前　言

一、ZLG72128 的诞生

在嵌入式系统中，数码管和键盘使用得十分广泛，特别是一些需要简单人机交互的应用场合：仪器仪表、工业控制器、条形显示器、控制面板等。在传统设计中，往往使用 MCU 的 I/O 口直接驱动数码管和键盘，这种设计有着明显的缺点：十分耗费系统的 I/O 资源（数码管和键盘都很耗费 I/O 口）和 CPU 资源（数码管和按键扫描均需占用 CPU 资源）。

为了解决传统设计中的缺陷，广州致远电子股份有限公司（http://www.zlg.cn/，后文简称 ZLG）研发设计了一款专用芯片：ZLG72128。该专用芯片可以同时管理 32 只按键和12 个数码管（或 96 个 LED，每个数码管实质由 8 个 LED 组成，共计 96 个 LED）。

主控 MCU 与 ZLG72128 之间采用标准 I^2C 接口通信，最少仅需 2 根线。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，数码管显示和键盘扫描完全由该专用芯片管理，这也会减轻主控 MCU 的 CPU 负担以及软件工程师的编程负担（无需开发数码管扫描和按键扫描相关的程序）。

为便于更好的适应行业需求，除了基础的数码管显示和键盘管理功能外，ZLG72128 还提供了丰富的扩展功能：对于数码管显示，为丰富显示效果，还支持闪烁、移位、段控制等功能；对于键盘管理，还提供了功能键、长按、连击计数等功能。

二、存在的问题

诚然，市面上已经有一些与 ZLG72128 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 ZLG72128 内部寄存器的含义），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 ZLG72128 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 ZLG72128 的软件工程师编写，也可作为了解 ZLG72128 的阅读资料。此外，书中讲解了部分与 ZLG72128 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。

周立功

2019 年 4 月 2 日

目 录

第1章 ZLG72128 简介

✍本章导读

ZLG72128是广州致远电子股份有限公司自行设计的数码管显示驱动及键盘扫描管理芯片，能够直接驱动 12 位共阴式数码管（或 96 只独立的 LED），同时还可以扫描管理多达32 个按键。通信采用 I²C 总线，与微控制器的接口最低仅需两根信号线。该芯片为工业级芯片，抗干扰能力强，在工业测控中已有大量应用。本章将对 ZLG72128作简要介绍。

1.1 特点

- 直接驱动 12 位 1 英寸以下的共阴式数码管或 96 只独立的 LED；
- 利用功率电路可以方便地驱动 1 英寸以上的大型数码管；
- 具有位闪烁、位消隐、段点亮、段熄灭、显示移位等强大功能；
- 具有 10 种数字和 21 种字母的译码显示功能，亦可直接向显示缓存写入显示数据；
- 软件配置支持 0 ~ 12 个数码管显示驱动模式（即配置数码管实际使用个数）；
- 与 MCU 之间采用 I²C 串行总线接口，一条 I²C 总线可挂接两片 ZLG72128 芯片；
- 能够管理多达 32 个按键（8 个功能键 + 24 个普通按键），自动消除抖动；
- 功能键如同电脑键盘上的 Ctrl、Shift 和 Alt 键；
- 普通按键具有连击计数功能，长按时可连续有效；
- 工作电压范围：3.0～5.0V；
- 工作温度范围：－40～＋85℃；
- 封装：TSSOP28。

1.2 引脚说明

ZLG72128 共计 28 个引脚，引脚排列详见图 1.1，引脚说明详见表 1.1。

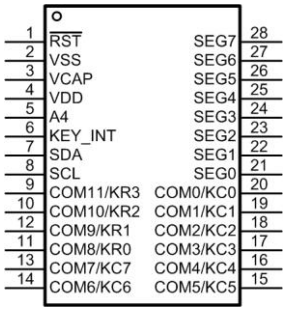


图 1.1 ZLG72128 管脚排列图

表 1.1 ZLG72128 引脚功能表

引脚号	引脚名称	功能描述
1	RST	复位信号，低电平有效
2	GND	接地
3	VCAP	外置电容端口
4	VDD	电源（3.0V ~ 5.0V）
5	A4	器件地址设置端口（决定了 7 位从机地址的第 4 位）
6	KEY_INT	键盘中断输出，低电平有效
7	SDA	I²C 总线数据信号
8	SCL	I²C 总线时钟信号
9 ~ 12	COM11 ~ 8 / KR3 ~ 0	数码管位选信号 11 ~ 8 / 键盘行信号 3 ~ 0
13 ~ 20	COM7~0 / KC7 ~ 0	数码管位选信号 7 ~ 0 / 键盘列信号 7 ~ 0
21 ~ 28	SEG0 ~ SEG7	数码管 a ~ dp 段

1.3 通信模型

ZLG72128 作为 I²C 从机，主控 MCU 可以通过 4 个 I/O 口完成对 ZLG72128 的控制，通信模型详见图 1.2。其中，RST 用于复位 ZLG72128，KEY_INT 用于 ZLG72128 检测到按键时，通过该引脚通知主机，SDA 和 SCL 用于 I²C 通信。图中的 KEY_INT 和 RST 使用了虚线表示，表明该引脚连接是可选的。实际应用中，为了减少 ZLG72128 对主控 MCU 的 I/O 占用，RST 引脚可以不由主控 MCU 控制，直接在外部设计一个 RC 复位电路（详见下节介绍的典型应用电路）即可。KEY_INT 中断信号输出引脚也可以不连接至主控 MCU（悬空即可），此时，主控 MCU 将无法得到按键事件的中断通知，则其必须以查询模式（例如，每隔 5ms 查询一次）检测是否有按键按下。

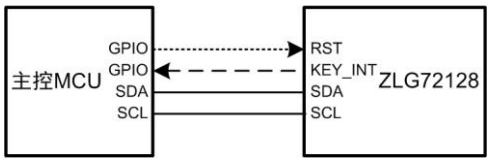


图 1.2　ZLG72128 通信模型

1.4 典型应用电路

如图 1.3 所示为 ZLG72128 的典型应用电路原理图，使用一片 ZLG72128 管理了 12 位数码管和 32 个按键。

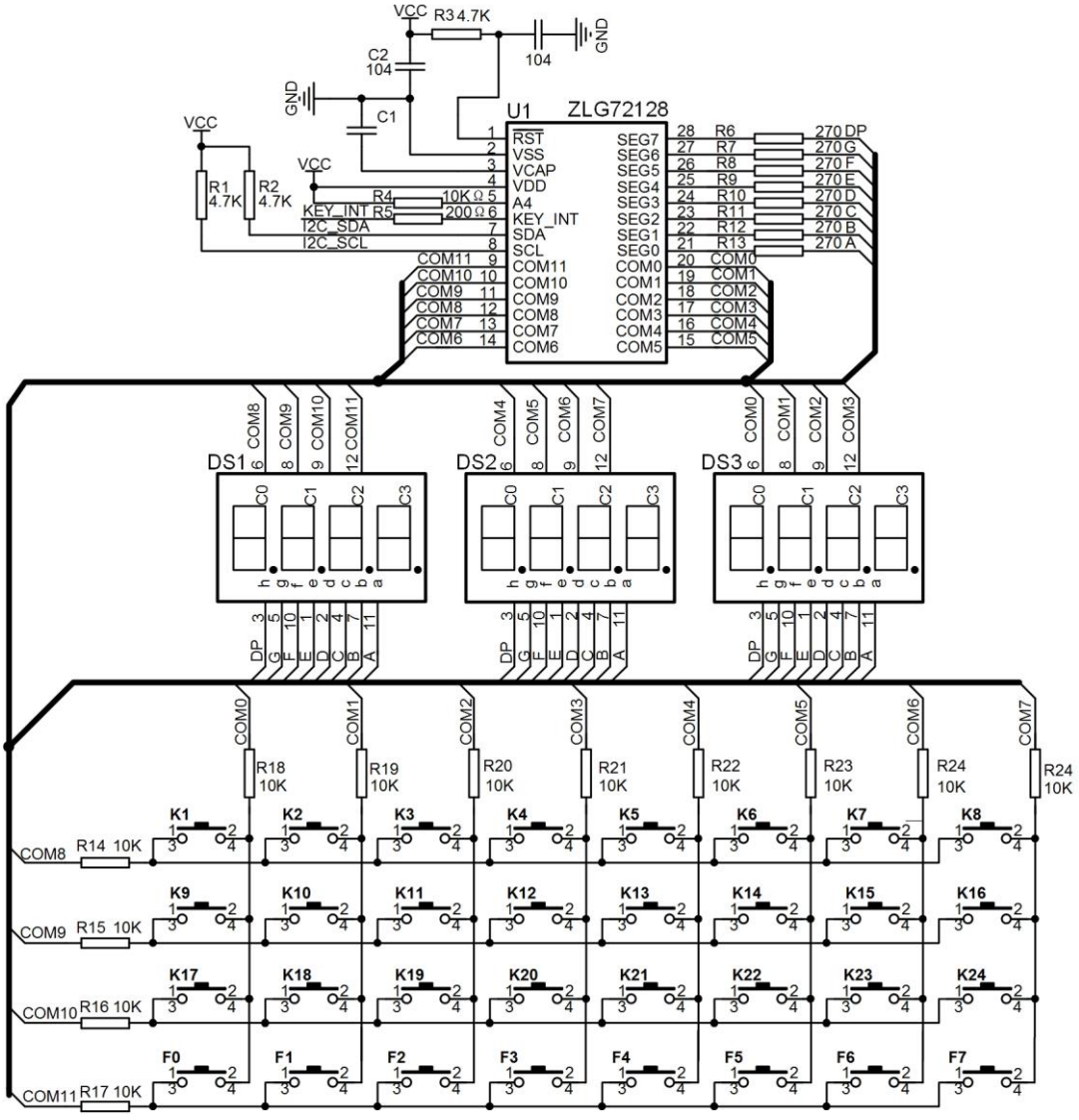


图 1.3　ZLG72128 典型应用电路

在实际应用中，使用的数码管和按键个数可能超过单片 ZLG72128 支持的最大数量，此时，可以在一路 I²C 总线上连接 2 片 ZLG72128，进而管理高达 24 位数码管和 64 个按键。应用电路详见图 1.4。

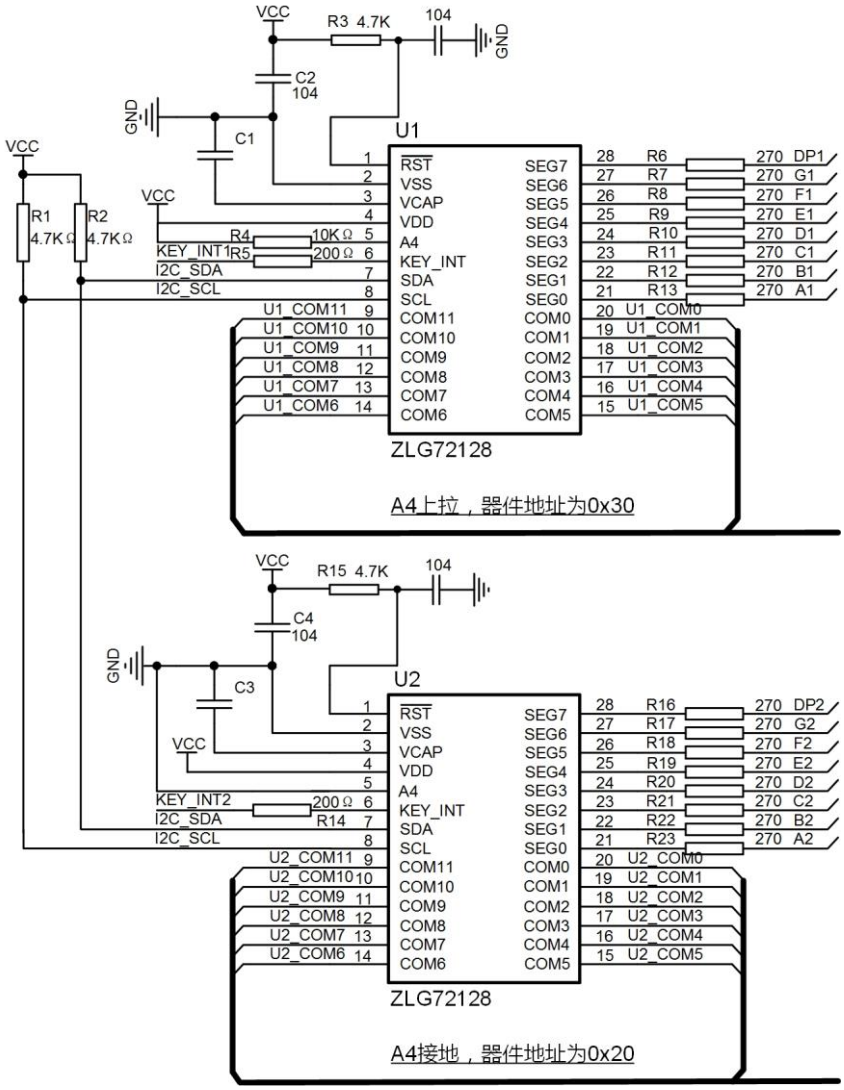


图 1.4　一路 I²C 总线上挂接两片 ZLG72128

其中一片 ZLG72128 的 A4 接高电平，另一片 ZLG72128 的 A4 接低电平，以保证两片 ZLG72128 的从机地址不同，有关 I²C 地址的更多细节，在 2.2 节中还会作进一步介绍。图中没有展示出数码管部分和按键部分的电路，这部分电路和图 1.3 所示的电路是完全相同的。

上面描述了在实际使用的数码管和按键个数过多（超过单片 ZLG72128 支持的最大数量）时的处理办法。与之对应的，还有另外一种情况：实际使用的数码管和按键个数较少，可能低于 ZLG72128 支持的最大数量，即不会用到全部的数码管和按键。此时，在设计硬件电路时，可以根据实际情况进行裁剪。键盘应按照整行和整列的方式进行裁剪，即部分行线和列线引脚不使用，以减少矩阵键盘的行数和列数，进而达到减少按键数量的目的。数码管对应的位选引脚为 COM0～COM11，如果不需要使用全部的 12 位数码管，则只需要从 COM0 开始，根据实际使用的数码管个数，依次将各个数码管的位选线与 COMx 引脚相连即可，例如，使用 8 个数码管，则 COM0～COM7 作为 8 个数码管的位选线。仅使用两位数码管和 4 个按键的典型原理图设计详见图 1.5。

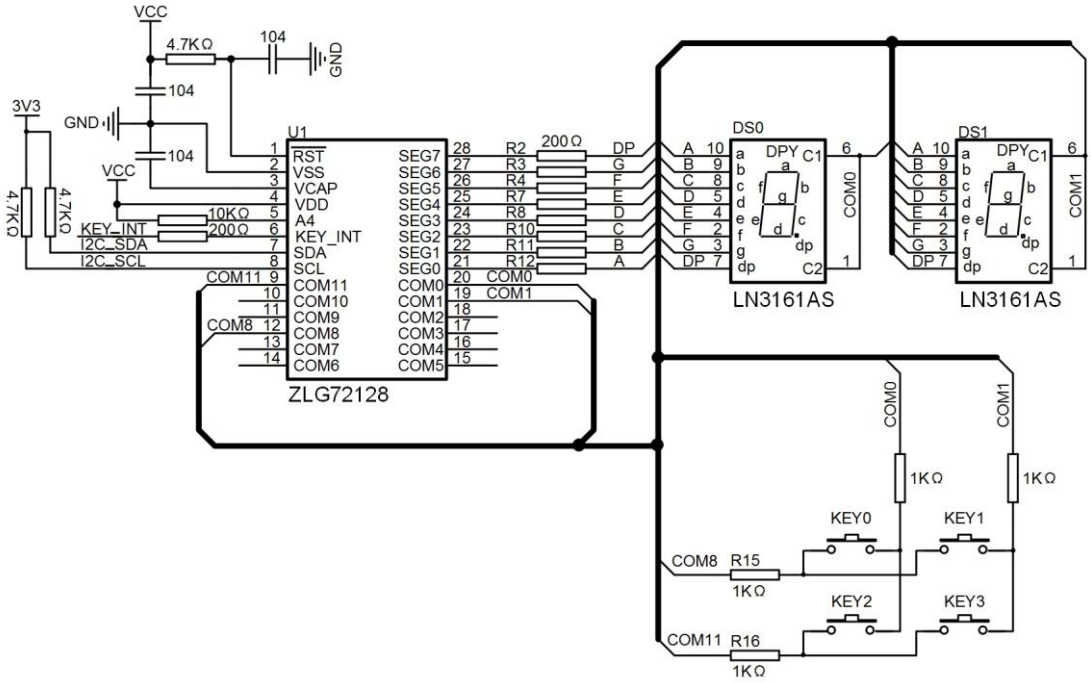


图 1.5　ZLG72128 典型应用电路（仅使用部分数码管和按键）

ZLG72128 只能直接驱动 12 位共阴式数码管，不能驱动共阳式数码管。在数码管的段与 ZLG72128 芯片引脚之间需要接一个限流电阻，其典型值为 270Ω。在实际应用中，可以可以适当减小电阻值，以增大数码管的亮度，阻值最小为 200Ω。在驱动一些大型数码管时，阻值最小时亮度可能依旧不够，此时，就必须加入外部功率驱动电路。

更多芯片相关的详尽说明（例如，外部功率驱动电路的设计）详见《ZLG72128 产品数据手册》，可以在 ZLG 官网（http://new.zlgmcu.com/Category_2395/Index.aspx#）获得最新的产品数据手册。

第2章 ZLG72128 功能详解

✍本章导读

ZLG72128 内部具有一系列寄存器。对于数码管显示，主控 MCU 通过 I²C 总线向指定寄存器中写入显示数据，ZLG72128 自动将数据对应的图形显示到数码管上；对于按键管理，ZLG72128 内部自动对按键进行扫描，扫描到按键事件后通过 KEY_INT 引脚通知主控 MCU，主控 MCU 可以通过 I²C 总线读取相应的键值寄存器，获取具体按下的按键。总之，数据的交互均是通过 I²C 总线读/写 ZLG72128 的内部寄存器实现的，如需深入理解 ZLG72128，直接通过 I²C 总线操作 ZLG72128，则必须对内部寄存器有一定的了解，本章将重点介绍各个内部寄存器的含义。实际上，ZLG 已经提供了通用驱动软件包，用户大可不必深入了解各个寄存器的含义，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 用户，ZLG72128 已经适配了标准接口，用户无需关心任何底层细节，仅需使用通用接口即可完成对 ZLG72128 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 ZLG72128 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 ZLG72128 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 寄存器详解

ZLG72128 内部共计 23 个寄存器，均为 8 位寄存器，它们的寄存器地址为 0x00 ~ 0x1B（中间有部分地址保留，暂未使用），各寄存器的介绍详见表 2.1。

表 2.1 ZLG72128 寄存器列表总览

寄存器名称	标识	地址	复位值	读/写特性
系统寄存器	SystemReg	0x00	0x00	只读（读清零）
键值寄存器	Key	0x01	0x00	只读（读清零）
连击计数器	RepeatCnt	0x02	0x00	只读
功能键寄存器	FunctionKey	0x03	0xFF	只读
命令缓冲区（2 个）	CmdBuf0、CmdBuf1	0x07、0x08	——	只写
闪烁控制寄存器	FlashOnOff	0x0B	0x77	读/写
消隐寄存器（2 个）	DispCtrl0、DispCtrl1	0x0C、0x0D	0x00	读/写
闪烁寄存器（2 个）	Flash0、Flash1	0x0E、0x0F	0x00	读/写
显示缓冲区（12 个）	DispBuf0 ~ DispBuf11	0x10 ~ 0x1B	0x00	读/写

2.1.1 系统寄存器

系统寄存器的位定义详见表 2.2。

表 2.2 系统寄存器位定义

位	标识	描述	复位值
7:1	——	保留	0
0	KeyAvi	0：无按键事件发生；1：有按键事件发生	0

ZLG72128 会实时监测按键的状态，当有普通键按下，或者功能键按下或释放时，视为产生了按键事件，系统寄存器的 bit0 值将变为 1，同时，通过 KEY_INT 引脚通知主控 MCU。

正常情况下，当有按键事件发生时，主控 MCU 只需要通过 KEY_INT 引脚判断就可以了。但特殊情况下，为了节省一个 I/O 口，可能不使用 KEY_INT 引脚，此时，主控 MCU 通过 I^2C 总线查询 KeyAvi 的值也可以判断当前是否有按键事件发生。但查询方式主要存在以下两个缺点：主控 MCU 每隔一段时间都要通过 I^2C 总线查询一次，耗费 CPU 资源；I^2C 总线处于频繁的活动状态，会在一定程度上导致功耗的增加，并且不利于抗干扰。

2.1.2 键值寄存器

当有按键事件发生时（系统寄存器的值为 1）键值寄存器中存放了普通按键的键值，其有效值为 1～24，分别对应 24 个普通按键，第 1 行第一个按键按下则值为 1，第 3 行最后一个按键按下则值为 24。特别地，若值为 0，则表明无普通键按下，此时，按键事件的发生可能是由于功能键按下或释放造成的。

键值寄存器中的值被读走后，将自动清 0，以便为存放下一次按键事件做好准备。

2.1.3 连击计数器

ZLG72128 为普通按键（第 1 行 ～ 第 3 行的按键）提供了连击计数功能。所谓连击，是指按住某个普通键不松手（即按键长按）时，也会连续产生按键事件。具体过程如下：首次按下按键时，产生一次按键事件（此时，连击计数器的值为 0），若按键一直保持按下状态，则在按住时间达到 1s 后，产生一次按键事件（连击计数器的值为 1），随后，ZLG72128 将以 200ms 的时间间隔产生按键事件，键值存入键值寄存器中，连击计数器的值在每次产生按键事件时加 1。这一过程对应的示意图详见图 2.1。

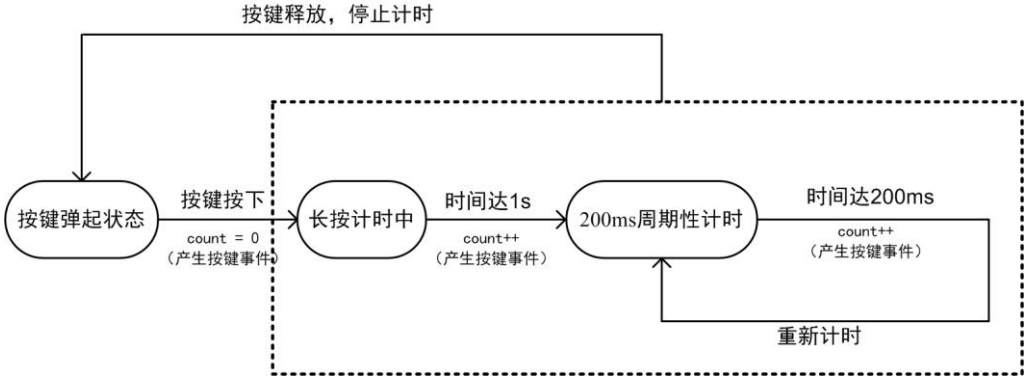


图 2.1　长按普通按键时的事件产生过程

简单地说，按键长按过程中，会启动按键长按计时，然后在时间达到特定值时产生按键事件，在计时过程中，只要按键释放，将停止计时，回到按键弹起状态。

这一特性与电脑上的键盘很类似，在字符输入时，按键按下时立即输入一个字符（产生按键事件），若按键保持按下，则一定时间后，将会以很快的速度连续输入字符（连续产生按键事件）。

注意，由于连击计数器为 8 位寄存器，因此，当 count 值递增到 255 后（长按时间约 52 秒，一般不会长按这么久），将不再加 1，但按键事件依然会每隔 200ms 产生一次。

2.1.4 功能键寄存器

ZLG72128 提供有 8 个功能键（图 1.3 中的 F0～F7）。功能键常常配合普通键一起使用，

就像电脑键盘上的 Shift、Ctrl 和 Alt 键。当然，功能键也可以单独使用，就像电脑键盘上的 F1～F12。功能键寄存器的值由 8 个功能键的状态组成，每一位的值与一个功能键对应，第 0 位（LSB）对应 F0，第 1 位对应 F1，第 7 位（MSB）对应 F7。若按键按下，则相应位的值为 0，否则，相应位的值为 1，初始状态，所有键均未按下，因此，功能键寄存器的初始值为 0xFF。只要功能键寄存器的值发生改变，就会产生一次按键事件：系统寄存器的值置 1，KEY_INT 输出低电平。例如，初始功能键寄存器的值为 0xFF，若 F0 按下，则第 0 位变为 0，值变为 0xFE，产生一次按键事件；当 F0 释放时，则第 0 位恢复为 1，值恢复为 0xFF，由于此时值也发生了改变，因此也会产生一次按键事件，即功能键的按下或释放均为产生按键事件，而普通按键释放时不会产生按键事件。

功能键寄存器的值实时反映 8 个功能按键的状态，读不会清 0，此外，功能按键不提供连击计数功能。

2.1.5 命令缓冲区

命令缓存区共包含两个只写寄存器：CmdBuf0、CmdBuf1。通过向命令缓冲区写入控制命令可以实现段寻址、下载显示数据等功能。当前支持的所有命令详见表 2.3。

按字节数目划分，命令可以分为单字节命令和双字节命令：单字节命令表示命令仅包含一个字节，发送命令时仅需向 CmdBuf0 中写入命令数据；双字节命令表示命令包含两个字节，发送命令时需依次向 CmdBuf0 和 CmdBuf1 中写入命令数据。

表 2.3　ZLG72128 命令总览

命令名称	标识	命令类型	简介
段寻址	SegOnOff	双字节命令	直接控制数码管的指定段
下载数据并译码	Download		直接指定显示的字符，内部自动解码
复位命令	Reset	单字节命令	复位，清除寄存器值默认值，包括显存内容
测试命令	Test		可用于测试 ZLG72128 是否正常工作
左移命令	ShiftLeft		控制数码管显示内容左移
循环左移命令	CyclicShiftLeft		控制数码管显示内容循环左移
右移命令	ShiftRight		控制数码管显示内容右移
循环右移命令	CyclicShiftRight		控制数码管显示内容循环右移
数码管扫描位数设置	Scanning		设置实际数码管的扫描位数

1． 段寻址

段寻址命令可用于直接控制某段数码管的亮灭，其为双字节命令，CmdBuf0 和 CmdBuf1 的位定义分别详见表 2.4 和表 2.5。

表 2.4　段寻址命令——CmdBuf0

位	标识	描述
7:4	cmd_type	命令码，固定为：0001
3:1	Reserved	保留位，当前应设置为 0
0	on	表示本次操作是点亮指定段（1），还是熄灭指定段（0）

注：所有命令中，CmdBuf0 的高 4 位为命令码，用以区分不同的命令。

表 2.5　段寻址命令——CmdBuf1

位	标识	描述
7:4	Bit addr	位地址，用以指定控制的数码管位，有效值：0 ~ 11
3:0	Seg addr	段地址，用以指定控制的数码管段，有效值：0 ~ 7，分别对应 a ~ dp 段

例如，要点亮第 2 个数码管对应的 dp 段（显示小数点），则双字节命令的值分别为：

```
CmdBuf0 = (0x01 << 4) | (1 << 0);
CmdBuf1 = (0x02 << 4) | (7 << 0);                    // 7 对应了 dp 段
```

2. 下载数据并译码

下载数据并译码命令，可用于直接传输要显示的字符数据（支持 10 种数字和 21 种字母），ZLG72128 内部自行解码，用户无需关心数据的解码操作。其同样为双字节命令，CmdBuf0 和 CmdBuf1 的位定义分别详见表 2.6 和表 2.7。

表 2.6　下载数据并译码命令——CmdBuf0

位	标识	描述
7:4	cmd_type	命令码，固定为：0010
3:0	Addr	数码管位地址，指定要设置显示数据的数码管，有效值：0 ~12

表 2.7　下载数据并译码命令——CmdBuf1

位	标识	描述
7	dp	小数点是否需要点亮：1，点亮；0，熄灭
6	flash	数码管是否闪烁：1，闪烁；0，不闪烁
5	Reserved	保留位，当前应设置为 0
4:0	data	表示要显示的数据，值范围：0 ~ 31，各个值对应的显示数据详见表 2.8

表 2.8　下载数据并译码命令的数据表

2 进制	16 进制	显示	2 进制	16 进制	显示	2 进制	16 进制	显示
00000	00H	0	01011	0BH	b	10110	16H	p
00001	01H	1	01100	0CH	C	10111	17H	q
00010	02H	2	01101	0DH	d	11000	18H	r
00011	03H	3	01110	0EH	E	11001	19H	t
00100	04H	4	01111	0FH	F	11010	1AH	U
00101	05H	5	10000	10H	G	11011	1BH	y
00110	06H	6	10001	11H	H	11100	1CH	c
00111	07H	7	10010	12H	i	11101	1DH	h
01000	08H	8	10011	13H	J	11110	1EH	T
01001	09H	9	10100	14H	L	11111	1FH	无显示
01010	0AH	A	10101	15H	o	——	——	——

3. 复位命令

复位命令用于清除寄存器设置，包括显存，执行复位命令后，将清除所有的数码管显示，复位命令为单字节命令，其位定义详见表 2.9。

表 2.9　复位命令——CmdBuf0

位	标识	描述
7:4	cmd_type	命令码，固定为：0011
3:0	Reserved	保留位，当前应设置为 0

由于保留为当前均设置为 0，因此，复位命令为固定值：0x30。

4. 测试命令

测试命令用于测试数码管显示是否正常，执行测试命令后，所有数码管的段将会以 1Hz 的频率闪烁，即所有段点亮 0.5s，然后再熄灭 0.5s，如此循环。测试命令为单字节命令，其位定义详见表 2.10。

表 2.10　测试命令——CmdBuf0

位	标识	描述
7:4	cmd_type	命令码，固定为：0100
3:0	Reserved	保留位，当前应设置为 0

由于保留位当前均设置为 0，因此，测试命令为固定值：0x40。

5. 移位命令

ZLG72128 支持 4 种移位方式：左移、循环左移、右移和循环右移。循环移位和普通移位的区别为：普通移位，移出的数据将丢失；循环移位，移出的数据将顺序添加至移位后留出的空位中，数据不会丢失。

例如，对于左移。普通移位后，左边移出的数据将丢失，右边空出的位将不显示任何内容。若数码管显示的内容为：0123456789Ab，在左移 2 位后，显示将变为：23456789Ab??（使用?表示左移后右边留出的空位）；若采用循环移位，则左边移出的数据将顺序添加至后边空出的位，即显示内容“0123456789Ab”在循环左移 2 位后，显示将变为：23456789Ab01。

移位命令为单字节命令，4 种移位方式都具有相应的命令，命令格式的定义详见表 2.11。

表 2.11　移位命令——CmdBuf0

位	标识	描述
7:4	cmd_type	命令码，4 种移位方式对应的命令码分别为： 0101 ： 左移 0110 ： 循环左移 0111 ： 右移 1001 ： 循环右移
3:0	BitShift	移位数，有效值： 1 ~ 11，0 或大于 11 的值无效。

各种移位方式对应的 CmdBuf0 值设置示例如下：

```
CmdBuf0 = (0x05 << 4) | (1);                    // 左移 2 位
CmdBuf0 = (0x06 << 4) | (3);                    // 循环左移 3 位
CmdBuf0 = (0x07 << 4) | (5);                    // 右移 5 位
CmdBuf0 = (0x08 << 4) | (8);                    // 循环右移 8 位
```

6. 数码管扫描位数设置

该命令用于设置实际的数码管扫描位数。ZLG72128 虽然最高可以支持 12 位数码管，但在实际应用中，不一定使用全部的 12 位数码管。例如，只使用 8 个数码管，则可以裁剪掉 COM8 ~ COM11 所对应的数码管（即这些位选线不连接数码管），COM0 ~ COM7 将分别作为 8 个数码管的位选线，这种情况下，由于高 4 位 COM 口没有连接数码管，因此在实际扫描时，可以仅扫描低 8 位，不扫描高 4 位，数码管的扫描位数减少后，有用的显示位由于分配的扫描时间更多，因而显示亮度可以得到一定程度上的提高。

数码管扫描位数设置命令为单字节命令，其位定义详见表 2.12。

表 2.12　数码管扫描位数设置命令——CmdBuf0

位	标识	描述
7:4	cmd_type	命令码，固定为：1001
3:0	BitNum	实际扫描位数，有效值：0 ~ 12，大于 12 时依然视为 12

例如，需要修改实际扫描位数为 8 位，则对应的命令值为：

```
CmdBuf0 = (0x09 << 4) | (8);                    // 数码管扫描位数设置为 8
```

2.1.6 闪烁控制寄存器

闪烁控制寄存器的值决定了闪烁频率和占空比，位定义详见表 2.13。

表 2.13　闪烁控制寄存器位定义

位	标识	描述	复位值
7:4	on_time	闪烁时，数码管点亮的持续时间 取值范围 0 ~ 15，实际时间值 = N × 50 + 150（ms）	7
3:0	off_time	闪烁时，数码管熄灭的持续时间 取值范围 0 ~ 15，实际时间值 = N × 50 + 150（ms）	7

高 4 位表示闪烁时亮的持续时间，低 4 位表示闪烁时灭的持续时间。它们的取值范围均为 0 ~ 15，实际时间值可以通过如下公式进行计算：

$$T = N \times 50 + 150(ms)$$

其中，N 为 on_time 或 off_time 的值，T 为对应的实际时间。

复位值为 7，可以求得对应的时间为 500ms，即默认情况下，一个闪烁周期内，数码管点亮的时间和熄灭的时间均为 500ms，即闪烁频率为 1Hz，占空比为 50%。

若将寄存器的值修改为 0x00，则点亮和熄灭的时间均为 150ms，此时，一个闪烁周期的时间最短，为 300ms，闪烁频率最快，约 3.33Hz。

若将寄存器的值修改为 0xFF，则点亮和熄灭的时间均为 900ms，此时，一个闪烁周期的时间最长，为 1800ms，闪烁频率最慢，约 0.56Hz。

特别注意，单独设置 FlashOnOff 寄存器的值，并不会观察到闪烁现象，还必须使能相应的数码管闪烁才能观察到闪烁现象。

2.1.7 消隐寄存器

消隐寄存器包含两个寄存器：DispCtrl0、DispCtrl1。这两个寄存器一起组成一个 16 位的控制值，DispCtrl0 作为高 8 位，DispCtrl1 作为低 8 位，详见表 2.14。

表 2.14　消隐寄存器

DispCtrl0								DispCtrl1							
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

在组成的 16 位数值中，高 4 位保留，暂未使用。低 12 位分别与一个数码管位对应，位 0 与 COM0 所接数码管对应，位 11 与 COM11 所接数码管对应。若相应位的值为 1，则对应的数码管位不显示，否则，对应的数码管位正常显示。复位值为 0x00，即所有数码管默认情况下都正常显示。

在实际应用中，可能需要显示的位数不足 12 位，例如只显示 8 位，这时可以把 DispCtrl0 的值设置为 0x0F，把 DispCtrl1 的值设置为 0x00，则仅扫描显示 8 位数码管。

2.1.8 闪烁寄存器

闪烁寄存器包含两个寄存器：Flash0、Flash1。这两个寄存器一起组成一个 16 位的控制值，Flash0 作为高 8 位，Flash1 作为低 8 位，详见表 2.15。

表 2.15　闪烁寄存器

Flash0								Flash1							
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

在组成的 16 位数值中，高 4 位保留，暂未使用。低 12 位分别与一个数码管位对应，位 0 与 COM0 所接数码管对应，位 11 与 COM11 所接数码管对应。若相应位的值为 1，则对应的数码管位闪烁，否则，对应的数码管位正常显示，不闪烁。复位值为 0x00，即所有数码管默认情况下都未开启闪烁。

在实际应用中，可能需要某些位闪烁，例如最后 2 位闪烁，这时可以把 Flash0 的值设置为 0x00，把 Flash1 的值设置为 0x03。

2.1.9 显示缓冲区

显示缓冲区包含 12 个寄存器：DispBuf0 ~ DispBuf11。用以存储 12 个数码管显示的内容，即显存。DispBuf0 控制 COM0 所接数码管的显示内容，DispBuf11 控制 COM11 所接数码管的显示内容，以此类推。显存内容的读/写操作均是允许的。

显存中的 8 位数据分别与 8 个数码管段（a、b、c、d、e、f、g、dp）一一对应，LSB 对应 a，MSB 对应 dp，详见表 2.16。

表 2.16　闪烁控制寄存器位定义

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
dp	g	f	e	d	c	b	a

例如，要通过直接写入缓存的方式显示一个数字“1”。根据 8 段数码管的排列方式，显示数字“1”需要点亮 b 段和 c 段，需要点亮的段对应位设置为 1，其它位设置为 0，各个位的设置情况详见表 2.17。

表 2.17　显示数字“1”的缓存内容

dp	g	f	e	d	c	b	a
0	0	0	0	0	1	1	0

由此可得，对应缓存需要设置的值即为：0x06。若需在显示数字“1”的同时，显示小数点，则各个位的设置情况详见表 2.18。

表 2.18　显示数字“1”（并显示小数点）的缓存内容

dp	g	f	e	d	c	b	a
1	0	0	0	0	1	1	0

由此可得，对应缓存需要设置的值为：0x86。

2.2　I²C 数据传输

2.2.1 从机地址

ZLG72128 作为 I²C 从机，当使用主控 MCU 操作 ZLG72128 时，传输的第一个字节为从机地址+读/写方向位，示意图详见图 2.2。

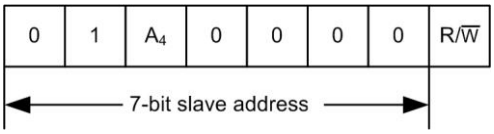


图 2.2　从机地址与读写命令组合示意图

ZLG72128 的 7 位 I²C 从机地址（未考虑读/写方向位）为：01$A_4$0000，其中，A_4 由 A4（#5）引脚的电平决定，若 A4 悬空或接高电平，则其 7 位 I²C 从机地址为：011 0000，即 0x30；若 A4 接低电平，则其 7 位 I²C 从机地址为：010 0000，即 0x20。由此可见，利用 A4 引脚可设置两个不同的器件地址，如此一来，在一条 I²C 总线就能连接两片 ZLG72128。

当考虑读/写方向位时，需要在 7 位从机地址后紧跟一个读/写方向位：0，写操作；1，读操作。例如，当 A4 悬空或接高电平时，写操作发送的地址为：01100000，即 0x60；读操作发送的地址为：01100001，即 0x61。

2.2.2 写数据

当器件寻址完成后，应发送一字节的寄存器地址，以指定从哪个寄存器开始写入，紧接着即可连续写入多字节数据（写入过程中，寄存器的地址会自动递增，以连续按顺序设置多个寄存器的值），通信过程示意图详见图 2.3。

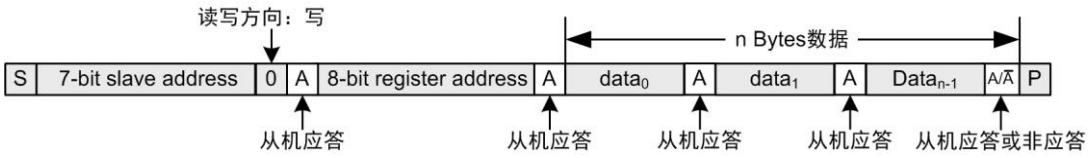


图 2.3　I²C 写数据——主机向从机发送数据示意图

图中，S 表示起始信号，P 表示停止信号。以灰底为背景的数据或信号表示由主控 MCU 发送至 ZLG72128，以白底为背景的数据或信号表示由 ZLG72128 发送至主机（如图中的应答信号）。这是 I²C 通信过程的一般表示法，若对 I²C 总线不甚了解（如起始信号、停止信号的定义等），可以阅读《ZLG72128 产品数据手册》或其它一些 I²C 标准规范文档。

2.2.3 读数据

读数据比写数据要多一个过程：写寄存器地址。即在读取数据前，首先也要执行一次写入操作，以将待读取的寄存器地址写入到 ZLG72128 中，整个通信过程的示意图详见图 2.4。

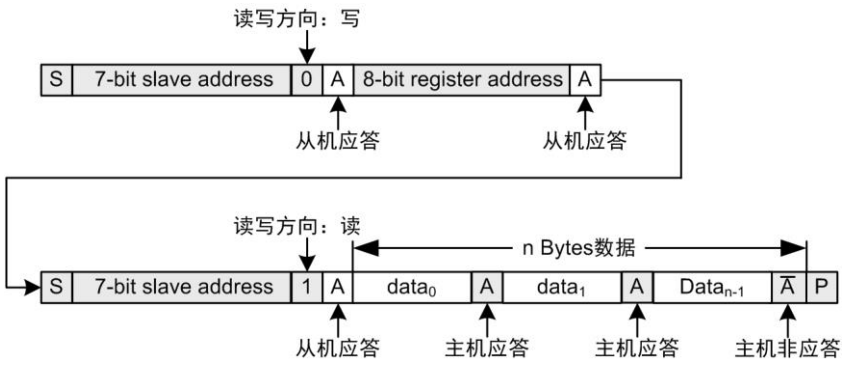


图 2.4　I²C 写数据——主机向从机发送数据示意图

在寄存器地址写入完成后，无需发送停止信号，直接再次发送起始信号，并将读/写方向修改为“读”即可，通常，将这种起始信号称之为“重复起始信号”，主要用于在通信过程中修改读/写方向。

第3章 ZLG72128 通用驱动软件包

✍本章导读

为了便于用户快速将 ZLG72128 应用到实际项目中，避免花费大量的时间去学习、了解 ZLG72128 的内部细节。特提供了 ZLG72128 驱动软件包，使用户可以通过软件包提供的各个接口完成对 ZLG72128 的基本操作，进而使用 ZLG72128 的数码管显示和按键管理功能。本章将对如何使用该软件包作详细的介绍。

在 3.3 和 3.4 节中，详细介绍了在不同平台中的适配方法以及 handle 的获取方法（ZLG72128 初始化）。这部分内容主要针对用户自有的特殊平台，若用户使用 AMetal、AWorks、Linux、Windows 等已经完成适配的平台，则可以跳过 3.3 和 3.4 节。直接阅读 3.5 节中介绍的各个功能接口以及平台相应的章节（例如：AMetal 对应第 4 章）。

3.1 软件包获取

ZLG72128 通用驱动软件包（为便于描述，后文将 ZLG72128 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 ZLG72128 数码管显示和按键管理功能的封装，使用户调用这些接口就可以快捷的使用 ZLG72128 提供的各种功能，同时，这些接口与具体平台或硬件无关，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 ZLG72128 资料集中，ZLG72128 资料集可以从致远电子官网（http://www.zlg.cn/）下载（或联系致远电子相关区域销售获取）。通用软件包的路径为：ZLG72128 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）。各文件简要说明详见表 3.1。

表 3.1 通用软件包文件说明

目录	文件名	功能描述
driver	zlg72128.h	各功能接口的声明，编写应用程序时查看
	zlg72128.c	各功能接口的实现，用户一般无需关心具体内容
	zlg72128_platform.h	与平台相关的接口声明、类型定义等
	zlg72128_platform.c	平台相关接口的实现，不同平台实现不同
examples	demo_zlg72128_entries.h	应用程序 demo 的入口函数声明
	demo_zlg72128_combination_key_test.c	组合键应用测试 demo
	demo_zlg72128_count_up_test.c	计数器测试 demo
	demo_zlg72128_digitron_test.c	数码管测试 demo
	demo_zlg72128_normal_key_test.c	普通键测试 demo

driver 目录下存放的文件是驱动的核心。其中，zlg72128.h 和 zlg72128.c 主要包含了 ZLG72128 功能接口的声明和实现，应用程序主要基于 zlg72128.h 文件中的接口编程；zlg72128_platform.c 和 zlg72128_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。

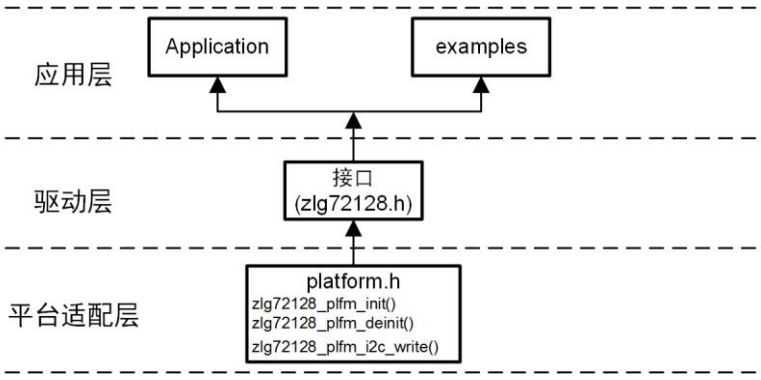


图 3.1 软件包基本结构

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 zlg72128.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 ZLG72128，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 ZLG72128 即可。基于此，若用户仅在这些平台中使用 ZLG72128，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 zlg72128.h 和 zlg72128.c 文件，它们实现了 ZLG72128 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 zlg72128_platform.h 和 zlg72128_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。

ZLG72128 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 ZLG72128 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 ZLG72128 的应用编程可能是相同的人（ZLG72128 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，使用户完全不需要关心 ZLG72128 的内部细节，降低了用户使用 ZLG72128 的难度，使用户可以更快的将 ZLG72128 应用到实际项目中。由主控 MCU 与 ZLG72128 之间的连接可知（详见图 1.2），主控 MCU 与 ZLG72128 之间主要通过 I²C 总线和 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，I²C 数据传输和 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 I²C 数据传输和 GPIO 控制方法。

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 I²C 的通信方法），实际上，本节讲解的适配方法主要是针对部分用户适配自有平台而言的，ZLG 已经对常用平台（AWorks、AMetal、Linux 和 Windows 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，直接使用下节介绍的各个功能接口操作 ZLG72128，非常便捷。需要适配的内容可以通过 zlg72128_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 zlg72128_platform.h 文件中，主要使用 typedef 定义了以下几个类型：

- bool_t ：布尔类型
- 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
- zlg72128_plfm_init_info_t ： 平台信息类型
- zlg72128_plfm_t ： 平台设备类型

它们的默认定义如下：

```
typedef     unsigned char       bool_t;
typedef     unsigned char       uint8_t;
typedef     unsigned short      uint16_t;
typedef     unsigned int        uint32_t;
typedef     void  *             zlg72128_plfm_init_info_t;
typedef     void  *             zlg72128_plfm_t;
```

对于前 4 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（zlg72128_plfm_init_info_t）和平台设备类型（zlg72128_plfm_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 ZLG72128 时，平台期望用户提供什么信息（比如引脚信息、中断号信息、I²C 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 ZLG72128 驱动时，需要保存的一些与 ZLG72128 相关的平台内部数据（例如，当前 I²C 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 ZLG72128 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 zlg72128_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

ZLG72128 驱动或 example 程序，可能需要使用“延时一段时间”的功能，不同平台可能具有不同的延时方法，因此，需要由平台根据实际情况实现一个延时函数。

该函数声明在 zlg72128_platform.h 文件中，声明如下：

```
void zlg72128_plfm_delay_ms (uint32_t ms);
```

由此可见，该函数无返回值，仅有一个表示延时时间（单位：毫秒）的参数。

在 zlg72128_platform.c 文件中，提供了一个简单的实现范例，详见程序清单 3.1。

程序清单 3.1　zlg72128_plfm_delay_ms()延时函数的默认实现

```
1    void zlg72128_plfm_delay_ms(int ms)
2    {
3        volatile    uint32_t    i;
4        while(ms--) {
5            for (i = 0; i < 1750; i++);
6    }
```

该延时函数的延时时间并不精确，也仅在 Cortex-M0@30MHz 的 MCU 中进行了简单的测试验证（保证其大约是毫秒级别的延时）。若需要精确的延时，用户可以基于定时器实现。

在特定的系统中，可能已经存在相应的延时函数，例如，在 AMetal 平台中，提供了 am_mdelay()函数，用于 ms 级别的延时，则该函数可以直接基于 am_mdelay()函数实现，详见程序清单 3.2。

程序清单 3.2　延时函数在 AMetal 中的实现

```
1    void zlg72128_plfm_delay_ms(int ms)
2    {
3        am_mdelay(ms);
4    }
```

3.3.4 其它平台相关函数

这类函数主要是平台相关的，且与 ZLG72128 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 zlg72128_platform.h 文件中声明，接口原型详见表 3.2。

表 3.2　软件包适配相关接口

函数原型	功能简介
int zlg72128_plfm_init (zlg72128_plfm_t *p_plfm, const zlg72128_plfm_init_info *p_plfm_init_info, void (*pfn_keyval_report) (void *, uint8_t reg [4]), void *p_key_arg);	平台初始化函数
int zlg72128_plfm_deinit(zlg72128_plfm_t *p_plfm);	平台解初始化函数
int zlg72128_plfm_i2c_write(zlg72128_plfm_t *p_plfm, uint8_t sub_addr, uint8_t *p_buf, uint8_t nbytes)	I^2C 写函数

这些接口在 zlg72128_platform.c 文件中实现。在默认的 zlg72128_platform.c 文件中，全部都是空函数，需要用户根据实际情况适配。

观察各个接口可以发现，所有函数的第一个参数均为 zlg72128_plfm_t*类型的 p_plfm，这里使用了一点面向对象编程的思想：将平台看作一个对象，基于平台相关的所有数据均保存在了 zlg72128_plfm_t 类型的平台设备中，通过该指针即可访问平台设备中的数据，以便进一步实现平台相关的功能。接下来，将详细介绍这几个接口的实现方法，以及 zlg72128_plfm_t 类型的定义。

1. 平台初始化函数

平台初始化函数用以完成与平台相关资源的初始化，例如，ZLG72128 需要使用到 KEY_INT 引脚、RST 引脚、I^2C 总线等，则这些资源相关的初始化操作都必须在平台初始化函数中完成。平台初始化函数会在后文介绍的 zlg72128_init()函数中被首先调用。

平台初始化函数的原型为：

```
int zlg72128_plfm_init ( zlg72128_plfm_t                 * p_plfm,
                         const zlg72128_plfm_init_info   *p_plfm_init_info,
                         void                            (*pfn_keyval_report) (void *, uint8_t reg0_3[4]),
                         void                             *p_key_arg);
```

该函数的返回值为 int 类型，主要用于返回初始化的结果，0 表示成功，非 0 表示失败（平台初始化失败时，zlg72128_init()也会随之失败）。参数有 4 个：p_plfm、p_plfm_init_info、pfn_keyval_report 和 p_key_arg。其中，p_plfm 为平台设备的指针，用于访问平台设备中的数据；p_plfm_init_info 为平台初始化时，可能需要用到的一些与平台相关的信息，例如引脚信息、I^2C 总线信息等；pfn_keyval_report 为键值上报函数；p_key_arg 为键值上报函数的参数。下面，分别对各个参数作详细介绍。

- p_plfm

p_plfm 是指向平台设备的指针，平台设备的类型为 zlg72128_plfm_t，该类型的具体定义同样由平台适配者完成。该类型主要用于保存一些必要的与平台相关的数据，例如，I^2C 传输可能有好几个状态，则可以将 zlg72128_plfm_t 定义为一个结构体类型，并存储一个状态变量，例如：

```
typedef struct {
    int state;
```

```
}zlg72128_plfm_t;
```

此外，参数 p_plfm_init_info 是一个指向平台信息的指针（该指针的具体用法在后文介绍 p_plfm_init_info 参数时详细介绍），如果在平台相关的其它函数中（例如，I^2C 写）需要使用到平台信息，则可以将该指针保存下来，以便后续使用，基于此，可以在 zlg72128_plfm_t 类型中新增一个成员：

```
typedef struct {
    int                                 state;
    const zlg72128_plfm_init_info       *p_plfm_init_info;
}zlg72128_plfm_t;
```

除此之外，参数 pfn_keyval_report 是一个函数指针，该指针指向的函数需要由平台调用（具体用法在后文介绍 pfn_keyval_report 参数时详细介绍），根据平台的实现，可能需要将该函数指针保存下来，以便在合适的时机调用，则可以在 zlg72128_plfm_t 类型中新增两个成员：

```
typedef struct {
    int                                 state;
    const zlg72128_plfm_init_info       *p_plfm_init_info;
    void                                (*pfn_keyval_report) (void *, uint8_t reg0_3[4]);
    void                                *p_key_arg;
}zlg72128_plfm_t;
```

总之，该类型可以根据需要任意定义，通常情况下，可以遵循这样一个原则：当期望平台相关的函数在运行结束后，保存一些数据，以便后续再使用，则可以在 zlg72128_plfm_t 类型中定义相应的成员，需要保存数据时，为该成员赋值即可。当然，若没有什么数据需要保存，则可以不使用 p_plfm 参数，zlg72128_plfm_t 类型保持默认的 void*即可。

ZLG72128 驱动会负责定义 zlg72128_plfm_t 类型的变量（开辟相应的内存空间），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_plfm 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。

平台设备中所有成员的初始化操作，通常都应在平台初始化函数中完成，范例程序详见程序清单 3.3。

程序清单 3.3　平台初始化函数实现范例

```
1   int zlg72128_plfm_init ( zlg72128_plfm_t                *p_plfm,
2                            const zlg72128_plfm_init_info  *p_plfm_init_info,
3                            void                           (*pfn_keyval_report) (void *, uint8_t reg0_3[4]),
4                            void                           *p_key_arg)
5   {
6       p_plfm->p_plfm_init_info    = p_plfm_init_info;
7       p_plfm->pfn_keyval_report   = pfn_keyval_report;
8       p_plfm->p_key_arg           = p_key_arg;
9       p_plfm->state               = 0;
10      return 0;
11  }
```

- p_plfm_init_info

平台在初始化 ZLG72128 时，可能需要使用到一些与平台相关的信息，例如引脚信息、

I²C 总线信息、I²C 设备从机地址等，这些信息可以包含在 zlg72128_plfm_init_info_t 类型中。在不同平台中，引脚等信息的表示方法可能不同，因而对应的类型可能不同。典型的，可以将 zlg72128_plfm_init_info_t 定义为一个结构体类型，用以保存平台相关的信息。

例如，系统使用 uint8_t 类型的 slv_addr 表示 I²C 设备从机地址，int 类型的 ID 表示 I²C 总线编号（系统中有多条 I²C 总线，指定使用哪条 I²C 总线），int 类型的数值标志引脚编号，则该类型可以定义为：

```
typedef struct {
    uint8_t     slv_addr;           // I²C 设备从机地址
    int         pin_rst;            // RST 引脚编号
    int         pin_key_int;        // KEY_INT 引脚编号
    int         i2c_id;             // I²C 总线编号
}zlg72128_plfm_init_info_t;
```

这些信息通常由用户根据实际的硬件连接提供，在特定的硬件中，这些信息通常不会变化，因而可以定义为 const 类型的常量。例如：

```
const zlg72128_plfm_init_info_t info = {
    0x30,
    PIO0_0,
    PIO0_1,
    0
};
```

基于此设计，后续硬件连接发生变化时，平台相关的代码也无需作任何变化，修改该结构体常量的值即可。

该类型和 zlg72128_plfm_t 类型相似，完全由平台适配者定义，其决定了用户在该平台下使用 ZLG72128 时，需要传递的一些与平台相关的基础信息，驱动在调用平台初始化函数时，原封不动的将平台相关的初始信息传递给平台初始化函数。该类型中所有成员的赋值与使用，都由平台相关函数负责，ZLG72128 驱动不会对其中的成员作任何操作，由此可见，软件包对于参数 p_plfm_init_info 只是进行传递和保存，参数中相关的成员也仅仅针对各自的平台，成员的使用也只与平台自身有关。

同理，若平台无需上层用户提供任何信息，则可以不使用 p_plfm_init_info 参数，zlg72128_plfm_init_info_t 类型保持默认的 void*即可。

- pfn_keyval_report

pfn_keyval_report 参数为平台需要伺机调用的“键值上报”函数，用户可通过调用此函数将获取到的键值上报给上层驱动。这里所谓的“键值”仅仅是一段与键值相关的数据：寄存器 0x00 ~ 0x03 这 4 个寄存器中的值，驱动适配者仅需选择在合适的时机上报这 4 个寄存器中的值即可，无需关心这 4 个寄存器中数据的具体含义，更不需要对其中的数据作任何检查和处理。换句话说，驱动适配者并不需要理解寄存器的含义。这里实际上仅仅定义了驱动适配者需要完成的 I²C 数据读操作：选择合适的时机，从寄存器起始地址 0 开始，连续读取 4 字节数据，然后将数据上报（不要对数据作任何处理）。上报形式为：

```
uint8_t     data[4];
plfm_i2c_read(0x00, 0x04, data);            // 从 0x00 寄存器地址开始，连续读取 4 字节数据，存储至 data
pfn_plfm_keyval_report(p_key_arg, data);    // 调用该函数将读取的 4 字节数据上报
```

其中，具体的 I²C 读取函数实现同样与平台相关。在调用该函数时，需要将 p_key_arg

作为该函数的第一个参数传递。针对 p_key_arg，平台适配者仅仅只需要简单传递即可，无需做其它任何操作。

什么时候是上报键值的合适时机呢？根据 ZLG72128 的特性，当有按键事件产生时，会通过 KEY_INT 引脚通知主控，因此，一个最佳的时机就是当产生 KEY_INT 中断时，这种“中断驱动”的方式有利于降低主控 MCU 的负担，主控 MCU 无需定期检查“是否有按键事件发生”。在部分需要低功耗的系统中，甚至可以将 KEY_INT 引脚用作唤醒引脚，可以将主控 MCU 从低功耗模式唤醒。在 KEY_INT 中断中上报键值的范例程序详见程序清单 3.4。

程序清单 3.4　在 KEY_INT 中断中上报键值

```
1   static void __int_callback (zlg72128_plfm_t *p_plfm)
2    {
3         uint8_t    data[4];
4         plfm_i2c_read(0x00, 0x04, data);
5         p_plfm->pfn_plfm_keyval_report(p_plfm->p_key_arg, data);
6    }
7
8   int zlg72128_plfm_init (zlg72128_plfm_t                  *p_plfm,
9                           const zlg72128_plfm_init_info_t  *p_plfm_init_info,
10                          void                             (*pfn_keyval_report) (void *, uint8_t reg0_3[4]),
11                          void                             *p_key_arg)
12  {
13        p_plfm-> pfn_plfm _keyval_report = pfn_keyval_report;      //保存函数
14        p_plfm->p_key_arg =  p_key_arg;                            //保存函数参数
15        gpio_trigger_connect(int_pin, __int_callback, p_plfm);     // 设置 GPIO 触发回调函数
16        gpio_int_enable(int_pin, TRIGGER_FALLING);                 // 开启引脚触发（下降沿）
17        // ....  其它处理
18  }
```

程序中没有在平台初始化函数中直接上报键值，而是在 KEY_INT 中断中上报键值，键值上报函数（pfn_keyval_report）需要在中断函数中使用，因此，在平台初始化函数中，将键值上报函数及其参数存储到了平台设备 p_plfm 中，以便在中断函数中取出调用。这就要求按照上文中 zlg72128_plfm_t 类型定义的举例，将 pfn_keyval_report 和 p_key_arg 保存在平台设备中，即：

```
typedef struct {
    // ......  其它成员
    void                            (*pfn_keyval_report) (void *, uint8_t reg0_3[4]);
    void                            *p_key_arg;
    // ......  其它成员
}zlg72128_plfm_t;
```

实际上，无论采用何种上报时机（后文还会介绍其它几种上报时机），通常都不会在平台初始化函数中直接上报（初始化时没有按键事件），而是在其它地方上报，因此，绝大部分情况下，平台适配者在定义 zlg72128_plfm_t 类型时，都可以按照上面的定义，将键值上报函数及其参数存于其中。

在程序清单 3.4 所示的中断处理函数中(__int_callback)，使用 I^2C 读取出了寄存器中的

数据并进行了上报。在实际应用中，I²C 读取数据通常耗费的时间在毫秒级别，对于部分实时性要求很高的场合，在中断中耗费如此长的时间可能不太合理，为此，也可以采用 I²C 异步传输的方式。例如，在引脚中断中只启动 I²C 读，然后函数立即结束，当 I²C 硬件完成数据读后，再调用相应的读取完成回调函数，在读取完成回调函数中进行键值的上报，范例程序详见程序清单 3.5。

程序清单 3.5　在 KEY_INT 中断中上报键值——I²C 采用异步读

```
1    void __i2c_read_complete (zlg72128_plfm_t *p_plfm)
2    {
3          p_plfm->pfn_plfm_keyval_report(p_plfm->p_key_arg, p_plfm-> key_data);
4    }
5    static void __int_callback (zlg72128_plfm_t *p_plfm)                    // KEY_INT 中断处理函数
6    {
7         // 启动读，读取完成调用 __i2c_read_complete，并将 p_plfm 作为 complete 函数的参数
8          plfm_i2c_read_start(0x00, 0x04, p_plfm-> key_data, __i2c_read_complete, p_plfm);
9    }
```

程序中，为了存储读取的数据，在平台设备中定义了一个 4 字节大小的缓存。即：

```
typedef struct {
    // ..... 其它成员
    uint8_t          key_data[4];
    // ..... 其它成员
}zlg72128_plfm_t;
```

此时，在 KEY_INT 中断中处理的事务就比较少了，对实时性的影响降低了很多。但是，在部分平台中（例如：没有硬件 I²C，只能使用 GPIO 模拟，或者即使存在硬件 I²C，实现异步方式也非常麻烦），可能不会选用这种方式，简单地，可以在中断中仅设置一个标志（如果在多任务环境中，则可以在中断中释放信号量），然后在主循环中查询该标志（如果在多任务环境中，则可以在任务中等待信号量），然后再读取出键值上报，范例程序详见程序清单 3.6。

程序清单 3.6　在主循环中上报键值——基于 KEY_INT 中断

```
1    static void __int_callback (zlg72128_plfm_t *p_plfm)                    // KEY_INT 中断处理函数
2     {
3          p_plfm->flag = 1;                                                  //  设置标志
4     }
5
6    // .....
7    int main()
8    {
9         while (1) {                                                        //  系统或任务主循环
10             if (flag == 1) {                                              //  检查标志
11                 uint8_t    data[4];
12                 plfm_i2c_read(0x00, 0x04, data);
13                 p_plfm->pfn_plfm_keyval_report(p_plfm->p_key_arg, data);
12                 p_plfm ->flag = 0;
```

```
14                  }
15          }
16  }
```

程序中，为了存储中断标志，在平台设备中定义了一个 flag 标志。即：

```
typedef struct {
    // ..... 其它成员
    volatile    uint8_t         flag;
    // ..... 其它成员
}zlg72128_plfm_t;
```

实际上，在裸机平台中，这种方式已经退化为了查询方式（需要不断轮询，当然，在多任务系统中，基于信号量的方式并不存在轮询）。这种情况下，为了避免再使用中断，耗费中断资源，也可以将查询 flag 标志修改为直接查询引脚电平，从而可以不再使用中断，范例程序详见程序清单 3.7。

程序清单 3.7　在主循环中上报键值——基于引脚电平

```
1   int main()
2   {
3       while (1) {                                                 // 系统或任务主循环
4           if (gpio_get(pin_key_int) == 0) {                       // KEY_INT 引脚为低电平
5               uint8_t    data[4];
6               plfm_i2c_read(0x00, 0x04, data);
7               p_plfm->pfn_plfm_keyval_report(p_plfm->p_key_arg, data);
8               p_plfm ->flag = 0;
9           }
10      }
11  }
```

当基于 KEY_INT 中断上报键值时，由于上报时已经明确产生了“按键事件”，因此，所有的上报都是有效的（键值数据中包含了有效的数据），这种工作机制效率最好。但在部分平台中，为了节省管脚，可能不耗费主控 MCU 的一个引脚与 KEY_INT 相连，此时，KEY_INT 悬空即可。这种情况下，就需要定期上报键值，例如，每隔 5ms 上报一次，范例程序详见程序清单 3.8。

程序清单 3.8　在主循环中上报键值——每隔 5ms

```
1   int main()
2   {
3       while (1) {                                                 // 系统或任务主循环
4           uint8_t    data[4];
5           plfm_i2c_read(0x00, 0x04, data);
6           p_plfm->pfn_plfm_keyval_report(p_plfm->p_key_arg, data);
7           zlg72128_plfm_delay_ms(5);                              // 延时 5ms
8       }
9   }
```

显然，查询方式效率较低，比较占用 MCU 资源，可能很多时候查询的结果都是“无按

键事件发生”。当然，是否有按键事件发生无需适配者判断，无论如何，都可以将键值上报，上层驱动会进行判断和处理。

在程序清单 3.8 中，直接在主循环中每隔 5ms 查询一次，当主循环中还要处理其它事务时，可能主循环中的处理逻辑会变得十分复杂（当然，在多任务中，可以单独开一个任务，在任务主循环中查询键值，则不会遇到这个问题），此时，也可以开一个定时器，将定时器的定时周期设定为 5ms，然后在定时器中断中上报键值，范例程序详见程序清单 3.9。

程序清单 3.9　使用定时器上报键值——每隔 5ms

```
1   static void __timeout_callback(zlg72128_plfm_t *p_plfm)
2    {
3           uint8_t    data[4];
4           plfm_i2c_read(0x00, 0x04, data);
5           p_plfm->pfn_plfm_keyval_report(p_plfm->p_key_arg, data);
6    }
7
8   int zlg72128_plfm_init (zlg72128_plfm_t                  *p_plfm,
9                           const zlg72128_plfm_init_info_t  *p_plfm_init_info,
10                          void                             (*pfn_keyval_report) (void *, uint8_t reg0_3[4]),
11                          void                             *p_key_arg)
12  {
13      p_plfm-> pfn_keyval_report = pfn_keyval_report;          // 保存函数
14      p_plfm->p_key_arg =  p_key_arg;                          // 保存函数参数
15      plfm_timer_enable(5, __timeout_callback,   p_plfm);      // 使能一个周期为 5ms 的定时器
16  }
```

此时，主循环无需做任何事情，程序结构性也变得更好。“伺机上报键值”是平台适配的一个核心工作，因此上文列举了很多具体适配的示例，以满足不同平台中各式各样的情况。综合来看，上报键值的时机主要有以下 4 种：

1) 在 KEY_INT 中断中上报；
2) 在 KEY_INT 中断中设置一个标志（或释放信号量），然后在主循环中检查标志（或等待信号量）；
3) 不使用 KEY_INT，直接在主循环中每隔一定时间（比如：5ms）上报一次；
4) 不使用 KEY_INT，使用定时器（比如：设定定时周期为 5ms），在定时器中断中上报。

- p_key_arg

参数 p_key_arg 为键值上报函数的参数，此参数由软件包内部定义并传入，驱动适配者无需关心其具体内容，仅需在调用 pfn_keyval_report 函数时，将其作为第一个传输传递即可。

上面介绍了平台初始化函数中各个参数的含义以及平台设备类型（zlg72128_plfm_t）的定义范例。在程序清单 3.3 中展示了平台初始化函数的第一个作用：为平台设备中的各个成员赋值。实际上，顾名思义，平台初始化函数主要需要完成平台相关的初始化操作，除了成员赋值外，还有一些硬件设备的初始化，与 ZLG72128 相关的硬件主要有：I^2C 总线、KEY_INT 引脚、复位引脚。因此，在平台初始化中，还需要根据实际情况完成这些硬件资源的初始化。

典型地，如果主控 MCU 通过一个 GPIO 连接至了 ZLG72128 的 RST 引脚（详见图 1.2），

则在初始化时，可以复位一次 ZLG72128，以确保 ZLG72128 在启动后正常可靠的工作，范例程序详见程序清单 3.10。

程序清单 3.10　平台初始化函数实现范例

```
1   static void __hw_reset (zlg72128_plfm_t *p_plfm)
2   {
3       int pin = p_plfm->p_plfm_init_info->pin_rst;        // 假定复位引脚编号存储在平台信息中
5
6       plfm_gpio_cfg(pin, GPIO_OUTPUT);                    // 配置 GPIO 为输出模式
7       plfm_gpio_set(pin, 0);                              // GPIO 输出低电平
8       zlg72128_plfm_delay_ms(1);                          // 延时 1ms
9       plfm_gpio_set(pin, 1);                              // GPIO 输出高电平
10      zlg72128_plfm_delay_ms(5);                          // 延时 5ms（确保 ZLG72128 已正常工作）
11  }
12  int zlg72128_plfm_init ( zlg72128_plfm_t                *p_plfm,
13                           const zlg72128_plfm_init_info  *p_plfm_init_info,
14                           void                           (*pfn_keyval_report) (void *, uint8_t reg0_3[4]),
15                           void                           *p_key_arg)
16  {
17      p_plfm->p_plfm_init_info    = p_plfm_init_info;
18      p_plfm->pfn_keyval_report   = pfn_keyval_report;
19      p_plfm->p_key_arg           = p_key_arg;
20      // 其它成员初始化
21
22      __hw_reset(p_plfm);                                 // 硬件复位
23
24      // .... I2C 初始化
25
26      return 0;
27  }
```

通常情况下，为了节省管脚，在设计硬件电路时，往往直接在 RST 引脚外部连接一个 RC 复位电路即可（详见图 1.3），这种情况下，主控 MCU 可以完全不理会 RST 引脚，无需在平台初始化时做任何复位相关的额外操作。

2. 平台解初始化函数

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 I²C 总线、KEY_INT 中断或定时器（若使用定时器定时上报键值）等。当 ZLG72128 不再被使用时，这些资源应该被相应的释放（或关闭），例如，可以关闭 I²C 控制器、关闭引脚中断等。与平台初始化函数相对应，这些操作在平台解初始化函数中完成。平台解初始化函数与平台初始化函数是相对应的，在平台初始化函数中打开、获得了什么资源，就应该在平台解初始化函数中关闭和释放。

平台解初始化函数会在后文介绍的 zlg72128_deinit()函数中被调用。平台解初始化函数的原型为：

```
int zlg72128_plfm_deinit(zlg72128_plfm_t    *p_plfm);
```

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，zlg72128_deinit()也会随之失败）。该参数仅 1 个 p_plfm 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_plfm 介绍时提到类型 zlg72128_plfm_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_plfm 中保存的数据进行相关资源的释放操作。范例详见程序清单 3.11。

程序清单 3.11　平台解初始化函数适配范例

```
1    int zlg_72128_plfm_deinit(zlg72128_plfm_t    *p_plfm)
2    {
3            plfm_pin_disable(p_plfm->int_pin);                  // 关闭引脚中断
4            plfm_i2c_deinit(p_plfm->i2c_handle);                // i2c 解初始化
5            // ..... 其它释放操作
6           return 0;
7    }
```

通过此范例程序可以看出，在适配平台解初始化函数时，若用户使用引脚中断获取键值，或者使用定时器轮询，则需要在平台解初始化函数中释放引脚，或者关闭定时器。同理，用户使用到的 I²C 相关信息也可以在此函数中进行释放，防止无用资源占用内存。

3. I²C 写函数

ZLG72128 模块与主控 MCU 之间采用的是 I²C 通信。由于在不同环境（操作系统或硬件 MCU）中，I²C 数据传输的具体实现方式各不相同，因此，该软件包将 I²C 数据传输的实现抽离了出来，交由各个具体平台实现。不同平台环境，需要根据实际情况，完成 I²C 数据的传输。不同平台在实现 I²C 数据传输时，无需关心通信数据的具体含义。对于用户来讲，通用软件包会提供一系列有意义的接口供用户使用，当用户调用这些接口时，会将用户对应的操作转换为与 ZLG72128 的通信命令，然后调用平台实现的 I²C 数据传输函数，将数据传输给 ZLG72128。

I²C 写函数用于传输数码管控制命令至 ZLG72128，其函数原型为：

```
int zlg72128_i2c_sync_write(zlg72128_plfm_t        *p_plfm,
                            uint8_t                 sub_addr,
                            uint8_t                *p_buf,
                            uint32_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 3.3。

表 3.3　I²C 同步写函数参数

名称	含义	详细描述
p_plfm	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
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_buf	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：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 I²C 数据传输时，最重要的职责就是根据这些参数信息完成 I²C 数据写。

熟悉 I²C 通信的读者可能会发现，在这些数据信息中，缺少了 I²C 通信必要的设备从机地址信息，在第一章节介绍 ZLG72128 模块时提到，ZLG72128 模块的从机地址与 A4 引脚的电平相关：高电平（或悬空）时地址为 0x30；低电平时地址为 0x20。因此，地址信息需要平台自行获取，若平台软件无法直接获取到 A4 引脚的电平，则可以交由用户提供该信息，简单地，可以在 zlg72128_plfm_init_info_t 类型中新增一个 slv_addr 成员，例如：

```
typedef struct {
    // ..... 其它成员
    uint8_t      slv_addr;      //设备 7-bit 从机地址
    // ..... 其它成员
}zlg72128_plfm_init_info_t;
```

在 I²C 写函数中，即可通过 p_plfm-> p_plfm_info->slv_addr 获取到从机地址信息（当然，前提是平台设备中保存了信息指针），即：

```
typedef struct {
    // ..... 其它成员
    zlg72128_plfm_init_info_t   *p_plfm_info;
    // ..... 其它成员
}zlg72128_plfm_t;
```

并在初始化时完成了该指针的赋值。范例程序详见程序清单 3.12。

程序清单 3.12　平台信息结构体保存设备从机地址范例

```
1   int zlg72128_plfm_init ( zlg72128_plfm_t                    *p_plfm,
2                            const zlg72128_plfm_init_info    *p_plfm_init_info,
3                            void    (*pfn_keyval_report) (void *, uint8_t reg0_3[4]),
4                            void                              *p_key_arg);
5   {
6       p->p_plfm_info = p_plfm_init_info;
7       //其他操作
8   }
9
10  int zlg72128_i2c_sync_write(zlg72128_plfm_t       *p_plfm,
11                              uint8_t                sub_addr,
12                              uint8_t               *p_buf,
13                              uint32_t               nbytes);
14  {
15      // 根据参数 sub_addr、p_buf、nbytes 及平台信息结构体中的 slv_addr 构建写函数
16      plfm_i2c_write(p_plfm->plfm_init_info->slv_addr, sub_addr, p_buf, nbytes);
17      //等待 I²C 传输完毕
18      wait();
19      return 0;
20  }
```

有关设备从机地址及寄存器地址的概念，可以参考 2.2 节的内容，以便正确实现 I²C 写

函数。

读者可能会有疑问，既然键值信息在 0x00 ~ 0x03 这 4 个寄存器中，为何不类似于 I²C 写函数，让平台字节适配 I²C 读函数，而要适配“键值上报函数”呢。主要有两个原因：

一是键值上报需要用户读取的寄存器（0x00~0x03）是固定的，不需要适配出读任意地址的“读函数”，因而不同平台可以伺机优化。而 I²C 写函数主要用于发送数码管控制命令至 ZLG72128，数据的地址和长度都会随着命令的不同而不同。

二是在不同平台中，期望读取这些键值数据的时机可能不尽相同（在平台初始化函数中介绍了 4 种可能的时机）。而通过 I²C 写函数发送命令至 ZLG72128 的时机是相对固定的（即上层用户调用相应的数码管控制功能接口时），没有过多的变化。

综合来看，采用直接上报键值的方式，可以为平台保留极大的灵活性，对调用时机和 I²C 读数据的方式（同步或异步）都没有任何限制，平台可以根据需要选择最佳方式进行适配。

平台适配的所有函数实际上都是被 ZLG72128 通用驱动调用的，调用时相关参数的赋值也是由 ZLG72128 通用驱动完成的，其会保证参数传递的正确性。

3.4 ZLG72128 的初始化

初始化函数的作用是完成 ZLG72128 模块的初始化，并获取到 ZLG72128 的操作“句柄”。在使用 ZLG72128 前，其应该首先被调用。获取 ZLG72128 的操作句柄是该函数一个非常重要的作用。在使用其它功能接口（下节详细介绍）时，均需将此处获取到的句柄作为各个功能接口的第一个参数。基于此，该函数通常被首先调用，进而将获取到的“句柄”作为其它功能接口的参数，以使用 ZLG72128 的各项功能。

初始化函数的原型为：

```
zlg72128_handle_t zlg72128_init(zlg72128_dev_t                *p_dev,
                                const zlg72128_devinfo_t    *p_devinfo);
```

1. 参数详解

由初始化函数的原型可知，其有两个参数：p_dev、p_devinfo。其中，p_dev 用于指向设备实例的指针；p_devinfo 为指向设备信息的指针。下面，分别对各个参数作详细介绍。

- p_dev

p_dev 为指向设备实例的指针。设备实例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 ZLG72128 相关的状态数据等，包括平台设备数据），p_dev 指针指向的即是一段内存空间的首地址。内存大小为 zlg72128_dev_t 类型数据占用的大小，基于此，可以先使用 zlg72128_dev_t 类型定义一个变量，即：

```
zlg72128_dev_t        dev;
```

则 dev 的地址（&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，zlg72128_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，zlg72128_dev_t 类型为 ZLG72128 设备类型，每个具体的 ZLG72128 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：

```
zlg72128_dev_t        dev;
```

若有多个 ZLG72128 对象，例如，系统中使用了两片 ZLG72128，则可以使用该类型定义多个对象，即：

```
zlg72128_dev_t      dev0;
zlg72128_dev_t      dev1;
```

当存在多个 ZLG72128 对象时，每个 ZLG72128 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 为指向设备信息的指针。设备信息描述了软件包需要使用到的一些与 ZLG72128 相关的数据信息，这些信息由用户决定并且通过初始化函数传入到软件包中，软件包内部不会修改这些数据信息。为了完成设备信息的定义，用户需要知晓设备信息类型（zlg72128_devinfo_t）的具体定义，该类型定义如下：

```
typedef struct zlg72128_devinfo {
    zlg72128_plfm_init_info_t   plfm_info;                  // 平台信息结构体
}zlg72128_devinfo_t;
```

由此可见，该信息目前仅包含了平台相关的信息，平台相关的信息由具体平台定义，不同平台可能不同，若在某一平台中，平台信息定义如下：

```
typedef struct {
    uint8_t     slv_addr;           // I²C 设备从机地址
    int         pin_rst;            // RST 引脚编号
    int         pin_key_int;        // KEY_INT 引脚编号
    int         i2c_id;             //I²C 总线编号
}zlg72128_plfm_init_info_t;
```

则根据实际情况，需要完成设备信息的定义。例如，slv_addr 为 0x30，RST 引脚编号为 PIO0_0，KEY_INT 引脚编号为 PIO0_1，I²C 总线编号为 0，则可以定义如下设备信息常量：

```
const zlg72128_devinfo_t info = {
    {
        0x30,
        PIO0_0,
        PIO0_1,
        0
    }
};
```

在不同平台中，该类型的定义可能不同，因此，不同平台下的初始化函数调用形式可能不同，通常情况下，不同平台在使用 ZLG72128 时，都会提供一套默认的配置，用户仅需在默认配置的基础上，进行简单的修改（例如，修改从机地址、引脚编号等）即可。

虽然设备信息仅包含了平台信息，但平台信息依然仅仅是设备信息的一个成员，并没有删除设备信息的定义， 这主要是出于结构性考虑，当未来增加一些与平台无关的信息时，直接添加到设备信息类型中即可。

通过上述描述可知，在初始化之前，用户需要完成时设备实例的定义以及设备信息常量的定义，初始化函数的调用范例详见程序清单 3.13。

程序清单 3.13　初始化函数使用范例

```
1   static zlg72128_dev_t __g_zlg72128_dev;
2
3   static const zlg72128_devinfo_t __g_zlg72128_devinfo = {
4       {
5           0x30,
6           PIO0_0,
7           PIO0_1,
8           0
9       }
10  }
11
12  int main()
13  {
14      zlg72128_init(&__g_zlg72128_dev,  &__g_zlg72128_devinfo);
15  }
```

2. 返回值

初始化函数的返回值为 ZLG72128 实例句柄，其类型为：zlg72128_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（详见 3.5 节中介绍的各个接口），均使用了一个 handle 参数，用以指定操作的 ZLG72128 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 ZLG72128 实例句柄。

获取 ZLG72128 实例句柄的语句为：

```
zlg72128_handle_t  handle = zlg72128_init(&__g_zlg72128_dev,  &__g_zlg72128_devinfo);
```

需要特别注意的是，若 handle 值为 NULL，则表明初始化失败，此时的 handle 不能被继续使用。只有当 handle 值不为 NULL 时，才表示获取到了一个有效的 handle。

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底层作过多的了解，在不同平台中，可以直接对外可以直接对外提供一个实例初始化函数，用以完成一个 ZLG72128 硬件设备的初始化，并获取到相应的 handle，范例程序详见程序清单 3.14。

程序清单 3.14　ZLG72128 实例初始化函数的实现范例（1）

```
1   static zlg72128_dev_t __g_zlg72128_dev;
2
3   static const zlg72128_devinfo_t __g_zlg72128_devinfo = {
4       {
5           0x30,
6           PIO0_0,
7           PIO0_1,
8           0
9       }
10  }
11
12  zlg72128_handle_t  zlg72128_inst_init (void)
```

```
{
    return zlg72128_init(&__g_dev, & __g_zlg72128_devinfo );
}
```

基于此，用户可以直接调用无参数的实例初始化函数完成 handle 的获取，即：

```
zlg72128_handle_t    handle = zlg72128_inst_init();
```

在实际应用中，若使用了两片 ZLG72128，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 ZLG72128 设备，范例程序详见程序清单 3.15。

程序清单 3.15　ZLG72128 实例初始化函数的实现范例（2）

```
static    zlg72128_dev_t      __g_dev0;                                    // 定义 ZLG72128 设备 0
static    zlg72128_dev_t      __g_dev1;                                    // 定义 ZLG72128 设备 1

static const zlg72128_devinfo_t __g_zlg72128_devinfo0 = {
    {
        0x20,          //7-bit 从机地址
        PIO0_0,
        PIO0_1,
        0
    }
}

static const zlg72128_devinfo_t __g_zlg72128_devinfo1 = {
    {
        0x30,          //7-bit 从机地址
        PIO0_2,
        PIO0_3,
        0
    }
}

zlg72128_handle_t    zlg72128_0_inst_init (void)
{
    return zlg72128_init(&__g_dev0,    &__g_zlg72128_devinfo0);
}

zlg72128_handle_t    zlg72128_1_inst_init (void)
{
    return zlg72128_init(&__g_dev1,    &__g_zlg72128_devinfo1);
}
```

基于此，用户可以分别调用相应的实例初始化函数获取与 ZLG72128 对应的 handle，即：

```
zlg72128_handle_t    handle0 = zlg72128_0_inst_init();                    // ZLG72128    0 初始化
zlg72128_handle_t    handle1 = zlg72128_1_inst_init();                    // ZLG72128    1 初始化
```

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

配者决定，一般来讲，无论如何适配，在适配完成后，都可以提供类似于实例初始化函数的接口（例如，部分平台可能提供 zlg7128_handle_get()这样的接口），以便用户快速获取到 ZLG72128 实例句柄，进而使用 ZLG72128 提供的各种功能。

3.5 功能接口

3.5.1 按键管理

ZLG72128 支持 32 个按键（4 行 8 列矩阵键盘），其中 3 行为普通键，同一时刻只能有一个普通键按下。最后一行为功能键，多个功能键可以同时按下。

为了在检测到按键事件（普通键按下或功能键状态改变）时，及时将按键事件通知到用户，进而对按键事件进行处理。ZLG72128 软件包提供了一个注册按键回调函数的接口，当按键事件发生时，将调用注册的回调函数通知用户。注册按键回调函数的接口原型为：

```
int    zlg72128_key_cb_set ( zlg72128_handle_t      handle,        // ZLG72128 实例句柄
                             zlg72128_key_cb_t      pfn_key_cb，// 按键回调函数
                             void                   *p_arg);       // 按键回调函数的参数
```

其中，handle 为 ZLG72128 的句柄；pfn_key_cb 为函数指针，其指向实际的按键处理函数，即按键事件回调函数；p_arg 为回调函数第一个参数的值。返回值为 0 时表示设置成功，否则，表示设置失败。

通过函数原型的声明可知，pfn_key_cb 指向的回调函数类型为 zlg72128_key_cb_t，该类型的具体定义如下：

```
typedef    void    (*zlg72128_key_cb_t) ( void          *p_arg,          // 用户自定义的参数
                                          uint8_t       key_val,         // 普通按键
                                          uint8_t       repeat_cnt,      // 普通按键重复计数
                                          uint8_t       funkey_val);     // 功能按键
```

由此可见，回调函数无返回值，但有 4 个参数，各个参数在系统调用回调函数时（检测到按键事件）由系统传递，主要用于传递按键事件相关的信息，以便用户根据按键信息作出处理。第一个参数 p_arg 为用户自定义的参数，其值为注册回调函数时 p_arg 参数（调用 zlg72128_key_cb_set()函数时的第三个参数）的设定值。key_val（普通键值）、repeat_cnt（普通按键重复计数）、funkey_val（功能按键）表示按键事件的相关信息，ZLG72128 可能的按键事件有以下 3 种：

1. 有普通键按下（普通键释放不作为按键事件）

当有普通键按下时，key_val 表示按下键的键值，键值有效范围为 1~24，普通键的键值已在 zlg72128.h 中定义为宏，宏名为 ZLG72128_KEY_X_Y，其中 X 表示行号（1～3），Y 表示列号（1～8），如第 2 行第 5 个键的键值为 ZLG72128_KEY_2_5。

2. 普通键一直按下（处于连击状态）

普通键按下保持时间超过 2s 后进入连击状态，处于连击状态时，每隔 200ms 左右会产生一个按键事件，并使用一个连击计数器对产生的按键事件计数，每产生一个按键事件连击计数器的值加 1，由于连击计数器的位宽为 8，其所能表示的最大值为 255，因此，当值达到 255 后不再加 1，但同样还会继续产生按键事件，直到按键释放，连击计数器清 0。处于连击状态时，key_val 表示按下键的键值，repeat_cnt 表示连击计数器的值。

在实际应用中，连击计数器的值一般不会达到 255，可以粗略计算一下达到 255 时所对应的时间：2s + 255 * 200ms = 53s，在实际应用中，一般不会出现“按键长按时间达到 53s 后执行某种动作”的需求。

3. 有功能键按下或释放（功能键按下或释放都会产生按键事件）

funkey_val 的值表示所有功能键的状态。最后一行最多 8 个键，从左至右分别为 F0~F7，与 funkey_val 的 bit0～bit7 一一对应，位值为 0 表示对应功能键按下，位值为 1 表示对应功能键未按下。当无任何功能键按下时，funkey_val 的值为 0xFF。只要 funkey_val 的值发生改变，就会产生一个按键事件，功能键不提供连击功能。可以使用 zlg72128.h 中的宏 ZLG72128_FUNKEY_CHEVK(funkey_val，funkey_num)来简单判断某一功能键是否按下。funkey_num 用于表示需要检测的功能键，值已经定义为宏，F0~F7 分别为 ZLG72128_FUNKEY_0~ZLG72128_FUNKEY_7。若对应键按下，则宏值为非 0；反之，则宏值为 0。例如，通过 funkey_val 判断 F0 是否按下可以参考如程序清单 3.16 所示。

程序清单 3.16　功能键是否按下的判断语句

```
1    if (ZLG72128_FUNKEY_CHECK(funkey_val, ZLG72128_FUNKEY_0)) {
2        //功能键 F0 按下的处理
3    } else {
4        //功能键 F0 未按下的处理
5    }
```

功能键类似 PC 机上的 Ctrl、Alt、Shift 等按键，使用普通键和功能键很容易实现组合键应用，注册按键回调函数范例程序详见程序清单 3.17。

程序清单 3.17　注册按键回调函数范例程序

```
1    #include “ametal.h”
2    #include “zlg72128.h”
3    // 自定义按键处理回调函数
4    static void __key_callback (void *p_arg, uint8_t key_val,uint8_t repeat_cnt, uint8_t funkey_val)
5    {
6        if (key_val == ZLG72128_KEY_1_1) {                                    // 第 1 行第 1 个按键按下
7            if (ZLG72128_FUNKEY_CHECK(funkey_val,  ZLG72128_FUNKEY_0)) {// 功能键 F0 按下
8                // 翻转 LED0
9            } else {                                                          // 功能键 F0 未按下
10               // 翻转 LED1
11           }
12       }
13   }
14   int main (void)
15   {
16       zlg72128_handle_t  zlg72128_handle =  zlg72128_inst_init();
17
18       // __key_callback 为注册的回调函数，回调函数的第一个参数不使用，设置为 NULL
19       zlg72128_key_cb_set(zlg72128_handle, __key_callback, NULL);
20       while (1){
21       }
22   }
```

若只按下第一行第一个键，则翻转 LED0 的状态，若在按下第一行第一个按键的同时，也按下了功能键 F0，则翻转 LED1 的状态，该示例简单的展示了按键注册函数的使用方法

以及按键的处理方法。

3.5.2 数码管显示

ZLG72128 支持 12 位共阴式数码管， ZLG72128 软件包提供了一系列数码管接口用于使用 ZLG72128 提供的数码管功能，包括控制数码管闪烁时间、控制数码管闪烁、控制数码管显示属性、设置数码管显示的字符、特殊字符显示等。相关接口原型详见表 3.4。

表 3.4 数码管显示接口

函数原型	功能简介
int zlg72128_digitron_disp_ctrl (zlg72128_handle_t handle, uint16_t ctrl_val);	显示属性（开或关）
int zlg72128_digitron_disp_char (zlg72128_handle_t handle, uint8_t pos, char ch, bool_t is_dp_disp, bool_t is_flash);	显示字符
int zlg72128_digitron_disp_str (zlg72128_handle_t handle, uint8_t start_pos, const char *p_str);	显示字符串
int zlg72128_digitron_disp_num (zlg72128_handle_t handle， uinit8_t pos， uinit8_t num， bool_t is_dp_disp， bool_t is_flash);	显示 0 ~ 9 的数字
int zlg72128_digitron_dispbuf_set (zlg72128_handle_t handle， uinit8_t start_pos， uinit8_t *p_buf， uinit8_t num);	直接设置显示段码
int zlg72128_digitron_seg_ctrl (zlg72128_handle_t handle, uinit8_t pos, char seg, bool_t is_on);	直接控制段的点亮或熄灭
int zlg72128_digitron_flash_ctrl (zlg72128_handle_t handle, uint16_t ctrl_val);	闪烁控制

续上表

函数原型	功能简介
int zlg72128_digitron_flash_time_cfg (zlg72128_handle_t handle, uint16_t on_ms, uint16_t off_ms);	闪烁持续时间
int zlg72128_digitron_shift (zlg72128_handle_t handle, uint8_t dir, bool_t is_cyclic, uint8_t num);	显示移位
int zlg72128_digitron_scan_set (zlg72128_handle_t handle, uint8_t num);	设置扫描位数
int zlg72128_digitron_disp_reset (zlg72128_handle_t handle);	复位显示
int zlg72128_digitron_disp_test (zlg72128_handle_t handle);	测试命令

虽然接口函数种类繁多，但各个接口函数的功能较为简单，下面将一一介绍各个接口函数的使用方法。

1. 显示属性(开或关)

显示属性控制是指控制哪些数码管显示，哪些数码管不显示。在默认的情况下，所有的数码管均处于打开显示状态，此时 ZLG72128 将扫描显示 12 位数码管。当实际上需要显示的位数不足 12 位时，可以使用该函数关闭某些位的显示，其函数原型为：

```
int zlg72128_digitron_disp_ctrl (
        zlg72128_handle_t       handle,         // ZLG72128 实例句柄
        uint16_t                ctrl_val);      //  控制值
```

其中，ctrl_val 为控制值，bit0~bit11 为有效位，分别对应数码管 0~11，位值为 0 时打开显示，位值为 1 时关闭显示。上电默认值为 0x0000，即所有位均正常显示。若只使用了数码管 0 ~ 7，则可以关闭数码管 8 ~ 11，范例程序详见程序清单 3.18。

程序清单 3.18　控制显示属性接口使用示例

```
1   zlg72128_digitron_disp_ctrl (
2       zlg72128_handle,                        //ZLG72128 实例句柄
3       0xF00);                                 //bit8~11 设置为 1，相应数码管关闭显示
```

2. 显示字符

在指定位置显示字符，ZLG72128 已经支持 0 ~ 9 这 10 个数字和常见的 21 种字母的自动译码显示，无需应用再自行译码，其函数原型为：

```
int   zlg72128_digitron_disp_char (
        zlg72128_handle_t   handle,             // ZLG72128 实例句柄
        uint8_t             pos,                //  字符显示位置
        char                ch,                 //  显示的字符
        bool_t              is_dp_disp,         //  是否显示小数点
        bool_t              is_flash);          //  是否闪烁
```

显示的字符必须是 ZLG72128 已经支持的可以自动完成译码的字符，包括字符'0'~'9'与 AbCdEFGHiJLopqrtUychT(区分大小写)。注意，若要显示数字 1.则 ch 参数应为字符'1'，而不是数字 1。若指定的字符不支持，则返回负值。只要成功显示，则返回 0。若需要显示一些自定义的图形，使用 zlg72128_digitron_dispbuf_set()直接设置显示的段码。比如，在数码管 0 显示字符 F，不显示小数点，不闪烁，使用方法详见程序清单 3.19。

程序清单 3.19　设置数码管显示字符接口使用示例

```
zlg72128_digitron_disp_char (
        zlg72128_handle,                    // ZLG72128 实例句柄
        0,                                  // 字符显示位置，0 号位置
        'F',                                // 显示的字符'F'
        ZLG72128_FALSE,                     // 不显示小数点
        ZLG72128_FALSE);                    // 不闪烁
```

3. 显示字符串

指定字符串显示的起始位置，开始显示一个字符串，其函数原型为：

```
int zlg72128_digitron_disp_str (
        zlg72128_handle_t   handle,         // ZLG72128 句柄
        uint8_t             start_pos,      // 字符串显示起始位置
        const char          *p_str);        // 字符串
```

字符串显示遇到字符结束标志"\0"将自动结束，或当超过有效的字符显示区域时，也会自动结束。显示的字符应确保是 ZLG72128 能够自动完成译码的，包括字符'0'~'9'与 AbCdEFGHiJLopqrtUychT（区分大小写）。如遇到有不支持的字符，对应位置将不显示任何内容。比如，从数码管 0 开始，显示字符串"0123456789"，其使用方法详见程序清单 3.20。

程序清单 3.20　显示字符串接口使用示例

```
zlg72128_digitron_disp_str (
        zlg72128_handle,                    // ZLG72128 实例句柄
        0，                                 // 从数码管 0 开始显示
        "0123456789");                      // 字符串"0123456789"
```

4. 显示 0 ~ 9 的数字

在指定位置显示 0~9 的数字，其函数原型为：

```
int   zlg72128_digitron_disp_num (
        zlg72128_handle_t   handle，        // ZLG72128 实例句柄
        uint8_t             pos，           // 数字显示位置
        uint8_t             num，           // 显示的数字
        bool_t              is_dp_disp，    // 是否显示小数点
        bool_t              is_flash);      // 是否闪烁
```

该函数仅用于显示一个 0~9 的数字，若数字大于 9，应自行根据需要分别显示各个位。注意，num 参数为数字 0~9，不是字符'0'~'9'。比如，在数码管 0 显示数字 8，不显示小数点，闪烁。接口使用方法详见程序清单 3.21。

程序清单 3.21　显示数字接口函数使用示例

```
zlg72128_digitron_disp_num (
```

```
2                                   zlg72128_handle，           // ZLG72128 实例句柄
3                                   0，                         // 数字显示位置，0 号位置
4                                   8，                         // 显示数字 8
5                                   ZLG72128_FALSE，            // 不显示小数点
6                                   ZLG72128_FALSE);            // 不闪烁
```

5. 直接设置显示段码

该函数用于设置各个数码管显示的段码，当需要显示一些不能自动译码显示的图形或者字符时，可以使用该函数灵活的显示各种各样的图形，其函数原型为：

```
int   zlg72128_digitron_dispbuf_set (
          zlg72128_handle_t   handle，                  // ZLG72128 实例句柄
          uint8_t              start_pos，              // 起始位置
          uint8_t             *p_buf，                  // 段码缓冲区
          uint8_t              num);                    // 本次设置段码的个数
```

该函数一次可以设置多个连续数码管显示的缓冲区内容，起始显示位置由 start_pos 指定，有效值为 0～11,连续显示数码管的个数由参数 num 指定，该函数将一次设置 start_pos～(start_pos+num-1)的各个数码管的显示内容。

段码为 8 位，bit0~bit7 分别对应段 a~dp。位值为 1 时，对应段点亮，位值为 0 时，对应段熄灭。如显示数字 1，则需要点亮段 b 和段 c，这就需要 bit1 和 bit2 为 1，因此段码为 00000110,即 0x06。其他显示图形可以以此类推。比如，在数码管 0~9 显示数字 0~9，可以使用该函数直接设置各个数码管显示的段码，接口使用方法详见程序清单 3.22。

程序清单 3.22　直接设置显示段码范例程序

```
1    uint8_t seg[10] = {0x3f, 0x06, 0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f};   // 0～9 的段码
2
3    zlg72128_digitron_dispbuf_set (
4                                   zlg72128_handle，           // ZLG72128 实例句柄
5                                   0,                          // 从数码管 0 开始显示
6                                   seg,                        // 要设置显示的数码管段码
7                                   10);                        // 共计设置 10 个数码管显示的段码
```

6. 直接控制段的点亮或熄灭

虽然已经提供了直接设置显示段码的函数，但为了更加灵活的显示一个图形，或控制图形的变换，ZLG72128 支持直接控制某个段的亮灭，其函数原型为：

```
int zlg72128_digitron_seg_ctrl (
          zlg72128_handle_t   handle,                   // ZLG72128 实例句柄
          uint8_t              pos,                     // 数码管的位置
          char                 seg,                     // 控制的段
          bool_t               is_on);                  // 是否点亮该段
```

pos 用于指定数码管的位置，有效值为 0～11，seg 表明要控制的段，有效值为 0~7，分别对应 a～dp。各个段已经在 zlg72128.h 文件中使用宏的形式定义好了。建议不要直接使用立即数 0 ~ 7，而应使用与 a ~ dp 对应的宏： ZLG72128_DIGITRON_SEG_A ~ ZLG72128_DIGITRON_SEG_DP。比如，在当前显示的基础上，需要在数码管 0 显示出小数点，其它内容不变，此时就可以直接使用该函数控制点亮数码管 0 的 dp 段，范例程序详见程序清单 3.23。

程序清单 3.23　直接控制单个段的亮灭范例程序

```
1    zlg72128_digitron_seg_ctrl (
2        zlg72128_handle,                    // ZLG72128 实例句柄
3        0,                                  //  控制数码管 0
4        ZLG72128_DIGITRON_SEG_DP,           //  控制 DP 段
5        ZLG72128_TRUE;                      //  点亮该段
```

7.　闪烁控制

控制数码管是否闪烁的函数原型为：

```
int zlg72128_digitron_flash_ctrl (
        zlg72128_handle_t    handle,        // zlg72128 句柄
        uint16_t             ctrl_val);     //  控制值
```

其中，ctrl_val 为控制值，bit0～bit11 为有效位，分别对应数码管 0～11，位值为 0 时不闪烁，位置为 1 时闪烁。上电默认值为 0x0000，即所有数码管均不闪烁。比如，控制所有数码管闪烁，接口使用方法详见程序清单 3.24。

程序清单 3.24　闪烁控制接口使用示例

```
1    zlg72128_digitron_flash_ctrl (
2                            zlg72128_handle,        // zlg72128 实例句柄
3                            0x0FFF);                // bit0 ~ bit11 设置为 1，所有数码管闪烁
```

8.　闪烁持续时间

当数码管闪烁时，设置其点亮和熄灭持续时间的函数原型为：

```
int zlg72128_digitron_flash_time_cfg (
        zlg72128_handle_t    handle,        // ZLG72128 实例句柄
        uint16_t             on_ms,         //  点亮的持续时间(ms)
        uint16_t             off_ms);       //  熄灭的持续时间(ms)
```

上电时，数码管 点亮和熄灭的持续时间默认值为 500ms。on_ms 和 off_ms 有效的时间值为 150、200、250、……、800、850、900，即 150ms ~ 900ms,且时间间隔为 50ms。若时间间隔不是这些值，应该选择一个最接近的值。比如，设置数码管以最快的频率闪烁，即亮灭时间均设置为最短：150ms，范例程序详见程序清单 3.25。

程序清单 3.25　闪烁持续时间接口使用示例

```
1    zlg72128_digitron_flash_time_cfg (
2                                zlg72128_handle,    // ZLG72128 实例句柄
3                                150,                //  点亮 150ms
4                                150);               //  熄灭 150ms
```

仅设置闪烁时间还不能立即看到闪烁现象，必须打开某位的闪烁开关后才能看到闪烁现象，详见 zlg72128_digitron_flash_ctrl()函数介绍。

9.　显示移位

ZLG72128 支持移位控制，可以使所有数码管根据命令进行移位。共支持 4 种移位方式：左移、循环左移、右移动和循环右移。其函数原型为：

```
int    zlg72128_digitron_shift (
```

```
    zlg72128_handle_t       handle,                 // ZLG72128 实例句柄
    uint8_t                 dir,                    // 移位方向
    bool_t                  is_cyclic,              // 是否循环移位
    uint8_t                 num);                   // 移动的位数，有效值 0~11
```

dir 指定移位方向，表示方向的宏值已经在 zlg72128.h 文件中使用宏的形式定义好了，应直接使用宏值作为 dir 参数的值，左移为 ZLG72128_DIGITRON_SHIFT_LEFT，右移为 ZLG72128_DIGITRON_SHIFT_RIGHT。

is_cyclic 为 ZLG72128_TRUE 时表明是循环移位，否则不是循环移位。如果不是循环移位，则移动后，右边空出的位（左移）或左边空出的位（右移）将不显示任何内容。若是循环移动，则空出的位将会显示被移除位的内容。参数 num 指定移动的位数，一次可以移动 0~11 位。大于 11 的值将被视为无效值。比如，要将当前数码管显示循环左移一位，则对应的范例程序详见程序清单 3.26。

程序清单 3.26　显示移位接口使用示例

```
1   zlg72128_digitron_shift (
2                           zlg72128_handle,                // ZLG72128 实例句柄
3                           ZLG72128_DIGITRON_SHIFT_LEFT,   // 左移
4                           ZLG72128_TRUE,                  // 是否循环移位
5                           1);                             // 移动 1 位
```

实际中，可能会发现移位方向与传入的命令恰恰相反，这是由于硬件设计的不同造成的。常见的，可能有以下两种硬件计方式：

1) 最右边为数码管 0，从左至右为: 11、10、9、8、7、6、5、4、3、2、1、0

2) 最左边为数码管 0，从左至右为: 0、1、2、3、4、5、6、7、8、9、10、11

这主要取决于硬件设计时，COM0 ~ COM11 引脚所对应数码管所处的物理位置。此处左移和右移的概念是以 ZLG72128 典型应用电路为参考的，其 COM0 对应的最右边的数码管，即最右边为数码管 0。那么左移和右移的概念分别为：

- 左移

数码管 0（最右侧数码管）显示切换到 1，数码管 1 显示切换到 2，.....，数码管 10 切换到 11。

- 右移

数码管 11（最左侧数码管）显示切换到 10，数码管 1 显示切换到 2，.....，数码管 10 切换到 11。

若硬件电路设计的数码管位置是相反的（如 COM0 对应的是最左边的数码管），则移位效果恰恰相反，此处只需要稍微注意即可。

10.　设置数码管扫描位数

该接口的功能是设置数码管扫描位数 num，num 的取值是 0~12。其函数原型为:

```
int zlg72128_digitron_scan_set(zlg72128_handle_t    handle,
                               uint8_t              num);
```

在使用过程中，如果用户不需要 12 位数码管显示，用户可通过参数 num 来设置数码管的扫描位数，以此来达到裁剪的作用。比如只使用 2 个数码管，则可通过该接口设置扫描位数为 2，范例程序详见程序清单 3.27

程序清单 3.27　设置数码管扫描位数接口使用示例

```
1    zlg72128_digitron_scan_set(handle,  2);
```

11.　复位显示

复位显示将数码管显示的内容清空，即所有数码管不显示任何内容。其函数原型为：

```
int zlg72128_digitron_disp_reset (zlg72128_handle_t    handle);
```

用户可调用此接口对ZLG72128复位，并熄灭所有数码管，范例程序详见程序清单3.28。

程序清单 3.28　复位显示范例程序

```
1    zlg72128_digitron_disp_reset (zlg72128_handle);
```

12.　测试命令

测试命令主要用于测试数码管的硬件电路是否连接正常，其函数原型为：

```
int    zlg72128_digitron_disp_test (zlg72128_handle_t    handle);
```

执行测试命令后，数码管段显示“8.8.8.8.8.8.8.8.8.8.8.”（即所有段点亮），并以0.5s的速率闪烁。用户在使用ZLG72128模块之前可以先调用此接口对模块的所有数码管进行测试，以检验数码管电路是否正常，I^2C数据通信是否正常。范例程序详见程序清单3.29。

程序清单 3.29　测试函数范例程序

```
1    zlg72128_digitron_disp_test(handle);                    // 测试命令
```

调用测试接口后，数码管将一直闪烁，可以通过调用“复位显示”函数复位数码管显示，结束测试。

3.5.3 解初始化

若用户不再使用ZLG72128，应调用解初始化函数，以释放与ZLG72128相关的资源。其函数原型为：

```
int zlg72128_deinit(zlg72128_handle_t handle);
```

其中，handle为初始化时返回的句柄。返回值为0时表示解初始化成功，非0时表示解初始化失败。解初始化之后，handle不再有效，为避免出错，用户可以将其设置为NULL。范例程序详见程序清单3.30。

程序清单 3.30　解初始化函数范例程序

```
1    zlg72128_handle_t    handle = //  获取 handle
2
3    // ....  使用各个功能
4
5    zlg72128_deinit(handle);                    //  使用结束，解初始化
6    handle = NULL;                              //  设置为 NULL，避免再使用
```

在解初始化之后，若还需继续使用ZLG72128的各个功能接口，则应重新初始化获取到一个有效的handle。

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_zlg72128_xxx_entry(zlg72128_handle_t handle);
```

其中，函数名中的“xxx”与具体应用的功能相关，例如数码管测试应用，其函数名可能为：demo_zlg72128_digitron_test_entry()。所有入口函数均有一个 zlg72128_handle_t 类型的 handle 参数，该参数通过 ZLG72128 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 ZLG72128 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 数码管显示测试

为了判断数码管是否工作正常，可以编写一个简单的数码管显示测试函数：系统启动时，数码管进入测试状态，数码管所有段全部点亮，即显示“8.8.8.8.8.8.8.8.8.8.”，并以 0.5s 的速率闪烁，历时 10 秒后，清空显示内容。

由于 ZLG72128 自带数码管测试命令，所以该项功能很容易实现，直接调用测试命令接口，延时 3s 后，复位数码管显示即可，范例程序详见程序清单 3.31。

程序清单 3.31　数码管显示测试范例程序

```
1   void demo_zlg72128_digitron_test_entry (zlg72128_handle_t handle)
2   {
3       zlg72128_digitron_disp_test(handle);                    // 测试命令
4       zlg72128_plfm_delay_ms(10000);                          // 延时 10 秒，使测试命令保持 10 秒
5       zlg72128_digitron_disp_reset(handle);                   // 复位数码管显示
6       while(1) {
7           zlg72128_plfm_delay_ms(10);
8       }
9   }
```

实际中，在不同平台下的延时函数实现可能不同，因此，运行该范例程序时，需要根据实际情况，完成对 zlg72128_plfm_delay_ms()函数的定义，以便应用程序正常工作。

3.6.2 普通键测试

为了测试各个普通键是否工作正常，实现一个简单的按键测试：按下任何一个普通键，数码管显示当前键的键值（1 ~ 24）。

对于普通键，当键按下时，可以通过按键回调函数直接获取到键值（1 ~ 24），获取到键值后，使用数码管显示接口将该值显示出来即可，范例程序详见程序清单 3.32。

程序清单 3.32　普通按键测试范例程序

```
1   #include "zlg72128.h"
2
3   volatile int __g_key_code;
4   volatile int __g_key_flag = 0;
5
6   // 普通按键实际处理函数
7   static void normal_key_process (
8       zlg72128_handle_t       handle,         // ZLG72128 操作句柄
9       uint8_t                 key_val)        // 普通键键值
```

```
{
    // 无普通键按下，清空显示
    if (key_val == 0) {
        zlg72128_digitron_disp_char(handle, 0, ' ', ZLG72128_FALSE, ZLG72128_FALSE);
        zlg72128_digitron_disp_char(handle, 1, ' ', ZLG72128_FALSE, ZLG72128_FALSE);
    }

    // 数码管 1 显示十位，十位为 0 时不显示（显示空格），不显示小数点，不闪烁
    if (key_val / 10 != 0) {
        zlg72128_digitron_disp_num(handle, 1,key_val / 10, ZLG72128_FALSE, ZLG72128_FALSE);
    } else {
        zlg72128_digitron_disp_char(handle, 1, ' ', ZLG72128_FALSE, ZLG72128_FALSE);
    }

    // 在数码管 0 显示个位，不显示小数点，不闪烁
    zlg72128_digitron_disp_num(handle, 0, key_val % 10, ZLG72128_FALSE, ZLG72128_FALSE);
}

static void __zlg72128_key_cb (void *p_arg, uint8_t key_val, uint8_t repeat_cnt, uint8_t funkey_val)
{
    if (!__g_key_flag) {                                    // 无按键事件待处理，填充新的键值
        __g_key_code = key_val
        __g_key_flag = 1;
    }
}

void demo_zlg72128_normal_key_test_entry (zlg72128_handle_t handle)
{
    zlg72128_key_cb_set(handle, __zlg72128_key_cb, NULL);   // 注册按键回调函数

    while (1) {
        if (__g_key_flag) {
            normal_key_process(handle, __g_key_code);
            __g_key_flag = 0;
        }
        zlg72128_plfm_delay_ms(1);
    }
}
```

该应用程序的入口函数为：demo_zlg72128_normal_key_test_entry()，在程序清单 3.32 中，按键处理代码并没有直接放在按键回调函数中。在按键回调函数中仅设置了一个标志，按键的实际处理在主循环中完成。在实际应用中，均不建议在中断中作过多的处理，当然，程序清单 3.32 中的处理方法是最基本的设置标志方法，存在诸多缺点（比如，按键过快时，可能丢键），更多有效的处理方法详见 8.1.2 节中介绍的通用解决办法。

3.6.3 组合键应用

ZLG72128 有 8 个功能键，功能键如同电脑键盘的 Ctrl、Shift 和 Alt 键，与其它普通按键组合可以实现丰富的功能，如 Ctrl+S（保存），Ctrl+A（全选），Ctrl+Z（撤销）等。这里，以功能键 F0 为例，展示其如何与普通按键组合使用。为方便观察，定义下列操作及对应的现象：

- F0 + K1：数码管显示循环左移；
- F0 + K2：数码管显示循环右移；
- F0 + K3：所有闪烁显示打开/关闭。

各种组合键对应的功能都有相应的 API，相应的范例程序详见程序清单 3.33。

程序清单 3.33　组合键应用范例程序

```
#include "zlg72128.h"

volatile int __g_key_code;
volatile int __g_funkey_val  = 0;
volatile int __g_key_flag    = 0;

// 组合键应用处理函数
static void combination_key_process (zlg72128_handle_t handle, uint8_t   key_val, uint8_t   funckey_val)
{
    static uint16_t flash = 0x0000;                         // 初始时，所有位均不闪烁
    if ((funckey_val & (1 << 0)) == 0) {                    // F0 按下
        switch (key_val) {
        case ZLG72128_KEY_1_1:                              // F0 + K1：循环左移
            zlg72128_digitron_shift(handle,
                                    ZLG72128_DIGITRON_SHIFT_LEFT,
                                    ZLG72128_TRUE,
                                    1);
            break;
        case ZLG72128_KEY_1_2:                              // F0 + K2：循环右移
            zlg72128_digitron_shift(handle,
                                    ZLG72128_DIGITRON_SHIFT_RIGHT,
                                    ZLG72128_TRUE,
                                    1);
            break;
        case ZLG72128_KEY_1_3:                              // F0 + K3：打开/关闭闪烁
            flash = ~flash;                                 // 取反，所有为闪烁状态改变
            zlg72128_digitron_flash_ctrl(handle, flash);
            break;
        default:
            break;
        }
    }
```

```
33  }
34
35  static void __zlg72128_key_cb (void *p_arg, uint8_t key_val, uint8_t repeat_cnt, uint8_t funkey_val)
36  {
37      if (!__g_key_flag) {                                        // 无按键事件待处理，填充新的键值
38          __g_key_code    = key_val
39          __g_funkey_val  = funkey_val;
40          __g_key_flag = 1;
41      }
42  }
43
44  void demo_zlg72128_combination_key_test_entry (zlg72128_handle_t handle)
45  {
46      zlg72128_key_cb_set(handle, __zlg72128_key_cb, NULL);   // 注册按键回调函数
47      //设置数码管 0 显示 0
48      zlg72128_digitron_disp_num(handle, 0, 0, ZLG72128_FALSE, ZLG72128_FALSE) ;
49      //设置数码管 1 显示 1
50      zlg72128_digitron_disp_num(handle, 1, 1, ZLG72128_FALSE, ZLG72128_FALSE) ;
51      while (1) {
52          if (__g_key_flag) {
53              // 功能键和普通键均有按下
54              if ((__g_funkey_val != 0xFF) && (__g_key_code != 0)) {
55                  combination_key _process(handle, __g_key_code, __g_funkey_val);
56              }
57              __g_key_flag = 0;
58          }
59          zlg72128_plfm_delay_ms(1);
60      }
61  }
```

该应用程序的入口函数为 demo_zlg72128_combination_key_test_entry()，与程序清单 3.32 类似，按键的处理同样没有直接在按键回调函数中处理。

3.6.4 计时应用

基于 ZLG72128 提供的数码管显示功能，可以实现一个简易的计时程序，数码管显示的数值从 0 开始递增，递增速率为每秒加 1，在递增过程中，按任意键清零，范例程序详见程序清单 3.34。

程序清单 3.34　计时应用范例程序

```
1   #include "zlg72128.h"
2
3   static volatile int __g_cur_count = 0;                          // 当前计时值
4
5   /*
6    * 显示一个 int 类型的数据，若 int 类型为 32 位，其范围为： -2147483648 ~ 2147483647
```

```
 * 显示时，和计算器之类的显示效果一致，采用右对齐方式，即数码管 0 始终显示个位数据。
 */
static void __digitron_disp_num (zlg72128_handle_t handle, int num)
{
    uint8_t    neg = 0;
    uint8_t    digi_num = 0;              // 当前显示位，初始为个位（假定 COM0 对应最右边数码管）

    if (num < 0) {
        neg = 1;                          // 负数，需要显示负号
        num = -1 * num;
    }

    /* 这里使用 do... while() 结构，主要是由于但 num 为 0 时，也应该显示 0 */
    do {
        zlg72128_digitron_disp_num(handle,
                                   digi_num++,
                                   num % 10,
                                   ZLG72128_FALSE,
                                   ZLG72128_FALSE);
        num /= 10;
    } while (num != 0);

    /* 显示负号 */
    if (neg) {
        zlg72128_digitron_disp_char(handle, digi_num++, '-', ZLG72128_FALSE, ZLG72128_FALSE);
    }
}

static void __zlg72128_key_cb (void *p_arg, uint8_t key_val, uint8_t repeat_cnt, uint8_t funkey_val)
{
    __g_cur_count = 0;
}

void demo_zlg72128_normal_key_test_entry (zlg72128_handle_t handle)
{
    zlg72128_key_cb_set(handle, __zlg72128_key_cb, NULL);    // 注册按键回调函数

    while (1) {
        __digitron_disp_num (handle, __g_cur_count);      // 显示数值
        __g_cur_count++;
        zlg72128_plfm_delay_ms(1000);                     // 延时 1s，延时函数与具体平台相关
    }
}
```

程序中，由于按键事件发生时，仅需将显示值设置为 0，处理时间非常短（与设置一

个标志的效率相当），因此，按键处理代码直接存放在了按键回调函数中。

由于按键回调函数通常在中断环境中运行，其可以打断主程序的运行，而按键回调函数中需要对计数值执行置 0 操作，主程序中需要对计数值执行加 1 操作，同时对计数值修改可能产生冲突，造成错误的结果。例如，__g_cur_count++;语句在执行过程中，可以需要 3 条 CPU 指令才能完成：

1) 将__g_cur_count 计数值读取到寄存器中；

2) 将寄存器的值加 1；

3) 再将寄存器的值写入到__g_cur_count 计数值中。

若主程序正在执行步骤（3），假定此时寄存器的值为 55，正准备将该值写入到__g_cur_count 中，而此时恰好有按键事件产生，将转而执行按键回调函数，则__g_cur_count 的值被修改为 0，按键回调结束后，继续执行主程序中的步骤（3），则会将寄存器的值（假定为 55）重新写入到__g_cur_count 中，则从现象上看，中断回调函数的处理被忽略了，等同于按键丢失，是一种错误的结果。

为了避免出现错误的结果，建议将主程序中的“加 1 操作”加上中断保护，避免修改过程中被回调函数打断，即：

```
int_disable();
__g_cur_count++;
int_enable();
```

上述代码仅作示意，实际中断的关闭和打开函数与具体平台相关。

3.6.5 应用程序入口函数声明

前面通过 4 个典型的应用范例展示了各个接口的使用方法，为了便于在实际应用中调用这些 Demo 对应的入口函数，将所有入口函数声明到一个名为 demo_zlg72128_entries.h 的头文件中，详见程序清单 3.35。

程序清单 3.35 应用范例入口函数声明（demo_zlg72128_entries.h）

```
1   #ifndef __DEMO_ZLG72128_ENTRIES_H
2   #define __DEMO_ZLG72128_ENTRIES_H
3
4   #include "zlg72128.h"
5
6   void demo_zlg72128_digitron_test_entry (zlg72128_handle_t handle) ;          // 数码管显示测试
7   void demo_zlg72128_normal_key_test_entry (zlg72128_handle_t handle) ;        // 普通键测试
8   void demo_zlg72128_combination_key_test_entry (zlg72128_handle_t handle) ; // 组合键测试
9   void demo_zlg72128_count_up_test_entry (zlg72128_handle_t handle);           // 计时应用
10
11  #endif
```

3.7 多任务环境下的使用

ZLG72128 软件包提供的接口仅仅是对 ZLG72128 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 ZLG72128 的情况。因此，在多任务环境中，若多个任务同时使用到了 ZLG72128，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 3.36。

程序清单 3.36　在多任务环境中使用 ZLG72128

```
1   mutex_t     mutex_zlg72128;   // 定义互斥量
2
3   void task1_entry (void)
4   {
5       while(1) {
6           mutex_take(&mutex_zlg72128);                                    // 获取信号量
7           zlg72128_digitron_disp_num(zlg72128_handle, 0, 8,    0, 0);     // 显示一个数字
8           mutex_give(&mutex_zlg72128);                                    // 释放信号量
9           // 其它操作
7       }
8   }
9
10  void task2_entry (void)
11  {
12      while(1) {
13          mutex_take(&mutex_zlg72128);                                    // 获取信号量
14          zlg72128_digitron_flash_ctrl(zlg72128_handle, 0x0FFF);          // 数码管闪烁
15          mutex_give(&mutex_zlg72128);                                    // 释放信号量
16          // 其它操作
17      }
18  }
19
20  void main ()
21  {
22      mutex_init(&mutex_zlg72128);                                        // 信号量初始化
23      // 创建并启动两个任务
24  }
```

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 ZLG72128，一般来讲，都不建议在中断中使用 ZLG72128 相关的接口，控制数码管的显示（I^2C 数据传输较慢，这会极大的影响系统的实时性）。

3.8　通用驱动软件包测试

为了加深读者对驱动包的理解，随软件包一起发布了软件包的单元测试代码。

3.8.1 了解测试对象

单元测试的第一步是了解测试对象。经过前面的描述，相信读者对软件包的各个接口已经比较熟悉，所以这里不再详细描述。在了解了软件包提供的各个接口后，读者还需要分析各个接口之间的依赖关系，以方便进行测试。

通过分析软件包的实现代码可以得出以下信息：

- 所有接口都需要使用 zlg72128_init()接口返回的设备句柄作为参数；
- 所有接口之间并不存在其他依赖关系；

- 查询键值需要使用 zlg72128_plfm_init()函数传入的回调函数；
- 键值寄存器的值由适配层自行读取并上报驱动，驱动解析并调用用户回调函数；
- 所有设置接口通过同步写实现，通过调用 zlg72128_i2c_sync_write 函数实现同步通信；
- zlg72128_deinit()函数直接调用平台的解初始化函数完成解初始化。

根据以上信息，在测试时可以在测试夹具中调用 zlg72128_init()函数以得到设备句柄，供其他测试用例使用，zlg72128_deinit()函数需要与 zlg72128_init()函数成对被调用，该函数的调用也放入测试夹具中；各个接口可分别进行测试，不需要考虑相互之间的依赖。

3.8.2 设计测试用例

在对测试对象进行了深入的了解后，就可以开始设计测试用例。由于各个接口之间不存在依赖关系，所以可以单独针对每个接口设计测试用例。根据各个接口的共性，读者在设计测试用例时可以从以下几个方面考虑：

- 使用指针作为参数时，需要考虑正常指针和空指针两种情况；
- 当某个参数允许值为一个范围时，需要考虑边界处的正常值和异常值；
- 当某个参数允许值为 0 和 1 时，除了要考虑 0 和 1 外，还需要考虑大于 1 的情况；
- 当传输函数返回错误时，对应的接口应当返回错误。

针对各个接口共性的部分，各个接口对应的测试用例基本一致。接下来还需要根据各个接口的特点针对性的设计测试用例。

- 查询键值：当没有键被按下时，键值回调函数不应该被调用；
- zlg72128_digitron_flash_time_cfg：由于时间不为 50 整数倍时会设置为近似值，需要考虑近似值为较低值和较高值两种情况；
- zlg72128_digitron_disp_char：字符的允许值为一个集合，需要针对集合内的每一个字符进行测试，同时需要考虑不在集合中的字符的情况；
- zlg72128_digitron_disp_str：由于字符后的小数点不单独占一位，需要考虑小数点在首位，小数点在正常字符后，连续的小数点三种情况；同时还需要考虑起始位置和长度都正常，两者之和超出范围的情况；
- zlg72128_digitron_dispbuf_set：需要考虑起始位置和长度都正常，两者之和超出范围的情况。

为各个接口设计的测试用例详见表 3.5。

表 3.5　接口测试用例

编号	接口	测试点名称	预期结果
1.1	zlg72128_init	正常初始化	返回 p_dev 地址
1.2		参数错误： a. 设备指针为空 b. 传输函数为空	返回 NULL
2.1	zlg72128_key_cb_set	正常设置	返回
2.2		参数错误： a. 设备句柄为空 b. 回调函数为空	返回-2

续上表

编号	接口	测试点名称	预期结果
3.1	键值查询	有键被按下	回调函数中得到正常的键值
3.2		没有键被按下	回调函数不被调用
3.3		设备句柄为空	不调用传输函数
4.1	zlg72128_digitron_flash_time_cfg	正常值 150,500,900	正确设置
4.2		近似值 174,175	设置为 150,200
4.3		异常值 100,149,901,950	设置为 150,150,900,900
4.4		设备句柄为空	返回-2
4.5		传输函数返回错误	返回-1
5.1	zlg72128_digitron_flash_ctrl	正常发送数据	正确发送数据
5.2		设备句柄为空	返回-2
5.3		传输函数返回错误	返回-1
6.1	zlg72128_digitron_disp_ctrl	正常发送数据	正确发送数据
6.2		设备句柄为空	返回-2
6.3		传输函数返回错误	返回-1
7.1	zlg72128_digitron_disp_char	位置 0,5,11	正常发送数据
7.2		分别发送各个字符	正常发送数据
7.3		is_dp_disp：0,1,2	正常发送数据
7.4		is_flash：0,1,2	正常发送数据
7.5		位置 12	返回-2
7.6		不在列表中的字符	返回-2
7.7		设备句柄为空	返回-2
7.8		传输函数返回错误	返回-1
8.1	zlg72128_digitron_disp_str	a. 以多个小数点开始 b. 中间有非连续小数点	a. 开始的每个小数点单独占一位 b. 中间的小数点不占位数
8.2		a. 以非小数点开始 b. 中间有连续小数点	连续的小数点单独占一位
8.3		起始位置：0,5,11	正常发送数据
8.4		起始位置：12	返回-2
8.5		起始位置+长度大于 12	只传输部分字符
8.6		0 字符字符串	直接返回 0
8.7		有不在列表中的字符	返回-2，从出错字符开始不再传输
8.8		字符串指针为空	返回-2
8.9		设备句柄为空	返回-2
8.10		传输函数返回错误	返回-1，之后的字符不再传输

续上表

编号	接口	测试点名称	预期结果
9.1	zlg72128_digitron_disp_num	位置 0,5,11	正常发送数据
9.2		数字 0,5,9	正常发送数据
9.3		is_dp_disp：0,1,2	正常发送数据
9.4		is_flash：0,1,2	正常发送数据
9.5		位置 12	返回-2
9.6		数字 10	返回-2
9.7		设备句柄为空	返回-2
9.8		传输函数返回错误	返回-1
10.1	zlg72128_digitron_dispbuf_set	起始位置：0,5,11	正常发送数据
10.2		起始位置：12	返回-2
10.3		起始位置+长度大于 12	只传输部分字节
10.4		设备句柄为空	返回-2
10.5		p_buf 为空	返回-2
10.6		传输函数返回错误	返回-1
11.1	zlg72128_digitron_seg_ctrl	pos：0,5,11	正常发送数据
11.2		seg：0, 4, 7	正常发送数据
11.3		is_on：0, 1, 2	正常发送数据
11.4		pos：12	返回-2
11.5		seg：8	返回-2
11.6		设备句柄为空	返回-2
11.7		传输函数返回错误	返回-1
12.1	zlg72128_digitron_shift	dir:0, 1, 2	正常发送数据
12.2		is_cyclic:0, 1, 2	正常发送数据
12.3		num:0, 5, 11	正常发送数据
12.4		num:12	返回-2
12.5		设备句柄为空	返回-2
12.6		传输函数返回错误	返回-1
13.1	zlg72128_digitron_disp_reset	正常传输	返回 0
13.2		设备句柄为空	返回-2
13.3		传输函数返回错误	返回-1
14.1	zlg72128_digitron_disp_test	正常传输	返回 0
14.2		设备句柄为空	返回-2
14.3		传输函数返回错误	返回-1
15.1	zlg72128_deinit	正常解初始化	返回 0
15.2		设备句柄为空	返回-2

zlg72128.h 文件中还实现了一个宏 ZLG72128_FUNKEY_CHECK 用于帮助开发者对各个功能值的状态进行检测，所以需要针对这个宏设计测试用例进行测试。为这个宏设计的测试用例详见表 3.6。

表 3.6　宏定义测试用例

编号	宏	测试点名称	预期结果
16.1	ZLG72128_FUNKEY_CHECK	分别检查第 0~7 位	返回正确状态

3.8.3 编写测试代码

当完成了测试用例的设计后，接下来就需要编写测试代码。在编写测试用例前，读者需要先完善平台相关文件 zlg72128_platform.h 和 zlg72128_platform.c。

在文件 zlg72128_platform.h 中，需要定义两个类型 zlg72128_plfm_init_info_t 和 zlg72128_plfm_t 用于表示平台相关数据，在测试时可以直接使用 int 类型。详见程序清单 3.37。

程序清单 3.37　与信号量相关的宏

```
1    typedef int zlg72128_plfm_init_info_t;
2    typedef int zlg72128_plfm_t;
```

由于 I²C 通信的函数需要在平台适配层实现，所以在测试时需要实现对应的测试桩。为了方便对通信过程的控制，需要实现对应的仿制对象，并在测试桩中调用仿制对象以实现通信。在调用 zlg72128_init()进行初始化时需要提供用于响应按键事件的回调函数，该函数也需要在测试桩和仿制对象中进行实现。仿制对象对象和测试桩的代码详见程序清单 3.38。

程序清单 3.38　仿制对象和测试桩

```
1    class I2CMock
2    {
3        public:
4            MOCK_METHOD4(zlg72128_plfm_init,
5                int(zlg72128_plfm_t* p_plfm,
6                    const zlg72128_plfm_init_info_t *p_plfm_init_info,
7                    void(*pfn_request_keyval_check)(void *p_arg, uint8_t reg0_3[4]),
8                    void *p_key_arg));
9        MOCK_METHOD4(zlg72128_plfm_i2c_write,
10               int(zlg72128_plfm_t* p_plfm,
11                   uint8_t sub_addr,
12                   uint8_t *p_buf,
13                   uint8_t nbytes));
14       MOCK_METHOD1(zlg72128_plfm_deinit, int(zlg72128_plfm_t* p_plfm));
15       MOCK_METHOD4(zlg72128_key_cb,
16               void(void *p_arg,
17                   uint8_t    key_val,
18                   uint8_t    repeat_cnt,
19                   uint8_t    funkey_val));
20   };
21   I2CMock *g_mock = NULL;
22
23   extern "C" {
```

```
24          int zlg72128_plfm_init(zlg72128_plfm_t *p_plfm,
25                  const zlg72128_plfm_init_info_t    *p_plfm_init_info,
26                  void(*pfn_request_keyval_check) (void *p_arg, uint8_t reg0_3[4]),
27                  void                                        *p_key_arg)
28          {
29                  return g_mock->zlg72128_plfm_init(p_plfm,
30                          p_plfm_init_info, pfn_request_keyval_check, p_key_arg);
31          }
23          int zlg72128_plfm_i2c_write(zlg72128_plfm_t *p_plfm,
333                 uint8_t                 sub_addr,
34                  uint8_t                *p_buf,
35                  uint8_t                nbytes)
36          {
37                  return g_mock->zlg72128_plfm_i2c_write(p_plfm,
38                          sub_addr, p_buf, nbytes);
39          }
40          int zlg72128_plfm_deinit(zlg72128_plfm_t *p_plfm)
41          {
42                  return g_mock->zlg72128_plfm_deinit(p_plfm);
43          }
44          void zlg72128_key_cb(void *p_arg,
45                  uint8_t    key_val,
46                  uint8_t    repeat_cnt,
47                  uint8_t    funkey_val)
48          {
49                  g_mock->zlg72128_key_cb(p_arg, key_val, repeat_cnt, funkey_val);
50          }
51  }
```

由于所有接口都需要使用 zlg72128_init()接口返回的设备句柄作为参数，所以需要在测试夹具中调用 zlg72128_init 函数生成句柄。同时仿制对象的生成与销毁也需要在测试夹具中完成。测试夹具的代码详见程序清单 3.39。

程序清单 3.39　测试夹具

```
1   class CZLG72128Test : public testing::Test
2   {
3   public:
4           static void SetUpTestCase()
5           {
6                   g_mock = new I2CMock();
7           }
8
9           static void TearDownTestCase()
10          {
11                  delete g_mock;
```

```
12          }
13
14          virtual void SetUp()
15          {
16                  m_dev_info.plfm_info = 0x22;
17
18                  EXPECT_CALL(*g_mock, zlg72128_plfm_init(NotNull(),
19                          Pointee(0x22), NotNull(), NotNull()))
20                          .Times(1)
21                          .WillOnce(DoAll(SetArgPointee<0>(0x11),
22                          SaveArg<2>(&zlg72128_keyval_report),
23                          SaveArg<3>(&p_key_arg),
24                          Return(0)));
25                  ASSERT_EQ(&m_dev, zlg72128_init(&m_dev, &m_dev_info));
26                  ASSERT_EQ(0, zlg72128_key_cb_set(&m_dev, zlg72128_key_cb, (void*)0x33));
27          }
28
29          virtual void TearDown()
30          {
31                  EXPECT_CALL(*g_mock, zlg72128_plfm_deinit(Pointee(0x11)))
32                          .Times(1).WillOnce(Return(0));
33                  zlg72128_deinit(&m_dev);
34                  testing::Mock::VerifyAndClearExpectations(g_mock);
35          }
36
37  protected:
38          zlg72128_dev_t m_dev;
39          zlg72128_devinfo_t m_dev_info;
40          void(*zlg72128_keyval_report) (void *p_arg, uint8_t reg0_3[4]);
41          void *p_key_arg;
42  };
```

在完成了测试桩和测试夹具后，接下来就可以为每一个测试用例编写对应的测试代码。读者可通过阅读软件包配套的测试代码以了解测试代码的编写。

第4章 在 AMetal 中使用 ZLG72128

✍本章导读

AMetal 平台已经完成了对 ZLG72128 的适配，可以在初始化之后，使用 ZLG72128 通用驱动软件包中提供的各个接口操作 ZLG72128，完成数码管显示和按键管理，无需关心任何底层细节，使用户可以更加快捷的将 ZLG72128 应用到实际项目中。除此之外，若选用 AMetal 平台，更加推荐的做法是直接使用 AMetal 提供的通用数码管接口和键盘管理接口完成数码管的显示以及按键的管理。这将使应用程序与数码管驱动方式无关，进而解除应用程序与 ZLG72128 之间的耦合，实现应用程序的“跨平台复用”。

4.1 使用 ZLG72128 通用软件包接口

在第三章中提到，AMetal 已经完成了对 ZLG72128 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 ZLG72128 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 ZLG72128 对象，当使用 AMetal 时，可以直接通过 ZLG72128 的实例初始化函数获得相应的 handle，其函数原型为：

```
zlg72128_handle_t    am_zlg72128_inst_init(void);
```

调用形式如下：

```
zlg72128_handle_t    handle = am_zlg72128_inst_init();
```

此处获取的 handle 即可用于 ZLG72128 通用软件包提供的各个功能接口（第 3.5 节进行了详细介绍）。

4.1.2 配置

通常情况下，实例初始化函数在模板工程中的 ZLG72128 配置文件中实现，配置文件的默认名称为： am_hwconf_zlg72128.c，以使用 LPC824 驱动 ZLG72128 为例，文件的具体内容详见程序清单 4.1。

程序清单 4.1 ZLG72128 配置文件内容示意

```
1    #include "ametal.h"
2    #include "am_lpc82x_inst_init.h"
3    #include "lpc82x_pin.h"
4    #include “zlg72128.h”
5
6    static am_zlg72128_dev_t        __g_zlg72128_dev;            // 定义一个 ZLG72128 实例
7
8    static const am_zlg72128_devinfo_t __g_zlg72128_devinfo = {
9        {
10                0x30,          // 从机地址
11                PIO0_6,        // 复位引脚
```

```
12              AM_TRUE,   // 通常情况下，均会使用中断引脚
13              PIO0_1,        // ZLG72128 的 KEY_INT 引脚与 LPC824 的 PIO0_1 连接
14              5              // 使用中断引脚时，该值无意义。若不使用中断引脚，则查询间隔为 5ms
15              am_lpc82x_i2c2_inst_init          // I²C 句柄获取函数
16              am_lpc82x_i2c2_inst_deinit        // I²C 解初始化函数
17  };
18  zlg72128_handle_t   am_zlg72128_inst_init (void)
20  {
21      return zlg72128_init(&__g_zlg72128_dev, &__g_zlg72128_devinfo);
22  }
```

注：若在用户所使用的工程中不存在该文件，则可以将程序清单 4.1 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_zlg72128.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。用户可能需要对其中的 6 个地方进行修改，视为 6 个配置项，各配置项的功能简介详见表 4.1，这些配置项都可以根据实际情况修改。

表 4.1　ZLG72128 用作通用功能时的配置项

序号	对应程序清单 4.1 的行号	配置项	默认值
1	10	从机地址	0x30
2	11	复位引脚	PIO0_6
3	12	获取按键事件的方式	AM_TRUE
4	13	中断引脚	PIO0_1
5	14	查询时间间隔	5
6	15	获取 ZLG72128 通信的 I²C 函数句柄	am_lpc82x_i2c2_inst_init
7	16	I²C 解初始化函数	am_lpc82x_i2c2_inst_deinit

下面，分别对 7 个配置项作详细介绍。

1. 从机地址

该配置项对应程序清单 4.1 的第 9 行，默认值为：0x30。该处设置的值表示 ZLG72128 的 7 位 I²C 从机地址，其值与 ZLG72128 的 A4 引脚电平相关：若为高电平，则地址为：0x30；若为低电平，则地址为：0x20。

2. 复位引脚

该配置项对应程序清单 4.1 的第 11 行，默认值为：PIO0_6。该处设置的值表示 ZLG72128 的 RST 引脚与主控 MCU（如 LPC824）连接的引脚号，若将 ZLG72128 复位引脚与主控 MCU 相接，则主控 MCU 可以通过该引脚控制 ZLG72128 的复位。这里即表示 LPC824 的 PIO0_6 与 ZLG72128 的 RST 引脚相连。一般情况下，为了节省管脚，复位引脚固定连接至一个 RC 复位电路，无需由 MCU 控制，此时，可将该处的值配置为-1。

3. 获取按键事件的方式

该配置项对应程序清单 4.1 的第 12 行，默认值为：AM_TRUE。虽然 ZLG72128 提供了 KEY_INT 引脚，用以按键事件产生时主动通知主控 MCU，但部分特殊情况下，为了节省 I/O，可能不使用该引脚。该配置项即用于配置是否使用中断引脚。若值为真，则表示使用中断引脚（中断引脚由第 4 个配置项指定）；否则，不使用中断引脚，采用查询模式

（查询时间间隔由第 5 个配置项指定）查询是否有按键事件产生。

4. 中断引脚

该配置项对应程序清单 4.1 的第 13 行，默认值为：PIO0_1。当使用中断模式（第 3 个配置项为真）时，该配置项有效，用于指定 ZLG72128 的 KEY_INT 引脚与主控 MCU 连接的引脚号。这里即表示 LPC824 的 PIO0_1 与 ZLG72128 的 KEY_INT 引脚相连。

5. 查询时间间隔

该配置项对应程序清单 4.1 的第 14 行，默认值为：5（单位：ms）。当使用查询模式（第 3 个配置项为 AM_FALSE）时，该配置项有效，用于指定查询的时间间隔，默认为 5ms，用户也可根据需要修改为其它值。

6. 与 ZLG72128 通信的 I²C 总线

该配置项对应程序清单 4.1 的第 15 行，默认值为：am_lpc82x_i2c**2**_inst_init。该行代码主要是为了获取 I²C 通信函数的句柄，以便驱动底层使用该句柄实现 I²C 数据的收发。LPC824 默认有 4 路 I²C： I²C0、I²C1、I²C2 和 I²C3。该配置项即用于配置使用哪条 I²C 总线与 ZLG72128 通信，默认配置中，使用了 I²C2，若期望使用 I²C1 与 ZLG72128 通信，则该行配置可以修改为：am_lpc82x_i2c**1**_inst_init。

7. I²C 解初始化函数

该配置项对应程序清单 4.1 的第 16 行，默认值为：am_lpc82x_i2c2_inst_deinit。该行代码提供了释放 I²C 句柄的方法，在上一个配置项中提供了获取了 I²C 句柄的方法，当 ZLG72128 正常使用时，会通过获取句柄方法获得 I²C 句柄，进而占用了 I²C 资源。当不使用 ZLG72128 时，驱动可以通过该配置项提供的方法释放之，以解除对相关资源的占用。该配置中释放的 I²C 总线必须与上一个配置项相对应，比如，默认对应的都是 I²C2。若上一个配置项期望使用 I²C1 与 ZLG72128 通信，则该行配置可以修改为：am_lpc82x_i2c1_inst_deinit。

由此可见，实例初始化函数本质上调用了 zlg72128_init()，并将用户配置信息（如复位引脚、中断引脚等信息）传入到了该函数中，其具体实现用户无需关心，简单的说，其内部实现了 I²C 数据传输，完成了对通用驱动库的适配，进而调用 zlg72128_init()完成了 ZLG72128 通用驱动库的初始化，此外，还对中断引脚进行了处理，配置引脚中断模式等。对于用户来讲，仅需根据实际情况完成相关配置信息的修改即可。

4.1.3 应用

当调用实例初始化函数获得 ZLG72128 句柄时，即可使用该句柄操作 ZLG72128，在 3.5 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 4.2。

程序清单 4.2 使用 ZLG72128 通用测试程序进行测试

```
1   #include "ametal.h"
2   #include "demo_zlg72128_entries.h"
3   #include "zlg72128.h"
4
5   extern zlg72128_handle_t    am_zlg72128_inst_init (void);                  // 声明实例初始化函数
6
7   int am_main (void)
8   {
9       zlg72128_handle_t    handle = am_zlg72128_inst_init();
```

```
10
11          demo_zlg72128_combination_key_test_entry(handle);                    // 启动组合键测试 Demo
12
13          while(1) {
14          }
15      }
```

由此可见，通过实例初始化函数可以快捷的获取到 ZLG72128 句柄，该句柄可用于通用软件包提供的各个功能接口。

4.2 使用 AMetal 提供的跨平台通用接口

为了使应用程序不与具体的硬件绑定，进而实现“跨平台复用”，AMetal 提供了一套通用接口。通用接口只与“抽象的”功能相关，而与“具体的”硬件无关。若应用程序基于通用接口编写，由于通用接口与硬件无关，因而应用程序不会与具体的硬件绑定，则在更换底层硬件时，应用程序无需作任何修改。

例如，对于数码管显示功能，AMetal 提供了一组通用数码管接口，无论底层使用的何种类型的数码管驱动方式（GPIO 驱动、HC595 驱动、ZLG72128 等专用芯片驱动），应用程序都可以使用这一组接口对数码管进行操作，显示相应的内容。

同理，针对按键管理功能，AMetal 也提供了一组接口，其接口形式也与具体硬件无关，无论底层使用的何种键盘驱动方式（独立按键、矩阵键盘、ZLG72128 等专用管理芯片），均可使用通用的键盘管理接口实现对按键的管理。

在实际应用中，使用通用接口操作某一器件（如 ZLG72128）前，必须先完成器件的初始化，以使该器件可以使用通用接口进行操作。接下来，首先介绍通用数码管接口和按键管理接口，使读者对 AMetal 提供的通用接口有一定的了解，然后介绍 ZLG72128 的初始化方法，在初始化过程中，可能涉及到一些具体的配置操作。若用户已经使用过 AMetal，并对数码管接口和按键管理接口较为熟悉，则可以跳过前两小节中对接口的介绍，直接阅读 4.2.3 节的内容，了解 ZLG72128 的初始化方法，完成 ZLG72128 的配置和初始化后即可使用。

4.2.1 通用数码管接口

AMetal 提供了一组通用数码管接口，详见表 4.2。

表 4.2 数码管通用接口（am_digitron_disp.h）

函数原型	功能简介
int **am_digitron_disp_decode_set** (int id, uint16_t (*pfn_decode)(uint16_t ch));	设置段码解码函数
int **am_digitron_disp_blink_set** (int id, int index, am_bool_t blink);	设置数码管闪烁
int **am_digitron_disp_at** (int id, int index, uint16_t seg);	显示指定的段码图形
int **am_digitron_disp_char_at** (int id, int index, const char ch);	显示字符
int **am_digitron_disp_str** (int id, int index, int len, const char *p_str);	显示字符串
int **am_digitron_disp_clr** (int id);	显示清屏

这些接口的具体用法将在后文详细介绍。粗略浏览各个接口可以发现，接口以“am_digitron_disp_”作为命名空间，命名中不再存在“zlg72128”关键字，表明这些接口与 ZLG72128 是无关的，即使以后更换了其它数码管管理芯片，也不会影响接口的使用，换句话说，“应用程序就不用再改了！”

所有接口的第一个参数为 id，id 的作用类似于 handle，用于指定具体使用的数码管显示

器。在系统中，每个数码管显示器都有一个唯一的 id 与之对应，例如，若系统中只使用了一个由 ZLG72128 驱动的显示器，则编号默认为 0；若系统中既存在使用 ZLG72128 驱动的数码管显示器，也存在使用 GPIO 驱动的数码管显示器，则它们的编号可能分别为 0 和 1。

若期望编号 0 与 ZLG72128 驱动的数码管显示器对应，则必须在使用各个接口操作 id 为 0 的数码管之前，指定 ZLG72128 对应的数码管显示器编号为 0，这一操作通常在 ZLG72128 的初始化过程中完成，关于 ZLG72128 的初始化，将在 4.2.3 节中详细介绍。

1. 设置段码解码函数

通过控制数码管各个段的亮灭，可以组合显示出多种图形，例如，对于 8 段数码管，要显示字符“1”，则需要点亮 b、c 两段，对应的编码值（即段码）为 0x60。解码函数用于对特定字符进行解码，以获取对应字符的编码值。根据编码值，可以知道在显示对应字符时，哪些段需要点亮（相应位为 1），哪些段需要熄灭（相应位为 0）。设置段码解码函数即用于用户自定义字符的解码函数，其函数原型为：

```
int am_digitron_disp_decode_set (int id, uint16_t (*pfn_decode) (uint16_t ch));
```

其中，id 表示数码管显示器的编号，通常，若系统只有一个数码管显示器，则 id 为 0。pfn_decode 为函数指针，其指向的函数即为本次设置的段码解码函数，解码函数的参数为 uint16_t 类型的字符，返回值为 uint16_t 类型的编码。

绝大部分情况下，对于 8 段数码管，常用字符图形（如字符'0' ~ '9'等）都具有默认编码，为此，AMetal 提供了默认的 8 段数码管解码函数，可以支持常见的字符'0' ~ '9'以及'A' 、'B'、 'C' 、'D'、 'E'、 'F'等字符的解码。其在 am_digitron_disp.h 文件中声明：

```
uint16_t am_digitron_seg8_ascii_decode (uint16_t ascii_char);
```

若无特殊需求，可以将该函数作为 pfn_decode 的实参传递，范例程序详见程序清单 4.3。

程序清单 4.3　am_digitron_disp_decode_set()范例程序

```
1    am_digitron_disp_decode_set(0, am_digitron_seg8_ascii_decode);
```

部分应用可能具有特殊需求，需要在显示某些字符时使用自定义的编码，比如，要使字符'O'的编码为 0xFC，则可以自定义如下解码函数：

```
1    uint16_t my_decode(uint16_t ch)
2    {
3        //…其它字符的解码
4        if (ch =='O') {
5            return 0xFC;
6        }
7    }
```

然后将该函数作为 pfn_decode 的实参传递即可：

```
am_digitron_disp_decode_set(0, my_decode);
```

2. 设置数码管闪烁

该函数可以指定数码管显示器的某一位数码管闪烁，其函数原型为：

```
int am_digitron_disp_blink_set (int id, int index, am_bool_t blink);
```

其中，id 为数码管显示器编号；index 为数码管索引，通常情况下，一个数码管显示器具有多个显示位，索引即用于指定具体操作哪一位数码管，例如，ZLG72128 最高可以驱动 12 位数码管，则该数码管显示器对应的位索引范围为：0 ~ 11；blink 表示该位是否闪

烁，若其值为 AM_TRUE，则闪烁，反之，则不闪烁，默认情况下，所有数码管均处于未闪烁状态。如设置 1 号数码管闪烁的范例程序详见程序清单 4.4。

程序清单 4.4　am_digitron_disp_blink_set()范例程序

```
1    am_digitron_disp_blink_set(0, 1, AM_TRUE);
```

3. 显示指定的段码图形

该函数用于不经过解码函数解码，直接显示段码指定的图形，可以灵活的显示任意特殊图形，其函数原型为：

```
int am_digitron_disp_at (int id, int index, uint16_t seg);
```

其中，id 为数码管显示器编号；index 为数码管索引；seg 为显示的段码。如在 8 段数码管上显示字符'-'，即需要 g 段点亮，对应的段码为 0x02（即：0000 0010），范例程序详见程序清单 4.5。

程序清单 4.5　am_digitron_disp_at()范例程序

```
1    am_digitron_disp_at(0, 1, 0x02);
```

4. 显示单个字符

该函数用于在指定位置显示一个字符，字符经过解码函数解码后显示，若解码函数不支持该字符，则不显示任何内容，其函数原型为：

```
int am_digitron_disp_char_at (int id, int index, const char ch);
```

其中，id 为数码管显示器编号，index 为数码管索引，ch 为显示的字符。比如，显示字符'H'的范例程序详见程序清单 4.6。

程序清单 4.6　am_digitron_disp_char_at()范例程序

```
1    am_digitron_disp_char_at (0, 1, 'H');
```

5. 显示字符串

该函数用于从指定位置开始显示一个字符串，其函数原型为：

```
int am_digitron_disp_str (int id, int index, int len, const char *p_str);
```

其中，id 为数码管显示器编号，index 为显示字符串的数码管起始索引，即从该索引指定的数码管开始显示字符串，len 指定显示的长度（显示该字符串所使用的数码管位数），p_str 指向需要显示的字符串。

实际显示的长度是 len 和字符串长度的较小值，若数码管位数不够，则多余字符不显示。部分情况下，显示所占用的数码管长度可能与字符串实际显示的长度不等，例如，显示字符串“1.”，其长度为 2，但实际显示时，字符“1”和小数点均可显示在一位数码管上，因此，该显示仅占用一位数码管。

显示"HELLO."字符串的范例程序详见程序清单 4.7。

程序清单 4.7　am_digitron_disp_str()范例程序

```
1    am_digitron_disp_str(0, 0, 5,"HELLO.");
```

若只存在两位数码管，则实际只会显示"HE"。

通常情况下，需要显示一些数字，如显示变量的值，此时，可以先将变量通过格式化字符串函数输出到字符串缓冲区中，然后再使用 am_digitron_disp_str()函数显示该字符串。

比如，显示一个变量 num 的值，范例程序详见程序清单 4.8。

程序清单 4.8　使用 am_digitron_disp_str()显示整数变量值的范例程序

```
1    int          num = 53;
2    char         buf[3];
3    am_snprintf(buf, 3, "%2d", num);
4    am_digitron_disp_str(0, 0, 2, buf);
```

其中，am_snprintf()与标准 C 函数 snprintf()函数功能相同，均用于格式化字符串到指定大小的缓冲区中，其函数原型为（am_vdebug.h）：

```
int am_snprintf (char *buf, size_t sz, const char *fmt, ...);
```

6. 显示清屏

该函数用于显示清屏，清除数码管显示器中的所有内容，其函数原型为：

```
int am_digitron_disp_clr (int id);
```

其中，id 为数码管显示器编号，范例程序详见程序清单 4.9。

程序清单 4.9　am_digitron_disp_clr()范例程序

```
1    am_digitron_disp_clr(0);
```

基于数码管通用接口，可以编写一个简易的 60s 倒计时程序，当倒计时还剩 5s 时，数码管闪烁。为便于复用，将应用程序存放到 app_digitron_count_down.c 文件中，并将其接口声明到 app_digitron_count_down.h 文件中，详见程序清单 4.10 和程序清单 4.11。

程序清单 4.10　倒计时应用程序实现（demo_digitron_count_down.c）

```
1    #include "ametal.h"
2    #include "am_digitron_disp.h"
3    #include "am_vdebug.h"
4    #include "am_delay.h"
5
6    static void __digitron_show_num (int id, int num)
7    {
8        char buf[3];
9        am_snprintf(buf, 3, "%2d", num);
10       am_digitron_disp_str(id, 0, 2, buf);
11   }
12
13   int demo_digitron_count_down (int id)                          // 应用函数入口
14   {
15       unsigned int sec = 60;
16       am_digitron_disp_decode_set(0, am_digitron_seg8_ascii_decode);   // 使用默认的解码函数
17       while (1) {
18           __digitron_show_num(id, sec);                          // 显示当前秒
19           if (sec > 0) {
20               sec--;
21           } else {
```

```
22                          sec = 60;
23                      }
24                      if (sec < 5) {                                  // 低于 5s，打开闪烁
25                          am_digitron_disp_blink_set(0, 1, AM_TRUE);
26                      } else {
27                          am_digitron_disp_blink_set(0, 1, AM_FALSE);
28                      }
29                      am_mdelay(1000);                                // 延时 1s
30                  }
31                  return AM_OK;
32              }
```

程序清单 4.11　倒计时应用程序接口声明（demo_digitron_count_down.h）

```
1   #pragma once
2   #include "ametal.h"
3
4   int demo_digitron_count_down (int id);                  // 应用程序入口
```

由此可见，要使用此应用程序，只需调用其入口函数 app_digitron_count_down()，并指定一个数码管显示器编号即可。应用程序与具体 MCU、数码管驱动方式无关，可以在任何 AMetal 平台上运行。使用 ZLG72128 启动该应用程序的范例详见程序清单 4.12。

程序清单 4.12　运行倒计时应用程序的主程序范例

```
1   #include "ametal.h"
2   #include "demo_digitron_count_down.h"
3
4   int am_main (void)
5   {
6       am_zlg72128_std_inst_init();            // ZLG72128 实例初始化
7       demo_digitron_count_down(0);            // 使用显示器编号为 0 的数码管
8       while (1) {
9       }
10  }
```

有关 ZLG72128 实例的初始化，将在 4.2.3 节中详细介绍。由此可见，若后续改用其它数码管驱动方式（例如，修改为 GPIO 直接驱动数码管的方式），则仅仅是主程序中的初始化语句发生变化，应用程序本身不需要作任何改变，换句话说，应用程序可以编译成库，独立开发、维护和部署。

4.2.2 通用键盘管理接口

在 AMetal 中，键盘的管理仅需使用到一个接口，其原型如下：

```
int am_input_key_handler_register(
    am_input_key_handler_t      *p_handler,
    am_input_cb_key_t             pfn_cb,
    void                        *p_arg);
```

其中，p_handler 为指向按键事件处理器的指针，pfn_cb 为指向用户自定义按键处理函

数的指针，p_arg 为按键处理函数的用户参数。

1. p_handler

am_input_key_handler_t 是按键事件处理器的类型，它是在 am_input.h 文件中使用 typedef 自定义的一个类型。即：

```
typedef struct am_input_key_handler am_input_key_handler_t;
```

在使用按键时，仅需使用该类型定义一个的按键事件处理器实例（对象），其本质是定义一个结构体变量。比如：

```
am_input_key_handler_t    key_handler;                    // 定义一个按键事件处理器实例（对象）
```

实例的地址（&key_handler）即可作为参数传递给函数的形参 p_handler。

应用程序无需直接操作实例（比如访问其中的成员），因此，用户无需对 am_input_key_handler_t 类型的具体定义作过多的了解（比如了解包含哪些成员以及成员的具体含义）。

2. pfn_cb

am_input_cb_key_t 是按键处理函数的指针类型，它是在 am_input.h 文件中使用 typedef 自定义的一个类型。即：

```
typedef void (*am_input_cb_key_t) (void *p_arg, int key_code, int key_state, int keep_time);
```

当有按键事件发生时（按键按下或按键释放），系统会自动调用 pfn_cb 指向的按键处理函数，以完成按键事件的处理。当该函数被调用时，系统会将按键相关的信息通过参数传递给用户，各参数的含义如下：

- p_arg

用户参数，即用户调用 am_input_key_handler_register()函数时设置的第三个参数的值。该值的含义完全由用户决定，系统仅作简单的传递工作。

- key_code

按键的编码，按键编码用于区分各个按键，通常情况下，一个系统中可能存在多个按键，比如，ZLG72128 最多支持 32 个按键， 为每个按键分配一个唯一的编码，当按键事件发生时，用户可以据此判断是哪个按键产生了按键事件。此外，出于可读性、可维护性等考虑，按键编码一般不直接使用数字，比如：1、2、3……而是使用在 am_input_code.h 文件中使用宏的形式定义的一系列编码，比如，KEY_1、KEY_2 等，用以区分各个按键；

- key_state

按键的状态（按下或释放），用户可以据此判断按键的状态，以便根据不同的状态作出不同的处理，各状态对应的宏定义详见表 4.3。

表 4.3 按键状态

宏名	含义
AM_INPUT_KEY_STATE_PRESSED	按键按下
AM_INPUT_KEY_STATE_RELEASED	按键释放

- keep_time

表示状态保持时间（单位：ms），常用于按键长按应用（例如，按键长按 3 秒关机），按键首次按下时，keep_time 为 0，若按键一直保持按下，则系统会以一定的时间间隔上报按键按下事件（调用 pfn_cb 指向的用户回调函数），keep_time 的值不断增加，表示按键按下已经保持的时间。特别地，若按键不支持长按功能，则 keep_time 始终为-1。

例如，做一个简单的按键提示应用：按键按下时，LED 点亮；按键释放时，LED 熄灭。相应的按键处理函数详见程序清单 4.13（假定按键对应的按键编码为 KEY_1）。

程序清单 4.13　按键处理函数范例程序——使用按键基本功能

```
1   static void __input_key_proc(void *p_arg, int key_code, int key_state, int keep_time)
2   {
3       if (key_code == KEY_1) {
4           if (key_state == AM_INPUT_KEY_STATE_PRESSED) {          // 有键按下
5               am_led_on(0);
6           }else if (key_state == AM_INPUT_KEY_STATE_RELEASED){// 按键释放
7               am_led_off(0);
8           }
9       }
10  }
```

函数名即可作为参数传递给 am_input_key_handler_register()函数的 pfn_cb 形参。

在按键处理中，可以使用 keep_time 信息实现按键长按功能。例如：使用按键长按功能模拟开关机，按键长按 3s，LED0 状态翻转，模拟切换“开关机”状态，LED0 亮表示“开机“；LED0 熄灭表示“关机”，按键处理范例程序详见程序清单 4.14。

程序清单 4.14　按键处理函数范例程序——使用按键长按功能

```
1   static void __input_key_proc(void *p_arg, int key_code, int key_state, int keep_time)
2   {
3       if (key_code == KEY_1) {
4           if ((key_state == AM_INPUT_KEY_STATE_PRESSED) && (keep_time == 3000)) {
5               am_led_toggle(0);                                   // 长按时间达到 3s,  LED0 状态翻转
6           }
7       }
8   }
```

3. p_arg

该参数的值会在调用事件处理回调函数（pfn_cb 指向的函数）时，原封不动的传递给事件处理函数的 p_arg 形参。如果不使用，则在调用 am_input_key_handler_register()函数时，将 p_arg 的值设置为 NULL，注册按键处理器的范例程序详见程序清单 4.15。

程序清单 4.15　按键处理函数范例程序

```
1   #include "ametal.h"
2   #include "am_input.h"
3
4   static am_input_key_handler_t g_key_handler;      // 事件处理器实例定义，全局变量，确保一直有效
5
6   int am_main (void)
7   {
8       am_input_key_handler_register(&g_key_handler, __input_key_proc, (void *)NULL);
9       while (1) {
10      }
11  }
```

注册按键处理器后，当有按键发生时，系统会自动调用注册按键处理器时指定的回调

函数，即程序清单 4.13 中的__input_key_proc ()函数。为了分离各个键的处理代码，可以注册多个按键事件处理器，每个处理器负责处理一个或多个键，详见程序清单 4.16。

程序清单 4.16　注册多个按键处理器范例程序

```
1   #include "ametal.h"
2   #include "am_input.h"
3   #include "am_led.h"
4
5   static void __input_key1_proc(void *p_arg, int key_code, int key_state, int keep_time)
6   {
7       if (key_code == KEY_KP0) {
8           // 处理按键 1
9       }
10  }
11
12  static void __input_key2_proc (void *p_arg, int key_code, int key_state, int keep_time)
13  {
14      if (key_code == KEY_KP1){
15          // 处理按键 2
16      }
17  }
18
19  static void __input_key3_proc (void *p_arg, int key_code, int key_state, int keep_time)
20  {
21      if (key_code == KEY_KP2){
22          // 处理按键 3
23      }
24  }
25
26  static am_input_key_handler_t g_key1_handler;
27  static am_input_key_handler_t g_key2_handler;
28  static am_input_key_handler_t g_key3_handler;
29
30  int am_main (void)
31  {
32      am_input_key_handler_register(&g_key1_handler, __input_key1_proc, NULL);
33      am_input_key_handler_register(&g_key2_handler, __input_key2_proc, NULL);
34      am_input_key_handler_register(&g_key3_handler, __input_key3_proc, NULL);
35      while (1){
36      }
37  }
```

通用键盘接口的特点是屏蔽了底层的差异性，使应用程序与底层 MCU、键盘的具体形式无关，可以轻松地实现应用程序的跨平台复用。

在实际的应用中，键盘的表现形式是多种多样的，比如，直接使用 GPIO 驱动的独立键

盘（一个或多个独立按键组成的键盘）和矩阵键盘、标准的 PS/2 接口键盘，以及使用 ZLG72128 芯片管理的外接键盘。虽然各种按键的检测方法都不尽相同，但对于应用程序来讲，进行按键管理的接口却是一样的。

4.2.3 ZLG72128 初始化

1. 实例初始化函数

在使用通用接口操作 ZLG72128 之前，需要完成 ZLG72128 的初始化。为便于用户使用，AMetal 提供了对应的实例初始化函数，直接调用即可完成初始化，其函数原型为：

```
int  am_zlg72128_std_inst_init(void);
```

其中，“std”关键字表示该实例初始化函数主要用于将 ZLG72128 用作通用功能，以便使用 AMetal 通用接口对其进行操作。由于通用数码管接口和键盘管理接口均无需传入 handle，因此，实例初始化函数未返回 handle，仅返回了一个用于表示成功（返回值为 0）或失败（返回值为非 0）的整型值。

该实例初始化函数的调用形式如下：

```
am_zlg72128_std_inst_init();
```

即在主程序中调用该函数即可完成 ZLG72128 通用功能的初始化，进而使用通用接口访问 ZLG72128。.

2. 配置

通常情况下，实例初始化函数在模板工程中的 ZLG72128 配置文件中实现，配置文件的默认名称为：am_hwconf_zlg72128.c，以使用 LPC824 驱动 ZLG72128 为例，其具体内容详见程序清单 4.17。

程序清单 4.17　ZLG72128 配置文件内容示意

```
1   #include "ametal.h"
2   #include "am_lpc82x_inst_init.h"
3   #include "lpc82x_pin.h"
4   #include "am_zlg72128.h"
5
6   static am_zlg72128_dev_t        __g_zlg72128_std_dev;           // 定义一个 ZLG72128 实例
7
8   // 2×2 键盘，共计 4 个按键，各按键对应的编码
9   am_local am_const int __g_zlg72128_key_codes[] = {
10      KEY_0,  KEY_1, KEY_2,  KEY_3
11  };
12
13  static const am_zlg72128_devinfo_t __g_zlg72128_std_devinfo = {
14      {
15          0x30,                                   // 从机地址
16          PIO0_6,                                 // 复位引脚
17          AM_TRUE,                                // 使用中断引脚
18          PIO0_1,                                 // 中断引脚
19          5,                                      // 查询时间间隔，使用中断引脚时，该值无意义
20          am_lpc82x_i2c2_inst_init                // I²C 句柄获取函数
```

```
21              am_lpc82x_i2c2_inst_deinit              // I²C 解初始化函数
22          },
23          {
24              0                                       // 数码管显示器的编号
25          },
26          500,                                        // 一个闪烁周期内，点亮的时间为 500ms
27          500,                                        // 一个闪烁周期内，熄灭的时间为 500ms
28          AM_ZLG72128_STD_KEY_ROW_0 | AM_ZLG72128_STD_KEY_ROW_3, // 实际使用的行标志
29          AM_ZLG72128_STD_KEY_COL_0 | AM_ZLG72128_STD_KEY_COL_1,  // 实际使用的列标志
30          __g_zlg72128_key_codes,                     // 按键编码信息
31          2                                           // 数码管个数为 2
32      };
33      int   am_zlg72128_std_inst_init (void)
34      {
35          return am_zlg72128_std_init(&__g_zlg72128_dev,
36                                      &__g_zlg72128_devinfo);
37      }
```

注：若在用户所使用的工程中不存在该文件，则可以将程序清单 4.17 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_zlg72128_std.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。用户可能需要对其中的 14 个地方作修改，视为 14 个配置项，各配置项的功能简介详见表 4.4，这些配置项都可以根据实际情况修改。

表 4.4　ZLG72128 用作通用功能时的配置项

序号	对应程序清单 4.17 的行号	配置项	默认值
1	15	从机地址	0x30
2	16	复位引脚	PIO0_6
3	17	获取按键事件的方式	AM_TRUE
4	18	中断引脚	PIO0_1
5	19	查询时间间隔	5
6	20	I²C 句柄获取函数	am_lpc82x_i2c2_inst_init
7	21	I²C 解初始化函数	am_lpc82x_i2c2_inst_deinit
8	24	数码管显示器 ID	0
9	26	数码管闪烁属性（点亮时间）	500ms
10	27	数码管闪烁属性（熄灭时间）	500ms
11	28	实际使用的键盘行线	第 0 行和第 3 行，共计 2 行
12	29	实际使用的键盘列线	第 0 列和第 1 列，共计 2 列
13	10	各个按键对应的编码	2（实际行数由配置项 9 决定）×2（实际列数由配置项 10 决定）键盘，共计 4 个按键，对应的编码由__g_zlg72128_key_codes 数组表示，分别为：KEY_0、KEY_1、KEY_2、KEY_3。
14	31	实际使用的数码管个数	2

下面，分别对这 14 个配置项作详细介绍。

（1） 从机地址

该配置项对应程序清单 4.17 的第 15 行，默认值为：0x30。该处设置的值表示 ZLG72128 的 7 位 I²C 从机地址，其值与 ZLG72128 的 A4 引脚电平相关：若为高电平，则地址为：0x30；若为低电平，则地址为：0x20。

（2） 复位引脚

该配置项对应程序清单 4.17 的第 16 行，默认值为：PIO0_6。该处设置的值表示 ZLG72128 的 RST 引脚与主控 MCU（如 LPC824）连接的引脚号，若将 ZLG72128 复位引脚与主控 MCU 相接，则主控 MCU 可以通过该引脚控制 ZLG72128 的复位。这里即表示 LPC824 的 PIO0_6 与 ZLG72128 的 RST 引脚相连。一般情况下，为了节省管脚，复位引脚固定连接至一个 RC 复位电路，无需由 MCU 控制，此时，可将该处的值配置为-1。

（3） 获取按键事件的方式

该配置项对应程序清单 4.17 的第 17 行，默认值为：AM_TRUE。虽然 ZLG72128 提供了 KEY_INT 引脚，用以在按键事件产生时主动通知主控 MCU，但部分特殊情况下，为了节省 I/O，可能不使用该引脚。该配置项即用于配置是否使用中断引脚。若值为真，则表示使用中断引脚（中断引脚由配置项 5 指定）；否则，不使用中断引脚，此时，将采用查询模式（查询时间间隔由配置项 6 指定）查询是否有按键事件产生。

（4） 中断引脚

该配置项对应程序清单 4.17 的第 18 行，默认值为：PIO0_1。当使用中断模式（配置项 4 的值为真）时，该配置项有效，用于指定 ZLG72128 的 KEY_INT 引脚与主控 MCU 连接的引脚号。这里即表示 LPC824 的 PIO0_1 与 ZLG72128 的 KEY_INT 引脚相连。

（5） 配置查询时间间隔

该配置项对应程序清单 4.17 的第 19 行，默认值为：5（单位：ms）。当使用查询模式（配置项 4 的值为假）时，该配置项有效，用于指定查询的时间间隔，默认为 5ms，用户也可根据需要修改为其它值。

（6） I²C 句柄获取函数

该配置项对应程序清单 4.17 的第 20 行，默认值为：am_lpc82x_i2c2_inst_init。该行代码主要是为了获取 I²C 通信函数的句柄，以便驱动底层使用该句柄实现 I²C 数据的收发。LPC824 默认有 4 路 I²C： I²C0、I²C1、I²C2 和 I²C3。该配置项即用于配置使用哪条 I²C 总线与 ZLG72128 通信，默认配置中，使用了 I²C2，若期望使用 I²C1 与 ZLG72128 通信，则该行配置可以修改为：am_lpc82x_i2c1_inst_init。

（7） I²C 解初始化函数

该配置项对应程序清单 4.17 的第 21 行，默认值为：am_lpc82x_i2c2_inst_deinit。该行代码提供了释放 I²C 句柄的方法，在上一个配置项中提供了获取了 I²C 句柄的方法，当 ZLG72128 正常使用时，会通过获取句柄方法获得 I²C 句柄，进而占用了 I²C 资源。当不使用 ZLG72128 时，驱动可以通过该配置项提供的方法释放之，以解除对相关资源的占用。该配置中释放的 I²C 总线必须与上一个配置项相对应，比如，默认对应的都是 I²C2。若上一个配置项期望使用 I²C1 与 ZLG72128 通信，则该行配置可以修改为：am_lpc82x_i2c1_inst_deinit。

（8） 数码管显示器 ID

该配置项对应程序清单 4.17 的第 24 行，默认值为：0。在 AMetal 中，为每个数码管显示器分配了一个唯一 ID，在使用通用接口操作数码管时，将通过 ID 指定要操作的数码

管，这里的默认值为 0，表明当使用通用接口控制 ID 为 0 的数码管时，实际上控制的即为 ZLGH72128 驱动的数码管。

（9） 闪烁属性（点亮时间）

该配置项对应程序清单 4.17 的第 26 行，默认值为：500ms。该配置项指定了数码管闪烁时，在一个闪烁周期内，数码管点亮的时间，通常情况下，数码管以 1Hz 频率闪烁，点亮和熄灭时间相等，点亮时间即为 500ms。

（10） 闪烁属性（熄灭时间）

该配置项对应程序清单 4.17 的第 27 行，默认值为：500ms。该配置项指定了数码管闪烁时，在一个闪烁周期内，数码管熄灭的时间，通常情况下，数码管以 1Hz 频率闪烁，点亮和熄灭时间相等，熄灭时间即为 500ms。

（11） 实际使用的键盘行线

该配置项对应程序清单 4.17 的第 28 行，默认值为：AM_ZLG72128_STD_KEY_ROW_0 | AM_ZLG72128_STD_KEY_ROW_3，即使用第 0 行和第 3 行。

ZLG72128 可以管理多达 32（4 行 8 列）个按键，但实际应用中，不一定使用到这么多，可以按行进行裁剪，以减少按键数目。该配置项用于表示实际使用了哪些行。ZLG72128 最多支持 4 行按键，分别对应 COM8 ~ COM11。该值由表 4.5 所示的宏值组成，使用多行时应将多个宏值相“或”。例如，使用了第 0 行和第 3 行，则该配置项的值为：

表 4.5 行使用宏标志

宏名	含义
AM_ZLG72128_STD_KEY_ROW_0	行 0
AM_ZLG72128_STD_KEY_ROW_1	行 1
AM_ZLG72128_STD_KEY_ROW_2	行 2
AM_ZLG72128_STD_KEY_ROW_3	行 3

```
AM_ZLG72128_STD_KEY_ROW_0  |
AM_ZLG72128_STD_KEY_ROW_3
```

（12） 实际使用的键盘列线

该配置项对应程序清单 4.17 的第 29 行，默认值为：AM_ZLG72128_STD_KEY_COL_0 | AM_ZLG72128_STD_KEY_COL_1，即使用第 0 列和第 1 列。

与上一个配置项类似，键盘设计时不仅可以按行进行裁剪，还可以按列进行裁剪，以减少按键数目。该配置项用于表示实际使用了哪些列。ZLG72128 最多支持 8 列按键，分别对应 COM0 ~ COM7。该值由表 4.6 所示的宏值组成，使用多列时应将多个宏值相“或”。例如，使用了第 0 列和第 1 列，则该配置项的值即为：

表 4.6 列使用宏标志

宏名	含义
AM_ZLG72128_STD_KEY_COL_0	列 0
AM_ZLG72128_STD_KEY_COL_1	列 1
……	
AM_ZLG72128_STD_KEY_COL_7	列 7

```
AM_ZLG72128_STD_KEY_COL_0  |
AM_ZLG72128_STD_KEY_COL_1
```

（13） 按键编码

该配置项对应程序清单 4.17 的第 10 行，默认值为：KEY_0, KEY_1, KEY_2, KEY_3。ZLG72128 可以管理多达 32（4 行 8 列）个按键，但实际应用中，不一定使用到这么多，实际使用到的行由配置项 9 决定，实际使用到的列由配置项 10 决定。在 AMetal 中，需要为每个按键分配一个唯一编码，便于应用程序使用。该配置项即用于配置各个按键对应的编码，默认为 KEY_0 ~ KEY_3（在 am_input_code.h 文件中定义的宏）。该处编码个数应该与实际

使用的按键个数相等，默认配置中，将键盘配置为了 2×2 的键盘，因此，编码个数应该为 4。用户可以根据需要，将编码修改为其它值。

（14） 实际使用的数码管个数

该配置项对应程序清单 4.17 的第 31 行，默认值为：2。ZLG72128 可以驱动多达 12 个数码管，但实际应用中，不一定使用到全部的数码管，该配置项用于表示实际使用的数码管个数。

3. 应用

在调用实例初始化函数完成 ZLG72128 通用功能的初始化后，即可使用 AMetal 提供的通用数码管接口和按键管理接口操作 ZLG72128，一个简单的应用范例详见程序清单 4.18。

程序清单 4.18 使用 ZLG72128 通用测试程序进行测试

```
1   #include "ametal.h"
2   #include "am_digitron_disp.h"
3
4   static void __input_key_proc(void *p_arg, int key_code, int key_state, int keep_time)
5   {
6       if (key_state == AM_INPUT_KEY_STATE_PRESSED) {
7           switch (key_code) {
8           case KEY_1:
9               // KEY1 键处理
10              break;
11          case KEY_2:
12              // KEY2 键处理
13              break;
14          case KEY_3:
15              // KEY3 键处理
16              break;
17          case KEY_4:
18              // KEY4 键处理
19              break;
20          default:
21              break;
22          }
23      }
24  }
25
26  static am_input_key_handler_t g_key_handler;      // 事件处理器实例定义，全局变量，确保一直有效
27  extern int    am_zlg72128_std_inst_init(void);    // 声明实例初始化函数
28
29  int am_main (void)
30  {
31      am_zlg72128_std_inst_init();                  // 完成 ZLG72128 初始化
32
33      am_input_key_handler_register(&g_key_handler, __input_key_proc, (void *)NULL);
```

```
34          am_digitron_disp_str (0, 0, 2, "01");                    // 显示字符串"01"
35          // 其它数码管操作
36          while(1) {
37          }
38  }
```

第5章 在 AWorks 中使用 ZLG72128

✍本章导读

AWorks 平台已经完成了对 ZLG72128 的适配，可以在使能设备之后，使用 ZLG72128 通用驱动软件包中提供的各个接口操作 ZLG72128，完成数码管显示和按键管理，无需关心任何底层细节，使用户可以更加快捷的将 ZLG72128 应用到实际项目中。除此之外，若选用 AWorks 平台，更加推荐的做法是直接使用 AWorks 提供的通用数码管接口和键盘管理接口完成数码管的显示以及按键的管理。这将使应用程序与数码管驱动方式无关，进而解除应用程序与 ZLG72128 之间的耦合，实现应用程序的“跨平台复用”。

5.1 设备使能及配置

在使用 ZLG72128 之前，必须使能 ZLG72128 硬件设备，并完成相关的配置。

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_ZLG72128_0 宏处于有效状态，即未被注释。

```
#define AW_DEV_ZLG72128_0
```

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 ZLG72128 设备的默认配置，此时，用户还需添加相应的配置文件（在下节介绍配置时会详细介绍）。

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_zlg72128.h 文件中，文件的示意内容详见程序清单 5.1。

程序清单 5.1 awbl_hwconf_zlg72128.h 文件内容

```
1   #ifndef __AWBL_HWCONF_ZLG72128_H
2   #define __AWBL_HWCONF_ZLG72128_H
3
4   #ifdef AW_DEV_ZLG72128_0
5
6   #include "aw_types.h"
7   #include "aw_input.h"
8   #include "driver/digitron_key/awbl_zlg72128.h"
9
10  /* 按键编码 */
11  aw_local aw_const int __g_key_codes[] = {
12          KEY_0, KEY_1,KEY_2, KEY_3
13  };
14
15  /* 设备信息 */
16  aw_local aw_const struct awbl_zlg72128_devinfo __g_zlg72128_devinfo = {
```

```
17          0,                      /* 数码管显示器 ID */
18          0x30,                   /* 从机地址        */
19          PIO1_17,                /* 复位引脚        */
20          PIO1_18,                /* 中断引脚        */
21          20,                     /* 按键查询时间间隔，使用中断引脚时，该值无意义 */
22          2,                      /* 数码管个数为 2  */
23          500,                    /* 一个闪烁周期内，点亮的时间为 500ms      */
24          500,                    /* 一个闪烁周期内，熄灭的时间为 500ms      */
25          AWBL_ZLG72128_KEY_ROW_0 | AWBL_ZLG72128_KEY_ROW_3,   /* 实际使用的行标志 */
26          AWBL_ZLG72128_KEY_COL_0 | AWBL_ZLG72128_KEY_COL_1,   /* 实际使用的列标志 */
27          __g_key_codes,          /* 按键编码， KEY_0 ~ KEY3                 */
28      };
29
30      /* 设备实例内存静态分配 */
31      aw_local struct awbl_zlg72128_dev __g_zlg72128_dev;
32
33      #define AWBL_HWCONF_ZLG72128_0                          \
34          {                                                   \
35              AWBL_ZLG72128_NAME,                             \
36              0,                                              \
37              AWBL_BUSID_I2C,                                 \
38              IMX28_I2C0_BUSID,                               \
39              (struct awbl_dev *)&__g_zlg72128_dev,           \
40              &(__g_zlg72128_devinfo)                         \
41          },
42
43      #else
44      #define AWBL_HWCONF_ZLG72128_0
45      #endif
46
47      #endif
```

特别地，若在用户获取的 SDK 中，不包含此文件，则可以自行新建一个名为 awbl_hwconf_zlg72128.h 的文件，并将程序清单 5.1 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

该文件的主要作用是完成 ZLG72128 的配置，并对外提供一个设备宏定义，即：AWBL_HWCONF_ZLG72128_0，仅关注与定义该宏相关的语句，忽略其它代码，即：

```
#ifdef   AW_DEV_ZLG72128_0
#define AWBL_HWCONF_ZLG72128_0                          \
    {                                                   \
        AWBL_ZLG72128_NAME,                             \
        // 其它信息...
```

```
    },
#else
#define AWBL_HWCONF_ZLG72128_0
#endif
```

由此可见，该宏可能有两种定义，具体定义被 AW_DEV_ZLG72128_0 宏控制，仅当 AW_DEV_ZLG72128_0 被有效定义时，AWBL_HWCONF_ZLG72128_0 的定义才包含实际内容，否则，AWBL_HWCONF_ZLG72128_0 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_ZLG72128_0 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。一个简单的示例片段详见程序清单 5.2。

程序清单 5.2　硬件设备列表（awbus_lite_hwconf_usrcfg.c）

```
1   aw_const struct awbl_devhcf g_awbl_devhcf_list[] = {        //  硬件设备列表
2       AWBL_HWCONF_IMX1050_NVIC                                //  中断控制器
3       AWBL_HWCONF_IMX1050_GPIO                                // GPIO
4       AWBL_HWCONF_GPIO_LED                                    // LED 设备
5       // ......  其它硬件设备
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 ZLG72128，则应该将 ZLG72128 对应的设备宏加入到硬件设备列表中，详见程序清单 5.3。

程序清单 5.3　硬件设备列表中加入 ZLG72128

```
1   aw_const struct awbl_devhcf g_awbl_devhcf_list[] = {        //  硬件设备列表
2       AWBL_HWCONF_IMX1050_NVIC                                //  中断控制器
3       AWBL_HWCONF_IMX1050_GPIO                                // GPIO
4       AWBL_HWCONF_GPIO_LED                                    // LED 设备
5       AWBL_HWCONF_ZLG72128_0                                  // ZLG72128
6       // ......  其它硬件设备
7   };
```

通常情况下，若在系统工程中存在 ZLG72128 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_ZLG72128_0 控制设备是否使能即可。

2．配置项详解

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，ZLG72128 的复位引脚与主控 MCU 连接的引脚号。在 ZLG72128 的配置文件中，用户可以修改的信息有 12 个，视为 12 个配置项，各配置项的功能简介详见表 5.1。

表 5.1　ZLG72128 用作通用功能时的配置项

序号	对应程序清单 5.1 的行号	配置项	默认值
1	12	各个按键对应的编码	2（键盘实际行数由配置项 9 决定）×2（键盘实际列数由配置项 10 决定）键盘，共计 4 个按键，对应的编码由__g_key_codes 数组表示，分别为：KEY_0、KEY_1、KEY_2、KEY_3。
2	17	数码管显示器 ID	0
3	18	从机地址	0x30
4	19	复位引脚	PIO1_17
5	20	中断引脚	PIO0_1
6	21	查询时间间隔	20
7	22	实际使用的数码管个数	2
8	23	数码管闪烁属性（点亮时间）	500ms
9	24	数码管闪烁属性（熄灭时间）	500ms
10	25	实际使用的键盘行线	第 0 行和第 3 行，共计 2 行
11	26	实际使用的键盘列线	第 0 列和第 1 列，共计 2 列
12	36	设备单元号	0
13	38	所处总线配置	i.MX28x 的 I^2C0

下面，分别对这 13 个配置项作详细介绍。

（1） 按键编码

该配置项对应程序清单 5.1 的第 12 行，默认值为：KEY_0, KEY_1, KEY_2, KEY_3。ZLG72128 可以管理多达 32（4 行 8 列）个按键，但实际应用中，不一定使用到这么多，实际使用到的行由配置项 9 决定，实际使用到的列由配置项 10 决定。在 AWorks 中，需要为每个按键分配一个唯一编码，便于应用程序使用。该配置项即用于配置各个按键对应的编码，默认为 KEY_0～KEY_3（在 aw_input_code.h 文件中定义的宏）。该处编码个数应该与实际使用的按键个数相等，默认配置中，将键盘配置为了 2×2 的键盘，因此，编码个数应该为 4。用户可以根据需要，将编码修改为其它值。

（2） 数码管显示器 ID

该配置项对应程序清单 5.1 的第 17 行，默认值为 0。在 AWorks 中，每个数码管显示器都分配了一个唯一 ID，在使用通用接口（接口将在 5.3.1 节介绍）操作数码管时，将通过 ID 指定要操作的数码管，这里的默认值为 0，表明当使用通用接口控制 ID 为 0 的数码管时，实际上控制的即为 ZLG72128 驱动的数码管。

（3） 从机地址

该配置项对应程序清单 5.1 的第 18 行，默认值为 0x30。该处设置的值表示 ZLG72128 的 7 位 I^2C 从机地址，其值与 ZLG72128 的 A4 引脚电平相关：若为高电平，则地址为：0x30；若为低电平，则地址为：0x20。

（4） 复位引脚

该配置项对应程序清单 5.1 的第 19 行，默认值为 PIO1_17。该处设置的值表示 ZLG72128 的 RST 引脚与主控 MCU（如 i.MX28x）连接的引脚号，若将 ZLG72128 复位引脚与主控 MCU 相接，则主控 MCU 可以通过该引脚控制 ZLG72128 的复位。这里即表示 i.MX28x 的

PIO1_17 与 ZLG72128 的 RST 引脚相连。一般情况下，为了节省管脚，复位引脚固定连接至一个 RC 复位电路，无需由 MCU 控制，此时，可将该处的值配置为-1。

（5） 中断引脚

该配置项对应程序清单 5.1 的第 20 行，默认值为 PIO1_18。该处设置的值表示 ZLG72128 的 KEY_INT 引脚与主控 MCU 连接的引脚号，这里即表示 i.MX28x 的 PIO1_18 与 ZLG72128 的 KEY_INT 引脚相连。

虽然 ZLG72128 提供了 KEY_INT 引脚，用以在产生按键事件时主动通知主控 MCU，但部分特殊情况下，为了节省 I/O，可能不使用该引脚，改用查询方式发现按键事件，此时，中断引脚的值可以设置为无效值：-1。查询时间间隔将由下一个配置项指定。

（6） 配置查询时间间隔

该配置项对应程序清单 5.1 的第 21 行，默认值为 20。当使用查询模式（配置项 5 的值为-1）时，该配置项有效，用于指定查询的时间间隔，默认为 20ms，用户也可根据需要修改为其它值。

（7） 实际使用的数码管个数

该配置项对应程序清单 5.1 的第 22 行，默认值为 2。ZLG72128 可以驱动多达 12 个数码管，但实际应用中不一定使用到全部的数码管，该配置项用于表示实际使用的数码管个数。

（8） 闪烁属性（点亮时间）

该配置项对应程序清单 5.1 的第 23 行，默认值为 500。该配置项指定了数码管闪烁时，在一个闪烁周期内，数码管点亮的时间，通常情况下，数码管以 1Hz 频率闪烁，点亮和熄灭时间相等，点亮时间即为 500ms。

（9） 闪烁属性（熄灭时间）

该配置项对应程序清单 5.1 的第 24 行，默认值为 500。该配置项指定了数码管闪烁时，在一个闪烁周期内，数码管熄灭的时间，通常情况下，数码管以 1Hz 频率闪烁，点亮和熄灭时间相等，熄灭时间即为 500ms。

（10） 实际使用的键盘行线

该配置项对应程序清单 5.1 的第 25 行，默认值为：AWBL_ZLG72128_KEY_ROW_0 | AWBL_ZLG72128_KEY_ROW_3。

ZLG72128 可以管理多达 32（4 行 8 列）个按键，但实际应用中，不一定使用到这么多，可以按行进行裁剪，以减少按键数目。该配置项用于表示实际使用了哪些行。ZLG72128 最多支持 4 行按键，分别对应 COM8～COM11。该值由表 5.2 所示的宏值组成，使用多行时应将多个宏值相“或”。例如，使用了第 0 行和第 3 行，则该配置项的值为：

表 5.2 行使用宏标志

宏名	含义
AWBL_ZLG72128 _KEY_ROW_0	行 0
AWBL_ZLG72128 _KEY_ROW_1	行 1
AWBL_ZLG72128 _KEY_ROW_2	行 2
AWBL_ZLG72128 _KEY_ROW_3	行 3

```
AWBL_ZLG72128_KEY_ROW_0     |
AWBL_ZLG72128_KEY_ROW_3
```

（11） 实际使用的键盘列线

该配置项对应程序清单 5.1 的第 26 行，默认值为：AWBL_ZLG72128_KEY_COL_0 | AWBL_ZLG72128_KEY_COL_1。

与上一个配置项类似，键盘设计时不仅可以按行进行裁剪，还可以按列进行裁剪，以减

少按键数目。该配置项用于表示实际使用了哪些列。ZLG72128 最多支持 8 列按键，分别对应 COM0 ~ COM7。该值由表 5.3 所示的宏值组成，使用多列时应将多个宏值相“或”。例如，使用了第 0 列和第 1 列，则该配置项的值即为：

```
AWBL_ZLG72128_KEY_COL_0      |
AWBL_ZLG72128_KEY_COL_1
```

表 5.3　列使用宏标志

宏名	含义
AWBL_ZLG72128_KEY_COL_0	列 0
AWBL_ZLG72128_KEY_COL_1	列 1
……	
AWBL_ZLG72128_KEY_COL_7	列 7

（12） 设备单元号

该配置项对应程序清单 5.1 的第 36 行，默认值为 0。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 I²C 总线挂载了多个（假定数目为 N）ZLG72128，则单元号可能分别为 0 ~ (N-1)。若存在多个 ZLG72128，则每个 ZLG72128 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 ZLG72128，则单元号固定为 0。

（13） 所处总线配置

该配置项对应程序清单 5.1 的第 38 行，默认值为 IMX28_I2C0_BUSID。ZLG72128 是一个 I²C 从机器件，该配置项描述了 ZLG72128 挂在哪条 I²C 总线上，以便主控 MCU 使用相应的 I²C 总线与 ZLG72128 通信。每条 I²C 总线对应的编号在 aw_prj_params.h 文件中定义，例如，在 i.MX28x 硬件平台中，默认有 2 条硬件 I²C 总线：I²C0、I²C1，它们对应的总线编号为 0 和 1，定义详见程序清单 5.4。

程序清单 5.4　I²C 总线编号定义

```
1    #define IMX28_I2C0_BUSID        0              // 硬件 I2C0
2    #define IMX28_I2C1_BUSID        1              // 硬件 I2C1
```

默认使用了 I²C0 与 ZLG72128 通信，若在硬件电路设计时，ZLG72128 的通信接口连接至了 I²C1，则可以将配置项修改为：IMX28_I2C1_BUSID。

5.2　使用 ZLG72128 通用软件包接口

在第三章中提到，AWorks 已经完成了对 ZLG72128 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 ZLG72128 即可。

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 ZLG72128 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（awbl_zlg72128.h）：

```
zlg72128_handle_t    awbl_zlg72128_handle_get(int unit);
```

在获取时，需要传入一个单元号，用以指定获取哪个 ZLG72128 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 ZLG72128 时，单元号默认为 0。基于此，获取 handle 的语句即为：

```
zlg72128_handle_t    handle = awbl_zlg72128_handle_get(0);
```

当调用实例初始化函数获得 ZLG72128 句柄时，即可使用该句柄操作 ZLG72128，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 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 5.5。

程序清单 5.5　使用 ZLG72128 通用测试程序进行测试

```
1    #include "aworks.h"
2    #include "demo_zlg72128_entries.h"
3    #include "zlg72128.h"
4    #include "awbl_zlg72128.h"                                    // 包含了 handle 获取函数的声明
5    int aw_main (void)
6    {
7        zlg72128_handle_t    handle = awbl_zlg72128_handle_get(0);
8        demo_zlg72128_combination_key_test_entry(handle);          // 启动组合键测试 Demo
9        while(1) {
10           aw_mdelay(10);
11       }
12   }
```

5.3　使用 AWorks 提供的跨平台通用接口

为了使应用程序不与具体的硬件绑定，进而实现“跨平台复用”，AWorks 提供了一套通用接口。通用接口只与“抽象的”功能相关，而与“具体的”硬件无关。若应用程序基于通用接口编写，由于通用接口与硬件无关，因而应用程序不会与具体的硬件绑定，则在更换底层硬件时，应用程序无需作任何修改。

例如，对于数码管显示功能，AWorks 提供了一组通用数码管接口，无论底层使用的何种类型的数码管驱动方式（GPIO 驱动、HC595 驱动、ZLG72128 等专用芯片驱动），应用程序都可以使用这一组接口对数码管进行操作，显示相应的内容。

同理，针对按键管理功能，AWorks 也提供了一组接口，其接口形式也与具体硬件无关，无论底层使用的何种键盘驱动方式（独立按键、矩阵键盘、ZLG72128 等专用管理芯片），均可使用通用的键盘管理接口实现对按键的管理。

本节将对 AWorks 中的通用数码管接口和按键管理接口作详细介绍，只要使能了 ZLG72128 设备，即可使用这些接口实现数码管的显示和按键的管理。若用户已经使用过 AWorks，并对数码管接口和按键管理接口较为熟悉，则可以跳过本章的后续内容。

5.3.1 通用数码管接口

AWorks 提供了操作数码管的通用接口，详见表 5.4。

表 5.4　通用数码管接口（aw_digitron_disp.h）

函数原型	功能简介
aw_err_t aw_digitron_disp_decode_set (int id, uint16_t (*pfn_decode)(uint16_t ch));	设置段码解码函数
aw_err_t aw_digitron_disp_blink_set (int id, int index, aw_bool_t blink);	设置数码管闪烁
aw_err_t aw_digitron_disp_at (int id, int index, uint16_t seg);	显示指定的段码图形
aw_err_t aw_digitron_disp_char_at (int id, int index, const char ch);	显示字符
aw_err_t aw_digitron_disp_str (int id, int index, int len, const char *p_str);	显示字符串
aw_err_t aw_digitron_disp_clr (int id);	显示清屏
aw_err_t aw_digitron_disp_enable (int id);	使能数码管显示
aw_err_t aw_digitron_disp_disable (int id);	禁能数码管显示

粗略浏览各个接口可以发现，接口以“aw_digitron_disp_”作为命名空间，命名中不再存在“zlg72128”关键字，表明这些接口与 ZLG72128 是无关的，所有接口的第一个参数为 id，id 的作用类似于 handle，用于指定具体使用的数码管显示器。在系统中，每个数码管显示器都有一个唯一的 id 与之对应，例如，若系统中只使用了一个由 ZLG72128 驱动的显示器，则编号默认为 0；若系统中既存在使用 ZLG72128 驱动的数码管显示器，也存在使用 GPIO 驱动的数码管显示器，则它们的编号可能分别为 0 和 1。

ZLG72128 对应的数码管显示器 ID 在 ZLG72128 配置时指定，详见表 5.1 中的配置项 2，其默认值为 0，后续接口举例时，均以 ID 为 0 作为范例。

1. 设置段码解码函数

通过控制数码管各个段的亮灭，可以组合显示出多种图形，例如，对于 8 段数码管，要显示字符“1”，则需要点亮 b、c 两段，对应的编码值（即段码）为 0x60。解码函数用于对特定字符进行解码，以获取对应字符的编码值。根据编码值，可以知道在显示对应字符时，哪些段需要点亮（相应位为 1），哪些段需要熄灭（相应位为 0）。设置段码解码函数即用于用户自定义字符的解码函数，其函数原型为：

```
aw_err_t    aw_digitron_disp_decode_set (int id, uint16_t (*pfn_decode) (uint16_t ch));
```

其中，id 表示设置数码管显示器的编号，通常，若系统只有一个数码管显示器，则 id 为 0。pfn_decode 为函数指针，其指向的函数即为设置的解码函数，解码函数的参数为 uint16_t 类型的字符，返回值为 uint16_t 类型的编码。

绝大部分情况下，对于 8 段数码管，字符'0' ~ '9'等都具有默认编码，为此，AWorks 提供了默认的 8 段数码管解码函数，可以支持常见的字符'0'~'9'以及 'A' 、'B'、 'C' 、'D'、 'E'、 'F'等字符的解码。其在 aw_digitron_disp.h 文件中声明：

```
uint16_t aw_digitron_seg8_ascii_decode (uint16_t ascii_char);
```

如无特殊需求，可以将该函数作为 pfn_decode 的实参传递，范例程序详见程序清单 5.6。

程序清单 5.6　aw_digitron_disp_decode_set()范例程序

```
1    aw_digitron_disp_decode_set(0, aw_digitron_seg8_ascii_decode);
```

若由于应用特殊需求，要求字符使用自定义的特殊编码，例如，要使字符'O'的编码为 0xFC，则可以自定义如下解码函数：

```
1    uint16_t my_decode(uint16_t ch)
2    {
3        //…其它字符的解码
4        if (ch =='O') {
5            return 0xFC;
6        }
7    }
```

然后将该函数作为 pfn_decode 的实参传递即可：

```
aw_digitron_disp_decode_set(0, my_decode);
```

注意，对于一个数码管显示器，只能设置一个解码函数。

2. 设置数码管闪烁

该函数可以指定数码管显示器的某一位数码管闪烁，其函数原型为：

```
aw_err_t    aw_digitron_disp_blink_set (int id, int index, aw_bool_t blink);
```

其中，id 为数码管显示器编号；index 为数码管索引，通常情况下，一个数码管显示器具有多个显示位，索引即用于指定具体操作哪一位数码管，例如，ZLG72128 最高可以驱动 12 位数码管，则该数码管显示器对应的位索引范围为：0～11；blink 表示该位是否闪烁，若其值为 TRUE，则闪烁，反之，则不闪烁，默认情况下，所有数码管均处于未闪烁状态。如设置 1 号数码管闪烁的范例程序详见程序清单 5.7。

程序清单 5.7　aw_digitron_disp_blink_set()范例程序

```
1    aw_digitron_disp_blink_set(0, 1, AW_TRUE);
```

3. 显示指定的段码图形

该函数用于不经过解码函数解码，直接显示段码指定的图形，可以灵活的显示任意特殊图形，其函数原型为：

```
int aw_digitron_disp_at (int id, int index, uint16_t seg);
```

其中，id 为数码管显示器编号；index 为数码管索引；seg 为显示的段码。如在 8 段数码管上显示字符'-'，即需要 g 段点亮，对应的段码为 0x02（即：0000 0010），范例程序详见程序清单 5.8。

程序清单 5.8　aw_digitron_disp_at()范例程序

```
1    aw_digitron_disp_at(0, 1, 0x02);
```

4. 显示单个字符

该函数用于在指定位置显示一个字符，字符经过解码函数解码后显示，若解码函数不支持该字符，则不显示任何内容，其函数原型为：

```
aw_err_t    aw_digitron_disp_char_at (int id, int index, const char ch);
```

其中，id 为数码管显示器编号，index 为数码管索引，ch 为显示的字符。比如，显示字符'H'的范例程序详见程序清单 5.9。

程序清单 5.9　aw_digitron_disp_char_at()范例程序

```
1    aw_digitron_disp_char_at (0, 1, 'H');
```

5. 显示字符串

该函数用于从指定位置开始显示一个字符串，其函数原型为：

```
int aw_digitron_disp_str (int id, int index, int len, const char *p_str);
```

其中，id 为数码管显示器编号，index 为显示字符串的数码管起始索引，即从该索引指定的数码管开始显示字符串，len 指定显示的长度（显示该字符串所使用的数码管位数），p_str 指向需要显示的字符串。

实际显示的长度是 len 和字符串长度的较小值，若数码管位数不够，则多余字符不显示。部分情况下，显示所占用的数码管长度可能与字符串实际显示的长度不等，例如，显示字符串“1.”，其长度为 2，但实际显示时，字符“1”和小数点均可显示在一位数码管上，因此，该显示仅占用一位数码管。

显示字符“HELLO.”的范例程序详见程序清单 5.10。

程序清单 5.10　aw_digitron_disp_str()范例程序

```
1    aw_digitron_disp_str(0, 0, 5,"HELLO.");
```

若只存在两个数码管，因此最终只会显示“HE”。

通常情况下，需要显示一些数字，如显示变量的值，此时，可以先将变量通过格式化字符串函数输出到字符串缓冲区中，然后再使用 aw_digitron_disp_str()函数显示该字符串。比如，显示一个变量 i 的值，范例程序详见程序清单 5.11。

程序清单 5.11　使用 aw_digitron_disp_str()显示整数变量值的范例程序

```
1    int          num = 53;
2    char         buf[3];
3    aw_snprintf(buf, 3, "%2d", num);
4    aw_digitron_disp_str(0, 0, 2, buf);
```

其中，aw_snprintf()与标准 C 函数 snprintf()函数功能相同，均用于格式化字符串到指定的缓冲区中，其函数原型为（aw_vdebug.h）：

```
int aw_snprintf (char *buf, size_t sz, const char *fmt, ...);
```

6. 显示清屏

该函数用于显示清屏，清除数码管显示器中的所有内容，其函数原型为：

```
int aw_digitron_disp_clr (int id);
```

其中，id 为数码管显示器编号，范例程序详见程序清单 5.12。

程序清单 5.12　aw_digitron_disp_clr()范例程序

```
1    aw_digitron_disp_clr(0);
```

7. 使能数码管显示

数码管默认是处于使能状态的，只有当被禁能后，才需要使用该函数重新使能。数码管仅在使能状态下才可以正常显示。

该函数用于显示清屏，清除数码管显示器中的所有内容，其函数原型为：

```
aw_err_t    aw_digitron_disp_enable (int id);
```

其中，id 为数码管显示器编号，范例程序详见程序清单 5.13。

程序清单 5.13　aw_digitron_disp_enable()范例程序

```
1    aw_digitron_disp_enable(0);
```

8. 禁能数码管显示

数码管默认处于使能状态，可以正常显示。清屏状态下只是清空了数码管显示的内容，数码管实际上还是处于工作状态，对于动态扫描类数码管，依然处于动态扫描状态，需要消耗 CPU 资源。若长时间不使用数码管，可以彻底关闭数码管显示器，关闭数码管扫描，节省 CPU 资源，甚至是关闭数码管的电源，降低系统功耗。关闭数码管显示器的函数原型为：

```
aw_err_t    aw_digitron_disp_disable (int id);
```

其中，id 为数码管显示器编号，范例程序详见程序清单 5.14。

程序清单 5.14　aw_digitron_disp_disable()范例程序

```
1    aw_digitron_disp_disable(0);
```

数码管被禁能后，将不能再正常显示，若需正常显示，必须使用 aw_digitron_disp_enable()接口重新使能数码管。

基于数码管通用接口，可以编写一个简易的 60s 倒计时程序，当倒计时还剩 5s 时，数码管闪烁，为便于复用，将应用程序存放到 app_digitron_count_down.c 文件中，并将其接口声明到 app_digitron_count_down.h 文件中，详见程序清单 5.15 和程序清单 5.16。

程序清单 5.15　倒计时应用程序实现（app_digitron_count_down.c）

```
1   #include "aworks.h"
2   #include "aw_digitron_disp.h"
3   #include "aw_vdebug.h"
4   #include "aw_delay.h"
5
6   static void __digitron_show_num (int id, int num)
7   {
8       char buf[3];
9       aw_snprintf(buf, 3, "%2d", num);
10      aw_digitron_disp_str(id, 0, 2, buf);
11  }
12
13  int app_digitron_count_down (int id)                                    // 应用函数入口
14  {
15      unsigned int sec = 60;
16
17      aw_digitron_disp_decode_set(0, aw_digitron_seg8_ascii_decode);      // 使用默认的解码函数
18
19      while (1) {
20          __digitron_show_num(id, sec);                                   // 显示当前秒
21          if (sec > 0) {
22              sec--;
23          } else {
24              sec = 60;
25          }
26          if (sec < 5) {                                                  // 低于 5s，打开闪烁
27              aw_digitron_disp_blink_set(0, 1, AW_TRUE);
28          } else {
29              aw_digitron_disp_blink_set(0, 1, AW_FALSE);
30          }
31          aw_mdelay(1000);                                                // 延时 1s
32      }
33      return AW_OK;
34  }
```

程序清单 5.16　倒计时应用程序接口声明（app_digitron_count_down.h）

```
1   #pragma once
2   #include "aworks.h"
3
```

```
4    int app_digitron_count_down (int id);                    // 应用程序入口
```

要使用此应用程序，只需调用其入口函数 app_digitron_count_down()，并指定一个数码管显示器编号即可。应用程序与具体 MCU、数码管驱动方式无关，可以在任何 AWorks 平台上运行，使用 ZLG72128 启动该应用程序的范例详见程序清单 5.17。

程序清单 5.17　运行倒计时应用程序的主程序范例

```
1    #include "ametal.h"
2    #include "app_digitron_count_down.h"
3
4    int am_main (void)
5    {
6        am_zlg72128_std_inst_init();          // ZLG72128 实例初始化
7        app_digitron_count_down(0);           // 使用显示器编号为 0 的数码管
8        while (1) {
9        }
10   }
```

由此可见，若后续改用其它数码管驱动方式（例如，修改为 GPIO 直接驱动数码管的方式），则仅仅是主程序中的初始化语句发生变化，应用程序本身不需要作任何改变，换句话说，应用程序可以编译成库，独立开发、维护和部署。

5.3.2 通用键盘接口

AWorks 实现了一个输入子系统架构，可以统一管理按键、鼠标、触摸屏等外部输入事件。这里以使用按键为例，讲述输入系统的使用方法。

对于键盘，无论是独立键盘、矩阵键盘还是外接的外围键盘管理芯片（如 ZLG72128），均可以使用输入系统进行管理。

对于用户来讲，要使用按键，即需要对外部输入的按键事件进行处理，为此，需要向系统中注册一个输入事件处理器，该处理器中，包含了用户自定义的事件处理函数，当有按键事件发生时，系统将自动回调事件处理器中的用户函数。

AWorks 提供了注册输入事件处理器的接口，其函数原型为：

```
aw_err_t aw_input_handler_register (
    aw_input_handler_t      *p_input_handler,
    aw_input_cb_t            pfn_cb,
    void                    *p_usr_data);
```

其中，p_input_handler 为指向输入事件处理器的指针，pfn_cb 为指向用户自定义的输入事件处理函数的指针，p_usr_data 为按键处理函数的用户参数。当输入事件发生时，系统会回调 pfn_cb 指向的用户处理函数，并将 p_usr_data 作为参数传递给用户处理函数。

1. 输入事件处理器

p_input_handler 指向输入事件处理器。aw_input_handler_t 为输入事件处理器类型，其在 aw_input.h 文件中定义，用户无需关心该类型的具体定义，仅需使用该类型定义输入事件处理器的实例即可，比如：

```
aw_input_handler_t      key_handler;          // 定义一个输入事件处理器实例，用于处理按键事件
```

其中，key_handler 为用户自定义的输入事件处理器，其地址可以作为 p_input_handler

的实参传递。

2. 用户自定义事件处理函数

pfn_cb 指向用户自定义的输入事件处理函数，其类型 aw_input_cb_t 为事件处理函数的类型，其在 aw_input.h 文件中定义如下：

```
typedef void (*aw_input_cb_t)( aw_input_event_t *p_input_data, void *p_usr_data);
```

当输入事件发生时，无论是按键事件，还是其它坐标事件，比如：鼠标、触摸屏等。均会调用 pfn_cb 指针指向的函数，当该函数被调用时，p_input_data 为输入事件相关的数据，包含事件类型（区分按键事件或坐标事件，比如：鼠标、触摸屏等）、按键编码、坐标等信息，用户可以根据这些数据作出相应的处理动作。p_usr_data 为用户自定义的参数，其值与注册事件处理器时传递的 p_usr_data 参数一致，若不使用该参数，则可以在注册事件处理器时，将 p_usr_data 参数的值设置为 NULL。

p_input_data 的类型为 aw_input_event_t 指针类型，aw_input_event_t 类型在 aw_input.h 文件中定义如下：

```
1    typedef struct aw_input_event {
2        int ev_type;                                    // 事件类型码
3    } aw_input_event_t;
```

其本质上是一个结构体类型，仅包含一个数据成员，用于表示事件的类型，若为按键事件，则该值为 AW_INPUT_EV_KEY；若为绝对事件（比如触摸屏上的触摸事件），则该值为 AW_INPUT_EV_ABS。

若 p_input_data 指向的数据中，ev_type 的值为 AW_INPUT_EV_KEY，则表示其指向的数据本质上是一个完整的按键事件数据，其类型为 aw_input_key_data_t，该类型在 aw_input.h 文件中定义如下：

```
1    typedef struct aw_input_key_data {
2        aw_input_event_t    input_ev;     // 事件类型
3        int                 key_code;     // 按键编码
4        int                 key_state;    // 按键状态
5        int                 keep_time;    // 按键保持时间，用于按键长按，时间单位为：ms
6    } aw_input_key_data_t;
```

该类型的第一个数据成员为 input_ev，其中包含了事件的具体类型。也正因为其第一个数据成员的类型为 aw_input_event_t，系统才可以在回调用户自定义的函数时，将 aw_input_key_data_t 类型的指针转换为指向 aw_input_event_t 类型的指针使用。

key_code 为按键编码，用于区分系统中多个不同的按键。ZLG72128 最多支持 32 个按键， 为每个按键分配一个唯一的编码，当按键事件发生时，用户可以据此判断是哪个按键产生了按键事件，各按键对应的编码在配置时指定（详见表 5.1 中的配置项 1），默认配置中，指定了 4 个按键的编码分别为：KEY_0、KEY_1、KEY_2、KEY_3。这些编码都是在 aw_input_code.h 文件中使用宏的形式定义的。

key_state 表示本次按键事件具体对应的按键状态，用以区分按键事件是按下事件还是释放事件。若该值不为 0，则表示按键按下；否则，表示按键释放。

keep_time 表示状态保持时间（单位：ms），常用于按键长按应用（例如，按键长按 3 秒关机）。按键首次按下时，keep_time 为 0，若按键一直保持按下，则系统会以一定的时间间隔上报按键按下事件（调用用户回调），keep_time 的值不断增加，表示按键按下已经保持的时间。特别地，若按键不支持长按功能，则 keep_time 始终为-1。

基于此，为了获取到更多的按键相关信息，比如：按键编码、按键状态（按下还是释放）等。可以将 p_input_data 强制转换为 aw_input_key_data_t 指针类型使用，详见程序清单 5.18。

程序清单 5.18　根据输入事件的类型使用数据

```
1   void key_process (aw_input_event_t *p_input_data, void *p_usr_data)
2   {
3       if (p_input_data->ev_type == AW_INPUT_EV_KEY) {                    // 处理按键事件
4           // 将数据转换为按键数据类型
5           aw_input_key_data_t *p_data = (aw_input_key_data_t *)p_input_data;
6           // 按键事件的处理代码
7           // 使用 key_code   : p_data->key_code
8           // 使用 key_state  : p_data->key_state
9           // 使用 keep_time  : p_data->keep_time
10      }
11  }
```

实际上，不同类型的输入事件，其需要包含的数据是不同的，例如，对于触摸屏事件，则需要包含横坐标和纵坐标。为了统一各种不同类型的事件处理函数类型，将 aw_input_event_t 类型的数据作为所有事件实际数据类型的第一个成员。这样，可以统一使用 aw_input_event_t 类型的指针指向实际的数据，以此统一事件处理函数的类型，用户在事件处理函数中，通过查看事件类型，即可进一步将该指针强制转换为指向实际数据类型的指针，使用其中更多的信息。如图 5.1 所示的类图表示了这种关系，实际数据类型均是从基类派生而来的，aw_input_prt_data_t 是指针型输入事件，比如触摸屏触摸事件等，其包含了具体坐标信息。

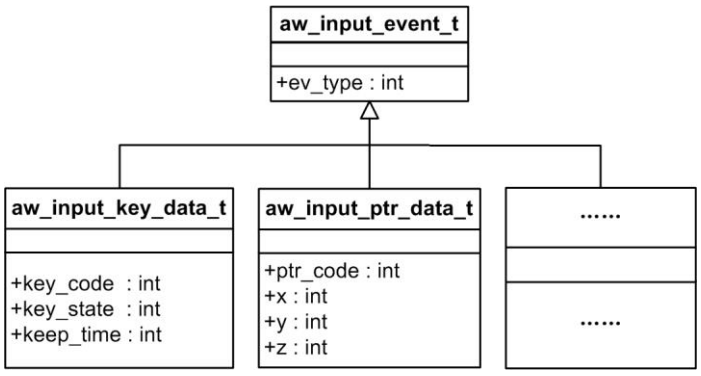


图 5.1　各种类型的输入事件对应的实际数据类图

例如，使用一个按键（按键编码为 KEY_0）控制 LED0，当按键按下时，则 LED0 点亮；当按键释放后，则 LED0 熄灭，相应的按键处理函数详见程序清单 5.19。

程序清单 5.19　按键处理函数范例程序（1）

```
1   void key_process (aw_input_event_t *p_input_data, void *p_usr_data)
2   {
3       if (p_input_data->ev_type == AW_INPUT_EV_KEY) {    // 处理按键事件
4           // 将数据转换为按键数据类型
5           aw_input_key_data_t *p_data = (aw_input_key_data_t *)p_input_data;
6           if (p_data->key_code == KEY_0) {                    // 编码为 KEY_0 的按键事件
7               if (p_data->key_state != 0) {                   // 值不为 0，按键按下
8                   aw_led_on(0);                               // 按键按下，点亮 LED0
9               } else {                                        // 值为 0，按键释放
10                  aw_led_off(0);                              // 按键释放，熄灭 LED0
```

```
11                  }
12              }
13          }
14      }
```

也可使用按键长按功能模拟开关机，例如：按键长按 3s，LED0 状态翻转，模拟切换“开关机”状态，LED0 亮表示“开机“；LED0 熄灭表示“关机”，按键处理范例程序详见程序清单 5.20。

程序清单 5.20　按键处理函数范例程序（2）

```
1   void key_process (aw_input_event_t *p_input_data, void *p_usr_data)
2   {
3       if (p_input_data->ev_type == AW_INPUT_EV_KEY) {    // 处理按键事件
4           // 将数据转换为按键数据类型
5           aw_input_key_data_t *p_data = (aw_input_key_data_t *)p_input_data;
6           if (p_data->key_code == KEY_0) {                    // 编码为 KEY_0 的按键事件
7               if ((p_data->key_state != 0) && (p_data->keep_time == 3000)) { // 长按时间达到 3s
8                   aw_led_toggle(0);                                          // LED0 状态翻转
9               }
10          }
11      }
12  }
```

完成按键处理函数的定义后，函数名可作为参数传递给 aw_input_handler_register()函数的 pfn_cb 形参。综合范例程序详见程序清单 5.21。

程序清单 5.21　注册事件处理器范例程序

```
1   #include "aworks.h"
2   #include "aw_led.h"
3   #include "aw_delay.h"
4   #include "aw_input.h"
5
6   static void __key_process (aw_input_event_t *p_input_data, void *p_usr_data)
7   {
8       if (p_input_data->ev_type == AW_INPUT_EV_KEY) {    // 处理按键事件
9           // 将数据转换为按键数据类型
10          aw_input_key_data_t *p_data = (aw_input_key_data_t *)p_input_data;
11          if (p_data->key_code == KEY_0) {            // 编码为 KEY_0 的按键事件
12              if (p_data->key_state != 0) {           // 值不为 0，按键按下
13                  aw_led_on(0);                       // 按键按下，点亮 LED0
14              } else {                                // 值为 0，按键释放
15                  aw_led_off(0);                      // 按键释放，熄灭 LED0
16              }
17          }
18      }
19  }
```

```
20
21  int aw_main()
22  {
23      static aw_input_handler_t    key_handler;
24      aw_input_handler_register(&key_handler, __key_process, NULL);
25      while(1) {
26          aw_mdelay(1000);
27      }
28  }
```

注册按键处理器后，当按键按下或释放时，均会调用注册按键处理器时指定的回调函数，即程序清单 5.21 中的__key_process()函数。

若系统中存在多个按键，且各个按键的处理毫不相关，为了分离各个按键的处理代码，可以注册多个按键事件处理器，每个处理器处理一个按键事件，范例程序详见程序清单 5.22。

程序清单 5.22　注册多个按键处理器范例程序

```
1   #include "aworks.h"
2   #include "aw_input.h"
3
4   static void __key0_process(aw_input_event_t *p_input_data, void *p_usr_data)
5   {
6       if (p_input_data->ev_type == AW_INPUT_EV_KEY) {
7           aw_input_key_data_t *p_data = (aw_input_key_data *)p_input_data;
8           if (p_data->key_code == KEY_0) {        // 编码为 KEY_0 的按键事件
9               // 处理按键 0
10          }
11      }
12  }
13
14  static void __key1_process(aw_input_event *p_input_data, void *p_usr_data)
15  {
16      if (p_input_data->ev_type == AW_INPUT_EV_KEY) {
17          aw_input_key_data_t *p_data = (aw_input_key_data *)p_input_data;
18          if (p_data->key_code == KEY_1) {        // 编码为 KEY_1 的按键事件
19              // 处理按键 1
20          }
21      }
22  }
23
24  static void __key2_process(aw_input_event_t *p_input_data, void *p_usr_data)
25  {
26      if (p_input_data->ev_type == AW_INPUT_EV_KEY) {
27          aw_input_key_data_t *p_data = (aw_input_key_data_t *)p_input_data;
28          if (p_data->key_code == KEY_2) {        // 编码为 KEY_2 的按键事件
29              // 处理按键 2
```

```
30              }
31          }
32      }
33
34      static void __key3_process(aw_input_event_t *p_input_data, void *p_usr_data)
35      {
36          if (p_input_data->ev_type == AW_INPUT_EV_KEY) {
37              aw_input_key_data_t *p_data = (aw_input_key_data_t *)p_input_data;
38              if (p_data->key_code == KEY_3) {          // 编码为 KEY_3 的按键事件
39                  // 处理按键 3
40              }
41          }
42      }
43
44      static aw_input_handler_t     g_key0_handler;
45      static aw_input_handler_t     g_key1_handler;
46      static aw_input_handler_t     g_key2_handler;
47      static aw_input_handler_t     g_key3_handler;
48
49      int aw_main (void)
50      {
51          aw_input_handler_register(&g_key0_handler, __input_key0_proc, NULL);
52          aw_input_handler_register(&g_key1_handler, __input_key1_proc, NULL);
53          aw_input_handler_register(&g_key2_handler, __input_key2_proc, NULL);
54          aw_input_handler_register(&g_key3_handler, __input_key3_proc, NULL);
55          while (1){
56              aw_mdelay(1000);
57          }
58      }
```

第6章 在 Linux 中使用 ZLG72128

✍本章导读

为便于在 Linux 中使用 ZLG72128，ZLG 针对 ZLG72128 的按键和数码管/LED 功能，分别提供了相应的 Linux 驱动支持。按键驱动基于标准的 input 子系统，LED 控制部分则采用 Linux 传统的字符方式实现。在应用层，使用 ZLG72128 的按键，采用标准 input 子系统编程即可；使用 ZLG72128 的数码管/LED，则需要通过 ioctl 系统调用来完成。

由于 Linux 内核变化，从 3.14 版本后引入了设备树，在驱动实现上有一些变化，本章将对有、无设备树的两种情况分别讲述。

6.1 驱动和源码编译说明

ZLG72128 的 Linux 驱动（含范例）源码可以在致远电子官网（http://www.zlgmcu.com/）下载，源码目录和相关文件说明详见表 6.1。

表 6.1 ZLG72128 驱动源码文件说明

目录	所含文件	说明
app/	zlg72128_lib.c、zlg72128.h、zlg72128_key_test.c、zlg72128_led_test.c、Makefile	范例程序
dev/	zlg72128_device.c、Makefile	驱动的 device 部分源码
drv/	zlg72128_driver.c、zlg72128.h、Makefile	驱动的 driver 部分源码

编译源码前，需要修改 dev/和 drv/目录下的 Makefile 文件，设置正确的内核源码路径以及交叉编译器，它们的默认值为：

```
LINUX_KERNEL_PATH=/home/zwj/work/kernel/linux-3.14/
CROSS_COMPILE=/home/zwj/ctools/arm-2014.05/bin/arm-none-linux-gnueabi-
```

编译完成后，将会得到 zlg72128_device.ko 和 zlg72128_driver.ko 两个驱动文件。

ZLG72128 能管理多达 32 只按键（24 只普通按键和 8 只功能按键），在实际应用中，对这些按键的键值定义可能都会有不同，为了适应这样的柔性需求，在驱动的具体实现中，将器件驱动分为 device 和 driver 两部分，将按键赋值放在设备部分实现，所有器件操作则放在驱动中完成。

在 device 实现部分，完成 platform_device 的注册，并提供与 driver 进行匹配所需要的 devname。在 driver 部分实现与 device 的匹配以及全部 ZLG72128 硬件操作，包括按键驱动实现、数码管驱动实现。另外，ZLG72128 的 I^2C 总线序号、中断以及 IO 复位，这些硬件资源在不同平台上都有可能会变化，在具体实现中，也提供了对这些可变因素的柔性支持。

本驱动实现了有、无设备树两种内核的兼容支持，在使用上基本一致，但有些细微区别，下面分别进行描述。

6.2 ZLG72128 Linux 驱动使用（无设备树）

6.2.1 模块加载

1. 驱动加载

加载驱动 zlg72128_driver.ko 时可传入 4 个参数，分别是 i2cline、sla、irq 和 rst，分别用于指定 I²C 总线、I²C 从机地址、IRQ 号以及复位引脚 IO 序号，各参数说明详见表 6.2。

表 6.2 zlg72128_driver 驱动参数说明

属性名	说明	默认值
i2cline	i2c 总线	2
sla	从机地址	0x30
irq	中断 gpio 号	62
rst	复位 gpio 号	63

以上参数可在加载驱动时根据具体硬件指定，或者修改源码。

假定在在一个主板上，外接了 1 片 ZLG72128 芯片，从机地址为 0x30，接在 I²C2 上，中断和复位分别连接到处理器的 GPIO1_30 和 GPIO1_31 上，则模块加载参数为：

```
# insmod zlg72128_driver.ko i2cline=3 sla=0x30   irq=62 rst=63
```

注：在 Linux 系统中，GPIO 序号 N 和引脚 GPIOx_y 的换算规则为：N=x*32+y。GPIO1_31 的序号 N=1*32+31=63,同理,GPIO1_30 的序号为 62。最新版 ZLG72128 允许 1 路 I2C 总线接最多两片 ZLG72128，通过不同从机地址进行区分。

2. 设备加载

device 驱动加载无需参数，直接用 insmod 命令加载即可，命令如下：

```
# insmod zlg72128_device.ko
```

6.2.2 自定义按键键值

驱动 driver 部分为所有按键定义了默认键值，详见程序清单 6.1。

程序清单 6.1 默认键值列表（zlg72128_driver.c）

```
static int key_list_def[ZLG72128_MAX_KEY_COUNT] = {
    KEY_1,   KEY_2,   KEY_3,   KEY_4,   KEY_5,   KEY_6,   KEY_7,   KEY_8,
    KEY_9,   KEY_A,   KEY_B,   KEY_C,   KEY_D,   KEY_E,   KEY_F,   KEY_G,
    KEY_H,   KEY_I,   KEY_J,   KEY_K,   KEY_L,   KEY_M,   KEY_N,   KEY_O,
    KEY_F1,  KEY_F2,  KEY_F3,  KEY_F4,  KEY_F5,  KEY_F6,  KEY_F7,  KEY_F8,
};
```

允许用户对键值进行修改，在驱动的 device 部分，提供了自定义键值列表，详见程序清单 6.2。

程序清单 6.2 自定义键值列表（zlg72128_device.c）

```
static int key_list[] = {
    KEY_1,   KEY_2,   KEY_3,   KEY_4,   KEY_5,   KEY_6,   KEY_7,   KEY_8,
    KEY_9,   KEY_A,   KEY_B,   KEY_C,   KEY_D,   KEY_E,   KEY_F,   KEY_G,
    KEY_H,   KEY_I,   KEY_J,   KEY_K,   KEY_L,   KEY_M,   KEY_N,   KEY_O,
```

```
    KEY_F1,  KEY_F2,  KEY_F3,  KEY_F4,  KEY_F5,  KEY_F6,  KEY_F7,  KEY_F8,
};
```

由此可见，默认键值和自定义键值列表是完全相同的。如果需要对键值进行修改，建议修改 device 部分的 key_list，而不是 driver 部分的 key_list_def。若修改了按键键值，则需要重新编译驱动使新的键值生效。

6.2.3 I/O 修改

zlg72128_dev_data 结构体描述了 ZLG72128 需要的 IO 资源，如程序清单 6.3 所示。

程序清单 6.3　struct　zlg72128_dev_data 类型定义（zlg72128_driver.c）

```
struct zlg72128_dev_data {
    int i2cline;        //    i2c 总线
    int sla;            //    i2c 设备地址
    int irq;            //    中断引脚
    int rst;            //    复位引脚
    int *key_list;      //    键盘值映射，第 0~23 个元素分别代表 K1~K23 普通按键键值，
                        //    第 24~31 个元素分别代表 F1~F8 特殊按键键值，0 为无效键值
};
```

各参数的含义见代码注释。在实际使用中，除键值列表外的 I/O 资源，可以通过加载参数指定，也可在源码中进行修改，例如：

```
static struct zlg72128_dev_data zlg72128_data = {
    .i2cline    = 2,
    .sla        = 0x30,
    .irq        = 62,
    .rst        = 63,
    .key_list   = key_list,
};
```

6.3　ZLG72128 Linux 驱动使用（设备树）

驱动源码针对高版本的 Linux 内核（Linux3.14 及以上）做了兼容性处理，模块的基本使用方法相同，但在修改参数方面有一些差异。

6.2.1 模块加载

驱动文件的加载方法与无设备树时的加载方法完全相同。

1．　驱动加载

加载 driver 驱动文件，也可以通过参数指定 I/O：

```
# insmod zlg72128_driver.ko i2cline=3 sla=0x30    irq=62 rst=63
```

2．　设备加载

直接用 insmod 命令加载即可，命令如下：

```
# insmod zlg72128_device.ko
```

6.3.2 设备树和 I/O 资源

基于设备树的内核，需要在对应设备树中增加设备子节点，在设备树中描述 ZLG72128

所需要的资源，包括 I/O 和键值，详见程序清单 6.4。

程序清单 6.4　zlg72128 设备树子节点

```
/ {
        ......
        zlg72128_0: zlg72128_0@30 {
                compatible = "zlg72128";
                i2cline = <2>;
                sla = <0x30>;
                irq = <62>;
                rst = <63>;
                keys_list = <
                    1   2   3   4   5   6   7   8
                    9   30  48  46  32  18  33  34
                    35  23  36  37  38  50  49  24
                    59  60  61  62  63  64  65  66
                  >;
        };
};
```

其中，设备树中的 zlg72128_0 部分为 zlg72128 的设备子节点，其相关的属性描述详见表 6.3。

表 6.3　zlg72128 设备树节点属性

序号	属性名	说明	默认值
1	compatible	驱动匹配名	必须为“zlg72128”
2	i2cline	i2c 总线	2
3	sla	从机地址	0x30
4	irq	中断 gpio 号	62
5	rst	复位 gpio 号	63
5	key_list	键盘值映射	-

其中 key_list 的键值映射顺序与程序清单 6.3 中 struct zlg72128_dev_data 的 key_list 成员的键值设置顺序一致。

此外，由于 ZLG72128 使用到 i2c 总线，所以必须保证设备树中对应 i2c 总线的 status 值为“okey”才能正常使用，详见程序清单 6.5。

程序清单 6.5　i2c 设备数节点设置

```
&i2c2 {
        pinctrl-names = "default";
        pinctrl-0 = <&i2c2_pins_default>;
        status = "okey";                                    // status  必须为  okey
        clock-frequency = <400000>;
};
```

- pinctrl-0：i2c 总线及相关引脚设置

- status：总线状态（okey 为使用，disabled 为不使用）
- clock-frequency：总线速率

对于 I²C 总线的设备树节点，在不同平台的内核上的配置方法会有差异，需要根据当前平台内核进行调整。

注意，设备树修改后需要重新进行交叉编译。

6.4 按键使用和编程参考

6.4.1 系统配置和使用

使用 ZLG72128 键盘的按键功能很简单，只要在系统环境变量中设置好输入设备的环境变量即可。假如，ZLG72128 键盘在某个系统中的输入设备节点为/dev/input/event1，下面分别讲述在命令行终端的使用和 Qt 中的配置。

在命令行终端，获取按键键值可用 hexdump 命令。输入 hexdump /dev/input/event1，然后按键盘的回车键，可以看到下面的信息：

```
# hexdump    /dev/input/event1
0000000 05dc 0000 3248 0009 0004 0004 001c 0007
0000010 05dc 0000 3248 0009 0001 001c 0001 0000
0000020 05dc 0000 3248 0009 0000 0000 0000 0000
0000030 05dc 0000 8a18 000a 0004 0004 001c 0007
0000040 05dc 0000 8a18 000a 0001 001c 0000 0000
```

在 Linux 系统中，回车键的键值为 28（0x1C）。

如果在 Qt 程序中使用 ZLG72128 键盘，只需在 Qt 的启动脚本中指定环境变量即可。程序清单 6.6 是 Qt 启动脚本中的一部分，其中 QWS_KEYBOARD 就是用于设置键盘设备的环境变量。

程序清单 6.6　Qt 中设置输入设备环境变量

```
devs_list=`ls /dev/input/event*`
has_ts=0
QWS_MOUSE_PROTO=""
QWS_KEYBOARD=""
for dev in $devs_list
do
    ./input_type "$dev" 2>/dev/null
    case $? in
    1)  QWS_MOUSE_PROTO="$QWS_MOUSE_PROTO ""Tslib:$dev"
        has_ts=1
        ;;
    2)  QWS_MOUSE_PROTO="$QWS_MOUSE_PROTO ""LinuxInput:$dev"
        ;;
    3)  QWS_KEYBOARD="$QWS_KEYBOARD ""LinuxInput:$dev"
        ;;
    *)  ;;
    esac
done
```

正确设置了键盘输入环境变量后，在 Qt 界面即可按标准键盘一样使用 ZLG72128 键盘。

6.4.2 C 编程范例

确定了 ZLG72128 的设备节点后，在 C 程序中获取按键键值，采用标准 input 子系统编程即可，程序清单 6.7 是一个通用的键盘按键测试范例。

程序清单 6.7　Linux 下键盘测试通用范例

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <fcntl.h>
4   #include <unistd.h>
5   #include <signal.h>
6   #include <string.h>
7   #include <poll.h>
8   #include <linux/input.h>
9
10  #define INPUT_DEV       "/dev/input/event1"                 /* 键盘对应的设备节点 */
11
12  int main(int argc,char *argv[])
13  {
14      unsigned char key_value;
15      int fd, ret;
16      struct pollfd fds;
17      struct input_event key;
18
19      if(argc == 2)
20          fd = open(argv[1], O_RDONLY);
21      else
22          fd = open(INPUT_DEV, O_RDONLY);
23      if(fd < 0) {
24          perror("open dev");
25          return 1;
26      }
27
28      fds.fd = fd;
29      fds.events = POLLIN;
30
31      while(1) {
32          ret = poll(&fds, 1, -1);
33          if(ret < 0)
34              return -1;
35
36          if(fds.revents == POLLIN){
37              read(fd, &key, sizeof(key));
38              if(key.type == EV_KEY){
```

```
39                              if(key.value == 0 || key.value == 1){
40                                      printf("key %d %s\n", key.code, (key.value)?"Press":"Release");
41                              }
42                      }
43              }
44      }
45      close(fd);
46      return 0;
47  }
```

编译并运行该程序，将在终端打印被按键的键值，松开按键也会打印提示信息。

6.5 数码管接口描述和编程

ZLG72128 的数码管控制功能在驱动中实现为普通字符型驱动，在 Linux 系统下的设备节点为“/dev/zlg72128-n”，其中 n 为 0~7（对应的硬件与内核中的设备注册先后顺序一致），通过 ioctl 方法完成具体功能，并设计了一组对应的命令，提供给应用程序调用。

在操作数码管之前，需要先调用 open()函数打开数码管设备，详见程序清单 6.8。

程序清单 6.8 打开数码管设备代码片段

```
1   #define LED_DEV       "/dev/zlg72128-0"
2   int fd;
3   //      ......
4   fd = open(LED_DEV, O_RDONLY);
5   if(fd < 0)   {
6       perror("open dev");
7       return 1;
8   }
```

6.5.1 命令及操作函数汇总

驱动实现了数码管的基本和高级功能，包括数码管闪烁、数码管移位、特定字符显示、数据段显示、数码管数据读取等。由于大部分命令需要传入的控制信息较多，所以在驱动中设计了一个与命令匹配的结构体，通过相应的结构体传入相关参数，各 ioctl 命令以及使用到的结构体关系详见表 6.4。

表 6.4 ioctl 命令汇总

ioctl 命令 （ZLG72128_ DIGITRON_*）	结构体	命令说明
DISP_CTRL	struct zlg72128_digitron_disp_ctrl_t	数码管有效显示扫描
DISP_CHAR	struct zlg72128_digitron_disp_char_t	让数码管显示特定字符
DISP_STR	struct zlg72128_digitron_disp_str_t	让数码管显示字符串
DISP_NUM	struct zlg72128_digitron_disp_num_t	让数码管显示数字
DISPBUF_SET	struct zlg72128_digitron_dispbuf_set_t	控制多个数码管的位段显示
SEG_CTRL	struct zlg72128_digitron_seg_ctrl_t	控制单个数码管的位段显示
FLASH_CTRL	struct zlg72128_digitron_flash_ctrl_t	数码管有效闪烁控制

ioctl 命令 （ZLG72128_ DIGITRON_*）	结构体	命令说明
FLASH_TIME_CFG	struct zlg72128_digitron_flash_time_cfg_t	控制数码管的闪烁延时功能
SHIFT	struct zlg72128_digitron_shift_t	控制数码管的移位功能
DISP_RESET	-	数码管复位命令
DISP_TEST	-	独立的数码管测试
RESET	-	zlg72128 硬件复位

注：表中的“-”表示该命令没有额外参数，不会使用到任何控制结构体。所有 ioctl 命令的前缀均为“ZLG72128_ DIGITRON_”，因此表中省略了前缀的显示，实际使用时，应加上前缀，例如，显示字符串命令宏为：ZLG72128_ DIGITRON_DISP_STR。

表 6.4 中的命令较多，对应的结构体类型也很多，使用起来较为不便，因此，提供了名为 zlg72128_lib.c 的驱动文件，其主要作用是将表 6.4 中的各条命令封装成与之对应的 API，使用户仅需调用 API 即可完成相应的操作，无需关心具体的命令以及对应的命令参数结构体类型。各个接口的原型详见表 6.5，下节将详细介绍各个接口的实现及使用方法，使读者进一步了解 ioctl 命令的用法及功能。

表 6.5　数码管操作函数

函数原型	功能简介
zlg72128_handle_t zlg72128_init(char *dev_path);	打开设备
int zlg72128_digitron_disp_ctrl (zlg72128_handle_t handle, int ctrl_val);	显示属性（开或关）
int zlg72128_digitron_disp_char (zlg72128_handle_t handle, unsigned char pos, char ch, unsigned char is_dp_disp, unsigned char is_flash);	设置数码管闪烁
int zlg72128_digitron_disp_str (zlg72128_handle_t handle, unsigned char start_pos, const char *p_str);	显示指定的段码图形
int zlg72128_digitron_disp_num (zlg72128_handle_t handle， unsigned char pos， unsigned char num， unsigned char is_dp_disp， unsigned char is_flash);	显示 0 ~ 9 的数字
int zlg72128_digitron_dispbuf_set (zlg72128_handle_t handle， unsigned char start_pos， unsigned char *p_buf，	直接设置显示段码

函数原型	功能简介
unsigned char num);	
int zlg72128_digitron_seg_ctrl (zlg72128_handle_t handle, unsigned char pos, char seg, unsigned char is_on);	直接控制段的点亮或熄灭

续上表

函数原型	功能简介
int zlg72128_digitron_flash_ctrl (zlg72128_handle_t handle, int ctrl_val);	闪烁控制
int zlg72128_digitron_flash_time_cfg (zlg72128_handle_t handle, unsigned char on_ms, unsigned char off_ms);	闪烁持续时间
int zlg72128_digitron_shift (zlg72128_handle_t handle, unsigned char dir, unsigned char is_cyclic, unsigned char num);	显示移位
int zlg72128_digitron_disp_reset (zlg72128_handle_t handle);	复位显示
int zlg72128_digitron_disp_test (zlg72128_handle_t handle);	测试命令
int zlg72128_digitron_reset (zlg72128_handle_t handle);	硬件复位

6.5.2 Linux 通用数码管接口函数详解

1. 打开设备

由表 6.5 中各个接口的原型可知，除打开设备接口外，其它功能接口均需传入一个 handle 参数，用以指定操作的 ZLG72128 对象。handle 可以直接通过 zlg72128_init()函数获得，其函数原型为（zlg72128.h）：

```
zlg72128_handle_t    zlg72128_init (char *dev_path);
```

函数功能的实现详见程序清单 6.9。

程序清单 6.9　zlg72128_init()函数实现

```
1    zlg72128_handle_t zlg72128_init (char *dev_path)
2    {
3        zlg72128_handle_t fd;
4        fd = open(dev_path,O_RDONLY);
5        if(fd < 0){
6            return ZLG72128_RETURN_ERROR;
7        }
8        return fd;
9    }
```

实际上，zlg72128_init()函数只是封装了 6.5 节中的 open 操作，在获取时，需要传入一个设备节点路径，用以指定获取哪个 ZLG72128 对应的 handle。节点路径的形式为：“/dev/zlg72128-n”，其中 n 为 0~7，其顺序与内核设备注册顺序一致。一般情况下，只连接单个 ZLG72128 时，设备节点路径为“/dev/zlg72128-0”。

基于此，获取 handle 的语句即为：

```
zlg72128_handle_t    handle = zlg72128_init("/dev/zlg72128-0");
```

关于返回值 handle 的参数类型 zlg72128_handle_t，在 Linux 中实际上是一个 int 型的文件描述符，其定义如下所示（zlg72128.h）：

```
typedef int zlg72128_handle_t;
```

在使用通用函数接口操作 zlg72128 时，其返回值分别使用 ZLG72128_RETURN_OK、ZLG72128_RETURN_ERROR、ZLG72128_RETURN_PARAMETER_ERROR 代表函数执行成功、执行失败、参数错误三种意义。

同时，驱动中也定义了 ZLG72128_TRUE（真）和 ZLG72128_FALSE（假），用于在程序中为二值参数赋值（布尔型），例如：小数点是否显示；数码管是否闪烁等。

2. 数码管显示属性设置

ZLG72128_DIGITRON_DISP_CTRL 命令用于设置数码管的显示属性（开或关），用到的结构体为 struct zlg72128_digitron_disp_ctrl_t，其定义详见程序清单 6.10，各成员说明详见表 6.6。

程序清单 6.10　struct zlg72128_digitron_disp_ctrl_t

```
1    struct zlg72128_digitron_disp_ctrl_t{
2            int ctrl_val;
3    };
```

表 6.6　struct zlg72128_digitron_disp_ctrl_t 成员说明

结构体	成员	说明
struct zlg72128_digitron_disp_ctrl_t	ctrl_val	bit0~bit11 为有效位，分别对应数码管 0~11，位值为 0 时打开显示，位值为 1 时关闭显示

命令封装后的函数为 zlg72128_digitron_disp_ctrl()，其具体实现详见程序清单 6.11。

程序清单 6.11　zlg72128_digitron_disp_ctrl 函数

```
1    int zlg72128_digitron_disp_ctrl (zlg72128_handle_t    handle, int ctrl_val)
2    {
3            struct zlg72128_digitron_disp_ctrl_t disp_ctrl_t;
4            memset(&disp_ctrl_t,0,sizeof(struct zlg72128_digitron_disp_ctrl_t));
5
6           disp_ctrl_t.ctrl_val = ctrl_val;
7            return ioctl(handle,ZLG72128_DIGITRON_DISP_CTRL,&disp_ctrl_t);
8    }
```

例如，将编号为 4~7 的数码管熄灭，其余 8 个数码管点亮，则使用 ioctl 的命令代码为：

```
struct zlg72128_digitron_disp_ctrl_t disp_ctrl_t;
disp_ctrl_t.ctrl_val      = 0x0f0;
ioctl(fd, ZLG72128_ DIGITRON_DISP_CTRL, &disp_ctrl_t);
```

也可以直接使用 zlg72128_digitron_disp_ctrl()函数完成同样的操作，即：

```
zlg72128_digitron_disp_ctrl(handle, 0x0f0);
```

3. 数码管显示特定字符

ZLG72128_DIGITRON_DISP_CHAR 命令用于在指定位置显示字符，用到的结构体为

struct zlg72128_digitron_disp_char_t，其定义详见程序清单 6.12，各成员说明详见表 6.7。

程序清单 6.12　struct zlg72128_digitron_disp_char_t

```
1    struct zlg72128_digitron_disp_char_t{
2        unsigned char pos;
3        char ch;
4        unsigned char is_dp_disp;
5        unsigned char is_flash;
6    };
```

表 6.7　struct zlg72128_digitron_disp_char_t 成员说明

结构体	成员	说明
struct zlg72128_digitron_disp_char_t	pos	字符显示位置(0~11)
	ch	显示的字符(空格为不显示)
	is_dp_disp	是否显示小数点(1 显示, 0 不显示)
	is_flash	是否闪烁(1 闪烁, 0 不闪烁)

其中，成员 ch 代表的显示的字符必须是 ZLG72128 已经支持的可以自动完成译码的字符，包括字符'0'~'9'与 AbCdEFGHiJLopqrtUychT(区分大小写)。注意，若要显示数字 1，则 ch 参数应为字符'1'，而不是数字 1。

命令封装后的函数为 zlg72128_digitron_disp_char()，其具体实现详见程序清单 6.13。

程序清单 6.13　zlg72128_digitron_disp_char()函数实现

```
1    int zlg72128_digitron_disp_char (zlg72128_handle_t    handle,
2                                     unsigned char         pos,
3                                     char                  ch,
4                                     unsigned char         is_dp_disp,
5                                     unsigned char         is_flash)
6    {
7        struct zlg72128_digitron_disp_char_t disp_char_t;
8        memset(&disp_char_t,0,sizeof(struct zlg72128_digitron_disp_char_t));
9
10       disp_char_t.pos = pos;
11       disp_char_t.ch = ch;
12       disp_char_t.is_dp_disp = is_dp_disp;
13       disp_char_t.is_flash = is_flash;
14
15       return ioctl(handle,ZLG72128_DIGITRON_DISP_CHAR,&disp_char_t);
16   }
```

例如，需要在编号为 1 的数码管上显示字符'0'，不显示小数点，不闪烁，则使用 ioctl 的命令代码为：

```
struct zlg72128_digitron_disp_char_t disp_char_t;
disp_char_t.pos = 1;
disp_char_t.ch = '0';
```

```
disp_char_t.is_dp_disp = 0;
disp_char_t.is_flash = 0;
ioctl(fd,ZLG72128_DIGITRON_DISP_CHAR,&disp_char_t);
```

也可以直接使用 zlg72128_digitron_disp_char()函数完成同样的操作，即：

```
zlg72128_digitron_disp_char (handle, 1, '0', 0, 0);
```

4. 数码管显示字符串

ZLG72128_DIGITRON_DISP_STR 命令用于设置数码管在特定位置开始显示特定字符串口，用到的结构体为 struct zlg72128_digitron_disp_str_t，其定义详见程序清单 6.14，各成员说明详见表 6.8。

程序清单 6.14　struct zlg72128_digitron_disp_str_t

```
1    struct zlg72128_digitron_disp_str_t{
2        unsigned char start_pos;
3        unsigned char p_str[12];
4    };
```

表 6.8　struct zlg72128_digitron_disp_str_t 成员说明

结构体	成员	说明
struct zlg72128_digitron_disp_str_t	start_pos	数码管起始显示位置
	p_str	需要在数码管上显示的字符串

字符串显示遇到字符结束标志"\0"将自动结束，或当超过有效的字符显示区域时，也会自动结束。显示的字符应确保是 ZLG72128 能够自动完成译码的，包括字符'0'~'9'与 AbCdEFGHiJLopqrtUychT（区分大小写）。如遇到有不支持的字符，对应位置将不显示任何内容。

命令封装后的函数为 zlg72128_digitron_disp_str()，其具体实现详见程序清单 6.15。

程序清单 6.15　zlg72128_digitron_disp_str()函数

```
1    int zlg72128_digitron_disp_str (zlg72128_handle_t    handle,
2                                    unsigned char        start_pos,
3                                    char                 *p_str)
4    {
5        struct zlg72128_digitron_disp_str_t disp_str_t;
6        memset(&disp_str_t, 0, sizeof(struct zlg72128_digitron_disp_str_t));
7
8        disp_str_t.start_pos = start_pos;
9        memcpy(disp_str_t.p_str,p_str,strlen(p_str));
10
11       return ioctl(handle,ZLG72128_DIGITRON_DISP_STR,&disp_str_t);
12   }
```

例如，需要从编号为 1 的数码管开始，显示字符串“0123456”，则使用 ioctl 的命令代码为：

```
struct zlg72128_digitron_disp_str_t disp_str_t = {1,{ '0', '1', '2', '3', '4', '5', '6', '\0'}};
ioctl(handle,ZLG72128_DIGITRON_DISP_STR,&disp_str_t);
```

也可以直接使用 zlg72128_digitron_disp_str()函数完成同样的操作，即：

```
zlg72128_digitron_disp_str(handle, 1, "0123456");
```

5. 数码管显示数字

ZLG72128_DIGITRON_DISP_NUM 命令用于在指定的数码管显示一个 0~9 的数字，用到的结构体为 struct zlg72128_digitron_disp_num_t，其定义详见程序清单 6.16，各成员说明详见表 6.9。

程序清单 6.16　struct zlg72128_digitron_disp_num_t

```
1    struct zlg72128_digitron_disp_num_t{
2        unsigned char pos;
3        unsigned char num;
4        unsigned char is_dp_disp;
5        unsigned char is_flash;
6    };
```

表 6.9　struct zlg72128_digitron_disp_num_t 成员说明

结构体	成员	说明
struct zlg72128_digitron_disp_num_t	pos	字符显示位置(0~11)
	num	显示的数字(0~9)
	is_dp_disp	是否显示小数点(1 显示, 0 不显示)
	is_flash	是否闪烁(1 闪烁, 0 不闪烁)

num 的值应该为数字 0~9，而不是字符'0'~'9'，若对应的成员值设置出错则返回 ZLG72128_RETURN_PARAMETER_ERROR。

命令封装后的函数为 zlg72128_digitron_disp_num()，具体实现详见程序清单 6.17。

程序清单 6.17　zlg72128_digitron_disp_num()函数

```
1    int zlg72128_digitron_disp_num (zlg72128_handle_t   handle,
2                                    unsigned char       pos,
3                                    unsigned char       num,
4                                    unsigned char       is_dp_disp,
5                                    unsigned char       is_flash)
6    {
7        struct zlg72128_digitron_disp_num_t disp_num_t;
8        memset(&disp_num_t,0,sizeof(struct zlg72128_digitron_disp_num_t));
9
10       disp_num_t.pos = pos;
11       disp_num_t.num = num;
12       disp_num_t.is_dp_disp = is_dp_disp;
13       disp_num_t.is_flash = is_flash;
14
15       return ioctl(handle,ZLG72128_DIGITRON_DISP_NUM,&disp_num_t);
16   }
```

例如，需要在编号为 1 的数码管上开始显示数字 1，不闪烁，不显示小数点，则使用 ioctl 的命令代码为：

```
struct zlg72128_digitron_disp_num_t disp_num_t = {1,1,0,0};
ioctl(handle,ZLG72128_DIGITRON_DISP_NUM,&disp_num_t);
```

也可以直接使用 zlg72128_digitron_disp_num()函数完成同样的操作，即：

```
zlg72128_digitron_disp_num(handle, 1, 1, 0, 0);
```

6. 控制多个数码管的位段显示

ZLG72128_DIGITRON_DISPBUF_SET 命令用于控制多个数码管的位段显示，使用到的结构体为 struct zlg72128_digitron_dispbuf_set_t，其定义详见程序清单 6.18，各成员说明详见表 6.10。

程序清单 6.18　struct zlg72128_digitron_dispbuf_set_t

```
1    struct zlg72128_digitron_dispbuf_set_t{
2        unsigned char start_pos;
3        unsigned char p_buf[12];
4        unsigned char num;
5    };
```

表 6.10　struct zlg72128_digitron_dispbuf_set_t 成员说明

结构体	成员	说明
struct zlg72128_digitron_dispbuf_set_t	start_pos	起始位置(0~11)
	p_buf	段码缓冲区
	num	本次设置段码的个数

当用户需要显示一些自定义的特殊图形时，可以使用该函数，直接设置段码。一般来讲，ZLG72128 已经提供了常见的 10 种数字和 21 种字母的直接显示，用户不必使用此函数直接设置段码，可以直接使用对应的函数显示数字或字母，为方便快速设置多个数码管位显示的段码，本命令可以一次连续写入多个数码管显示的段码。在使用时需要注意的是，若 start_pos + num 的值超过 12 ，则多余的缓冲区内容被丢弃。

命令封装后的函数为 zlg72128_digitron_dispbuf_set()，具体实现详见程序清单 6.19。

程序清单 6.19　zlg72128_digitron_dispbuf_set()函数

```
1    int zlg72128_digitron_dispbuf_set (zlg72128_handle_t    handle,
2                                       unsigned char        start_pos,
3                                       unsigned char       *p_buf,
4                                       unsigned char        num)
5    {
6        struct zlg72128_digitron_dispbuf_set_t dispbuf_set_t;
7
8        memset(&dispbuf_set_t,0,sizeof(struct zlg72128_digitron_dispbuf_set_t));
9
10       dispbuf_set_t.start_pos = start_pos;
11       memcpy(dispbuf_set_t.p_buf,p_buf,num);
```

```
12          dispbuf_set_t.num = num;
13
14          return ioctl(handle,ZLG72128_DIGITRON_DISPBUF_SET,&dispbuf_set_t);
15      }
```

例如，需要设置编号为 0~11 的 12 个数码管显示的段码分别为“0x09，0x09，0x09，0x09，0x40，0x40，0x40，0x40，0x09，0x09，0x09，0x09”，则使用 ioctl 的命令代码为：

```
struct zlg72128_digitron_dispbuf_set_t dispbuf_set_t =
{0,{0x09,0x09,0x09,0x09,0x40,0x40,0x40,0x40,0x09,0x09,0x09,0x09},12};
ioctl(handle,ZLG72128_DIGITRON_DISP_NUM,&dispbuf_set_t);
```

也可以直接使用 zlg72128_digitron_dispbuf_set()函数完成同样的操作，即：

```
unsigned char buf[12] = {0x09,0x09,0x09,0x09,0x40,0x40,0x40,0x40,0x09,0x09,0x09,0x09};
zlg72128_digitron_dispbuf_set (handle, 0, buf, 12);
```

7. 控制单个数码管的位段显示

ZLG72128_DIGITRON_SEG_CTRL 命令用于控制单个数码管的位段显示，使用到的结构体为 struct zlg72128_digitron_seg_ctrl_t，其定义详见程序清单 6.20，各成员说明详见表 6.11。

程序清单 6.20　struct zlg72128_digitron_seg_ctrl_t

```
1       struct zlg72128_digitron_seg_ctrl_t{
2           unsigned char pos;
3           char seg;
4           unsigned char is_on;
5       };
```

表 6.11　struct zlg72128_digitron_seg_ctrl_t 成员说明

结构体	成员	说明
struct zlg72128_digitron_seg_ctrl_t	pos	数码管位置(0~11)
	seg	控制的段（0~7）
	is_on	是否点亮该段（1 点亮，0 熄灭）

在设置数码管位段时，建议不要直接使用立即数 0 ~ 7，而应使用与 a ~ dp 对应的宏：AM_ZLG72128_DIGITRON_SEG_A ~ AM_ZLG72128_DIGITRON_SEG_DP。

命令封装后的函数为 zlg72128_digitron_seg_ctrl()，其具体实现详见程序清单 6.21。

程序清单 6.21　zlg72128_digitron_seg_ctrl()函数

```
1       int zlg72128_digitron_seg_ctrl (zlg72128_handle_t   handle,
2                                       unsigned char       pos,
3                                       char                seg,
4                                       unsigned char       is_on)
5       {
6           struct zlg72128_digitron_seg_ctrl_t seg_ctrl_t;
7           memset(&seg_ctrl_t,0,sizeof(struct zlg72128_digitron_seg_ctrl_t));
8
9           seg_ctrl_t.pos = pos;
```

```
10          seg_ctrl_t.seg = seg;
11          seg_ctrl_t.is_on = is_on;
12
13          return ioctl(handle,ZLG72128_DIGITRON_SEG_CTRL,&seg_ctrl_t);
14      }
```

例如，需要把编号为 5 的数码管的 G 段点亮，则使用 ioctl 的命令代码为：

```
struct zlg72128_digitron_seg_ctrl_t seg_ctrl_t =    {5, AM_ZLG72128_DIGITRON_SEG_G,1};
ioctl(handle,ZLG72128_DIGITRON_SEG_CTRL,&seg_ctrl_t);
```

也可以直接使用 zlg72128_digitron_seg_ctrl()函数完成同样的操作，即：

```
zlg72128_digitron_seg_ctrl (handle, 5, AM_ZLG72128_DIGITRON_SEG_G, 1);
```

8. 数码管有效闪烁控制

ZLG72128_DIGITRON_FLASH_CTRL 命令用于控制数码管的闪烁功能，用到的结构体为 struct zlg72128_digitron_flash_ctrl_t，其定义详见程序清单 6.22，各成员说明详见表 6.12。

程序清单 6.22　struct zlg72128_digitron_flash_ctrl_t

```
1       struct zlg72128_digitron_flash_ctrl_t{
2           int ctrl_val;
3       };
```

表 6.12　struct zlg72128_digitron_flash_ctrl_t 成员说明

结构体	成员	说明
struct zlg72128_digitron_flash_ctrl_t	ctrl_val	闪烁控制值（bit0~bit11 有效）

ctrl_val 为控制值，bit0 ~ bit11 为有效位，分别对应数码管 0 ~ 11，为 0 时不闪烁，为 1 时闪烁。

命令封装后的函数为 zlg72128_digitron_flash_ctrl()，具体实现详见程序清单 6.23。

程序清单 6.23　zlg72128_digitron_flash_ctrl 函数

```
1       int zlg72128_digitron_flash_ctrl (zlg72128_handle_t    handle,
2                                         int                  ctrl_val)
3       {
4           struct zlg72128_digitron_flash_ctrl_t flash_ctrl_t;
5           memset(&flash_ctrl_t,0,sizeof(struct zlg72128_digitron_flash_ctrl_t));
6           flash_ctrl_t.ctrl_val = ctrl_val;
7           return ioctl(handle,ZLG72128_DIGITRON_FLASH_CTRL,&flash_ctrl_t);
8       }
```

例如，需要控制 12 路数码管，中间四个闪烁，其他不闪烁，则使用 ioctl 的命令代码为：

```
struct zlg72128_digitron_flash_ctrl_t flash_ctrl_t =    {0x0f0};
ioctl(handle,ZLG72128_DIGITRON_FLASH_CTRL,&flash_ctrl_t);
```

也可以直接使用 zlg72128_digitron_flash_ctrl()函数完成同样的操作，即：

```
zlg72128_digitron_flash_ctrl (handle, 0x0f0);
```

9. 控制数码管的闪烁延时功能

ZLG72128_DIGITRON_FLASH_TIME_CFG 命令用于控制单路数码管闪烁延时功能，用到的结构体为 struct zlg72128_digitron_flash_time_cfg_t，其定义详见程序清单 6.24，各成员说明详见表 6.13。

程序清单 6.24　struct zlg72128_digitron_flash_time_cfg_t

```
1    struct zlg72128_digitron_flash_time_cfg_t{
2        unsigned char on_ms;
3        unsigned char off_ms;
4    };
```

表 6.13　struct zlg72128_digitron_flash_time_cfg_t 成员说明

结构体	成员	说明
struct zlg72128_digitron_flash_time_cfg_t	on_ms	数码管点亮时间（0~15）
	off_ms	数码管熄灭时间（0~15）

上电时，数码管点亮和熄灭的持续时间默认值为 500ms。on_ms 和 off_ms 的值经计算后所得的有效延时时间值（150ms+n*50ms）为 150、200、250、……、800、850、900，即 150ms ~ 900ms，且时间间隔为 50ms。若时间间隔不是这些值，应该选择一个最接近的值。

命令封装后的函数为 zlg72128_digitron_flash_time_cfg()，具体实现详见程序清单 6.25。

程序清单 6.25　zlg72128_digitron_flash_time_cfg()函数

```
1    int zlg72128_digitron_flash_time_cfg (zlg72128_handle_t    handle,
2                                          unsigned int         on_ms,
3                                          unsigned int         off_ms)
4    {
5        struct zlg72128_digitron_flash_time_cfg_t flash_time_cfg_t;
6        memset(&flash_time_cfg_t,0,sizeof(struct zlg72128_digitron_flash_time_cfg_t));
7
8        flash_time_cfg_t.on_ms = (on_ms - 150) / 50;
9        flash_time_cfg_t.off_ms = (off_ms - 150) / 50;
10       return ioctl(handle,ZLG72128_DIGITRON_FLASH_TIME_CFG,&flash_time_cfg_t);
11   }
```

例如，需要设置数码管闪烁时点亮时间为 500ms 和熄灭时间为 500ms，则使用 ioctl 的命令代码为：

```
struct zlg72128_digitron_flash_time_cfg_t flash_time_ctrl_t =    {7,7}; //延时时间  = 150ms + 7*50ms = 500ms
ioctl(handle,ZLG72128_DIGITRON_FLASH_TIME_CTRL,&flash_time_ctrl_t);
```

也可以直接使用 zlg72128_digitron_flash_time_cfg()函数完成同样的操作，即：

```
zlg72128_digitron_flash_time_cfg (handle, 500,500);
```

10.　控制数码管的移位功能

ZLG72128_DIGITRON_SHIFT 命令用于控制数码管的移位功能，用到的结构体为 struct zlg72128_digitron_shift_t，其定义详见程序清单 6.26，各成员说明详见表 6.14。

程序清单 6.26　struct zlg72128_digitron_shif_t

```
1    struct zlg72128_digitron_shift_t{
2        unsigned char dir;
3        unsigned char is_cyclic;
4        unsigned char num;
5    };
```

表 6.14　struct zlg72128_digitron_shift_t 成员说明

结构体	成员	说明
struct zlg72128_digitron_shift_t	dir	移动方向（0 为右移，1 为左移）
	is_cyclic	是否循环移位（1 为循环移位，0 为不循环移位）
	num	移动的位数（0~11，0 为不移位，其他值无效）

成员 dir 代表移动方向，建议使用 ZLG72128_DIGITRON_SHIFT_RIGHT 和 ZLG72128_DIGITRON_SHIFT_LEFT 来设置右移和左移；如果不是循环移位，则移动后，右边空出的位（左移）或左边空出的位（右移）将不显示任何内容；若是循环移动，则空出的位将会显示被移除位的内容。

移位功能是与硬件设计电路密切相关的，若硬件电路设计数码管位置是相反的，则移位效果可能也恰恰是相反的，此处只需要稍微注意即可。

命令封装后的函数为 zlg72128_digitron_shift()，其具体实现详见程序清单 6.27。

程序清单 6.27　zlg72128_digitron_shif()函数

```
1    int zlg72128_digitron_shift (zlg72128_handle_t    handle,
2                                 unsigned char        dir,
3                                 unsigned char        is_cyclic,
4                                 unsigned char        num)
5    {
6        struct zlg72128_digitron_shift_t shift_t;
7        memset(&shift_t,0,sizeof(struct zlg72128_digitron_shift_t));
8
9        shift_t.dir = dir;
10       shift_t.is_cyclic = is_cyclic;
11       shift_t.num = num;
12       return ioctl(handle,ZLG72128_DIGITRON_SHIFT,&shift_t);
13   }
```

例如，需要数码管显示向左循环移动 2 位，则使用 ioctl 的命令代码为：

```
struct zlg72128_digitron_shift_t    shift_t =    { ZLG72128_DIGITRON_SHIFT_LEFT, ZLG72128_TRUE,2};
ioctl(handle,ZLG72128_DIGITRON_SHIFT,&shift_t);
```

也可以直接使用 zlg72128_digitron_disp_shift()函数完成同样的操作，即：

```
zlg72128_digitron_shift (handle, ZLG72128_DIGITRON_SHIFT_LEFT, ZLG72128_TRUE,2);
```

11.　数码管复位

ZLG72128_DIGITRON_DISP_RESET 为数码管显示复位命令，使用后可以熄灭所有数

码管的所有 LED 段。

该命令封装后的函数为 zlg72128_digitron_disp_reset()，其具体实现详见程序清单 6.28。

程序清单 6.28　zlg72128_digitron_disp_reset()函数

```
1    int zlg72128_digitron_disp_reset (zlg72128_handle_t handle)
2    {
3        return ioctl(handle, ZLG72128_DIGITRON_DISP_RESET,0);
4    }
```

例如，需要复位数码管的显示，则使用 ioctl 的命令代码为：

```
ioctl(handle,ZLG72128_DIGITRON_DISP_RESET,0);
```

也可以直接使用 zlg72128_digitron_disp_reset()函数完成同样的操作，即：

```
zlg72128_digitron_disp_reset (handle);
```

12.　独立数码管测试

ZLG72128_DIGITRON_DISP_TEST 为数码管测试命令，使用后所有数码管的所有 LED 段以 0.5s 的速率闪烁，用于测试数码管是否显示正常。

该命令封装后的的函数是 zlg72128_digitron_disp_test，其具体实现详见程序清单 6.29。

程序清单 6.29　zlg72128_digitron_disp_test()函数

```
1    int zlg72128_digitron_disp_test (zlg72128_handle_t handle)
2    {
3        return ioctl(handle, ZLG72128_DIGITRON_DISP_TEST,0);
4    }
```

例如，需要启动数码管的测试功能，则使用 ioctl 的命令代码为：

```
ioctl(handle,ZLG72128_DIGITRON_DISP_TEST,0);
```

也可以直接使用 zlg72128_digitron_disp_test()函数完成同样的操作，即：

```
zlg72128_digitron_disp_test (handle);
```

13.　硬件复位

ZLG72128_DIGITRON_RESET 为 ZLG72128 的硬件复位命令，用于器件复位，复位后所有数码管的设置将失效（包括默认配置），一般情况下用不到此功能。

该命令封装后的函数为 zlg72128_digitron_reset()，其具体实现详见程序清单 6.30。

程序清单 6.30　zlg72128_digitron_reset()函数

```
1    int zlg72128_digitron_reset (zlg72128_handle_t handle)
2    {
3        return ioctl(handle, ZLG72128_DIGITRON_RESET,0);
4    }
```

例如，需控制 ZLG72128 硬件复位，则使用 ioctl 的命令代码为：

```
ioctl(handle,ZLG72128_DIGITRON_RESET,0);
```

也可以直接使用 zlg72128_digitron_reset()函数完成同样的操作，即：

```
zlg72128_digitron_reset (handle);
```

6.5.3 数码管范例

程序清单 6.31 所示代码是前述所有编程接口的综合性范例，其展示了一些简单图形（字符串）的显示方法。

程序清单 6.31　数码管接口使用范例

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <sys/ioctl.h>
#include "zlg72128.h"

#define LED_DEV        "/dev/zlg72128-0"

int main(int argc,char *argv[])
{
        zlg72128_handle_t handle;
        int ret;

        handle = zlg72128_init(LED_DEV);
        if(handle == ZLG72128_RETURN_ERROR){
                printf("zlg72128_init error\n");
                return handle;
        }

        // 控制数码管的显示属性（打开显示）
        ret = zlg72128_digitron_disp_ctrl (handle, 0);
        if(ret != ZLG72128_RETURN_OK){
                printf("Error ----------%d----------\n",__LINE__);
                goto error;
        }

        // 从 0 位置开始显示一个字符串"012356789"
        ret = zlg72128_digitron_disp_str (handle, 0, "0123456789");
        if(ret != ZLG72128_RETURN_OK){
                printf("Error ----------%d----------\n",__LINE__);
                goto error;
        }

        // 设置第 10 个数码管显示的字符'1'
        ret = zlg72128_digitron_disp_char (handle, 10, '1', ZLG72128_TRUE, ZLG72128_TRUE);
        if(ret != ZLG72128_RETURN_OK){
```

```
40              printf("Error ----------%d----------\n",__LINE__);
41              goto error;
42          }
43
44          // 设置第 11 个数码管显示的数字 8
45          ret = zlg72128_digitron_disp_num (handle, 11, 8, ZLG72128_TRUE, ZLG72128_TRUE);
46          if(ret != ZLG72128_RETURN_OK){
47              printf("Error ----------%d----------\n",__LINE__);
48              goto error;
49          }
50
51          // 设置数码管闪烁时间，闪烁时点亮持续的时间 500ms 和熄灭持续的时间 500ms
52          ret = zlg72128_digitron_flash_time_cfg (handle, 500, 500);
53          if(ret != ZLG72128_RETURN_OK){
54              printf("Error ----------%d----------\n",__LINE__);
55              goto error;
56          }
57
58          // brief 控制（打开或关闭）数码管闪烁，中间四个 LED 闪烁其他不闪烁
59          ret = zlg72128_digitron_flash_ctrl (handle,0x0f0);
60          if(ret != ZLG72128_RETURN_OK){
61              printf("Error ----------%d----------\n",__LINE__);
62              goto error;
63          }
64
65          // 循环左移两位
66          ret = zlg72128_digitron_shift (handle, ZLG72128_DIGITRON_SHIFT_LEFT, ZLG72128_TRUE, 2);
67          if(ret != ZLG72128_RETURN_OK){
68              printf("Error ----------%d----------\n",__LINE__);
69              goto error;
70          }
71          sleep(5);
72
73          // 直接设置数码管显示的内容（段码）
74          unsigned char buf[12] = {0x09,0x09,0x09,0x09,0x40,0x40,0x40,0x40,0x09,0x09,0x09,0x09};
75          ret = zlg72128_digitron_dispbuf_set (handle, 0, buf, 12);
76          if(ret != ZLG72128_RETURN_OK){
77              printf("Error ----------%d----------\n",__LINE__);
78              goto error;
79          }
80          sleep(5);
81
82          // 控制第 5 个数码管的 G 段熄灭
83          ret = zlg72128_digitron_seg_ctrl (handle,
```

```
84                                          5,
85                                          AM_ZLG72128_DIGITRON_SEG_G,
86                                          ZLG72128_FALSE);
87          if(ret != ZLG72128_RETURN_OK){
88                  printf("Error ----------%d----------\n",__LINE__);
89                  goto error;
90          }
91
92          // 控制第 6 个数码管的 G 段熄灭
93          ret = zlg72128_digitron_seg_ctrl (handle,
94                                          6,
95                                          AM_ZLG72128_DIGITRON_SEG_G,
96                                          ZLG72128_FALSE);
97          if(ret != ZLG72128_RETURN_OK){
98                  printf("Error ----------%d----------\n",__LINE__);
99                  goto error;
100         }
101         sleep(5);
102
103         // 复位命令，将数码管的所有 LED 段熄灭
104         ret = zlg72128_digitron_disp_reset (handle);
105         if(ret != ZLG72128_RETURN_OK){
106                 printf("Error ----------%d----------\n",__LINE__);
107                 goto error;
108         }
109         sleep(5);
110
111         // 测试命令，所有 LED 段以 0.5S 的速率闪烁，用于测试数码管显示是否正常
112         ret = zlg72128_digitron_disp_test (handle);
113         if(ret != ZLG72128_RETURN_OK){
114                 printf("Error ----------%d----------\n",__LINE__);
115                 goto error;
116         }
117
118 error:
119         return ret;
120 }
```

第7章 在 Windows 中使用 ZLG72128

✍本章导读

为便于在Windows中使用ZLG72128,ZLG针对ZLG72128的按键和数码管/LED功能，分别提供了相应的 Windows 驱动支持。本驱动软件包主要是 ZLG72128 通用软件包的Windows平台扩展包，实现了ZLG72128通用软件包的I²C的通信细节，同时保留原来通用软件包的接口基本不变为前提而实现的。

由于在PC中没有提供中断接口，所以按键信息只能通过轮询模式获得，本驱动软件包内部实现线程自动轮询按键信息，用户也可以通过设置回调函数来获取按键信息。

同时由于PC没有提供I²C接口,所以为了与PC对接,需要使用到串口转I²C或者USB转I²C的转换器，所以在本驱动软件包中出现I²C的转换器软件包，由于市面上I²C的转换器类型很多,所以本章节使用SC18IM700这一款转换器为例子来讲述如何使用Windows版本的ZLG72128软件包，如果用户想使用其他的转换器，可以查考SC18IM700转换器的接口定义，写出一个接口相同的转换器即可以接入Windows版本的ZLG72128软件包。

7.1 驱动和源码编译说明

ZLG72128的Windows驱动（含范例）源码存于ZLG72128资料集中，ZLG72128资料集可以从致远电子官网(http://www.zlg.cn/)下载(或联系致远电子相关区域销售获取)。Windows 驱动（含范例）源码的路径为：ZLG72128 资料集/04.软件设计指南/05. 驱动软件——Windows。源码目录和相关文件说明详见表 7.1。

表 7.1 ZLG72128 驱动源码文件说明

目录	文件	说明
/	zlg72128_win_test.c	范例程序
/src/zlg72128_driver	zlg72128.c、zlg72128.h、zlg72128_platform.c、zlg72128_platform.h	ZLG72128 通用软件包和 Windows 平台驱动源码
/src/ zlg72128_windrv	windrv_commond.h	Windows 平台公共定义头文件
/src/ zlg72128_windrv_hwconf	zlg72128_windrv_hwconf.h	驱动配置文件
/src/IIC	windrv_IIC_commond.h、windrv_IIC_SC18IM700.c、windrv_IIC_SC18IM700.h	SC18IM700 转换器的源码

一般在Windows上面编译都是使用Visual Studio。首先在Visual Studio中创建一个C++或者 C 的空项目，然后把上面的所有代码导入项目中，最后通过编译就会生成一个对应的可执行文件EXE，该可执行文件EXE就是由上面的zlg72128_win_test.c文件生成的测试范例程序。

7.2 使用 ZLG72128 通用软件包接口

在第三章中提到，Windows 平台已经完成了对 ZLG72128 通用软件包的适配，若用户在 Windows 中使用 ZLG72128，则无需自行适配。

7.2.1 获取 handle

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 ZLG72128 对象，Windows 驱动包提供了 zlg72128_windev_init()函数来直接获取 handle，其在 zlg72128_windrv_hwconf.h 文件中声明，函数原型为：

```
static      zlg72128_handle_t    zlg72128_windev_init(void);
```

调用形式如下：

```
zlg72128_handle_t    handle = zlg72128_windev_init();
```

此处获取的 handle 即可用于 ZLG72128 通用软件包提供的各个功能接口。

注意：Windows 系统默认没有 I²C 接口，无法直接与 ZLG72128 通信（I²C 从机器件）通信。为了与 I²C 从机器件通信，需要使用到 I²C 转换器（例如：UART 转 I²C）。在驱动默认设置中，采用的是型号为 SC18IM700 的 I²C 转换器（串口转 I²C），使用的串口是 COM1。若需使用其他串口，则可以通过简单的配置实现（下节介绍）；若需使用其它型号的 I²C 转换器，则需要重新适配 I²C 转换器（将在 7.3 节详细介绍）。

7.2.2 配置

在实际情况中，串口也可能为 com2，这时就需要修改 zlg72128_windrv_hwconf.h 配置文件中的参数，文件的具体内容详见程序清单 7.1。

程序清单 7.1　ZLG72128 配置文件内容示意

```
1    #ifndef _ZLG72128_WINDRV_HWCONF_H__
2    #define _ZLG72128_WINDRV_HWCONF_H__
3
4    #include "../IIC/windrv_IIC_SC18IM700.h"
5    #include "../zlg72128_driver/zlg72128.h"
6
7    //     SC18IM700 串口
8    #define    ZLG72128_WINDRV_COM_NAME                    "COM1"
9
10   //     ZLG72128 的从机地址
11   #define    ZLG72128_WINDRV_SLV_ADDR                    0x30
12
13   //     ZLG72128 的轮询时间间隔
14   #define    ZLG72128_WINDRV_THREAD_INTERVAL_TIME        5
15
16   //     定义一个 ZLG72128 实例
17   static    zlg72128_dev_t     g_dev;
18   //     定义一个 平台信息实例
19   static    zlg72128_devinfo_t    g_devinfo;
20
```

```
21  static  zlg72128_handle_t    zlg72128_windev_init(void){
22
23          g_devinfo.plfm_info.slv_addr   =   ZLG72128_WINDRV_SLV_ADDR;
24          g_devinfo.plfm_info.thread_interval_time   =   ZLG72128_WINDRV_THREAD_INTERVAL_TIME;
25
26          //      设置 IIC 的函数表
27          g_devinfo.p_info.vt   =   Get_IIC_SC18IM700_Vt_Table();
28          //      设置 IIC 的操作句柄
29          g_devinfo.p_info.p_IIC_arg   =   SC18IM700_create(ZLG72128_WINDRV_COM_NAME);
30
31          //      如果输入的参数没有问题，
32          //      返回来的 zlg72128_handle 对象和上面的 dev 其实指向同一个内存空间的
33          return zlg72128_init(&g_dev,   &g_devinfo);
34  }
```

其中，默认的 3 个配置项使用了宏的形式进行了定义，各配置项的功能简介详见表 7.2，这些配置项都可以根据实际情况修改。

表 7.2　ZLG72128 用作通用功能时的配置项

序号	配置宏	配置项	默认值
1	ZLG72128_WINDRV_COM_NAME （对应程序清单 7.1 的第 8 行）	串口号	COM1
2	ZLG72128_WINDRV_SLV_ADDR （对应程序清单 7.1 的第 11 行）	从机地址	0x31
3	ZLG72128_WINDRV_THREAD_INTERVAL_TIME （对应程序清单 7.1 的第 14 行）	键值轮询间隔	5ms

由此可见，该配置文件的存在，使得用户可以快速通过 zlg72128_windev_init()函数获取到 handle，进而快速开发应用程序。即使有一些配置并不相同（例如，串口号），也可以通过修改配置宏的值快速实现配置。

在 zlg72128_windrv_hwconf.h 配置文件中，默认采用的是型号为 SC18IM700 的转换器，主要通过程序清单 7.1 的第 27 行和第 29 行体现：

```
g_devinfo.plfm_info.vt            =   Get_IIC_SC18IM700_Vt_Table();
g_devinfo.plfm_info.p_IIC_arg     =   SC18IM700_create(ZLG72128_WINDRV_COM_NAME);
```

这两行代码完成了设备信息中部分成员的赋值。其中，SC18IM700_create()函数用以获取 SC18IM700 的 I^2C 转换器句柄，而 ZLG72128_WINDRV_COM_NAME 表示使用的串口，Get_IIC_SC18IM700_Vt_Table()函数用以获取 SC18IM700 的函数表指针。在实际应用中，若用户使用的是其它型号的 I^2C 转换器，则仅需实现与 Get_IIC_SC18IM700_Vt_Table()和 SC18IM700_create()相对应的两个函数，并将上述两行代码替换为调用新的函数实现，具体这两个函数的功能与实现方法，将在 7.3 节中详细介绍。特别地，若用户选用的 I^2C 转换器恰好是 SC18IM700，则可以跳过 7.3 节的内容，直接阅读后续内容。

7.2.3 应用

通过 zlg72128_windrv_hwconf.h 配置文件中的 zlg72128_windev_init()函数获取到 handle 后，即可基于通用软件包提供的接口（功能接口详解详见 3.5 节）快速操作 ZLG72128，详

见程序清单 7.2。

程序清单 7.2　应用兼容接口代码

```
#include "zlg72128_windrv_hwconf.h"
#include <stdio.h>

void    main(void){
        int error = 0;
        zlg72128_dev_t      dev;
        zlg72128_devinfo_t    devinfo;

        zlg72128_handle_t      zlg72128_handle    =      zlg72128_windev_init();

        if (zlg72128_handle    ==    NULL){
                printf("zlg72128_init_windev failed \n");
                return ;
        }

        //      复位显示
        error    =    zlg72128_digitron_disp_reset(zlg72128_handle);

        //      从 0 位置开始显示一个字符串"01234567"
        error    =    zlg72128_digitron_disp_str(zlg72128_handle,    0,    "01234567");
        if (error    !=    0)    printf("Error ----------%d----------\n", __LINE__);

        printf("key down to exit and zlg72128_close_windev \n");
        getchar();

        //      执行完 zlg72128 后需要释放 zlg72128 句柄
        zlg72128_deinit (zlg72128_handle);
        zlg72128_handle = NULL;
}
```

上面的代码只是简单的演示了如何使用 zlg72128_windrv_hwconf.h 配置文件中的 zlg72128_windev_init()函数来快速开发，当然也需要采用 SC18IM700 作为 I^2C 转换器和配置相同的串口，如果用户想摆脱 zlg72128_windrv_hwconf.h 配置文件和了解更加详细配置接口，请详细看 7.3 节。

由于接口的通用性，因此，也可以直接运行基于 ZLG72128 功能接口编写的 demo 程序，例如，运行组合键测试 demo（实现代码详见程序清单 3.32），主程序范例详见程序清单 7.3。

程序清单 7.3　使用 ZLG72128 通用测试程序进行测试

```
#include "demo_zlg72128_entries.h"
#include "zlg72128.h"
#include "zlg72128_windrv_hwconf.h"

```

```
5   int am_main (void)
6   {
7       zlg72128_handle_t    handle = zlg72128_windev_init();
8
9       demo_zlg72128_combination_key_test_entry(handle);                    // 启动组合键测试 Demo
10
11      while(1) {
12      }
13  }
```

7.3 I²C 转换器适配

7.3.1 I²C 转换器使用说明

1. SC18IM700 转换器说明

在 Windows 系统中没有提供 I²C 接口，所以需要加外部的转换器来转换为 I²C，由于市面上有很多 PC 的 I²C 转换器，每一种转换器的编程都可能会不一样，而本章采用了 SC18IM700 转换器作为例子来讲述如何在 Windows 上面通过 I²C 转换器来控制 ZLG72128。

在 ZLG72128 的 Windows 驱动软件包中，提供了一套 I²C 的接口，如果用户使用其他的转换器，只要实现驱动软件包中的 I²C 的接口就可以接入驱动软件包中使用，详见 7.3.5。而本章采用的 SC18IM700 是串口转 I²C 的形式，可以理解为操作 SC18IM700 就是操作串口，会涉及到串口的编程，如果不了解串口编程的，可以阅读其他一些串口编程的文章。

写入数据流程大致是，先通过 Windows 的串口发送数据，然后通过 SC18IM700 把串口数据转换到 I²C，然后再发送给 ZLG72128 芯片；而读取数据的流程是 ZLG72128 芯片把 I²C 的数据发送给 SC18IM700，然后 SC18IM700 把数据转换为串口数据，返回来给 Windows 程序接收读取数据，整个通信过程的示意图详见图 7.1。

图 7.1 SC18IM700 通信流程图

2. 发送数据

给 ZLG72128 发送数据，在 2.2.2 节中有详细讲述，通过 SC18IM700 转换的话，需要在发送的数据中增加一些字节，才可以发送成功。

例如给 SC18IM700 发送数码管测试命令，第一个字节必须是 0x53 表示开始，然后第二个字节为 0x60（从机地址+写标记），第三个字节为 0x2 表示需要写入的命令长度为 2（需要写入的命令为第四和第五个字节），第四个字节为 0x7 表示需要写入的寄存器地址（定位到 0x7 号寄存器），第五个字节为 0x40 表示把 0x40 写入到 0x7 号寄存器（数码管测试命令），第六个字节为 0x50 为结束， 发送字节组成详见图 7.2。

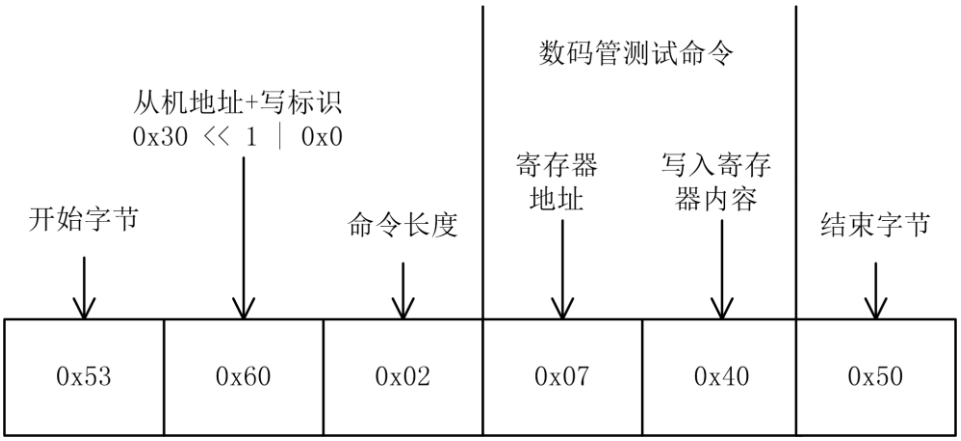


图 7.2　数码管测试命令字节

其中包含了发送命令 0x7 和 0x40，这两个字节表示的命令为数码管测试命令，也可以换成其他的命令来发送，有些命令的字节长度为 1，所以发送这些命令的时候在第三个字节就需要设置为 0x1，如果不明白命令字节的含义，请阅读 2.1 寄存器详解。

3． 读取数据

类似的，ZLG72128 读取数据在 2.2.3 节中也有详细讲述，但是在使用 SC18IM700 转换的时候，发送命令字节也有所不同。例如，读取按键信息，一开始需要定位到按键寄存器的地址上面，然后再读取寄存器信息，按键的寄存器地址分别为 0x00，0x01，0x02 和 0x03，所以需要定位寄存器地址到 0x00，然后读取 4 个字节就可以获取到按键信息。

读取按键信息的第一条命令，第一个字节是 0x53 表示开始，第二个字节是 0x60（从机地址+写标记，该字节主要是用来定位，而不是真的写入数据），第三个字节是 0x01 命令字节长度（这里的长度一定是 0x01，写入的命令只会是寄存器的地址，而不会在寄存器中写入数据），第四个字节是 0x00 表示需要读取定位的寄存器地址（定位到 0x00 号寄存器），第五个字节是 0x50 表示结束，字节组成详见图 7.3。

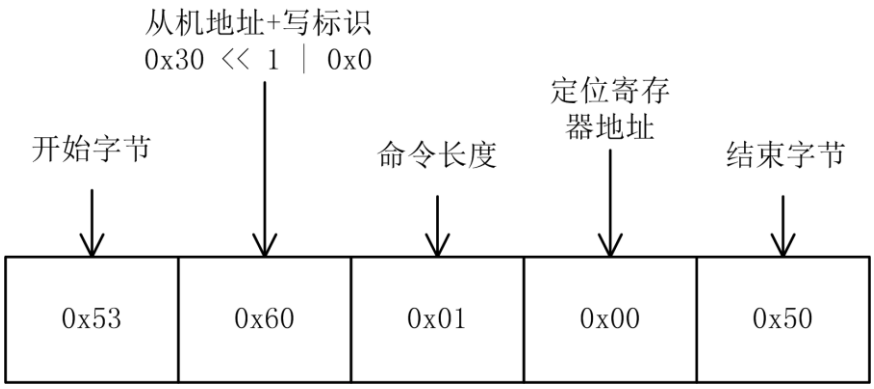


图 7.3　定位 0x00 寄存器命令字节

读取按键信息的第二条命令，第一个字节是 0x53 表示开始，第二个字节是 0x61（从机地址+读标记），第三个字节是 0x04 读取字节个数（读取四个寄存器的内容），第四个字节是 0x50 表示结束，字节组成详见图 7.4，当这两条命令发送完毕后，就可以获取 ZLG72128 返回来的数据了。

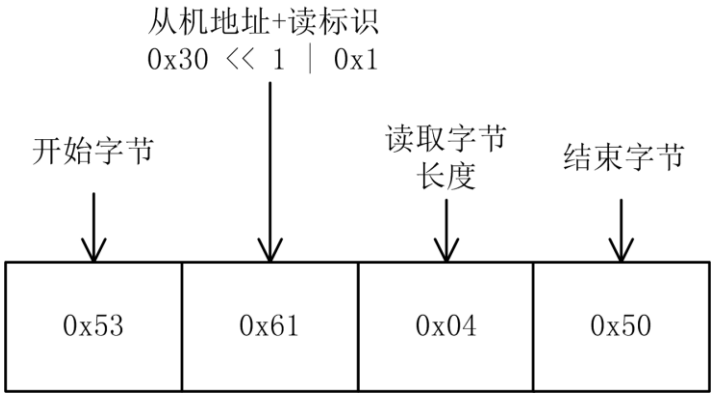


图 7.4　读取寄存器命令字节

当然也可以把上面两条命令合并成一条命令，但是需要注意的是，每条命令只能有一个结束字节，所以该合并命令的第一个字节为 0x53 表示开始，第二个字节为第二个字节是 0x60（从机地址+写标记），第三个字节是 0x01 命令字节长度，第四个字节是 0x00 表示定位的寄存器地址，第五个字节为 0x53 表示下一条命令的开始，第六个字节为 0x61（从机地址+读标记），第七个字节是 0x04 读取字节个数（读取四个寄存器的内容），第八个字节是 0x50 表示结束，字节组成详见图 7.5。

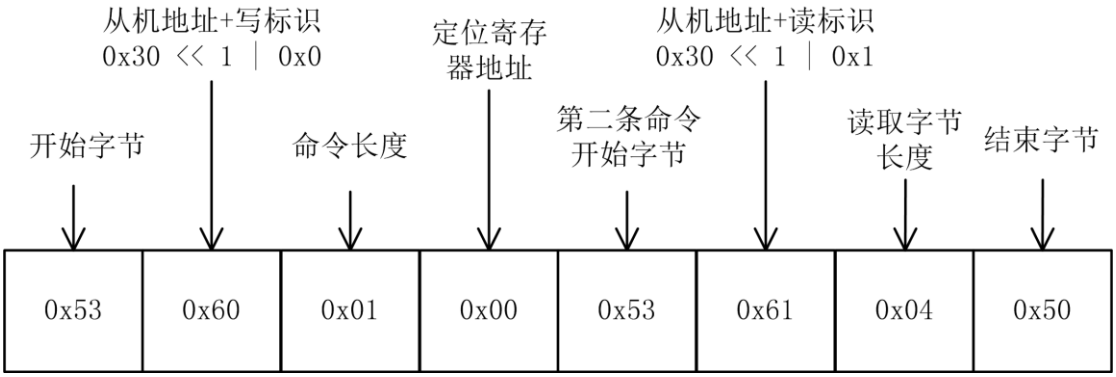


图 7.5　定位和读取同时方式命令字节

4．设置波特率

在串口通信中，有一个会影响传输速度的参数，该参数是波特率。SC18IM700 默认的波特率是 9600bps，速度会相对慢了一些，尤其在快速读取按键信息的时候可能速度会跟不上，但是可以通过设置 SC18IM700 的波特率提高 SC18IM700 的串口读写速度。

同样的是通过发送串口数据来修改 SC18IM700 的波特率，但是这次发送数据的对象是 SC18IM700 而不是 ZLG72128，所以发送数据的格式有所不同了。

在 SC18IM700 中有两个寄存器来存放波特率的信息，寄存器名称分别是 BRG0 和 BRG1，BRG0 寄存器的地址为 0x00，BRG1 的寄存器地址为 0x01，而波特率可以通过如下公式进行计算 BRG0 和 BRG1 的值：

$$(BRG1, BRG0) = 7372800 / \text{波特率} - 16$$

其中$(BRG1, BRG0)$是一个通过公式算出来的 4 位十六进制结果，该十六进制结果的高 2 位为 BRG1 的值，低 2 位为 BRG0 的值。

例如，需要设置的波特率为 115200bps，通过公式计算得出$(BRG1, BRG0)$等于 0x0030 的值，然后该值的高 2 位为 0x00，低 2 位为 0x30，所以 BRG1 的值为 0x00，而 BRG0 的

值为 0x30，只要把对应的值写入寄存器即可修改串口波特率。

读取 SC18IM700 的 BRG0 和 BRG1 寄存器数据命令，第一个字节为 0x52 表示读取 SC18IM700 寄存器数据，第二个字节为 0x00 表示读取 0x0 号寄存器的数据（0x0 号寄存器就是 BRG0 寄存器），第三个字节为 0x01 表示读取 0x1 号寄存器的数据（0x1 号寄存器就是 BRG1 寄存器），第四个字节为 0x50 表示结束本命令，字节组成详见图 7.6。

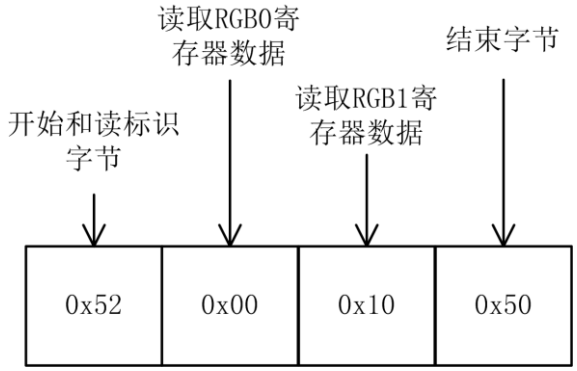


图 7.6　读取波特率命令字节

如果断电重启 SC18IM700 后，SC18IM700 的波特率会恢复默认值 9600bps，这时候读取 BRG0 和 BRG1 寄存器数据，能够获取到 BRG0 为 0xF0，BRG1 为 0x02，通过上面的公式验证，SC18IM700 的默认波特率就是 9600bps，接下来就是发送修改波特率的命令。

发送修改 SC18IM700 波特率为 115200bps 的命令，第一个字节为 0x57 表示写入 SC18IM700 寄存器数据，第二个字节为 0x00 表示定位到 SC18IM700 的 0x0 号寄存器地址（0x0 号寄存器就是 BRG0 寄存器），第三个字节为 0x30 表示上面公式计算等到的 BRG0 的值，第四个字节为 0x01 表示定位到 SC18IM700 的 0x01 号寄存器地址（0x01 号寄存器就是 BRG1 寄存器），第五个字节为 0x00 表示上面公式计算等到的 BRG1 的值，第六个字节为 0x50 表示结束本命令，字节组成详见图 7.7。

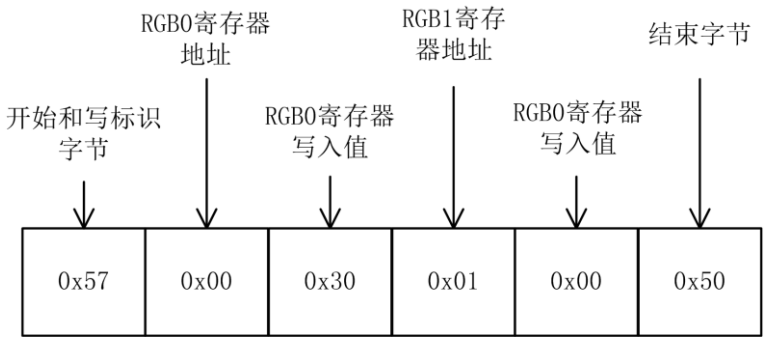


图 7.7　修改波特率为 115200bps 命令字节

注意：在 Windows 上面串口的波特率设置不一样的话，会导致通信出错，如果 SC18IM700 的波特率比 Windows 串口设置的波特率要高的时候，发送读取 SC18IM700 的 BRG0 和 BRG1 寄存器数据命令，会导致 SC18IM700 重启硬件，SC18IM700 的波特率会恢复到默认的 9600bps，只有在波特率相同的情况下，才可以正常的操作。

由于操作波特率比较复杂，在 ZLG72128 的 Windows 驱动软件包中，SC18IM700_create_ex 和 SC18IM700_create_DCB 函数可以同步设置 Windows 串口波特率和 SC18IM700 的波特率，用户只需要设置需要的波特率就可以了，这两个函数具体用法请看

创建 SC18IM700 结构体函数章节，但是注意波特率设置必须大于 110bps。

7.3.2 创建 SC18IM700 结构体函数

创建 SC18IM700 结构体函数的作用是创建出一个包含 SC18IM700 串口配置信息的结构体，该结构体在 ZLG72128 初始化的时候会使用到。相关接口原型详见表 7.3。

表 7.3 创建 SC18IM700 结构体函数接口

函数原型	功能简介
PIIC_SC18IM700_Handle_t SC18IM700_create(const char *com_name);	创建 SC18IM700 的 I²C 转换器句柄，简单版本
PIIC_SC18IM700_Handle_t SC18IM700_create_ex(const char *com_name, DWORD baud_rate, BYTE byte_size, BYTE stop_bits, char EofChar);	创建 SC18IM700 的 I²C 转换器句柄，复杂版本，用于设置各种串口参数的
PIIC_SC18IM700_Handle_t SC18IM700_create_DCB(const char *com_name, const DCB *m_DCB);	创建 SC18IM700 的 I2C 转换器句柄，windows 设置版本

1. SC18IM700_create()函数

SC18IM700_create()函数的原型为：

```
PIIC_SC18IM700_Handle_t SC18IM700_create (const char *com_name);
```

该函数为创建 I²C 转换器句柄的简单版本，只需要通过 com_name 参数输入串口名称就可以创建出一个 SC18IM700 的 I²C 转换器句柄，用于快速测试，注意的是默认波特率设置是 9600bps，如果想修改波特率的话，就不能使用这个函数。

返回值为 I²C 转换器句柄，特别地，若创建失败，则返回值为 NULL。

2. SC18IM700_create_ex()

SC18IM700_create_ex()函数的原型为：

```
PIIC_SC18IM700_Handle_t SC18IM700_create_ex( const char *com_name,
                                             DWORD baud_rate,
                                             BYTE byte_size,
                                             BYTE stop_bits,
                                             char EofChar);
```

该函数为创建 I²C 转换器句柄的复杂版本，该函数有 5 个参数，其中：com_name 同样表示设置 COM 口的名字；baud_rate 表示期望设置的波特率，内部会同时修改 PC 串口和 SC18IM700 的波特率；byte_size 表示数据位宽，这里应该设置为 8 位；stop_bits 表示停止位，这里应该设置为 1 位；Eofchar 表示停止字符。使用该函数主要是用来设置波特率，以解决 SC18IM700_create()函数没法设置波特率的问题。

返回值同样为 I²C 转换器句柄，特别地，若创建失败，则返回值为 NULL。

3. SC18IM700_create_DCB()函数

SC18IM700_create_DCB()函数的原型为：

```
PIIC_SC18IM700_Handle_t    SC18IM700_create_DCB (  const  char    *com_name,
                                                   const  DCB     *m_DCB);
```

SC18IM700_create_DCB 函数为 Windows 串口配置版本，可以让用户传入 windows 的串口设置参数来配置串口，参数 DCB 为 windows 的串口配置信息，可以通过 windows 的 API 函数 GetCommState 函数来获取 m_DCB 结构体对象，又或者可以根据自己的需要自行创建出来，该函数是给用户提供修改串口的所有设置参数使用的，但是 SC18IM700_create_DCB 函数本质上和上面的 SC18IM700_create 函数或者 SC18IM700_create_ex 函数是没有任何区别，返回的均是 I²C 转换器句柄。

7.3.3 I²C 转换器函数表

I²C 转换器函数表主要是提供给 ZLG72128 的 Windows 驱动软件包中调用，实现多态调用适配不同的 I²C 转换器的接口，本节是按照 SC18IM700 为例子，提供了一个获取 I²C 转换器函数表的函数，其函数原型为：

```
const P_windev_IIC_vt_t Get_IIC_SC18IM700_Vt_Table (void);
```

该函数返回的是一个常量指针类型，该指针会指向 IIC_SC18IM700_vt_vtable 对象，其中包含了很多函数的实现，其详细内容可以参见 7.3.5 节。

该函数获取到的指针将通过 ZLG72128 的 Windows 驱动软件包的初始化方法传入 ZLG72128 驱动中，以便在 ZLG72128 的 Windows 驱动软件包内部使用。

一般用户使用的转换器为 SC18IM700 的话，是不需要了解内部原理的，直接调用即可，如果使用其他的转换器的话就需要详细了解，详见 7.3.5。

7.3.4 串口配置

提供了 set_SC18IM700_CommTimeouts 函数和 set_SC18IM700_SetupComm 函数，分别是设置串口的读写时间和设置串口读写缓冲区大小。如果不清楚具体要设置为多少的话，可以不调用函数，使用系统默认提供的默认参数即可。相关接口详表 7.4。

表 7.4 创建 SC18IM700 结构体函数接口

函数原型	功能简介
BOOL set_SC18IM700_CommTimeouts(PIIC_SC18IM700_Handle_t phandle, const COMMTIMEOUTS *p_time);	设置串口的读写时间
BOOL set_SC18IM700_SetupComm(PIIC_SC18IM700_Handle_t phandle, DWORD dwInQueue, DWORD dwOutQueue);	设置串口的读写缓冲区大小

1. set_SC18IM700_CommTimeouts()函数

set_SC18IM700_CommTimeouts()函数的原型为：

```
void    set_SC18IM700_CommTimeouts(PIIC_SC18IM700_Handle_t    phandle,
                                   const  COMMTIMEOUTS        *p_time);
```

该函数用于设置串口的读写时间，具有 2 个参数，其中：phandle 为 SC18IM700 的 I²C

转换器句柄；p_time 为指向时间信息的指针。COMMTIMEOUTS 类型为 windows 串口的读写时间类型，其定义如下：

```
typedef      struct      _COMMTIMEOUTS {
      DWORD     ReadIntervalTimeout;              /* Maximum time between read chars.   */
      DWORD     ReadTotalTimeoutMultiplier;       /* Multiplier of characters.          */
      DWORD     ReadTotalTimeoutConstant;         /* Constant in milliseconds.          */
      DWORD     WriteTotalTimeoutMultiplier;      /* Multiplier of characters.          */
      DWORD     WriteTotalTimeoutConstant;        /* Constant in milliseconds.          */
} COMMTIMEOUTS,     *LPCOMMTIMEOUTS;
```

2． set_SC18IM700_SetupComm()函数

set_SC18IM700_SetupComm()函数的原型为：

```
void       set_SC18IM700_SetupComm(      PIIC_SC18IM700_Handle_t      phandle,
                                         DWORD                        dwInQueue,
                                         DWORD                        dwOutQueue);
```

该函数用于设置串口的读写缓冲区大小，具有 3 个参数，其中：phandle 为 SC18IM700 的 I²C 转换器句柄；dwInQueue 为串口写入缓冲区的大小；dwOutQueue 为串口读取缓冲区的大小。通常情况下，只需要在读写字节比较大的时候可以通过这个函数调整缓冲区大小，但是一般情况下是不需要修改的。

7.3.5 I²C 转换器适配方法

在 ZLG72128 的 Windows 驱动软件包中，提供了用户自定义 I²C 转换器适配方案的接入。若用户使用的是 SC18IM700 转换器方案，就不需要看本节内容，仅当用户使用的是其它 I²C 转换器时，才需要仔细本节内容，掌握如何将自定义 I²C 转换器接口接入 ZLG72128 的 Windows 驱动软件包中。本节是采用 SC18IM700 的转换器作为例子来讲解如何实现接口，ZLG72128 的 Windows 驱动软件包适配其他的 I²C 转换器。

ZLG72128 的 Windows 驱动软件包的 I²C 转换器的函数接口表，函数表的类型定义详见程序清单 7.4。

程序清单 7.4　I²C 转换的的函数接口表

```
1     // I²C 转换器初始化函数，用来执行 I2C 转化器的初始化
2     typedef     int(*windev_IIC_Init) (void       *p_arg);
3
4     // I²C 转换器关闭释放函数，用来释放关闭 I2C 转化器的系统资源
5     typedef     int(*windev_IIC_Close) (void     *p_arg);
6
7     // I²C 转换器读取信息函数，用来获取 ZLG72128 信息的函数，一般用来获取按钮信息的函数
8     typedef     int(*windev_IIC_Read) (void                    *p_arg,          unsigned   char          slv_addr,
9                                        unsigned   char          sub_addr,        unsigned   char          *p_buf,
10                                       unsigned   int           nbytes);
11
12    // I²C 转换器发送信息函数，用来给 ZLG72128 发送各种指令消息函数
13    typedef     int(*windev_IIC_Write) (void                   *p_arg,          unsigned   char          slv_addr,
14                                        unsigned   char         sub_addr,        unsigned   char          *p_buf,
```

```
15                                          unsigned    int          nbytes);
16
17  //      转接器的函数表类型
18  typedef struct _windev_IIC_vt
19  {
20          windev_IIC_Init             init_func;              //      I²C 转换器初始化函数
21          windev_IIC_Close            close_func;             //      I²C 转换器关闭释放函数
22          windev_IIC_Read             read_func;              //      I²C 转换器读取信息函数
23          windev_IIC_Write            write_func;             //      I²C 转换器发送信息函数
24  } windev_IIC_vt_t;
```

该函数表的类型原型（windev_IIC_vt_t）定义在 windrv_IIC_commond.h 文件中，其中包含了四个成员变量，它们的类型都是函数指针，分别指向 I²C 转换器的不同函数。

1. SC18IM700 的静态常量函数表对象

```
static    const    windev_IIC_vt_t    IIC_SC18IM700_vt_vtable    =    {    IIC_SC18IM700_Init,
                                                                          IIC_SC18IM700_Close,
                                                                          IIC_SC18IM700_Read,
                                                                          IIC_SC18IM700_Write };
```

IIC_SC18IM700_vt_vtable 对象为 SC18IM700 的函数表对象，该对象主要是用于 ZLG72128 初始化的时候使用的，通过调用 Get_IIC_SC18IM700_Vt_Table 函数来获取，并且传入 SC18IM700 的配置结构体指针，同理也可以适配其他转换器。

该对象是静态常量对象，其成员变量的函数指针指向了四个函数，详见表 7.5。

表 7.5　SC18IM700 的函数接口

函数原型	功能简介
static int IIC_SC18IM700_Init(PIIC_SC18IM700_Handle_t p_arg)	初始化函数
static int IIC_SC18IM700_Close(PIIC_SC18IM700_Handle_t *p_arg)	关闭释放资源函数
static int IIC_SC18IM700_Read(PIIC_SC18IM700_Handle_t p_arg, unsigned char slv_addr, unsigned char sub_addr, unsigned char *p_buf, unsigned int nbytes)	读取数据函数
static int IIC_SC18IM700_Write(PIIC_SC18IM700_Handle_t p_arg, unsigned char slv_addr, unsigned char sub_addr, unsigned char *p_buf, unsigned int nbytes)	写入命令数据函数

2. 初始化函数

初始化函数的原型为：

```
static    int    IIC_SC18IM700_Init (PIIC_SC18IM700_Handle_t        p_arg);
```

该函数主要是用来初始化串口和设置串口参数等，参数 p_arg 是 I²C 转换器句柄（可以通过 SC18IM700_create()、SC18IM700_create_ex()或 SC18IM700_create_DCB()函数得到）。需要注意的是，该函数并不直接对外开放（内部静态函数），其会作为全局 IIC_SC18IM700_vt_vtable 对象中的 init_func 成员变量的值。

返回值表示执行的结果，0 表示成功；其他值表示失败。

3. 关闭释放函数

关闭释放函数的原型为：

```
static    int    IIC_SC18IM700_Close(PIIC_SC18IM700_Handle_t        *p_arg)
```

该函数主要是用来关闭和释放串口资源，同时也释放由 SC18IM700_create()等创建函数创建出来的 SC18IM700 配置结构体对象，参数 p_arg 是 SC18IM700 的配置结构体指针的指针。该函数同样不直接对外开放，其作为全局 IIC_SC18IM700_vt_vtable 对象中的 close_func 成员变量的值。

返回值表示执行的结果，0 表示成功；其他值表示失败。

4. 读取数据函数

读取数据函数的原型为：

```
static  int  IIC_SC18IM700_Read( PIIC_SC18IM700_Handle_t        p_arg,
                                 unsigned  char                 slv_addr,
                                 unsigned  char                 sub_addr,
                                 unsigned  char                 *p_buf,
                                 unsigned  int                  nbytes);
```

该函数主要用于读取 ZLG72128 的寄存器信息，例如按键信息。其具有 5 个参数：p_arg 是 SC18IM700 的配置结构体指针；slv_addr 表示从机地址；sub_addr 表示寄存器地址；p_buf 指向一段缓存，用以保存读取寄存器后的数据；nbytes 表示读取数据的字节数，其应与 p_buf 缓存的实际大小保持一致，若读取字节个数大于 p_buf 字节数组的内存空间，会导致栈溢出问题。该函数同样不直接对外开放，其作为全局 IIC_SC18IM700_vt_vtable 对象中的 read_func 成员变量的值。

返回值表示执行的结果，0 表示成功；其他值表示失败。

5. 写入数据函数

写入数据函数的原型为：

```
static  int  IIC_SC18IM700_Write( PIIC_SC18IM700_Handle_t        p_arg,
                                  unsigned  char                 slv_addr,
                                  unsigned  char                 sub_addr,
                                  unsigned  char                 *p_buf,
                                  unsigned  int                  nbytes);
```

该函数主要是用于向 ZLG72128 的寄存器写入（发送）信息，例如数码管显示控制命令。其具有 5 个参数：p_arg 是 SC18IM700 的配置结构体指针；slv_addr 表示从机地址；sub_addr 表示寄存器地址；p_buf 指向一段缓存，用以保存待写入寄存器的数据；nbytes 表示写入数据的字节数。该函数同样不直接对外开放，其作为全局 IIC_SC18IM700_vt_vtable 对象中的 write_fun 成员变量的值。

返回值表示执行的结果，0 表示成功；其他值表示失败。

第8章 其它注意事项

✍本章导读

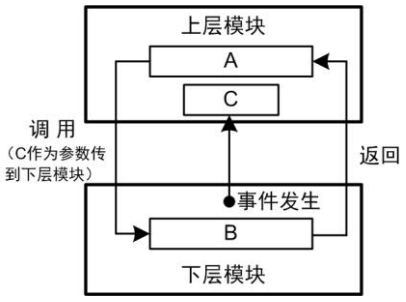
在实际应用中，部分用户可能因为软件设计问题造成最终的软件效率较低，甚至是不能运行，例如，为了设计简便，直接在按键回调函数中处理按键事件，在部分平台中，这种设计可能造成一系列问题。本章将介绍一些类似的需要用户特别注意的地方，以便用户设计出更加优良的程序。

8.1 不建议直接在按键回调函数中处理按键事件

8.1.1 原因分析

前面讲解了在几种平台中使用 ZLG72128 的方法，无论何种平台，对按键处理的大致思路是相同的，均使用了“回调机制”：实际按键处理程序由用户完成，用户将实际按键处理函数以回调函数的形式注册到系统之中，当有按键事件发生时，系统自动调用注册的回调函数，以完成按键的实际处理。

回调函数机制很好的满足了著名的好莱坞（Hollywood）扩展原则：“不要调用我，让我调用你”，当下层需要传递信息给上层时，则采用回调函数指针接口隔离变化。通过倒置依赖的接口所有权，创建一个更灵活、更持久和更易于修改的结构。



回调机制示意图详见图 8.1。在按键应用中，系统驱动为下层模块，应用代码为上层模块，按键处理函数为回调函数（即 C），应用代码（A）将回调函数的地址通过注册接口（B）传递到下层模块中，下层模块实现按键的检测，当检测到按键事件时，通过存储的地址自动调用回调函数，进而完成按键的处理。

通常情况下，按键回调函数的运行环境是中断环境。以 ZLG72128 为例，当 ZLG72128 检测到按键事件时，会通过 KEY_INT 引脚将按键事件通知到主控 MCU，主控 MCU 通常会使用 I/O 中断来接收此信号，以便及时发现按键事件。当中断产生时，表明有按键事件发生，进而读取键值，然后调用用户注册的回调函数通知用户 ，由此可见，“调用用户注册的回调函数以通知用户”这一过程实则是在中断环境中完成的。其它一些按键检测方式，如定时查询，几乎都是在中断（定时器中断）中完成的。

在嵌入式系统中，实时性是一项重要的指标，为了保证系统的实时性，通常都不建议在中断中耗费大量的时间（如等待某一事件、执行过多的处理、延时等操作），以避免影响任务的正常运行（中断占用时，主循环或主任务均得不到执行）或其它中断的产生（同等优先级或更低优先级的中断），降低系统的实时性。在绝大部分系统中，都要求中断处理应该尽可能快的结束。在实际应用中，应在主循环或主任务中执行按键的实际处理，而在按键回调函数中，仅通过某种机制通知到主循环或主任务即可。

8.1.2 解决办法

1. 使用标志进行处理

在裸机平台中，可以使用简单的标志进行处理，范例程序详见程序清单 8.1。

程序清单 8.1　使用简单的标志进行处理

```
1    #include "ametal.h"
2    #include "am_input.h"
3
4    static am_input_key_handler_t g_key_handler;      // 事件处理器实例定义，全局变量，确保一直有效
5
6    volatile int __g_key_code;
7    volatile int __g_key_flag = 0;
8
9    static void __input_key_proc(void *p_arg, int key_code, int key_state, int keep_time)
10   {
11       if (key_state == AM_INPUT_KEY_STATE_PRESSED) {              // 有键按下
12           __g_key_code = key_code;
13           __g_key_flag = 1;
14       }
15   }
16
17   int am_main (void)
18   {
19       int key_code;
20
21       am_input_key_handler_register(&g_key_handler, __input_key_proc, (void *)NULL);
22
23       while (1) {
24           if (__g_key_flag) {
25               key_code = __g_key_code;
26               __g_key_flag = 0;
27               if (key_code == KEY_1) {
28                   // 按键实际处理
29               }
30           }
31       }
32   }
```

程序中，增加了__g_key_flag 标志，按键回调函数中将该标志设置为 1，并将键值存储在__g_key_code 全局变量中。主循环中不断判断该标志，不为 0 时即可进行处理。

在程序清单 8.1 中，按键回调函数仅在按键按下时才将__g_key_flag 标志设置为 1，意味着仅处理按键按下事件，这是绝大部分应用的实际情况（不处理按键释放），若按键释放事件也需要处理，则可以在按键回调函数中将按键状态也存储到一个全局变量中，进而在主循环中根据按键状态作出不同的处理动作。

虽然这种处理方式使用起来非常简单，但是有诸多缺点，主要有以下两点：

- 需要额外的保护机制

程序清单 8.1 仅是一个示意性代码，实际中，若采用这种方式，由于__g_key_flag 标志在主循环和回调函数中均进行了修改（主程序置 0，回调函数置 1），为了避免冲突，应该

在修改全局变量的值时，增加中断锁，使修改过程不会被打断。

- 按键事件丢失

主程序中可能还会处理其它事务，若按键事件产生的频率过高，则按键事件很有可能因为不能及时处理而丢失（__g_key_code 还未处理时，被新到的按键事件覆盖），显然，这在部分可靠性要求很高的系统中，是无法容忍的。

2. 使用环形缓冲区进行处理

使用单个标志无法缓存键值，可能导致按键事件的丢失，为了避免出现这种情况，可以增加一个缓存，按键回调函数仅负责向缓存中添加键值，而主循环负责从缓存中取出键值进行处理，键值处理的顺序应该是“先按先处理”，即缓存中的数据应该“先进先出”，可以使用“队列”结构进行管理，简单地，在 AMetal 平台中，可以使用已经提供的“环形缓冲区”工具，范例程序详见程序清单 8.2。

程序清单 8.2　使用环形缓冲区进行处理

```
1   #include "ametal.h"
2   #include "am_input.h"
3   #include "am_rngbuf.h"
4
5   static am_input_key_handler_t   g_key_handler;                  // 事件处理器实例定义
6
7   static struct am_rngbuf         __g_key_rngbuf;                 // 定义环形缓冲区
8   static char                     __g_key_buf[20 * sizeof(int) + 1]; // 定义可以装载 20 个键值的缓存
9
10  static void __input_key_proc(void *p_arg, int key_code, int key_state, int keep_time)
11  {
12      if (key_state == AM_INPUT_KEY_STATE_PRESSED) {                  // 有键按下
13          am_rngbuf_put(&__g_key_rngbuf, (const char *)&key_code, sizeof(int));   // 将键值存入缓存
14      }
15  }
16
17  int am_main (void)
18  {
19      int key_code;
20
21      am_input_key_handler_register(&g_key_handler, __input_key_proc, (void *)NULL);
22
23      am_rngbuf_init(&__g_key_rngbuf, __g_key_buf, sizeof(__g_key_buf));
24
25      while (1) {
26          /* 环形缓冲区中存在键值，取出进一步处理 */
27          if (am_rngbuf_nbytes(&__g_key_rngbuf) >= sizeof(int)) {
28              am_rngbuf_get(&__g_key_rngbuf, (char *)&key_code, sizeof(int)); // 从缓存中取出键值
29              if (key_code == KEY_1) {
30                  // 按键实际处理
31              }
```

```
32                  }
33              }
34      }
```

程序中使用到了环形缓冲区，以 am_rngbuf_*开头的函数即为环形缓冲区相关的接口，各接口的简要说明详见表 8.1，更详细的说明可以参见《AMetal API 参考手册》。

表 8.1　环形缓冲区接口简介

函数原型	功能简介
int **am_rngbuf_init** (struct am_rngbuf *p_rb, char *p_buf, size_t size);	初始化一个环形缓冲区
size_t **am_rngbuf_put** (am_rngbuf_t rb, const char *p_buf, size_t nbytes);	向 ringbuf 中填入数据，p_buf 指向待填入数据的首地址，nbytes 指定写入数据的数据量（字节数）
size_t **am_rngbuf_get** (am_rngbuf_t rb, char *p_buf, size_t nbytes);	从 ringbuf 中取出数据。p_buf 指向数据缓存，用于存储取出的数据，nbytes 指定取出数据的数据量（字节数）
size_t **am_rngbuf_nbytes** (am_rngbuf_t rb);	获取 ringbuf 中已经装载的数据量（字节数）

下面对初始化环形缓冲区作一个简要的说明。在初始化之前，应完成 ringbuf 实例的定义和缓存空间的的定义。

- 实例定义：

struct am_rngbuf rngbuf;

- 缓存空间定义：

缓存空间即一段被用于环形缓冲区管理的内存，可以根据实际应用需要，使用 char 类型定义一段指定大小的内存空间，假定应用需要使用的内存空间大小为 64 字节，则缓存空间可以定义如下：

```
char buf[64+1];
```

若需存储 20 个 int 类型的键值，则缓存空间可以定义如下：

```
char buf[20 * sizeof(int) + 1];
```

注意，在定义缓存空间时，定义的大小应该比实际使用的大小多出 1 字节，多出的单字节空间在环形缓冲区内部使用，用户无需关心。

完成实例及缓存空间的定义后，即可调用 am_rngbuf_init()函数完成环形缓冲区的初始化，初始化时，各参数的值应设置为：p_rb 指向定义的 ringbuf 实例（即&rngbuf）；p_buf 指向定义的缓存空间（即 buf）；size 为缓存空间的大小，实际可以使用的大小为 size-1。初始化语句即为：

```
am_rngbuf_init(&rngbuf, buf, sizeof(buf));
```

完成环形缓冲区的初始化之后，即可调用其它功能接口操作环形缓冲区，各个功能接口的第一个参数为 rb，其本质上就是指向 ringbuf 实例的指针，即&rngbuf，am_rngbuf_t 类型等同于 struct am_rngbuf 指针类型：

```
typedef struct am_rngbuf *am_rngbuf_t;
```

在程序清单 8.2 中，按键回调函数仅在按键按下时，才会将按键编码装载到了环形缓

冲区中，实际中，若对按键的状态（key_state）或按键保持的时间（keep_time）也感兴趣，则可以将这些信息一并写入到环形缓冲区中。

3. 使用消息队列进行处理

上述程序是以 AMetal 平台为例，若在 AWorks 平台中，由于 AWorks 内置对实时内核的支持，因此，可以使用“消息队列”来解决，即在按键回调函数中发送消息，在任务循环中获取消息并处理，范例程序详见程序清单 8.3。

程序清单 8.3　使用消息队列进行处理

```
1   #include "aworks.h"
2   #include "aw_msgq.h"
3   #include "aw_input.h"
4
5   AW_MSGQ_DECL_STATIC(msgq_key, 20, sizeof(int));        // 定义消息队列实体，可存储 20 条消息
6
7   static void __key_process (aw_input_event_t *p_input_data, void *p_usr_data)
8   {
9       if (p_input_data->ev_type == AW_INPUT_EV_KEY) {                    // 仅处理按键事件
10          aw_input_key_data_t *p_data = (aw_input_key_data_t *)p_input_data;
11
12          if (p_data->key_state != 0) {                                  // 仅处理按键按下事件
13              AW_MSGQ_SEND(msgq_key,
14                           &p_data->key_code,
15                           sizeof(int),
16                           AW_MSGQ_NO_WAIT,
17                           AW_MSGQ_PRI_NORMAL);
18          }
19      }
20  }
21
22  int aw_main()
23  {
24      int                         err;
25      int                         key_code;
26      static aw_input_handler_t   key_handler;
27
28      AW_MSGQ_INIT(msgq_key, 10, sizeof(__mail_key_info_t), AW_MSGQ_Q_FIFO);
29      aw_input_handler_register(&key_handler, __key_process, NULL);
30
31      while(1) {
32          err = AW_MSGQ_RECEIVE(msgq_key,
33                                &key_code,
34                                sizeof(int),
35                                AW_MSGQ_WAIT_FOREVER);
36          if (err == AW_OK) {
```

```
37                      if (key_code == KEY_0) {
38                          // 按键处理
39                      }
40                  }
41              }
42      }
```

程序中使用到了消息队列，以 AW_MSGQ_*开头的宏即为消息队列相关的接口，简要说明详见表 8.2，更详细的说明可以参见《AWorks API 参考手册》。

表 8.2　消息队列相关的宏（aw_msgq.h）

宏原型	功能简介
AW_MSGQ_INIT(msgq, msg_num, msg_size, options)	初始化消息队列
AW_MSGQ_RECEIVE(msgq, p_buf, nbytes, timeout)	从消息队列中获取一条消息
AW_MSGQ_SEND(msgq, p_buf, nbytes, timeout, priority)	发送一条消息到消息队列
AW_MSGQ_TERMINATE(msgq)	终止消息队列

下面对初始化消息队列作简要的说明。在初始化之前，应完成消息队列实例的定义和缓存空间的的定义。它们的定义可以直接通过 AW_MSGQ_DECL_STATIC()宏完成，该宏的原型如下：

```
AW_MSGQ_DECL_STATIC(msgq, msg_num, msg_size)
```

其中，参数 msgq 为消息队列实例的标识名。msg_num 和 msg_size 用于分配存储消息的空间，msg_num 表示消息的最大条数，msg_size 表示每条消息的大小（字节数）。用于存储消息的总内存大小即为：msg_num×msg_size。例如，使用 AW_MSGQ_DECL_STATIC()定义一个标识名为 msgq_test 的消息队列实例，且消息的最大数目为 20，每条消息为一个 int 类型数据：

```
AW_MSGQ_DECL (msgq_test, 20, sizeof(int));                    // 20 条消息的空间，每条消息为 int 类型数据
```

完成消息队列的定义后，即可使用 AW_MSGQ_INIT()对其进行初始化操作，该宏的原型为：

```
AW_MSGQ_INIT(msgq, msg_num, msg_size, options)
```

其中，msgq 为 AW_MSGQ_DECL_STATIC()定义的消息队列。msg_num 表示消息队列可以存储的消息条数，其值必须与定义消息队列实体时的 msg_num 相同。msg_size 表示每条消息的大小（字节数），其值必须与定义消息队列实体时的 msg_size 相同。options 为消息队列的选项，其决定了阻塞于此消息队列（等待消息中）的任务的排队方式，可以按照任务优先级（# AW_MSGQ_Q_PRIORITY）或先进先出（# AW_MSGQ_Q_FIFO）的顺序排队，本例中，应采用先进先出的排队方式，即：

```
1    AW_MSGQ_INIT (msgq_test, 20, sizeof(int), AW_MSGQ_Q_FIFO);
```

完成初始化后，可以调用其它接口操作消息队列，各个功能接口的第一个参数为 msgq，其为 AW_MSGQ_DECL_STATIC()定义的消息队列。

在程序清单 8.3 中，按键回调函数仅在按键按下时才向消息队列中发送消息，意味着仅处理按键按下事件，这是绝大部分应用的实际情况（不处理按键释放），若按键释放事件也需要处理，则在回调函数中，可以将按键编码、按键状态等信息打包到一条消息中一并发送。