User mannual

M2378-CFNS20

ARM 嵌入式工业控制模块

V1.01 Date: 2008/04/16

产品软件手册

类别	内容
关键词	M2378-CFNS20I、MiniARM、嵌入式工控模块
摘　要	讲解 M2378-CFNS20I 软件资源的使用方法。

广州致远电子有限公司

ZHIYUAN ELECTRONICS

修订历史

版本	日期	原因
V1.00	200803/27	创建文档
V1.01	200804/16	修改文档格式，修改几处引用错误

销售与服务网络（一）

广州周立功单片机发展有限公司

地址：广州市天河北路 689 号光大银行大厦 12 楼 F4 邮编：510630
电话：(020)38730916 38730917 38730972 38730976 38730977
传真：(020)38730925
网址：www.zlgmcu.com

广州专卖店

地址：广州市天河区新赛格电子城 203-204 室
电话：(020)87578634 87569917
传真：(020)87578842

北京周立功

地址：北京市海淀区知春路 113 号银网中心 712 室（中发电子市场斜对面）
电话：(010)62536178 62536179 82628073
传真：(010)82614433

杭州周立功

地址：杭州市登云路 428 号浙江时代电子市场 205 号
电话：(0571)88009205 88009932 88009933
传真：(0571)88009204

深圳周立功

地址：深圳市深南中路 2070 号电子科技大厦 A 座 24 楼 2403 室
电话：(0755)83781788（5 线）
传真：(0755)83793285

上海周立功

地址：上海市北京东路 668 号科技京城东座 7E 室
电话：(021)53083452 53083453 53083496
传真：(021)53083491

南京周立功

地址：南京市珠江路 280 号珠江大厦 2006 室
电话：(025)83613221 83613271 83603500
传真：(025)83613271

重庆周立功

地址：重庆市石桥铺科园一路二号大西洋国际大厦（赛格电子市场）1611 室
电话：(023)68796438 68796439
传真：(023)68796439

成都周立功

地址：成都市一环路南二段 1 号数码同人港 401 室（磨子桥立交西北角）
电话：(028)85439836 85437446
传真：(028)85437896

武汉周立功

地址：武汉市洪山区广埠屯珞瑜路 158 号 12128 室（华中电脑数码市场）
电话：(027)87168497 87168297 87168397
传真：(027)87163755

西安办事处

地址：西安市长安北路 54 号太平洋大厦 1201 室
电话：(029)87881296 83063000 87881295
传真：(029)87880865

销售与服务网络（二）

广州致远电子有限公司

地址：广州市天河区车陂路黄洲工业区 3 栋 2 楼
邮编：510660
传真：(020)38601859
网址：www.embedtools.com （嵌入式系统事业部）
www.embedcontrol.com （工控网络事业部）
www.ecardsys.com （楼宇自动化事业部）

技术支持：

CAN-bus：

电话：(020)22644381 22644382 22644253
邮箱：can.support@embedcontrol.com

MiniARM：

电话：(020)28872684
邮箱：miniarm.support@embedtools.com

编程器：

电话：(020)38681856 28872449
邮箱：programmer@embedtools.com

ARM 嵌入式系统：

电话：(020)28872347 28872377 22644383 22644384
邮箱：ARM.Support@zlgmcu.com

iCAN 及模块：

电话：(020)28872344 22644373
邮箱：ican@embedcontrol.com

以太网及无线：

电话：(02 0)22644385 22644386
邮箱：wireless@embedcontrol.com
Ethernet.support@embedcontrol.com

分析仪器：

电话：(020)22644375 28872624 28872345
邮箱：tools@embedtools.com

楼宇自动化：

电话：(020)22644376 22644389
邮箱：mjs.support@ecardsys.com
mifare.support@zlgmcu.com

销售：

电话：(020)22644249 22644399 28872524 28872342
28872349 28872569 28872573

维修：

电话：(020)22644245

目 录

1. LPC2300 基础驱动程序库使用

1.1 LPC2300 基础驱动程序库

嵌入式系统中硬件与软件是一个有机的结合体，驱动程序是软硬件协调的交点，也是任何嵌入式系统中最重要的组成部分之一，底层驱动程序的可靠性直接影响到系统的运行效率和稳定性。

开发高效率、性能稳定的驱动程序，不仅要求工程师具有良好的程序编写水平，同时还要求对底层硬件接口技术非常熟悉。在面对一大堆的寄存器说明时，往往使很多平时专注于应用程序开发的工程师对于嵌入式系统望而却步。即使是有过底层开发经验的程序员，也往往希望能够有更轻松的开发方式出现，以减少其浪费在驱动程序编写上的时间。

LPC2300 系列 ARM（LPC2364/2366/2368/2378）是 NXP 半导体最新推出的 ARM7 内核微控制器，专为工业通信连接而设计，其内部具有非常丰富的外设，包括多路 UART、多路 CAN2.0B 控制器、USB 控制器、SSP、定时器等等，同时每种外设又具有很多灵活实用的功能，使其很容易嵌入各种工业应用场合，但功能丰富的同时也带来了系统开发尤其是驱动开发的复杂。

为了有效帮助工程师减少产品开发“0”阶段的时间，这里提供一种基于 API 方式操作底层硬件的开发方法。通过对 LPC2300 系列 ARM 片内外设寄存器的封装，生成一套用于 LPC2300 系列 ARM 的底层驱动库。

使用 LPC2300 驱动库开发，设计人员只需要调用 API 接口函数即可使用各种外设，无需理会其寄存器有多么复杂。通过封装的 API 函数，大大减少了与片内设备通信时所需的代码量。维护工作变得简单，驱动库使用了风格尽量统一的函数接口，使得程序的可移植性也大大提高，如表 1.1 所示。

表 1.1 开发模式对比

	使用驱动库	自行开发驱动
开发周期	短	长
代码量	小	大
维护工作	简单	复杂
移植	方便	难度大
开发难度	低	高

1.1.1 概述

ZLG/Driver2300 驱动函数库配合 LPC2300 系列 ARM 使用，为用户提供了大部分片内外设的底层硬件驱动程序，但不包括 USB、SD 接口等，支持的常用部件，如

表 1.2 所示。

表 1.2 支持外设列表

	外设名称	说明
1	GPIO	支持全部 P0、P1、P2、P3、P4 端口，支持高速 GPIO 或低速 GPIO 操作。
2	中断控制器	在特权模式统一管理中断，提供 FIQ、IRQ 的所有中断设置接口
3	定时器	提供定时、计数、捕获外部事件、匹配输出等功能，当事件发生时均可以对是否产生中断、复位或停止 TC 等操作进行组合配置
4	PWM	6 路单边沿或 3 路双边沿 PWM 波输出，PWM 控制器可作为定时器使用
5	外部中断	4 个外部中断输入设置函数及中断处理函数，可任意配置触发极性和边沿
6	UART	支持 4 个 UART 口的通用串行通信功能及自动波特率设置，同时提供 UART1 Modem 通信控制、UART3 IrDA 通信控制和 4 个串口的 RS485 通信控制，并支持最多 6 个用户外扩 UART
7	I^2C	支持 I2C0、I2C1 和 I2C2 的主机操作，最高速率 400Kbps
8	SSP/SPI	支持 SSP0、SSP1 接口的 SPI 主机操作，以及 SPI0 主、从机操作
9	ADC	提供片内 10 位 ADC 接口函数
10	DAC	提供片内 10 位 DAC 接口函数
11	系统时钟	通过 PLLInit()函数完成 CPU、外设等所有时钟设置
12	RTC	提供 RTC 设置、时间读取、中断响应接口函数
13	功率控制	支持 3 种低功耗模式：空闲模式、睡眠模式和掉电模式，多个中断源唤醒
14	外部存储器控制器	可单独配置 BANK0~BANK1，读写时序，提供读写操作函数
15	看门狗	操作片内看门狗控制 CPU 复位或者中断
16	CAN	支持 CAN1、CAN2 数据通信接口函数，带硬件验收滤波功能

另外，USB、SD 接口等提供专门的软件包支持。使用驱动库函数，工程师不必关心硬件细节，只要懂 C 语言就可直接通过 API 函数操作 LPC2300 系列 ARM 开发产品。

注：本函数库可在 LPC2300 系列 ARM 的多种开发平台上使用，本章示范例程以 SmartARM2300 开发板为平台（M2378-CFNS20 模块）进行测试。暂时不支持其它硬件平台的应用，后续版本将对不同平台进行测试以解决平台兼容问题。

1.2 引脚连接

1.2.1 函数说明

“LPC2300PinCfg.c”文件提供了函数 PinInit()，用于初始化引脚连接。

1.2.2 使用方法

在使用引脚的相应功能之前，首先需要将文件“LPC2300PinCfg.c”和“LPC2300PinCfg.h”添加到 LPC2300 工程中；然后在“LPC2300PinCfg.h”中正确配置引脚；最后调用“PinInit()”函数使能引脚功能配置即可使用。

1.3 系统时钟配置

1.3.1 函数说明

只有一个时钟配置的接口函数 PLLInit()。

1.3.2 使用方法

进行正确的参数设置，再直接调用此函数即可。

1.3.3 示范例程

具体使用方法详见功率控制小节的示范例程。

1.4 功率控制

1.4.1 函数说明

功率控制只包含一个函数 PowerSetMode()。器件复位后的功率设置如表 1.3 所示。

表 1.3 LPC2300 外设功率设置位

PCONP（DevSel）	对应外设	复位值	PCONP（DevSel）	对应外设	复位值
0	保留位	0	15：18	保留位	NA
1	Timer0	1	19	I^2C1	1
2	Timer1	1	20	保留	0
3	UART0	1	21	SSP0	1
4	UART1	1	22	Timer2	0
5	PWM0	1	23	Timer3	0
6	PWM1	1	24	UART2	0
7	I^2C0	1	25	UART3	0
8	SPI0	1	26	I^2C 2	1
9	RTC	1	27	I^2S	0
10	SSP1	1	28	SD 卡控制器	0
11	EMC	1	29	GPDMA	0
12	ADC	1	30	以太网控制器	0
13	CAN1	1	31	USB	0
14	CAN2	1			

注：复位值为 0 表示关闭对应外设；为 1 表示打开对应外设；NA 则不确定，一般设置为 0。

1.4.2 使用方法

使用函数 PowerSetMode()可进行功率模式设置，亦可单独关闭某个外设从而达到节电的目的。以下对两者的使用方法进行详细说明。

1 系统功率模式设置

如果系统经过设置进入低功耗模式，则必须正确设置唤醒源才能使处理器恢复正常工作状态。唤醒后的 CPU 不会自动配置 PLL，需要调用 PLLInit()函数重新进行时钟配置。具体使用方法如程序清单 1.1 所示。

程序清单 1.1 功率模式设置

```
char    PowModeArg[] = "Mode＝1 WakeUpSrc=0 BodMode=1 BodRstEn＝0";
char    pllArg[] = "Fosc=12000000 CclkRatio=4 PclkRatio=2 ClkSrc=1 UsbEn=1";
PowerSetMode(PCON0, SET_PCONMODE, PowModeArg);
                        // 进入掉电模式，外部中断 0 唤醒，掉电检测使能，掉电复位禁止

// 唤醒后执行的第一行代码如下：
PLLInit(PLL0, pllArg, NULL);                                    // 重新配置系统时钟
```

在 SET_PCONMODE 命令参数设置完成后，还可使用 SET_WAKEUP_SRC 命令设置其他的唤醒源。可同时设置多个唤醒中断源来唤醒 CPU，但每调用此函数一次只能设置一个。

2 外设打开/关闭设置

LPC2300 系列 ARM 的某些外设默认状态为打开，某些则默认为关闭，参考表 1.3。可调用 PowerSetMode()关闭一些不用的外设以节能，如程序清单 1.2 所示。

程序清单 1.2 关闭外设

```
char PowModeArg[] = " DevSel=0xffe87ffa";                  // 定时器 1 关闭，其他外设打开
PowerSetMode(PCON0, SET_PCONPMODE, PowModeArg);
```

DevSel 参数的值要由用户自行计算，使用不是很方便。为此驱动库提供了宏定义，使用方法请参考后面的范例 2。

1.4.3 示范例程

1 系统功率模式设置

设置 CPU 进入掉电模式，掉电检测、复位使能，使用外部中断 1 来唤醒 CPU，具体的程序如程序清单 1.3 所示。

程序清单 1.3 CPU 进入掉电模式程序

文件 1：“LPC2300PinCfg.h”

```
#define     P2_11_FNUC          P2_11_EINT1                     // 外部中断 1 输入
```

文件 2：“main.c”

```
void TASK1(void *pdata)
{
    // 系统功率控制参数，掉电模式，外部中断 1 唤醒，掉电检测使能，掉电检测复位使能
    char   powerArg[] = "Mode=1 WakeupSrc=1 BodMode=1 BodRstEn=1";

    char   pllArg[80];                                          // PLL 配置参数缓存
    char   EintWakeMode[] = "WakeUp=1";
    char   EintIntMode[] = "IntMode=0";
    pdata = pdata;

    // 通过 sprintf()函数来获取 PLL 初始化参数，可以避免人为计算出错
    // 使用外部振荡器，其频率为宏 Fosc 定义的值：12000000，使能 USB
    // 宏 Fosc、Fcclk、Fpclk 均在 config.h 文件中定义，请不要改动
    sprintf(pllArg,
            "Fosc=%u CclkRatio=%u PclkRatio=%u ClkSrc=1 UsbEn=1",
             Fosc, (Fcclk / Fosc), (Fpclk / (Fcclk / 4)));

           // 外部中断 1 使能唤醒功能，若功率控制参数中使能了外部中断 1 唤醒，则可省略
           // EINTSetMode(EINT1, SET_WAKEMODE, EintWakeMode);

           EINTSetMode(EINT1, SET_INTMODE, EintIntMode);  // 下降沿中断

           // 唤醒后产生中断，如果不设置中断服务函数，则只唤醒不产生中断
           // SetVICIRQ(EINT1_IRQ_CHN, 7, (uint32)EINT1_ISR);
    while (1)
    {
        PowerSetMode(PCON0, SET_PCONMODE, powerArg);  // 进入掉电模式

     /* 如果设置了中断服务函数，不需要 ExINClr()函数，下次也能进入调电模式；如果不
        设置中断服务函数，只有唤醒后调用了 ExINClr()函数，下次才能进入调电模式    */
        EINTClr(EINT1);

        PLLInit(PLL0, pllArg, NULL);                            // 唤醒后要配置系统时钟
        GpioClr(P1_27);                                         // 控制蜂鸣器鸣叫一声
        OSTimeDly(OS_TICKS_PER_SEC / 2);
        GpioSet(P1_27);
    }
}
```

使用杜邦线把 JP1 上的 P2.11 引脚和 KEY1 相连，BEEP 和 P1.27 使用跳线器短接。下载程序至 SmartARM2300 开发板，断电，取下 JTAG，重开电源运行。这时蜂鸣器不鸣叫，说明进入了掉电模式。按 KEY1 键，蜂鸣器会鸣叫一声，说明 CPU 被唤醒并执行了蜂鸣器控制代码，之后又进入掉电模式。

2 外设打开/关闭设置

为方便用户使用，本驱动库将 DevSel 的位与外设的对应关系进行了宏定义，具体见 power.h 头文件。下面的例程将演示如何使用这些宏定义来进行外设功率控制设置。如设置定时器 1、UART0 和 UART1 关闭，UART3 打开，其他部件使用默认设置，参数 DevSel 的值可由宏 PC_CUSTOM 获得，具体设置如程序清单 1.4 所示。

程序清单 1.4 外设功率控制设置

文件 1：“power.h”

```
#define    PCTIM1      (OFF << 2)          //  定时器 1
#define    PCUART0     (OFF << 3)          //  UART0
#define    PCUART1     (OFF << 4)          //  UART1
#define    PCUART3     (ON << 25)          //  UART3
```

文件 2：“main.c”

```
void TASK1 (void *pdata)
{
    char   pconpArg[32];                                  // 缓存外设功率设置参数
    pdata = pdata;
    sprintf(pconpArg, “DevSel=%u”, PC_CUSTOM);            // 获取所需的外设功率控制参数
    PowerSetMode(PCON0, SET_PCONPMODE, pconpArg);         // 设置外设功率
    while (1) {
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

1.5 外部存储器控制器

1.5.1 函数说明

本驱动库对 EMC 的操作提供了 3 个接口函数，它们分别是外部存储器配置函数 EMCInit()、对外部存储器写函数 EMCWrite()和对外部存储器读函数 EMCRead()。

1.5.2 使用方法

1 初始化 MEC

需根据所使用的外部存储器的读写时序正确配置外部存储器控制器 EMC 的初始化参数，一些不关心的参数可使用默认值。调用 EMCInit()函数完成 EMC 的初始化。

2 读写操作

对存储器进行读写操作，调用 EMCWrite()或 EMCRead()函数。

1.5.3 示范例程

SmartARM2300 核心板在 BANK0 上外扩了一片 128KB 容量的 IS63LV1024。通过 BANK0 可以访问其低 64KB 空间。下面的范例向 IS63LV1024 的地址 0x00 写入 5 个字节数据，再读回判断是否正确。具体的程序如程序清单 1.5 所示。

程序清单 1.5 EMC 实验程序

文件 1：main.c

```
void TASK1 (void *pdata)
{
    uint8 wdata[5] = {0xaa, 0xa5, 0x5a, 0xfe, 0x55};          // 使用 8 位宽的数据类型
    uint8 rdata[5];                                           // 使用 8 位宽的数据类型
    // EMC 初始化字符串参数，小端模式，片选低电平有效，读等待 32 个 CCLK 时钟周期，
    // 写等待 33 个 CCLK 时钟周期，读写转换延时 16 个 CCLK 时钟周期
    char emcArg[] = "Endian=0 CsPol=0 ReadDly=0x1f WriteDly=0x1f TurnDly=0xf";
    uint32 i;
    pdata = pdata;

    EMCInit(BANK0, emcArg, NULL);                             // EMC 初始化

    while (1) {
        EMCWrite(BANK0, DATA_8BIT, 0x0, wdata, 5);            // 写入 5 个字节数据
        OSTimeDly(1);

        EMCRead(BANK0, DATA_8BIT, 0x0, rdata, 5);             // 读取写入的 5 个数据
        for(i = 0; i < 5; i++)                                // 判断读取的数据和写入的数据是否相同
        {
```

```
            if (wdata[i] != rdata[i]) break;
        }

        if (i == 5) {                                   // 如果相同蜂鸣器鸣叫一声
            GpioClr(P1_27);
              OSTimeDly(OS_TICKS_PER_SEC / 5);
            GpioSet(P1_27);
        }
        else{                                                   // 不同则一直鸣叫蜂鸣器
            GpioClr(P1_27);
            while (1) {
                OSTimeDly(OS_TICKS_PER_SEC);
            }
        }
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

全速运行程序，若听到蜂鸣器有规律的鸣叫，则表示数据读写正确；若蜂鸣器一直鸣叫，则表示数据读写有误，出错的原因可能是总线时序配置不正确。

1.6 中断管理

1.6.1 函数说明

本驱动库总共提供了 7 个函数用于中断设置，其中 SetVICIRQ()、FreeVICIRQ()、GetVICIRQState()、DisableVICIRQ()、ReEnableVICIRQ()五个函数用于 IRQ 中断模式的管理；SetVICFIQ()、FreeVICFIQ()两个函数管理 FIQ 中断资源。

1.6.2 使用方法

中断的使用分为FIQ和IRQ两种，分别使用SetVICIRQ()和SetVICFIQ()去设置并使能相应中断。在调用这两个函数之前，应该已设置好对应通道的属性。

例如：设置定时器1为IRQ中断，优先级为2，指向中断响应函数TIMER1_ISR()。应先打开定时器1并设置好工作方式，然后调用函数SetVICIRQ(TIMER1_IRQ_CHN,2,(uint32) TIMER1_ISR)即可使用。具体的使用范例会在后继的章节中体现，在此不再单独涉及。

注：中断的设置应特别小心，错误的操作会导致死机!

1.7 GPIO 应用

1.7.1 函数说明

P0 和 P1 端口有高速和低速两种模式，P2、P3、P4 端口硬件只支持高速模式。使用 GpioSpeedHigh()和 GpioSpeedLow()来设置 P0 和 P1 端口速度模式选择。

GPIO 的操作函数分为置位、清零、读取、取反 4 类，无论高低速均可以通过 GpioSet()、GpioClr()、GpioGet()和 GpioCpl()来完成对所有 GPIO 端口的设置与操作。

1.7.2 使用方法

1 引脚连接

确定需要将某个引脚作为 GPIO 以后，首先应该配置引脚连接，如程序清单 1.6 所示。同时还可以设置引脚内部是否连接上/下拉电阻，请参照引脚连接小节。在系统初始化时，调用“PinInit()”函数使能引脚连接配置。

程序清单 1.6 引脚 P0.00 功能配置

```
// P0.00
#define    P0_00_GPIO      0x00        // GPIO
#define    P0_00_RD1       0x01        // CAN-1 控制器接收引脚
#define    P0_00_TXD3      0x02        // UART3 发送引脚
#define    P0_00_SDA1      0x03        // I2C-1 数据线（开漏）
```

```
#define     P0_00_FNUC      P0_00_GPIO      // P0.00 设为 GPIO 引脚
```

2 访问模式

P0、P1 默认为慢速 GPIO 端口，P2、P3、P4 硬件上只支持高速模式，如果需要将 P0、P1 设置为高速模式，就应该使用 GpioSpeedHigh()函数，如程序清单 1.7 所示。

程序清单 1.7 高速访问模式

```
GpioSpeedHigh();                    // 设置 P0 和 P1 口为高速访问模式
                                    // P0 与 P1 口只能同时配置为高速或低速
```

3 操作端口

完成引脚配置后即可对相应的 GPIO 端口进行置位、清零、取反等操作，具体操作请参照函数说明的函数表格中范例一栏。

1.7.3 示范例程

蜂鸣器是工业应用中非常广泛的指示元件，这里演示如何利用 LPC2300 系列 ARM 的 GPIO 端口控制直流蜂鸣器鸣叫。由于蜂鸣器的工作电流要大于 LPC2300 系列 ARM 的 GPIO 驱动输出电流，故控制蜂鸣器时通常需要加入一级三极管来驱动，如图 1.1 所示驱动电路即为 SmartARM2300 工控平台上使用的蜂鸣器电路。此处蜂鸣器驱动电压为 3.3V，由 P1.27 控制，通过跳线 JP1-9 选择连接。

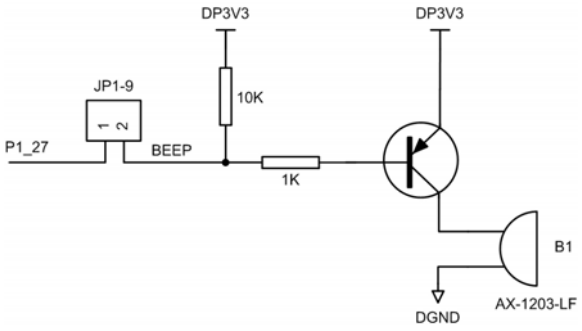

图 1.1 蜂鸣器控制电路

当 P1.27 引脚输出低电平时，三极管饱和导通，蜂鸣器鸣叫；反之，当 P1.27 引脚输出高电平时，三极管 Q4 截止，蜂鸣器停止鸣叫。

1 低速模式控制

使用 P1.27 低速模式控制蜂鸣器的代码如程序清单 1.8 所示。

程序清单 1.8 GPIO 端口控制蜂鸣器例程（1）

文件 1：“LPC2300PinCfg.h”

```
#define     P1_27_FNUC      P1_27_GPIO      // P1.27 配置为 GPIO
```

文件 2：“main.c”

```
……
PinInit();                                  // 引脚初始化
……
void TASK1(void *pdata)
{
    pdata = pdata;
    GpioSpeedLow();                         // P0、P1 设置为低速访问模式
    while(1) {
            GpioSet(P1_27);                 // 控制蜂鸣器鸣叫
            OSTimeDly(OS_TICKS_PER_SEC / 4);
            GpioClr(P1_27);
            OSTimeDly(OS_TICKS_PER_SEC / 4);
    }
}
```

2 高速模式控制

使用 P1.27 高速模式控制蜂鸣器的代码如程序清单 1.9 所示。

程序清单 1.9 GPIO 端口控制蜂鸣器例程（2）

文件 1：“LPC2300PinCfg.h”

```
#define          P1_27_FNUC          P1_27_GPIO          // P1.27 配置为 GPIO
```

文件 2：“main.c”

```
……
PinInit();                                              // 引脚初始化
……
void TASK1(void *pdata)
{
    pdata = pdata;

    GpioSpeedHigh();                                    // P0、P1 设置为高速访问模式
    while(1) {
        GpioSet(P1_27);                                 // 控制蜂鸣器鸣叫
        OSTimeDly(OS_TICKS_PER_SEC / 4);
        GpioClr(P1_27);
        OSTimeDly(OS_TICKS_PER_SEC / 4);
    }
}
```

1.8 定时器

1.8.1 函数说明

定时器（Timer）驱动库提供 8 个用户 API 函数，分别为 TimerInit()、TimerSetMode()、TimerStart()、TimerStop()、TimerReset()、TimerGetTc()、TimerGetCR()、TimerISR()主要完成对定时器 1、2 或 3 的初始化、模式设定、启动、停止、重启以及 TC 值或捕获值的获取等功能。

1.8.2 使用方法

1 连接引脚

使用定时器时，如需使用捕获或匹配输出功能，首先应该配置引脚连接，如程序清单 1.10 所示。同时还可以设置引脚内部是否连接上/下拉电阻。在系统初始化时，调用“PinInit()”函数使能引脚连接配置。

程序清单 1.10 引脚功能配置

```
// P0.04
#define      P0_04_FNUC          P0_04_CAP20     // P0.4 设为定时器 2 的捕获输入引脚
```

2 接口初始化

初始化主要是对定时器的接口参数以及中断等的设置，如程序清单 1.11 所示。在使用字符串进行参数传入时，可以只输入部分参数，未输入的参数将忽略或被设为默认值。

程序清单 1.11 定时器初始化

```
char TimerArg[ ] = "Mode=0";
char TimerModeArg [ ] = "Channel=0 Time=5000 ActCtrl =2 OutCtrl =2";

TimerInit(TIMER1, TimerArg, NULL);              // 定时器 1 初始化为定时模式：Mode=0;
TimerSetMode(TIMER1, SET_MATMODE, TimerModeArg);
                                                /*   设置定时器 1 的工作模式为匹配输出模式
                                                     Channel=0，匹配输出通道设置为 0
                                                     Time=5000，匹配时间为 5ms
                                                     ActCtrl=2，匹配后将 T1TC 复位
```

```
                                        OutCtrl=2，匹配后使对应外部匹配输出反 */
```

3 中断设置

如果在接口初始化中设置为产生中断，则在此必须调用函数 SetVICIRQ()或 SetVICFIQ()进行中断初始化和使能，参见程序清单 1.12；如没有使用中断，则可省略此步。

程序清单 1.12 中断设置

```
SetVICIRQ(5, 2, (uint32)TIMER1_ISR);        // 设置定时器 1 为 IRQ 中断，中断优先级 2,
                                            // 中断服务响应函数为 TIMER1_ISR
```

或

```
SetVICFIQ(5);                               // 设置定时器 1 为 FIQ 中断
```

4 启动定时器

完成初始化后即可使用函数 TimerStart()启动定时器。参见程序清单 1.13。

程序清单 1.13 启动定时器

```
TimerStart(TIMER1, NULL);                   // 启动定时器
```

1.8.3 示范例程

LPC2300 系列 ARM 的定时/计数器主要有 2 个模式（定时模式和计数模式）、6 大功能（每个模式都具有输入捕获和匹。配输出 2 个功能，加之各自具有的定时功能和计数功能），其中，定时器模式下的定时、匹配输出和输入捕获功能及计数器模式下的外部计数功能是常用的 4 个功能，以下几节分别对这几个功能，结合驱动库的使用方法进行详细说明。

注：示范例程全部以 SmartARM2300 开发板为平台进行调试。

1 定时功能

LPC2300 系列 ARM 的定时器最基本的应用是定时。以下用定时器 1 的定时功能，采用 IRQ 中断方式处理，每隔 0.5 秒取反蜂鸣器控制管脚；其中，伴随的硬件动作是复位 TC 并产生中断，匹配无输出。因为无匹配输出，所以引脚连接配置可省略。定时器 1 的具体功能配置见程序清单 1.14。

程序清单 1.14 定时器 1 定时功能配置

文件 1：“LPC2300PinCfg.h”

```
#define     P1_27_FNUC          P1_27_ GPIO          // P1.27 引脚设置为 GPIO
```

文件 2：“main.c”

```
……
PinInit();
……

void TASK0 (void *pdata)
{
    char    TimerArg[ ] = "Mode=0";
    char    TimerModeArg[ ] = "Channel=0 Time=500000 ActCtrl=3 OutCtrl=2";
    pdata = pdata;

    TimerInit(TIMER1, TimerArg, NULL);          // 定时器 1 初始化为定时模式：Mode=0;
    TimerSetMode(TIMER1, SET_TIMERMODE, TimerModeArg);
                                                /* 设置定时器 1 的工作模式为定时模式
                                                   Channel=0，选择通道 0
                                                   Time=500000，0.5s 定时
                                                   ActCtrl=3，匹配复位 T1TC 并产生中断  */
    SetVICIRQ(TIMER1_IRQ_CHN, 2, (uint32)Time1ISR);
                                                // 设置定时器 1 中断优先级为 2
     TimerStart(TIMER1, NULL);                  // 启动定时器

    while(1)
    {
```

```
        OSTimeDly(OS_TICKS_PER_SEC / 2);
    }
}
```

由于已经在“Timer.h”文件中为用户提供了中断服务函数，中断处理代码只需在函数 TIMER1_ISR 中添加，具体见程序清单 1.15。

程序清单 1.15　定时器 1 中断处理

```
void TIMER1_ISR(void)
{
        //------------------添加用户处理代码----------------------
        GpioCpl(P1_27);                                 // 取反 BEEP 管脚
        //--------------------------------------------------------------
        TimerISR(TIMER1);                               // 清除中断标志
        VICVectAddr = 0x00;                             // 中断处理结束
}
```

将 SmartARM2300 工控开发板 JP1 的 P1.27 和 BEEP 短接。编译程序进入 AXD，全速运行程序，可以观察到蜂鸣器每隔 0.5 秒响一声。用户可以尝试修改定时的时间长度来观察现象。

2　匹配输出功能

LPC2300 系列 ARM 的定时器另外一个重要的功能是匹配输出，可设置为定时一段时间，时间到达时硬件自动置位、清零或匹配输出。

现以定时器 2 在定时模式下匹配输出 2.5Hz 的方波（见程序清单 1.16）为例，具体演示本部分驱动库的使用方法。输出波形可使用蜂鸣器监听，亦可使用示波器观察。

程序清单 1.16　定时器 2 匹配输出功能配置

文件 1：“LPC2300PinCfg.h”

```
#define     P0_07_FNUC          P0_07_MAT21// P0.7 引脚设置为定时器 2 匹配输出通道 1
```

文件 2：“main.c”

```
……
PinInit();                                              // 引脚初始化
……
void TASK0 (void *pdata)
{
    char    TimerArg[ ] = "Mode=0";
    char    TimerModeArg [ ] = "Channel=1 Time=200000 ActCtrl=2 OutCtrl=2";
    pdata = pdata;
    TimerInit(TIMER2, TimerArg, NULL);                  // 定时器 2 初始化为定时模式：Mode=0;
    TimerSetMode(TIMER2, SET_MATMODE, TimerModeArg);
                                                        /* 设置定时器 2 的工作模式为匹配输出模式
                                                        Channel=1，匹配输出通道设置为 1
                                                        Time=200000，匹配时间为 0.2s
                                                        ActCtrl=2，匹配后将 T1TC 复位
                                                        OutCtrl=2，匹配输出翻转
                                                        */
    TimerStart(TIMER2, NULL);                           // 启动定时器 2
    while(1)
    {
        OSTimeDly(OS_TICKS_PER_SEC / 2);
    }
}
```

使用杜邦线将 SmartARM2300 工控开发板 JP1 上的 P0.7 和 BEEP 相连。

编译程序进入 AXD，全速运行程序，P0.7 即可输出 2.5Hz 的方波。可通过蜂鸣器响声或者示波器进行观察。使用广州致远电子有限公司的 LA1024 逻辑分析仪监视 P0.7 管脚，可得如图 1.2 所示的波形图。

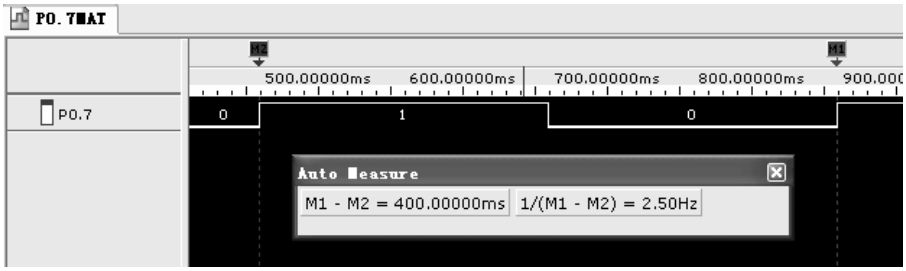

图 1.2　管脚波形图

3　输入捕获功能

LPC2300 系列 ARM 具有捕获外部输入信号的功能，当相应管脚出现符合条件的外部电平信号时，定时器就会将当前计数值 TC 保存到捕获寄存器 CR 中，并可设置产生中断。

本例程使用定时器 3 的输入捕获 CAP3.0（P0.23）捕获外部下降沿，当出现下降沿时，T3TC 的值被装入 T3CR0，程序根据 T3CR0 的值控制蜂鸣器鸣叫的频率。

程序清单 1.17　定时器 3 捕获功能配置

文件 1：“LPC2300PinCfg.h”

```
#define     P0_23_FNUC          P0_23_CAP30 // P0.23 引脚设置为定时器 3 捕获输出通道 0
#define     P1_27_FNUC          P1_27_ GPIO         // P1.27 引脚设置为 GPIO
```

文件 2：“main.c”

```
void User_Delay(uint32 time)                        // 软件延时，用于控制蜂鸣器频率
{
    while(time--);
}

void TASK0 (void *pdata)
{
    char        TimerArg[]  = "Mode=0";
    char        TimerModeArg[] = "Channel=0 ActCtrl=2";

    uint32    timePrev    = 0;                      // 上一次的捕获值
    uint32    timeNow     = 0;                      // 当前捕获值
    uint32    timeBeep    = 0;                      // 蜂鸣器鸣叫时间

    pdata = pdata;

    TimerInit(TIMER3, TimerArg, NULL);              // 定时器 3 初始化为定时模式：Mode=0;

    // 设置定时器 3 为定时模式，Channel=0：捕获输入通道设置为 0，ActCtrl=2：下降沿捕获
    TimerSetMode(TIMER3, SET_CAPMODE, TimerModeArg);

    TimerStart(TIMER3, NULL);                       // 启动定时器 3
    while(1)
    {
        timeNow = TimerGetCR(TIMER3, 0);
        if (timeNow > timePrev)
        {     // 捕获的值可能很大，除以 50 减少该值以便能观察到效果
            timeBeep = (timeNow - timePrev) / 50;
        }
        else if (timeNow < timePrev)
        {
            timeBeep = (timePrev - timeNow) / 50;
        }
        timePrev = timeNow;

        GpioClr(P1_27);
        User_Delay(timeBeep);
        GpioSet(P1_27);
```

```
            User_Delay(timeBeep);

            OSTimeDly(1);
        }
}
```

使用杜邦线将 SmartARM2300 工控开发板 JP13 上的第 6 脚（JP13 的第 6 脚通过一个 100 欧姆的电阻连接到 P0.23）和 JP1 上的 KEY1 连接，并且使用跳线器短接 P1.27 和 BEEP。编译程序进入 AXD，全速运行程序。每按下 KEY1 按键一次，蜂鸣器鸣叫的频率就会变化一一这是由于每次捕获的 T3TC 的值彼此不相同的原故，按键间隔越长，鸣叫频率越慢。

4 外部计数功能

LPC2300 系列 ARM 的定时器可通过 CAPn.m 管脚对外部脉冲进行计数。

为了更加方便、直观地观察到上述功能的实现，本例程通过定时/计数器 2 在计数模式下的匹配输出和计数功能相互配合，通过设置匹配寄存器，预定义一个计数上限，并设置匹配输出翻转；当计数值到达匹配值时，匹配通道 0 输出翻转控制蜂鸣器，从而相互验证。具体实现见程序清单 1.18。

程序清单 1.18 定时器 2 外部计数功能配置

文件 1：“LPC2300PinCfg.h”

```
#define     P0_04_FNUC          P0_04_CAP20         // P0.4 设置为定时器 2 捕获输入通道 0
……
#define     P0_06_FNUC          P0_06_MAT20         // P0.6 设置为定时器 2 匹配输出通道 0
```

文件 2：“main.c”

```
……
PinInit();                                          // 引脚初始化
……
void TASK1 (void *pdata)
{
    char    TimerArg[ ] = "Mode=1 CAPChn=0x01";
    char    TimerModeArg [ ] = "Channel=0 Time=5 ActCtrl=2 OutCtrl=2";
    pdata = pdata;

    TimerInit(TIMER2, TimerArg, NULL);              // 定时器 2 初始化为计数模式：Mode=1
                                                    // 捕获通道 0 为下降沿触发
    TimerSetMode(TIMER2, SET_MATMODE, TimerModeArg);
                                                    /* 匹配通道 0 为匹配输出模式
                                                    Channel=0，选择通道 0
                                                    Time=5，计数到 5 时匹配输出
                                                    ActCtrl=2，匹配后将 T2TC 复位
                                                    OutCtrl=2，匹配输出翻转
                                                    */
    TimerStart(TIMER2, NULL);                       // 启动计数器器
    while(1)
    {
        OSTimeDly(OS_TICKS_PER_SEC / 2);
    }
}
```

使用杜邦线将 SmartARM2300 工控开发板 JP1 上的 P0.4 和 P0.6 分别与 KEY1 和 BEEP 连接。

编译程序进入 AXD，全速运行程序后，每连续按 5 次（由于抖动的原因可能次数小于 5），蜂鸣器引脚电平状态翻转，控制其鸣叫，其频率与按键的速率有关。

1.9 异步串行接口

1.9.1 函数说明

异步串行口（UART）驱动库提供 7 个用户 API 函数，分别为 UartInit()、UartSetMode()、

UartRead()、UartWrite()、UartISR()、UARTn_ISR (void)、rs485Input()主要完成 4 个异步串行口的通用串行通信功能及自动波特率设置，同时提供 UART1 Modem 通信控制、UART3 IrDA 通信控制和 4 个串口的 RS-485 通信控制。并支持最多 6 个外扩 UART。

1.9.2 使用方法

1 引脚连接

在使用 UART 之前，首先应该配置引脚连接，如程序清单 1.19 所示。同时还可以设置引脚内部是否连接上/下拉电阻。在系统初始化时，调用“PinInit()”函数使能引脚连接配置。

程序清单 1.19 引脚功能配置

```
#define     P0_02_FNUC      P0_02_TXD0      // P0.2 配置为 UART0 发送引脚
#define     P0_03_FNUC      P0_03_RXD0      // P0.3 配置为 UART0 接收引脚
```

2 接口初始化

在调用其它功能函数之前，必须先调用函数 UartInit()对 UART 进行初始化操作，如程序清单 1.20 所示。

程序清单 1.20 接口初始化

```
char    UartArg[ ] = "BaudRate=115200 RxBufSize=512 TxBufSize=256 FifoLen=8";
UartInit(UART0, UartArg, NULL);                 // 初始化 UART0
```

3 模式设置（可选）

如需改变波特率、奇偶校验、停止位或使用 Modem、RS-485 等功能，必须调用 UartSetMode()函数进行具体设置。更改数据位、停止位和奇偶校验如程序清单 1.21 所示。

程序清单 1.21 串口模式设置

```
char UartModeArg[] = "DataBits=8 StopBits=2 Parity=1";      // 数据位 8，停止位 2，奇校验
UartSetMode(UART0, SET_CTRLMODE, UartModeArg);
```

4 设置中断

如程序清单 1.22 所示，设置 UART0 的中断。

程序清单 1.22 设置并使能 UART 中断

```
SetVICIRQ(UART0_IRQ_CHN, 7, (uint32)UART0_ISR);     // 设置 UART0 中断，优先级为 7
```

5 数据读写

完成初始化和设置好中断后即可进行相应的数据读写操作，如程序清单 1.23 所示。

程序清单 1.23 数据读写

```
uint8   rcv_buf[5];                             // 接收数据字符数组
UartRead(UART0, rcv_buf, 3, NULL);              // 读取 3 个数据
```

或

```
uint8   send_buf[5];
UartWrite(UART0, send_buf, 3, NULL);            // 发送 3 个数据
```

1.9.3 示范例程

1 串口通信

现以 UART0 为例，使用 SmartARM2300 工控开发板为平台，演示 UART 驱动库的部分函数的使用方法。上位机向串口发送数据，串口把接收到的数据再回发给上位机。如果接收到字符’A’，则调用用户钩子函数 fun()，在 fun()函数中控制蜂鸣器鸣叫。如程序清单 1.24 所示，仅供参考。

程序清单 1.24 UART0 使用示例

文件 1：“LPC2300PinCfg.h”

```
#define      P0_02_FNUC          P0_02_TXD0          // P0.2 设置为 UART0 TXD
#define      P0_03_FNUC          P0_03_RXD0          // P0.3 设置为 UART0 RXD
```

文件 2：“main.c”

```
void fun(void *pp)                                          // 接收到字符'A'时调用的函数
{
    GpioCpl(P1_27);                                         // 取反蜂鸣器引脚
}
void TASK1      (void *pdata)
{
      int32                 k;
      uint8                 buf[50] = {0};                  // 数据收发缓冲
      UART_HOOKS            uarthook = {0};                 // 钩子函数挂接结构体
    char                    uartInitArg[] = "BaudRate=115000 RxBufSize=64 TxBufSize=64";
    //char              uartModeArg[] = "TimeOut=10";

      uarthook.cSpringChar      = 'A';                        // 触发字符
    uarthook.pSpringFunction = fun;                         // 收到触发字符后的处理函数

    /* UART0 初始化，波特率为 115200，收/发缓冲都为 64 字节，其他参数取默认值
       即数据位数为 8，停止位数为 1，硬件 FIFO 长度为 8，外设时钟 Fpclk 取系统默认值 */
      k = UartInit(UART0, uartInitArg, &uarthook);          // UART0 初始化
      SetVICIRQ(UART0_IRQ_CHN, 6, (uint32)UART0_ISR);       // 设置 UART0 中断
      //UartSetMode(UART0, SET_TIMEOUT, uartModeArg);       // 设置读超时因子

    while(1)
    {
        k = UartRead (UART0, (uint8 *)buf, 40, NULL);       // 读数据
        if(k > 0)
        {
            k = UartWrite(UART0, (uint8*)buf, k, NULL);     // 发送数据
        }
       OSTimeDly(OS_TICKS_PER_SEC / 10);
    }
}
```

全速运行程序后，通过串口调试软件向串口发送字符，在接收窗口将观察到接收到的数据，如所示。若所发数据中包含有字符‘A’，蜂鸣器将鸣叫或停止（取决于‘A’字符的个数）。

图 1.3 串口收发数据

2 自动波特率设置

自动波特率通过函数 UartSetMode()启动和设置自动波特率测量模式。自动波特率功能可用于测量基于“AT”协议（Hayes 命令）的输入波特率。

一旦启动了自动波特率测量，将要求对方发送特定格式的起始数据进行波特率测量，若不发送数据，系统波特率不会改变，若发送的起始字符为其他字符则可能出现波特率测量错误。自动波特率测量时对起始数据的格式要求如表 1.4 所示。

表 1.4 自动波特率测量时对数据的要求

自动波特率模式	测量字节 Bit[1]	测量字节 Bit[0]	数据举例
模式 0	0	1	0x01、0x41
模式 1	0	1	0x01、0x41
	1	1	0x03、0x43

自动波特率的范围为 600～115200。下面的例程在 UART0 初始化后启动一次自动波特率设置，如程序清单 1.25 所示。

程序清单 1.25 UART0 自动波特率

文件 1：“LPC2300PinCfg.h”

```
#define    P0_02_FNUC        P0_02_TXD0                    // P0.2 设置为 UART0 TXD
#define    P0_03_FNUC        P0_03_RXD0                    // P0.3 设置为 UART0 RXD
```

文件 2：“main.c”

```
void TASK1 (void *pdata)
{
    int32     k;
    uint8     buf[50];                                    //  数据缓冲
    char      uartInitArg[] = "BaudRate=9600 RxBufSize=64 TxBufSize=64";
    char      uartModeArg[] = "AbrCtrl=0";

    pdata = pdata;

    /* UART0 初始化，波特率为 9600，收发缓冲都为 64 字节，其他参数取默认值
       即数据位数为 8，停止位数为 1，硬件 FIFO 长度为 8，外设时钟 Fpclk 取系统默认值 */
    UartInit(UART0, uartInitArg, NULL);
    UartSetMode(UART0, SET_AUTOBAUD, uartModeArg);          //  使能自动波特率，采用模式 0
    SetVICIRQ(UART0_IRQ_CHN, 7, (uint32)UART0_ISR);     //  设置串口中断，优先级 7
    while(1)
    {
        k = UartRead (UART0, (uint8*) buf, 40, NULL);
        if(k > 0)
        {
            k= UartWrite(UART0, (uint8 *)buf, k, NULL);
        }
        OSTimeDly(OS_TICKS_PER_SEC / 10);
    }
}
```

拔掉 SmartARM2300 工控开发板上的 ISP 跳线器，下载程序后去掉 JTAG，全速运行程序。在 PC 机上打开串口调试软件，波特率设在 600～115200 范围内即可。首次发送数据时，第一个数据要求符合表 1.4 的格式，如字符‘A’，之后下位机已经和 PC 机同步了波特率，才可以发送任意数据。

为了演示，程序中将下位机波特率设为了 9600，现设置 PC 机以 115200 的波特率发送数据，并在首次发送数据时发送符合波特率测量要求的数据块如“A……”，如图 1.4 所示。

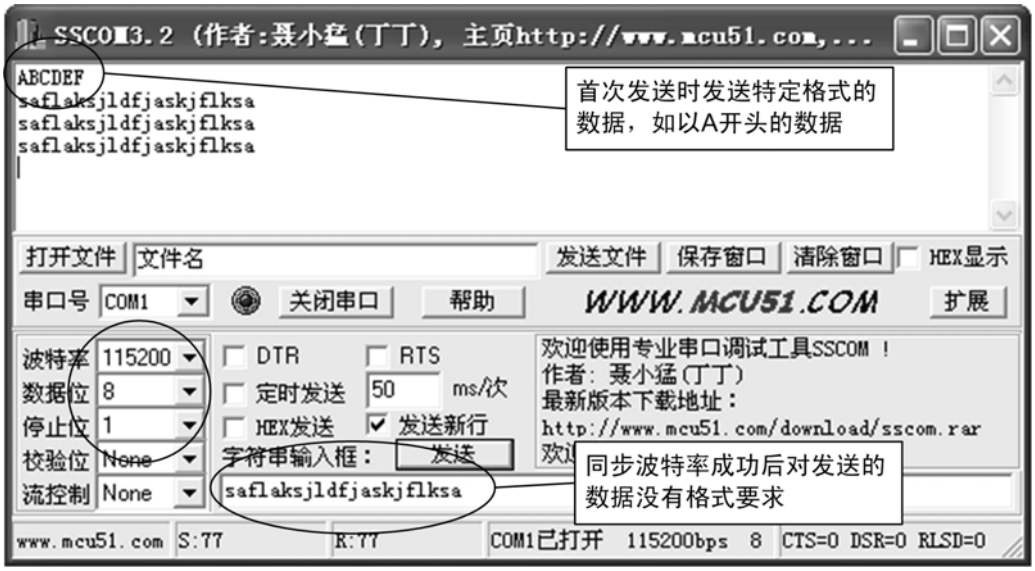

图 1.4　以 115200 波特率向下位机发送数据

若首次发送的数据块的第一个数据不符合规定，则下位机波特率测量会出错，下位机收到的将会是乱码，PC 机收到的也会是乱码。

3　UART3 IrDA 通讯

将 UART3 配置成 IrDA 模式就使得 LPC2300 具备发送和接收 IrDA 格式脉冲的能力。其引脚的配置及数据收发和普通串口相同。UART3 IrDA 功能的使用只需在串口初始化后使用 UartSetMode()函数将 IrDA 使能并配置好参数即可。但要注意 UART3 的 IrDA 功能是半双工的，在同一时刻只能作为发送端或接收端。

SmartARM2300 开发板上具有一个 HSDL-3602 红外模块，可使用两块板子进行收发实验。下面的例程中发送端不停地发送字符‘A’～‘Z’，接收端通过串口 0 向上位机发送收到的字符。IrDA 通信发送见程序清单 1.26，接收见程序清单 1.27，仅供参考。

程序清单 1.26　UART3 IrDA 发送

文件 1：“LPC2300PinCfg.h”

```
#define     P4_28_FNUC          P4_28_TXD3      // P4.28 设置为 UART3 TXD
#define     P4_29_FNUC          P4_29_RXD3      // P4.29 设置为 UART3 RXD
```

文件 2：“main.c”

```
#define     IR_EN               P3_26       //  使 HSDL-3602 工作在 SIR 模式的引脚
void TASK1   (void *pdata)
{
    char   uartInitArg[] = "BaudRate=115200 RxBufSize=32 TxBufSize=256 FifoLen=8";
    char   uartModeArg[] = "IrDACtrl=2 PulseDiv=1";
    uint8   strMsg[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ\r\n";

    pdata = pdata;

    while(UartInit(UART3, uartInitArg, NULL) == OPERATE_FAIL);
                                                    //  初始化串口 3
    UartSetMode(UART3, SET_IrDA, uartModeArg);   //  设置 IrDA
    SetVICIRQ(UART3_IRQ_CHN, 7, (uint32)UART3_ISR);
                                                    //  设置 UART3 中断，优先级为 7
    GpioClr(IR_EN);                             //  置 HSDL-3602 工作在 SIR 模式

    while(1)
```

```
    {       // 发送字符'A'到'Z'
            UartWrite(UART3, strMsg, (sizeof(strMsg) - 1), NULL);
            OSTimeDly(OS_TICKS_PER_SEC / 4);
    }
}
```

程序清单 1.27　UART3 IrDA 接收

文件 1：“LPC2300PinCfg.h”

```
#define     P4_28_FNUC          P4_28_TXD3    // P4.28 设置为 UART3 TXD
#define     P4_29_FNUC          P4_29_RXD3    // P4.29 设置为 UART3 RXD
```

文件 2：“main.c”

```
#define     IR_EN               P3_26         // 置 HSDL-3602 红外模块工作在 SIR 模式的引脚
#define     RCV_BUF_SIZE        32
void TASK1 (void *pdata)
{
    uint32 Len = 0;                                     // 接收数据长度
    uint8   rcv_buf[RCV_BUF_SIZE];                  // 接收数据字符数组
    char    uart0Arg[] = "BaudRate=115200 RxBufSize=256 TxBufSize=32";
    char    uart3Arg[] = "BaudRate=115200 RxBufSize=32 TxBufSize=256";
    char    uartModeArg[] = "IrDACtrl=2 PulseDiv=1";

    pdata = pdata;

    while(UartInit(UART0, uart3Arg, NULL) == OPERATE_FAIL);
                                                        // 初始化串口 0
    SetVICIRQ(UART0_IRQ_CHN, 7, (uint32)UART0_ISR);
                                                        // 设置 UART0 中断，优先级为 7

    while(UartInit(UART3, uart0Arg, NULL) == OPERATE_FAIL);
                                                        // 初始化串口 3
    UartSetMode(UART3, SET_IrDA, uartModeArg);          // 设置 IrDA
    SetVICIRQ(UART3_IRQ_CHN, 6, (uint32)UART3_ISR);
                                                        // 设置 UART3 中断，优先级为 6
    GpioClr(IR_EN);                                 // 置 HSDL-3602 工作在 SIR 模式

    while(1)
    {
        Len = UartRead(UART3, rcv_buf, RCV_BUF_SIZE, NULL);
        if (Len > 0)
        {
            UartWrite(UART0, rcv_buf, Len, NULL);
        }
        OSTimeDly(1);
    }
}
```

将收发程序分别下载到两块 SmarARM2300 开发板，将两块板子上 JP1 的 IR_EN、IR_TX 和 IR_RX 跳线短接，使红外收发器件相互对准，将接收板串口 0 与上位机相连。最后全部全速运行，在上位机上打开一个串口调试软件，设置好各项参数，就可以看到 IrDA 通讯的效果，如图 1.5 所示。

图 1.5　IrDA 通讯效果

4　UART1 Modem 通讯控制

使用 UART1 的 Modem 控制功能和使用普通 3 线串口一样，只需在初始化串口后使用 UartSetMode()函数使能 Modem 控制功能，就可用 UartWrite()对 Modem 发送数据，UartRead()可接收 Modem 的返回数据。本驱动库对 DCD、DSR 和 RI 信号不做检测，使用时请注意。

本例程使用 SmartARM2300 开发板和 ZWG-13A（GPRS Modem）调试通过。使用串口 0 接收上位机发送的 AT 命令，通过串口 1 转发给 Modem，从串口 1 接收到的 Modem 调试回显信息再通过串口 0 回传到上位机供观察。如所图 1.6 示。

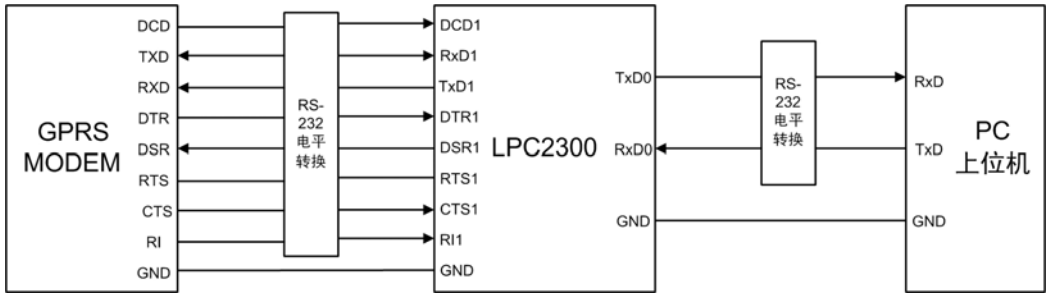

图 1.6　MCU 与 Modem 通信示意

具体代码如程序清单 1.28 所示，仅供参考。

程序清单 1.28　UART1 Modem 控制功能演示程序

文件 1：“LPC2300PinCfg.h”

```
#define    P0_02_FNUC          P0_02_TXD0          // P0.2 设置为 UART0 TXD
#define    P0_03_FNUC          P0_03_RXD0          // P0.3 设置为 UART0 RXD
……
#define    P2_00_FNUC          P2_00_TXD1          // UART1 的 Modem 控制线引脚
#define    P2_01_FNUC          P2_01_RXD1
#define    P2_02_FNUC          P2_02_CTS1
#define    P2_03_FNUC          P2_03_DCD1
#define    P2_04_FNUC          P2_04_DSR1
#define    P2_05_FNUC          P2_05_DTR1
#define    P2_06_FNUC          P2_06_RI1
#define    P2_07_FNUC          P2_07_RTS1
```

文件 2：“main.c”

```
#define  MAX_AT_CMD_LEN    32                    // AT 命令最大长度
……
```

```
PinInit();
……

// 任务 1，接收 Modem 返回的数据并通过 UART0 送回上位机显示
void TASK1 (void *pdata)
{
    int32    len=0;
    char        uartArg[] = "BaudRate=115200 RxBufSize=50 TxBufSize=50";
    char     uartModeArg0[] = "TimeOut=20";
    char     uartModeArg1[] = "FlowType=3";
    uint8    buf[MAX_AT_CMD_LEN];                          // 数据缓冲

     pdata = pdata;
     UartInit(UART0, uartArg, NULL);                       // 初始化 UART0
     UartSetMode(UART0, SET_TIMEOUT, uartModeArg0);        // 设置接收超时为 20ms
     SetVICIRQ(UART0_IRQ_CHN, 7, (uint32)UART0_ISR);       // 设置 UART0 中断，优先级 7

     UartInit(UART1, uartArg, NULL);                       // 初始化 UART1
     UartSetMode(UART1, SET_MODEM, uartModeArg1);          // 使能 Modem 控制，使用硬件流控制
     SetVICIRQ (UART1_IRQ_CHN, 8, (uint32)UART1_ISR);      // 设置 UART1 中断，优先级 8

   while (1)
     {
          len = UartRead(UART1, buf, 20, NULL);            // 接收 Modem 发来的数据
          if (len > 0)
          {
               strcat((char *)buf, “\r\n”);
               UartWrite(UART0, buf, len, NULL);           // 通过 UART0 将数据发送给 PC
          }
          OSTimeDly(5);
     }
}

void TASK2(void *pdata)
{
     uint32    len;
     uint8     buf[MAX_AT_CMD_LEN];                        // 数据缓冲
     pdata = pdata;

     while (1)
     {
          len = UartRead(UART0, buf, 20, NULL);            // 从 UART0 接收 PC 发来的数据
          if (len > 0)
          {
               UartWrite(UART1, buf, len, NULL);      // 将数据通过 UART1 转发给 Modem
               memset(buf, 0, 20);                    // 把 buf 清空
          }
          OSTimeDly(OS_TICKS_PER_SEC / 8);
     }
}
```

将串口 0 连接 PC 机的串口，串口 1 连接 Modem，然后下载程序并全速运行程序。在 PC 机上打开超级终端，设置正确的串口参数后就可发送 AT 命令和观察结果，如发送“ATE1”为打开调试回显功能，“ATH”为挂机，“AT+CSQ”为显示信号强度，“ATD+电话号码”为拨打电话，显示的情形如图 1.7 所示。注意输入一个命令后要按回车键。

图 1.7　GPRS Modem ZWG－13A 控制

5　RS-485 通讯控制

PC 机串口连接到 RS-232 转 RS-485 模块上，模块的另一端连接到 MiniARM 底板的 RS-485 接线上。如图 1.8 所示。

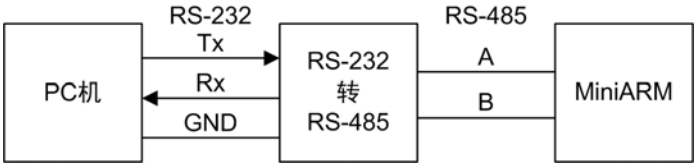

图 1.8　MiniARM 通过 RS-485 与 PC 机通讯示意

通过 PC 上位机的串口向 MiniARM 发送数据，MiniARM 把接收到的数据再回发到 PC 上位机。使用 RS-485 时串口初始化必须有“RS485Dir”参数，且参数值应正确。以串口 0 为例，使用 P0.6 作方向控制引脚，使用自定义的钩子函数进行方向引脚控制，具体程序如程序清单 1.29 所示。

程序清单 1.29　RS-485 通讯使用示例

文件 1：“LPC2300PinCfg.h”

```
#define     P0_02_FNUC          P0_02_TXD0      // P0.2 设置为 UART0 TXD
#define     P0_03_FNUC          P0_03_RXD0      // P0.3 设置为 UART0 RXD
#define     P0_06_FNUC          P0_06_GPIO      // P0.6 设置为 GPIO
```

文件 2：“main.c”

```
// 数据块发送完毕后调用的钩子函数，可添加其他处理代码
void sendEndHook(void *pp)
{
    rs485Input(UART0);                          // 发送完毕后置 RS-485 的控制引脚方向为输入
    // 添加其他处理代码
    // ……
}

void TASK1      (void *pdata)
{
    int32               k;
    uint8           buf[50];                    // 数据缓冲
    char            rs485Arg[32];               // RS-485 参数缓存
    UART_HOOKS      uarthook = {0};             // 用户钩子函数挂接结构体
    char            uartInitArg[] = "BaudRate=9600 RxBufSize=64 TxBufSize=64";
```

```
    uarthook.pSendEnd = sendEndHook;                    // 挂接用户钩子函数
    UartInit(UART0, uartInitArg, &uarthook);            // UART 初始化
    sprintf(rs485Arg, "RS485Dir=%u", P0_06);            // 获取 RS-485 方向控制引脚 P0.6 的编号值

    UartSetMode(UART0, SET_RS485, rs485Arg);            // 使能 RS-485，设置其方向控制引脚
    SetVICIRQ(UART0_IRQ_CHN, 7, (uint32)UART0_ISR);
    while(1)
    {
        k = UartRead (UART0, (uint8*) buf, 40, NULL);
        if(k > 0)
         {
            k= UartWrite(UART0, (uint8 *)buf, k, NULL);
         }
        OSTimeDly(OS_TICKS_PER_SEC / 10);
    }
}
```

正确连接硬件，全速运行程序。PC 机使用串口调试软件向 MiniARM 发送数据，并接收回发数据，如图 1.9 所示。

图 1.9　RS-485 通讯效果

1.10 I2C 接口应用

1.10.1　函数说明

I^2C 驱动提供了 7 个用户可用的 API 函数，分别为 I2cInit()、I2cSetMode()、I2cRead()、I2cWrite()、I2cGetFlag()、I2C1_ISR()、I2C2_ISR()主要完成对 I^2C 接口 1、2 的初始化、模式设定、数据读写、状态获取以及用户中断处理等功能。随着 I^2C 总线技术的推出。很多电子厂商都推出了许多带 I^2C 总线接口的器件，大量应用于视频、音像、存储及通讯等领域，其中一些通用的 I^2C 接口器件的种类、型号及寻址字节见

表 1.5。

表 1.5　常用 I²C 接口通用器件的种类、型号及寻址字节

种　　类		型　　号	器件地址及寻址字节							
			MSB							LSB
E²PROM	128 字节	CAT24C01	1	0	1	0	A2	A1	A0	R/$\overline{W}$
	256 字节	CAT24C02	1	0	1	0	A2	A1	A0	R/$\overline{W}$
	512 字节	CAT24C04	1	0	1	0	A2	A1	A8	R/$\overline{W}$
	1K 字节	CAT24C08	1	0	1	0	A2	A9	A8	R/$\overline{W}$
	2K 字节	CAT24C16	1	0	1	0	A10	A9	A8	R/$\overline{W}$
	4K 字节	CAT24C32	1	0	1	0	A2	A1	A0	R/$\overline{W}$
	8K 字节	CAT24C64	1	0	1	0	A2	A1	A0	R/$\overline{W}$
	16K 字节	CAT24C128	1	0	1	0	X	X	X	R/$\overline{W}$
	32K 字节	CAT24C256	1	0	1	0	0	A1	A0	R/$\overline{W}$
串行 FRAM（铁电存储器）	64K 字节	FM24C512	1	0	1	0	A2	A1	A15	R/$\overline{W}$
	32K 字节	FM24C256	1	0	1	0	A2	A1	A0	R/$\overline{W}$
	8K 字节	FM24C64	1	0	1	0	A2	A1	A0	R/$\overline{W}$
	2K 字节	FM24C16	1	0	1	0	A2	A1	A0	R/$\overline{W}$
	2K 字节	FM24C16A	1	0	1	0	A2	A1	A0	R/$\overline{W}$
	512 字节	FM24CL04	1	0	1	0	A2	A1	NC	R/$\overline{W}$
	512 字节	FM24C04A	1	0	1	0	A2	A1	A0	R/$\overline{W}$
实时时钟/日历芯片		PCF8563	1	0	1	0	0	0	1	R/$\overline{W}$
键盘及 LED 驱动器		ZLG7290	0	1	1	1	0	0	0	R/$\overline{W}$
温度传感器		LM75A	1	0	0	1	A2	A1	A0	R/$\overline{W}$
带 32×4 位 RAM 低复用率的通用 LCD 驱动器		PCF8562	0	1	1	1	0	0	SA0	R/$\overline{W}$
通用低复用率 LCD 驱动器		PCF8576D	0	1	1	1	0	0	SA0	R/$\overline{W}$
内嵌 I²C 总线、E²PROM、RESET、WDT 功能的电源监控器件		CAT1161/2	1	0	1	0	A10	A9	A8	R/$\overline{W}$
内嵌 I²C 总线、E²PROM、RESET 功能的电源监控器件		CAT1024/5	1	0	1	0	0	0	0	R/$\overline{W}$

注:（1）A0、A1 和 A2 对应器件的管脚 1、2 和 3， SA0 也为器件的引脚;
（2）A8、A9、A10 和 A15 对应存储阵列地址子地址。

1.10.2　使用方法

1　引脚连接

在使用 I²C 接口时，首先应该配置引脚连接，如程序清单 1.30 所示。同时还可以设置引脚内部是否连接上/下拉电阻。在系统初始化时，调用“PinInit()”函数使能引脚连接配置。

程序清单 1.30　引脚功能配置

```
#define     P0_10_FNUC          P0_10_SDA2      // 配置 P0.10 为 I2C2
#define     P0_11_FNUC          P0_11_SCL2      // 配置 P0.11 为 I2C2
```

2 接口初始化

使用函数 I2cInit()对 I²C 接口初始化，如程序清单 1.31 所示。在函数入口给定总线速率，其它初始化操作函数内部会自动完成（使用 I²C0 时可以省略此步骤）。

程序清单 1.31 接口初始化

```
char    pI2C[ ] = " Speed=400000";
I2cInit (I2C1, pI2C, NULL);                     // I2C1 初始化，总线速率为 400Kb/s
```

3 中断设置

I²C 状态的每次改变都会引发中断，所以有关 I²C 的状态处理全部都是在中断处理函数中进行的。在 I²C 接口初始化后，必须使用函数 SetVICIRQ()设置 I²C 中断，确定中断优先级以及中断服务函数的入口地址（使用 I²C0 时可以省略此步骤）。

程序清单 1.32 中断设置

```
SetVICIRQ(I2C1_IRQ_CHN, 2, (uint32)I2C1_ISR);       // 设置 I2C 中断
```

4 数据的发送和接收

完成上述几步工作后即可使用函数 I2cWrite()和 I2cRead()对 I²C 设备进行数据的发送和接收操作了，参考程序清单 1.33。

程序清单 1.33 数据的发送和接收

```
#define     FM24CL04        0xA0                    // 定义从器件地址
uint8 T_Buf[10] = {0,1,2,3,4,5,6,7,8,9 };           // 发送数据指针
uint16      Sub_Addr = 0x00;                        // 器件子地址
I2cWrite(I2C0, FM24CL04, &Sub_Addr ,2,T_Buf, 10); // 发送 10 个数据
while(I2C_WRITE_END != I2cGetFlag(I2C0))            // 等待发送完成
{
    OSTimeDly(1);
}
```

或

```
#define     FM24CL04        0xA0                    // 定义从器件地址
uint8 R_Buf[12] = {0};                              // 接收缓冲指针
uint16       Sub_Addr = 0x00;                       // 器件子地址
I2cRead(I2C0, FM24CL04, &Sub_Addr ,2,R_Buf,10);     // 接收 10 个数据或
while(0 == (I2C_READ_END & I2cGetFlag(I2C0)))       // 等待接收完成
{
    OSTimeDly(1);
    if((I2C_NOT_GET_BUS | I2C_ACK_ERR) & I2cGetFlag(I2C0))
    {
        while(1);
    }
}
```

1.10.3 示范例程

1 读写铁电存储器－FM24CL04

铁电存储器能兼容 RAM 的一切功能，并且和 ROM 技术一样，是一种非易失性的存储器。铁电存储器在这两类存储类型间搭起了一座跨越沟壑的桥梁：一种非易失性的 RAM，并且被认为未来可能是取代各类存贮器的超级存贮器。

Ramtron 公司的铁电存储器技术到现在已经相当的成熟，拥有串行总线接口（I²C 和 SPI）、并行总线接口的铁电器件，目前容量最大的已经达到了 4Mbit。

I²C 接口的铁电存储器为了兼容以前的 E²PROM 芯片，在接口上沿用了 E²PROM 芯片的封装，使得可以完全代替 E²PROM 芯片，除了具有掉电保存数据的特点外，还具有零等待、随机读写的特性。FM24CL04 就是一种 I²C 接口的铁电存储器，容量是 512 字节，容量更大的有 FM24C16、FM24C64、FM24C256、FM24C512 等。

在 MiniARM 系列核心板上有一个 FM24CL04，其应用电路如图 1.10 所示。

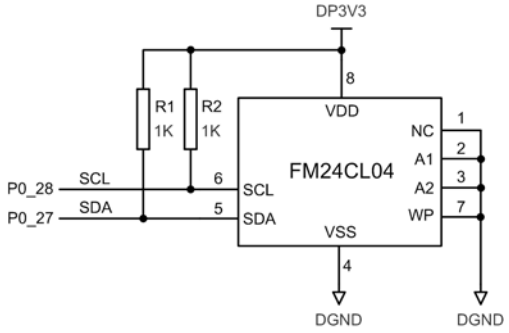

图 1.10　FM24CL04 应用电路原理

由图 1.10 可见，FM24CL04 器件的地址 A1、A2 都接地，由

表 1.5 可知其访问地址是 0xA0。由于 LPC2300 系列 ARM 的 P0.27、P0.28（I^2C0 的 SDA0、SDA0）是开漏输出，因此需要接上拉电阻 R1、R2。

由于 FM24CL04 是随机存储的，并且没有写等待周期，因此可以从任何一个地址开始进行连续的若干个写操作，FM24CL04 具有子地址，并且由于其容量大于 256，子地址是双字节的，所以要使用双字节写操作。

对 FM24CL04 的 0xEA 地址到 0x014D 地址连续写入 100 个字节的数据的操作代码如程序清单 1.34 所示。

程序清单 1.34 I^2C 使用示例

文件 1：“main.c”

```
#define             FM24CL04            0xA0                    // 定义从器件地址
void   TASK2 (void *pdata)
{
      uint8   T_Buf[100] = {0 };                                  // 发送数据指针
      uint8   R_Buf[100] ;                                        // 接收缓冲
      uint16   SubAddr = 0x00;                                    // 器件子地址
      uint8   i;

      pdata = pdata;
      for(i=0; i<100; i++)
            T_Buf[i] = i;

     SubAddr = 0x00;                                              // 从机子地址
     I2cWrite(I2C0, FM24CL04, (uint8*)&SubAddr,TWO_BYTE_SADDR, T_Buf, 100);
                                                                  // 发送 100 个数据
     while(0 == (I2C_WRITE_END & I2cGetFlag(I2C0)))               // 等待发送完成
     {
           OSTimeDly(1);
     }
     I2cRead(I2C0, FM24CL04, (uint8*)&SubAddr, TWO_BYTE_SADDR,R_Buf,100);
                                                                  // 接收 100 个数据
     while(0 == (I2C_READ_END & I2cGetFlag(I2C0)))                // 等待接收完成
     {
      OSTimeDly(1);
     }
     for(i=0; i<100; i++)
     {
           if(R_Buf[i] != T_Buf[i])
                 break;
     }
     if(i ==100)                                                  // 接收数据无误，BEEP 一声
     {
           GpioClr(P1_27);
           OSTimeDly(OS_TICKS_PER_SEC / 2);
           GpioSet(P1_27);
           OSTimeDly(OS_TICKS_PER_SEC / 2);
      }
      else                                                        // 接收数据有误，BEEP 两声
      {
           GpioClr(P1_27);
           OSTimeDly(OS_TICKS_PER_SEC / 2);
           GpioSet(P1_27);
           OSTimeDly(OS_TICKS_PER_SEC / 2);

           GpioClr(P1_27);
           OSTimeDly(OS_TICKS_PER_SEC / 2);
           GpioSet(P1_27);
           OSTimeDly(OS_TICKS_PER_SEC / 2);
      }
      while(1)
      {
```

```
            OSTimeDly(OS_TICKS_PER_SEC);
        }
}
```

将 SmartARM2300 工控开发板 JP1 的 P1.27 和 BEEP 短接。编译程序进入 AXD 调试环境，全速运行程序，如果对 FM24CL04 读写校验成功则蜂鸣器鸣叫一声，否则鸣叫两声。

2 I²C 主、从机通讯

利用 I²C1 的主、从操作模式，在两套 SmartARM2300 开发板上进行主、从机通讯实验。具体示范例程见程序清单 1.35、程序清单 1.36 和程序清单 1.37 所示。

程序清单 1.35 I²C1 引脚连接配置

```
#define         P0_00_FNUC      P0_00_SDA1              // P0.0 设置为 I2C1   SDA
#define         P0_01_FNUC      P0_01_SCL1              // P0.1 设置为 I2C1   SCL
```

程序清单 1.36 I²C1 主机通讯代码

```
#define         SAL_ADDR        0xFE                    //  定义器件地址
void    TASK1(void *pdata)
{
        char      UartArg[ ] = "BaudRate=115200 RxBufSize=512 TxBufSize=256";
     uint8    UartRev[30];                               //  数据收发缓冲
        uint8    SubAddr;
        uint8    DataLength;
        pdata = pdata;
        while((I2cInit(I2C1, "Speed=400000", NULL)) == OPERATE_FAIL) // I2C 初始化
        {
                OSTimeDly(1);
        }
        SetVICIRQ(I2C1_IRQ_CHN, 2, (uint32)I2C1_ISR);   //  设置 I2C 中断
        while(UartInit(UART0, UartArg, NULL) == OPERATE_FAIL)   //  初始化 UART0
        {
            OSTimeDly(1);
        }
        SetVICIRQ(UART0_IRQ_CHN, 3, (uint32)UART0_ISR);     //  设置 UART0 中断
        while(1)
        {
                if(UartRead(UART0,UartRev,25,NULL))
                {
                        if((UartRev[0]+UartRev[1]) > 20)
                                continue;
                        SubAddr = UartRev[0];
                        DataLength = UartRev[1];
                        memset(UartRev,0,(DataLength+2));
                        I2cRead(I2C1, SAL_ADDR, &SubAddr, ONE_BYTE_SADDR, UartRev, DataLength);
                                                        //  接收 DataLength 个数据
                        while(0 == (I2C_READ_END & I2cGetFlag(I2C1)))   //  等待接收完成
                        {
                                OSTimeDly(1);
                                if((I2C_NOT_GET_BUS | I2C_ACK_ERR) & I2cGetFlag(I2C1))
                                {
                                        GpioClr(P1_27);
                                        OSTimeDly(OS_TICKS_PER_SEC/10);
                                        GpioSet(P1_27);
                                        OSTimeDly(OS_TICKS_PER_SEC/10);
                                        DataLength=0;
                                        break;
                                }
                        }
                        UartWrite(UART0,UartRev,DataLength,NULL);
                        UartWrite(UART0,(uint8*)"\r\n",2,NULL);
                }
                OSTimeDly(OS_TICKS_PER_SEC/10);
        }
}
```

程序清单 1.37 I²C1 从机通讯代码

```
uint8   SalBuf[20]={'1','2','3','4','5','6','7','8','9','0','a','b','c','d','e','f','g','h','i','j'};
void    TASK1 (void *pdata)
{
        uint8 I2C_Buf[30] = {0 };
        I2C_SALHOOKS SalHooks;
        pdata = pdata;
        SalHooks.pSendBuf = SalBuf;
        SalHooks.pRevBuf    = I2C_Buf;
        while(I2cInit(I2C1, "Mode=1 SalAddr=0xFE SubAddrType=1", &SalHooks) == OPERATE_FAIL)
        {                                                                    // I2C 初始化
                OSTimeDly(1);
    }
        SetVICIRQ(I2C1_IRQ_CHN, 2, (uint32)I2C1_ISR);                        //  设置 I2C 中断
        while(1)
        {
                OSTimeDly(OS_TICKS_PER_SEC/10);
        }
}
```

分别将上述代码下载到两块调试板，用三根杜邦线将各自的 P0.0、P0.1 及 GND 相连接，并用串口线将 I²C 主机与 PC 机相连，全速运行程序，在上位机串口软件中输入“从机子地址 读取数据个数”格式的命令，得到运行结果如图 1.11 所示：

图 1.11 I²C1 主、从机通讯结果

1.11 SPI 接口应用

1.11.1 函数说明

SPI 驱动提供了 8 个用户可用的 API 函数，分别为 SPIInit()、SPISetMode()、SPIRead()、SPIWrite()、SPIFIFOState()、SPIFIFOFlush()、SPIISR()、SPI0()主要完成对 SPI 接口 0 的初始化、模式设定、数据读写、收发缓冲队列状态获取、收发缓冲队列清空以及用户中断处理等功能。

1.11.2 表使用方法

1 引脚连接

如程序清单 1.38 所示，在 LPC2300PinCfg.h 文件中把 SPI 的各个相关引脚设置为 SPI 功能，否则 SPI 设备将不起作用。其中 P0.16（SSEL）在作主机时可不用作 SSEL 功能而作为 GPIO 使用。

程序清单 1.38 SPI 引脚设置

```
#define          P0_15_FNUC          P0_15_SCK          // P0.15 作为 SPI0 的串行时钟
```

```
#define     P0_17_FNUC      P0_17_MISO      // P0.17 作为 SPI0 主机输入从机输出端
#define     P0_18_FNUC      P0_18_MOSI      // P0.18 作为 SPI0 主机输出从机输入端
#define     P0_16_FNUC      P0_16_SSEL      // P0.16 作为 SPI0 从机选择
```

2 初始化 SPI

初始化 SPI 设备的程序如程序清单 1.39 所示。

程序清单 1.39 初始化 SPI

```
char    Spi_arg[ ] = "Mode=0 Cpha=0 Cpol=1 Lsbf=0 Speed=100000 CsPin=11 RWMode=1 \
                      RxBufSize=16 TxBufSize=16";
SPIInit(SPI0, Spi_arg, NULL);                   // SPI 主机设备初始化
```

3 设置中断函数

SPI 的数据收发是在中断中完成的，因此需要设置中断函数。如果要同时使用 SPI0 和 SSP0，则需设置的中断函数为 SPI0_SSP0_ISR()，因为 SPI0 和 SSP0 共用同一个中断通道号 10。SPI 中断的设置如程序清单 1.40 所示。

程序清单 1.40 中断处理函数

```
SetVICIRQ(SPI0_IRQ_CHN, 7, (uint32)SPI0_ISR);       // 设置 SPI0 中断，优先级为 7

// 若同时使用 SPI0 和 SSP0，则中断设置如下
SetVICIRQ(SPI0_IRQ_CHN, 7, (uint32)SPI0_SSP0_ISR);  // 设置 SPI0 和 SSP0 中断
```

4 读写数据

调用函数 SPIWrite()或 SPIRead()进行数据的读写操作。

1.11.3 示范例程

在 SmartARM2300 开发板上有一片 74HC595，可通过 SPI 接口对其进行数据读写操作，在其并行输出端连接了 8 个 LED 灯，本例程将通过 SPI0 接口控制与 74HC595 相连的 8 个 LED 产生流水灯效果。

请事先将 JP1 上 74HC595 的相关信号线与 JP3 上 SPI0 信号线用杜邦线连接。本例程中 SPI0 接口作主机，向 74HC595 写数据来产生流水灯效果，如程序清单 1.41 所示。

程序清单 1.41 SPI 主机程序

文件 1：LPC2300PinCfg.h

```
#define     P0_15_FNUC      P0_15_SCK       // P0.15 作为 SPI0 的串行时钟
#define     P0_17_FNUC      P0_17_MISO      // P0.17 作为 SPI0 主机输入从机输出端
#define     P0_18_FNUC      P0_18_MOSI      // P0.18 作为 SPI0 主机输出从机输入端
#define     P0_16_FNUC      P0_16_GPIO      // P0.16 作为 GPIO，用作从机片选引脚
```

文件 2：main.c

```
……
PinInit();                                      // 初始化引脚配置
……
// 流水灯显示数据
uint8 const DISP_TAB[] = {
    0x00, 0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3F, 0x7F,
    0xFF, 0x7F, 0x3F, 0x1F, 0x0F, 0x07, 0x03, 0x01
};
// SPI 初始化参数
char    const spiArg[] = "Mode=0 Cpha=0 Cpol=1 Lsbf=0 Speed=100000 CsPin=16 \
RWMode=1 RxBufSize=16 TxBufSize=32";

void TASK1    (void *pdata)
{
    uint32 i;
    uint8       rcvDat;

    pdata = pdata;
```

```
    if (OPERATE_FAIL == SPIInit(0, (char *)spiArg, NULL))// SPI 初始化
    {                                                    // 出错处理
        while (1);
    }
    if (0 == SetVICIRQ(SPI0_IRQ_CHN, 8, (uint32)SPI0_ISR))    // 设置中断
    {
        while (1);
    }

    while (1)
    {
        // 流水灯演示
        for(i = 0; i < (sizeof(DISP_TAB) / sizeof(DISP_TAB[0])); i++)
        {
            SPIWrite(SPI0, (uint8 *)(DISP_TAB + i), 1, NULL);
            OSTimeDly(OS_TICKS_PER_SEC / 5);
            SPIRead(SPI0, &rcvDat, 1, NULL);                  // 可在此设置断点观察回读的数据
        }
    }
}
```

1.12 SSP 接口应用

1.12.1 函数说明

SSP 驱动提供对 2 路同步串行接口的操作函数，目前只支持 SPI 模式。分别为 SspInit()、SspSetMode()、SspRead()、SspWrite()、SspISR()、SSP0_ISR()、SSP1_ISR()、SPI0_SSP0_ISR()。主要功能包括对 SSP 接口的初始化、参数设置、读数据、写数据以及相应的中断函数设置。

1.12.2 使用方法

使用驱动库操作 SSP 接口非常简单，工作流程图如图 1.12 所示。

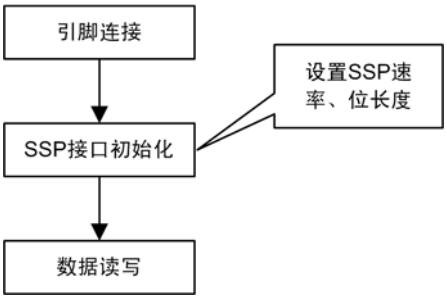

图 1.12 SSP 工作流程

1 引脚连接

与其它外设的使用一样，使用 SSP 接口时，首先应该配置引脚连接，如程序清单 1.42 所示。同时还可以设置引脚内部是否连接上/下拉电阻。在系统初始化时，必须调用“PinInit()”函数使能引脚连接配置。

程序清单 1.42 引脚功能配置

```
#define     P0_06_FNUC      P0_06_SSEL1
#define     P0_07_FNUC      P0_07_SCK1
#define     P0_08_FNUC      P0_08_MISO1
#define     P0_09_FNUC      P0_09_MOSI1
```

2 接口初始化

接口初始化包括参数以及中断的设置，如程序清单 1.43 所示。可只输入部分参数，未输入的参数将被设为默认值。

使用字符串进行参数传入时，具体请参照函数接口小节的参数设置部分。字符串中的参

数的具体内容及意义请参考 SSP 参数设置的说明。

程序清单 1.43　初始化 SSP 接口

```
char SspArg[] = "BusType=0 Mode=0 Speed=1000000 DataBits =8 \
                 ClkPol=0 CsPin=121 CsPol=0 RxBufSize=16 TxBufSize=16";
SspInit(SSP0, SspArg, NULL);                                    // SSP0 初始化
```

3　中断设置

SSP 的数据收发处理采用中断方式，故应设置中断服务函数才能工作。如果用户要同时使用 SPI0 和 SSP0 则中断服务函数应为 SPI0_SSP0_ISR()，因为 SPI0 和 SSP0 共享同一个中断通道号 10。中断的设置方法如程序清单 1.44 所示。

程序清单 1.44　中断设置

```
SetVICIRQ(SSP0_IRQ_CHN, 8, (uint32)SSP0_ISR);                   // 设置 SSP0 中断

// 若同时使用 SPI0 和 SSP0，则中断设置如下
SetVICIRQ(SSP0_IRQ_CHN, 8, (uint32)SPI0_SSP0_ISR);              // 设置 SPI0 和 SSP0 中断
```

4　数据读写

完成初始化后即可使用驱动程序操作相应的 SSP 设备进行数据的读写。

程序清单 1.45　SSP 收发数据

```
uint8 sData = 0;
uint8 rdbuf[10];
……
SspWrite(SSP0, &sData, 1, NULL);                                // 发送一帧数据
……
SspRead(SSP0, (void *)rdbuf, 10, NULL);                         // 读取 10 帧数据
```

1.12.3　示范例程

本驱动库支持 SSP 的 SPI 主机模式，代替独立 SPI 与其它外设通信。

用户可令 SSP 工作在 SPI 主机模式，控制 SmartARM2300 工控开发平台上的 74HC595 驱动 8 盏 LED 闪烁以形成流水灯，相关电路如所示。

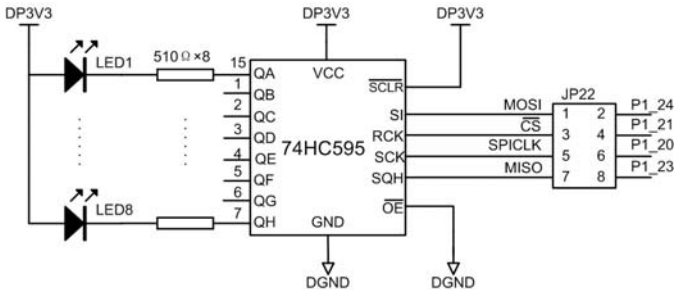

图 1.13　SSP 控制 74HC595 驱动 LED 示意图

为将处理器 SSP 引脚的信号引出，用户需将 JP1 上的/CS、MOSI、SCLK、MISO 分别和 P1.21、P1.24、P1.20、P1.23 短接，程序流程如图 1.14 所示。

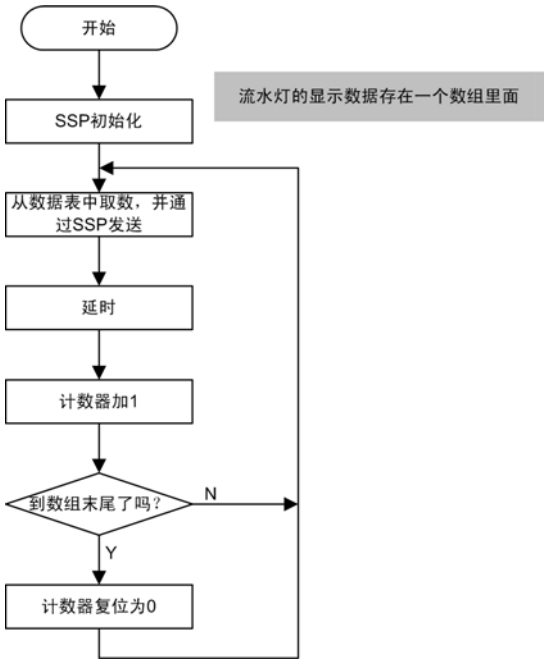

图 1.14　SSP SPI 主机模式应用示例流程图

如程序清单 1.46 所示的例程演示了使用 SSP0 数据发送的过程。

程序清单 1.46　SSP0 使用例程

文件 1：“LPC2300PinCfg.h”

```
#define     P1_20_FNUC          P1_20_SCK0
#define     P1_21_FNUC          P1_21_GPIO                  // P1.21 选择 GPIO 功能，用作从机片选
#define     P1_23_FNUC          P1_23_MISO0
#define     P1_24_FNUC          P1_24_MOSI0
```

文件 2：“main.c”

```
// 流水灯显示数据
uint8 const DISP_TAB[] = {
    0x00, 0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3F, 0x7F,
    0xFF, 0x7F, 0x3F, 0x1F, 0x0F, 0x07, 0x03, 0x01
};
char   const sspArg[] = {                                   // SSP 初始化参数
    "Speed=12000000 DataBits=8 OpMode=0 ClkPol=0 CsPin=121 ClkPha=0 \
RxBufSize=4 TxBufSize=4"
};

void TASK0 (void *pdata)
{
    pdata = pdata;
    if (OPERATE_FAIL == SspInit(SSP0, (char *)sspArg, NULL))        // SSP0 初始化
    {                                                               // 出错处理
        while (1);
    }
    if (0 == SetVICIRQ(SSP0_IRQ_CHN, 8, (uint32)SSP0_ISR))         // 设置中断
    {                                                               // 出错处理
        while (1);
    }
    while (1)
    {
        uint32  i;
        uint8       sndDat;
        uint8       rcvDat;
        // 流水灯演示
        for(i = 0; i < (sizeof(DISP_TAB) / sizeof(DISP_TAB[0])); i++)
```

```
        {
            sndDat = DISP_TAB[i];
            SspWrite(SSP0, &sndDat, 1, NULL);
            OSTimeDly(OS_TICKS_PER_SEC / 5);
            SspRead(SSP0, &rcvDat, 1, NULL); //  可在此设置断点观察回读的数据
        }
    }
}
```

程序运行，LED 将按照设置的花样不停地闪烁。这说明 SSP 成功地以 SPI 主机方式将要发送的数据发送给了 74HC595，并驱动 LED 显示成功。

同时在程序运行时，使用逻辑分析仪监视 SSP 总线，观察到一个数据帧的完整波形如图 1.15 所示。

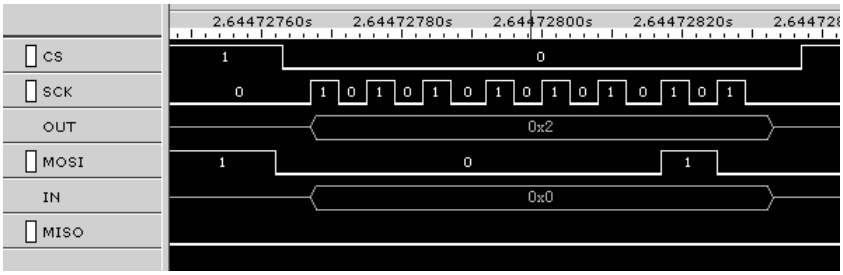

图 1.15　SSP 作 SPI 主机的总线时序

现对图 1.15 所示的波形作简要说明：

- CS 总线

使用 SSP 接口作为 SPI 总线应用时，SSEL 引脚可以自动输出 CS 信号时序，同时驱动库也允许用户自行定义任意 GPIO 作为 SPI 从机的片选信号，在设备初始化时定义即可。

从时序图可看出，在数据帧开始发送时，SSP 上的 CS 清“0”选通设备 74HC595，然后在数据帧发送结束时置“1”，取消对设备 74HC595 的选择状态。

- 时钟极性和 SCK 总线

在 SSP 总线初始化时设定“ClkPol=0”，即总线空闲时 SCK 为低电平。由图中波形可知该设置生效。

1.13 外部中断

1.13.1　函数说明

外部中断驱动提供了 4 个用户可用的 API 函数，分别为 EINTSetMode ()、EINTClr()、EINTISR()、EINT0_ISR()、EINT1_ISR()、EINT2_ISR()、EINT3_ISR()主要完成对外部中断 0、1、2、3 的模式设定、中断标志清除以及用户中断处理等功能。

1.13.2　使用方法

1　引脚连接

使用外部中断前，同样需要在 LPC2300PinCfg.h 文件中进行相应的引脚功能配置，如程序清单 1.47 所示为设置 P2.11 为外部中断 1。同时还可以设置引脚内部是否连接上/下拉电阻。在系统初始化时，必须调用函数 PinInit()使能引脚连接配置。

程序清单 1.47　引脚连接

```
// P2.11
#define    P2_11_GPIO          0x00          // GPIO
#define    P2_11_EINT1         0x01          //  外部中断 1 输入
#define    P2_11_MCIDAT1       0x02          // SD、MMC 接口数据线 1
#define    P2_11_I2STXCLK      0x03          // I2S 发送时钟 SCK，由主机驱动，从机接收

#define    P2_11_FNUC          P2_11_EINT1
```

2　模式设置

使用函数 EINTSetMode ()对外部中断进行模式设置，如程序清单 1.48 所示。

程序清单 1.48 模式设置

```
char    EintModeArg[] = "Mode=0";
EINTSetMode(EINT1, SET_INTMODE, EintModeArg);        // EINT1，下降沿触发
```

3 中断设置

如程序清单 1.49 所示，对外部中断进行中断响应函数设置。在“EINT.h”文件中为用户提供了中断服务函数，用户可以在 EINT0_ISR()、EINT1_ISR()、EINT2_ISR()、EINT3_ISR()中添加中断响应内容。

程序清单 1.49 中断设置

```
SetVICIRQ(EINT1_IRQ_CHN, 5, (uint32) EINT1_ISR); //  外部中断 1 为 IRQ 中断，优先级 5
```

合理设置外部中断的工作模式和向量中断控制器后，即可触发外部中断，进入服务程序，对外部事件进行响应。

1.13.3 示范例程

SmartARM2300 工控板上提供的按键可以用于模拟产生外部中断事件。将 JP1 的 P2.11(外部中断 1)引脚用杜邦线接到按键 KEY1（P0.6）上。由于芯片内部的引脚可以设置为上拉模式，因此，在外面不需要连接上拉电阻。

外部中断模式设置为下跳沿触发，如程序清单 1.50 所示。全速运行程序后，每按一次 KEY1，将进入一次中断服务程序，取反蜂鸣器引脚电平，控制其鸣叫。

程序清单 1.50 外部中断使用示例

文件 1：“LPC2300PinCfg.h”

```
#define      P2_11_FNUC            P2_11_EINT1  //  外部中断 1 输入
```

文件 2：“EINT.h”

```
void EINT1_ISR(void)
{
     //中断处理
     GpioCpl(P1_27);                                    //  取反蜂鸣器控制引脚
     EINTISR(EINT1);                                          //  外部中断 1 结束
}
```

文件 3：“main.c”

```
……
PinInit();                                                    //  引脚初始化
……
void   TASK1(void *pdata)
{
    char    EintModeArg[] = "IntMode=0";
    pdata = pdata;

      EINTSetMode(EINT1, SET_INTMODE, EintModeArg);          // EINT1，下降沿
      SetVICIRQ(EINT1_IRQ_CHN, 10, (uint32)EINT1_ISR);//  中断优先级为 10
      while (1)
      {
            OSTimeDly(OS_TICKS_PER_SEC);
}
```

1.14 ADC 应用

1.14.1 函数说明

ADC 驱动提供了 7 个用户可用的 API 函数，分别为 ADCInit()、ADCSetMode()、ADCStart()、ADCStop()、ADCRead()、ADCISR()和 ADC0_ISR ()主要完成对 AD 转换的初始化，模式设置，启动停止，数据读取及用户中断处理等功能。

1.14.2 使用方法

1 引脚连接

在对 ADC 设备初始化之前，应该配置引脚连接使对应的 AINn 引脚作为 ADC 采集输入端，如程序清单 1.51 所示。否则，AD 转换值不可靠。在系统初始化时，调用“PinInit()”函数使能引脚连接配置。

程序清单 1.51 引脚功能配置

```
#define     P0_23_FNUC      P0_23_AD00      // P0.23 作为 AIN0 输入
#define     P0_24_FNUC      P0_24_AD01      // P0.24 作为 AIN1 输入
```

2 初始化 ADC

初始化 ADC 设备如程序清单 1.52 所示。

程序清单 1.52 初始化 ADC

```
char    AdcArg[] = "Mode=0 Channel=0x01 ActCtrl=0x00 Speed=100000";

// 软件模式，选择通道 0，软件控制立即启动转换，ADC 时钟为 100kHz
ADCInit(ADC0, AdcArg,NULL);                     // 初始化 ADC
```

3 设置中断

程序清单 1.53 设置并使能 ADC 中断

```
SetVICIRQ(ADC0_IRQ_CHN, 7, (uint32)ADC0_ISR);  // 设置 ADC0 中断，优先级为 7
```

4 启动 ADC 转换

调用函数 ADCStart()启动 ADC 转换。

5 读取采样值

ADCRead()函数读取对应通道 AD 转换值，该值需要根据外部参考电压转换为实际电压值。转换公式如下：

$$U = \frac{V_{REF}}{2^N - 1} \times VALUE$$

其中 U 为实际电压值，VALUE 为 ADCRead()读取的采样结果，N 为 AD 转换的精度（3～10 位），Vref 为参考电压值。

设采集精度为 10bit，参考电压为 3.3V，读取采样数据的代码如程序清单 1.54 所示。

程序清单 1.54 读取 ADC 转换值

```
int32     AdcValue;                             // 采样电压值（单位：mV）

if (ADCRead(ADC0, &AdcValue, 1, NULL))          // 正确读到一个采样值
{
    AdcValue = AdcValue * 3300 / 1023 ;         // 参考电压 3.3V，转换后的电压以 mV 为单位
}
```

1.14.3 示范例程

以下范例均在 SmartARM2300 开发板上调试通过。该开发板已将 AD 转换器的输入通道 5（AIN5）连接到一个电位器 W1 上，短接 JP4 将模拟电源加到 W1 上。调节 W1 可在 AIN5 上获得 0～2.5V 的模拟输入。此外该开发板给 CPU 的 AD 转换器提供的基准源也是 2.5V。

使用本驱动库编写 AD 采样程序时，建议在中断服务函数使用信号量来通知处理任务转换结束，详见下面的示范例程。

1 软件控制模式 ADC 转换

示例一：软件控制立即启动方式

从 A/D 采样通道 5 输入一个 0～2.5V 的模拟电压信号，采样得到的电压值通过串口发

送到上位机显示。具体的程序代码如程序清单 1.55 所示。

程序清单 1.55　软件控制 ADC 单通道采样程序

文件 1：“LPC2300PinCFG.h”

```
#define      P1_31_FNUC            P1_31_AD05   // P1.31 作为 AIN5 输入
```

文件 2：“main.c”

```
#define             Vref          2500                    // 参考电压 2.5V
OS_EVENT      *semADCDone;                                // 转换结束信号量

char    uart_para[] = "BaudRate=115200 RxBufSize=4 TxBufSize=64";

// 软件模式，选择通道 5，立即启动转换，ADC 时钟为 100kHz
// 转换精度 10 位(软件模式下只能是 10 位)
char adc_para[]= "Mode=0 Channel=0x20 ActCtrl=0x00 Speed=100000";

void TASK0 (void *pdata)
{
      char Send_buf[14];                                    // 串口发送缓冲
      int32 AdcValue[1];
      uint8 err;
      pdata = pdata;

      semADCDone = OSSemCreate(NULL);
      if (semADCDone == NULL)
    {
        while(1);
    }

      UartInit(UART0, uart_para, NULL);                     // 初始化 UART
      SetVICIRQ(UART0_IRQ_CHN, 7, (uint32)UART0_ISR);
      ADCInit(ADC0, adc_para, NULL);                        // 初始化 ADC
      SetVICIRQ(ADC0_IRQ_CHN, 6, (uint32)ADC0_ISR);

      while(1)
      {
            ADCStart(ADC0, NULL);
            OSSemPend(semADCDone, 0, &err);

            if(ADCRead(ADC0, AdcValue, 1, NULL))    // 读取采样
            {
                  AdcValue[0] = AdcValue[0] * Vref / 1023;
                  // 发送到上位机观察
                  sprintf(Send_buf, "V=%dmV AIN%d\r\n", AdcValue[0], 5);
                  UartWrite(UART0, (uint8 *)Send_buf, strlen(Send_buf), NULL);
            }
            OSTimeDly(OS_TICKS_PER_SEC / 2);
      }
}
```

文件 3：“ADC.h”

```
__inline void ADC0_ISR (void)
{
     extern OS_EVENT *semADCDone;            // 声明引用的外部信号量

      ADCISR(ADC0);                          // AD 转换结束处理
     OSSemPost(semADCDone);                  // 发送转换完成信号量
      VICVectAddr = 0x00;                    // 中断处理结束
}
```

编译成功后通过 JTAG 下载到 SmartARM2300 开发板，全速运行程序，打开串口调试软件，设置波特率为 115200，数据位 8 位，停止位 1 位，无奇偶校验，在接收区观察到的采样结果如图 1.16 所示。

图 1.16　单通道采样结果

示例二：定时 ADC 转换

采用定时器的捕获功能启动 AD 转换，将转换的采样结果通过串口发送到上位机。具体的程序如程序清单 1.56 所示。

程序清单 1.56　定时 AD 采样程序

文件 1：“LPC2300PinCFG.h”

```
#define      P1_31_FNUC          P1_31_AD05        // P1.31 作为 AIN5 输入
```

文件 2：“main.c”

```
#define            Vref          2500              //  参考电压 2.5V
OS_EVENT     *semADCDone;                          //  转换结束信号量

char   uart_para[] = "BaudRate=115200 RxBufSize=8 TxBufSize=64";

//  软件控制模式，选择通道 5，上升边沿出现在 MAT1.0 时启动转换，ADC 时钟为 1M
char   adc_para[] = "Mode=0 Channel=0x20 ActCtrl=11 Speed=1000000";

char   Timer_para[] = "Mode=0";                     //  定时器 1 初始化为定时模式
//  设置定时器 1 的工作模式为匹配输出模式：选择匹配输出通道 0，匹配时间为 500ms
//  匹配后将 T1TC 复位，匹配后使对应外部匹配输出反转
char   TimerModeArg[] = "Channel=0 Time=500000 ActCtrl=2 OutCtrl=2";

void TASK0 (void *pdata)
{
      char Send_buf[14];                            //  串口发送缓冲

      int32 AdcValue;
      uint8 err;
      pdata = pdata;

      semADCDone = OSSemCreate(NULL);
      if (semADCDone == NULL)
    {
        while(1);
    }

      UartInit(UART0,uart_para, NULL);              //  初始化 UART
      SetVICIRQ(UART0_IRQ_CHN, 7, (uint32)UART0_ISR);
```

```
    ADCInit(ADC0, adc_para, NULL);                  // 初始化 ADC
    SetVICIRQ(ADC0_IRQ_CHN, 6, (uint32)ADC0_ISR);

    TimerInit(TIMER1, Timer_para, NULL);
    TimerSetMode(TIMER1, SET_MATMODE, TimerModeArg);

    TimerStart(TIMER1, NULL);                       // 启动定时器
    ADCStart(ADC0, NULL);                           // 启动 AD 转换

    while(1)
    {
        OSSemPend(semADCDone, 0, &err);
        if(ADCRead(ADC0, &AdcValue, 1, NULL))  // 读取采样值
        {
            AdcValue = AdcValue * Vref / 1023;
            // 发送到上位机观察
            sprintf(Send_buf, "V=%dmV AIN%d\r\n", AdcValue, 5);
            UartWrite(UART0, (uint8 *)Send_buf, strlen(Send_buf), NULL);
        }
        OSTimeDly(10);
    }
}
```

文件 3：“ADC.h”

```
__inline void ADC0_ISR (void)
{
    extern OS_EVENT *semADCDone;                // 声明引用的外部信号量

    //ADCStop(ADC0, NULL);                      // 停止 ADC
     ADCISR(ADC0);                              // AD 转换结束处理
    OSSemPost(semADCDone);                      // 发送转换完成信号量
     VICVectAddr = 0x00;                        // 中断处理结束
}
```

程序编译成功后下载到 SmartARM2300 开发板，全速运行程序，在打开串口调试软件，正确设置后可观察到如图 1.17 所示的转换结果。

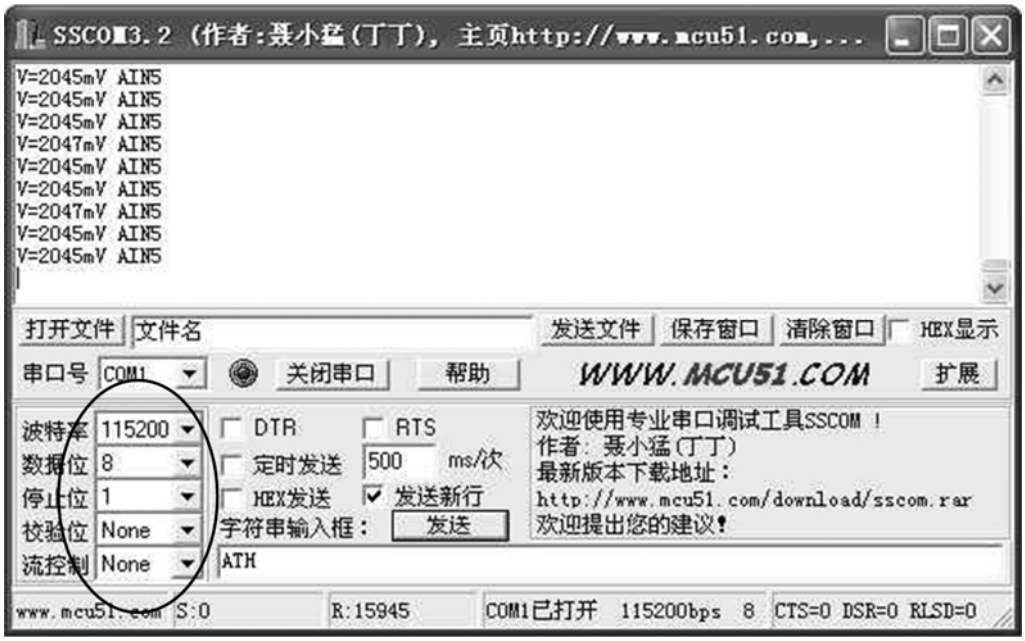

图 1.17　定时器匹配触发 AD 转换结果

2　突发（burst）模式多通道 ADC 转换

在 burst 模式下，顺序转换通道 0～5 的电压值，并把转换后的电压值通过串口发送到上位机。具体的程序见程序清单 1.57 所示。

程序清单 1.57　burst 模式多通道采样程序

文件 1：“LPC2300PinCFG.h”

```
#define      P0_23_FNUC          P0_23_ AD00  // P0.23 作为 AIN0 输入
#define      P0_24_FNUC          P0_24_ AD01  // P 0.24 作为 AIN1 输入
#define      P0_25_FNUC          P0_25_ AD02  // P0.25 作为 AIN2 输入
#define      P0_26_FNUC          P0_26_ AD03  // P0.26 作为 AIN3 输入
#define      P1_31_FNUC          P1_31_ AD05  // P1.31 作为 AIN5 输入
```

文件 2：“main.c”

```
#define          ADC_CHNS          5              // ADC 采样通道数
#define            Vref          2500             //  参考电压 2.5V

OS_EVENT      *semADCDone;                        //  转换结束信号量

char    uart_para[] = "BaudRate=115200 RxBufSize=8 TxBufSize=64";

// 突发模式，选择通道 0～3 和 5，转换精度 10 位，ADC 时钟为 100kHz
char    adc_para[]= "Mode=1 Channel=0x02F Preci=10 Speed=100000";

void TASK0 (void *pdata)
{
      char    Send_buf[14];                         //  串口发送缓冲

      int32    AdcValue[ADC_CHNS];
      uint8    err;
      int32    i;

      pdata = pdata;

      semADCDone = OSSemCreate(NULL);              //  创建信号量
      if (semADCDone == NULL)
    {
        while(1);
    }

      UartInit(UART0, uart_para, NULL);            //  初始化 UART
      SetVICIRQ(UART0_IRQ_CHN, 7, (uint32)UART0_ISR);

      ADCInit(ADC0, adc_para, NULL);               //  初始化 ADC
      SetVICIRQ(ADC0_IRQ_CHN,6,(uint32)ADC0_ISR);

      while(1)
      {
            ADCStart(ADC0, NULL);
            OSSemPend(semADCDone, 0, &err);        //  等待信号量

            // 读取  ADC_CHNS 个通道的采样值
            if(ADC_CHNS == ADCRead(ADC0, AdcValue, ADC_CHNS, NULL))
            {
                  for (i = 0; i < ADC_CHNS; i++)
                  {
                   AdcValue[i] = AdcValue[i] * Vref / 1023;
                         // 发送到上位机观察
                   sprintf(Send_buf, "V=%4dmV AIN%d\r\n", AdcValue[i], (i == 4) ? 5 : i);
                   UartWrite(UART0, (uint8 *)Send_buf, strlen(Send_buf), NULL);
                   OSTimeDly(1);
                }
               UartWrite(UART0, (uint8 *)"\r\n", 2, NULL);
            }
            OSTimeDly(OS_TICKS_PER_SEC);
```

```
    }
}
```

文件 3：“ADC.h”

```
__inline void ADC0_ISR (void)
{
    extern OS_EVENT *semADCDone;          // 声明引用的外部信号量

    ADCStop(ADC0, NULL);                  // 停止 ADC
     ADCISR(ADC0);                        // AD 转换结束处理
    OSSemPost(semADCDone);                // 发送转换完成信号量
     VICVectAddr = 0x00;                  // 中断处理结束
}
```

*注：由于 burst 模式下硬件会不停的循环自动转换，当设置的转换时钟较高时就会产生频繁的中断，影响其他任务的实时性，严重时会引起系统崩溃。故建议在中断服务函数中调用函数 ADCStop()停止 AD 转换，在用户处理任务中读完转换值后再重新启动。

程序编译成功后下载到 SmartARM2300 开发板，全速运行程序，打开串口调试软件，正确设置串口参数，观察到的采样情况如图 1.18 所示。

图 1.18 多通道 AD 转换结果

1.15 DAC 应用

1.15.1 函数说明

DAC 驱动提供了 2 个用户可用的 API 函数，分别为 DacInit()和 DacWrite()来完成对 DAC 转换的初始化及数据输出功能。

1.15.2 使用方法

1 引脚连接

在使用 DAC 设备之前，首先应该配置引脚连接，使数据从相应的引脚输出，如程序清单 1.58 所示。同时还可以设置引脚内部是否连接上/下拉电阻。在系统初始化时，调用“PinInit()”函数使能引脚连接配置。

程序清单 1.58 引脚连接

```
// P0.26
#define    P0_26_GPIO      0x00          // GPIO
#define    P0_26_AD03      0x01          // ADC-0，通道 3
```

```
#define     P0_26_AOUT      0x02            // DAC 输出
#define     P0_26_RXD3  0x03                // UART-3 接收引脚

#define     P0_26_FNUC      P0_26_AOUT      // P0.26 作为 DAC 输出
```

2 初始化 DAC

使用函数 DacInit()对 DAC 接口初始化，如程序清单 1.59 所示。

程序清单 1.59 初始化 DAC

```
char   DacArg[ ] = "DacVref=2500";
DacInit(DAC0, DacArg, NULL);                // DAC 初始化
```

3 数据输出

完成初始化后，即可使用函数 DacWrite()从对应引脚输出指定的模拟电压数据，如程序清单 1.60 所示。

程序清单 1.60 数据输出

```
uint16 Vo1[45];
DacWrite(DAC0, (uint16 *)Vo1, 45, NULL);        //  数据输出
```

1.15.3 示范例程

1 正弦波输出

DAC 的使用比较简单，如程序清单 1.62 所示，介绍了如何使用 DAC 输出正弦波的方法。要输出一个正弦波形，则需要一个正弦数组，从 0～2π平均取 45 个点（即 360 度，每 8 度取一个点），将得到的正弦值组成一个数组，在 Matlab 里面的计算方法如下：

```
>> x=0:2*pi/45:2*pi;
>> y=(sin(x)+1)*1024/2.5;
>> int16(y)
```

得到的数组如程序清单 1.61 所示。

程序清单 1.61 正弦数组

```
uint16   SinTable[45] = {                       //  正弦表，2π周期内取 45 点，每 8 度一点
    410,467,523,576,627,673,714,749,778,799,813,819,817,807,789,764,732,694,
    650,602,550,495,438,381,324,270,217,169,125, 87, 55, 30, 12,   2,   0,   6,
    20, 41, 70,105,146,193,243,297,353
};
```

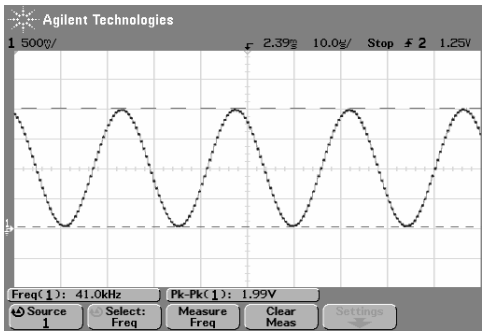

图 1.19 用示波器观察 DAC 输出正弦波

编译程序启动 AXD 调试，全速运行程序，使用示波器观察 SmartARM2300 工控开发平台上 CON8 的第四个引脚，可以看到如图 1.19 所示的正弦波。

图 1.19 是使用示波器的探头直接测量 D/A 的输出电压，如果外接一个简单的 RC 滤波器，截止频率大于 160KHz，即可得到更平滑的正弦波形，有兴趣的用户可以自己设计 RC 滤波电路观察。

程序清单 1.62 DAC 使用例程

文件 1：“LPC2300PinCfg.h”

```
#define      P0_26_FNUC            P0_26_AOUT        // P0.26 作为 DAC 输出
```

文件 2：“main.c”

```
uint16   SinTable[45] = {                                  // 正弦表，2π周期内取 45 点，每 8 度一点
    410,467,523,576,627,673,714,749,778,799,813,819,817,807,789,764,732,694,
    650,602,550,495,438,381,324,270,217,169,125, 87, 55, 30, 12,   2,   0,   6,
    20, 41, 70,105,146,193,243,297,353
};

void TASK0 (void *pdata)
{
    char          DacArg[] = "DacVref=2500";
    uint32 i = 0;
    uint16 Vo1[45];

     pdata = pdata;
     DacInit(DAC0, DacArg, NULL);                          // DAC 初始化
     for(i=0; i<45; i++)
     {
          Vo1[i] = (uint16)(SinTable[i] * 2500 / 1023 + 1);      // "+1"修正小数误差
     }
     while(1)
     {
          DacWrite(DAC0, (uint16 *)Vo1, 45, NULL);          // 输出正弦波
     }
}
```

2 DAC 工业应用方案

在工业应用现场，距离较远的电气设备、仪表之间进行信号传输时，通常存在干扰，导致系统不稳定甚至误操作。除系统内、外部干扰影响外，还有一个十分重要的原因就是各种仪器设备的接地处理问题。一般情况下，设备外壳都需接大地，电路系统也要有公共参考地。然而，由于各仪表设备的参考接地点之间通常会存在电势差，因而形成接地环路，由于地线环流会带来共模及差模噪声及干扰，常常导致系统工作不正常。

一个理想的解决方案是对设备进行电气隔离，这样原本相互联接的地线网络变为相互独立的单元，相互之间的干扰也将大大减小。

在工业自动化控制系统应用中，广泛采用 4~20mA 电流来传输控制与检测信号。4~20mA 电流环路抗干扰能力强，线路简单，可用来传输几十甚至几百米长的模拟信号。

LPC2300 系列 ARM 的工控开发平台配套有 ADC/DAC 信号调理模块，此模块主要包括：电源电路、A/D 调理电路以及 D/A 调理电路，结构如图 1.20 所示。这里我们主要描述 D/A 调理电路的原理。

由于 LPC2300 系列 ARM 的 D/A 参考电压 V_{REF} 使用了 ADR525 产生的 2.5V 精密电压源。而一般的工业信号是 0～5V，因此 LPC2300 系列 ARM 的 DAC 的模拟输出电路需经过放大到 0~5V 或者 V/I 转换成 0~20mA 信号方可输出。

DAC 输出模拟电路的原理图如图 1.20 所示，可以分为 2 部分：输出信号放大部分和 V/I 变换部分。

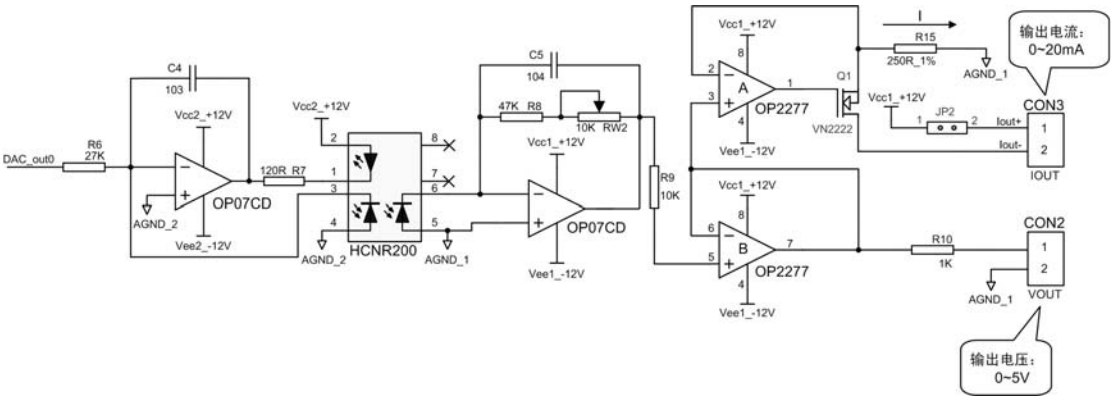

图 1.20　DAC 的输出调理电路图

隔离电路采用 HCNR200/201，如图 1.21 所示，设输入端电压为 VDAC_out，输出端电压为 Vout。

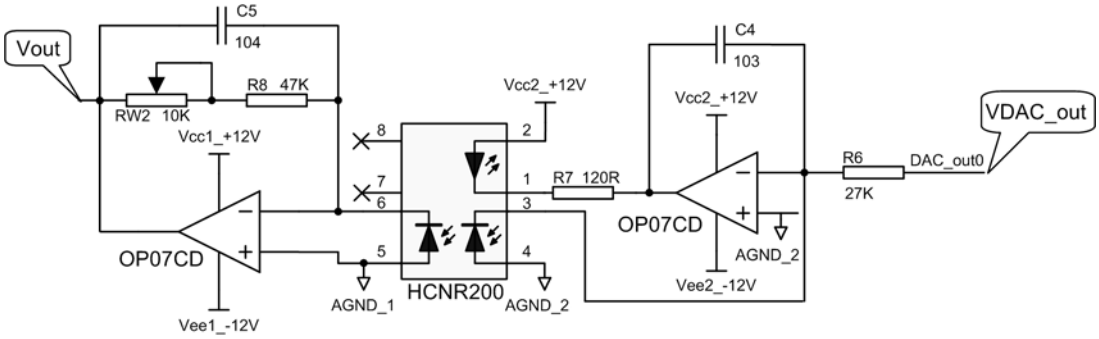

图 1.21　隔离电路：AMP=2

根据 HCNR200/201 的特性：

$$V_{out} = \frac{R_8 + R_{W2}}{R_6} V_{DAC\_out}$$

其中 R6= 27K，RW2 = 10K，R8 = 47K。调节 RW2 的值等于 7KΩ可以保证输出信号被放大 1 倍，这样就能将 LPC2300 系列 ARM 的 DAC 输出的 0～2.5V 放大到 0～5V。实现 V/I 变换电路如图 1.22 所示。

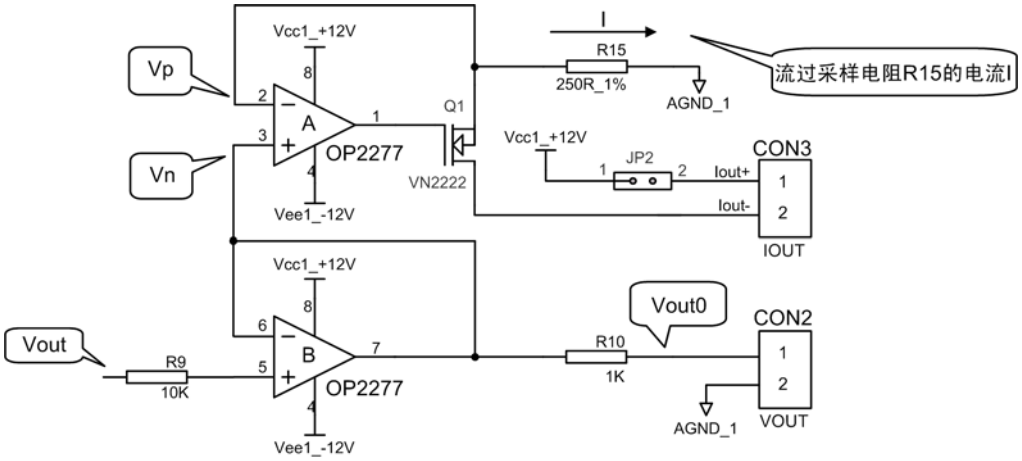

图 1.22　V/I 变换：0~5V/0~20mA

在图 1.22 中 Vout0 的范围为 0~5V，同样可知 Vn 为 0~5V，由于运放的开环增益很大，运放输出电压要高于 Vn 上的电压，使 MOS 管 Q6 导通，这样在电阻 R15 上的电流大小为：

I = Vn / R15，范围为 0~20mA，根据 MOS 管的特点，流过 MOS 管的漏极电流与源极电流是相等的，这样在 CON3 上外接负载，即可获取到 0~20mA 的电流。

1.16 PWM 应用

1.16.1 函数说明

PWM 驱动提供了 7 个用户可用的 API 函数，分别为 PWMInit()、PWMSetMode()、PWMStart()、PWMStop()、PWMReset()、PWMISR ()、PWM1_ISR()来完成 PWM 调制器的初始化、参数设置、启动输出、停止输出、重新启动函数及中断服务处理等功能。

1.16.2 使用方法

1 引脚连接

在使用 PWM 功能时，需要对引脚连接模块进行正确设置。例如，当进行程序清单 1.63 所示设置时，P2.2 将选择 PWM3 功能。在系统初始化时，必须调用“PinInit()”函数使能引脚连接配置。

程序清单 1.63 引脚功能配置

```
#define     P2_02_FNUC          P2_02_PWM13     // P2.2 选择 PWM-1，通道 3
```

2 初始化 PWM

按照输入的参数初始化 PWM 调制器，完成对 PWM 的初始化。如初始化设置为 PWM 输出模式，周期为 10000μs，PWMMR0 与 TC 匹配时复位 TC，如程序清单 1.64 所示。

程序清单 1.64 初始化 PWM

```
char   PwmArg [] = “Mode=0 Time=10000 ActCtrl=2”;
PWMInit(PWM1, PwmArg, NULL);          // 初始化 PWM1，周期 10000uS，匹配时复位 PWMTC
```

如果 PWM 做定时器使用，定时时间为 10ms，匹配时复位 PWMTC 并产生中断。则设置参数如下：

```
char   arg[] = "Mode=1 Time=10000 ActCtrl=3";
```

3 设置模式

接下来需要设置 PWM 的参数、模式。假设 PWM 初始化的周期为 10ms，通道 1 单边沿输出占空比为 30%，通道 2 双边沿输出占空比为 20%，则其设置如程序清单 1.65 所示。

程序清单 1.65 设置 PWM 模式

```
PWMSetMode(PWM1, SET_CTRLCHN,"Channel=1 Time=3000 Edge=0 ActCtrl=0 OutCtrl=1");
                // 通道 1，PWMMR1 匹配值为 3ms，单边沿，匹配时无动作，使能输出
 PWMSetMode(PWM1, SET_CTRLCHN,"Channel=2 Time=5000 Edge=1 ActCtrl=0 OutCtrl=1");
                // 通道 2，PWMMR2 匹配值为 5ms，双边沿，匹配时无动作，使能输出
```

如果作定时器使用，则需设置各通道的定时特性。如设置通道 1 的定时时间为 4ms，匹配时产生中断，如程序清单 1.66 所示。

程序清单 1.66 设置定时器模式

```
char   PwmModeArg[] = "Channel=1 Time=4000 ActCtrl=1";
PWMSetMode(PWM1, SET_CTRLCHN, PwmModeArg);
                                                  // 通道 1，定时 4ms，匹配时产生中断
```

4 启动/停止 PWM 定时器

设置好 PWM 的模式、周期、占空比数据以后即可启动 PWM 控制器输出对应波形，如程序清单 1.67 所示。

程序清单 1.67 启动 PWM 输出

```
PWMStart(PWM1);                                   // 启动 PWM1 输出
```

停止 PWM 输出见程序清单 1.68。

程序清单 1.68 停止 PWM 设备

```
PWMStop(PWM1);                                  // 停止 PWM
```

1.16.3 示范例程

1 PWM 输出

PWM 输出周期为 10000μs，通道 2 输出占空比 50%的单边沿波形，通道 5 输出占空比 70%的单边沿波形，通道 4 输出 10%占空比的双边沿正脉冲波形，如程序清单 1.69 所示。

程序清单 1.69 PWM 波形输出

文件 1：“LPC2300PinCfg.h”

```
#define     P1_20_FNUC      P1_20_PWM12     // P1.20 选择 PWM-1，通道 2
#define     P1_21_FNUC      P1_21_PWM13     // P1.21 选择 PWM-1，通道 3
#define     P1_23_FNUC      P1_23_PWM14     // P1.23 选择 PWM-1，通道 4
#define     P1_24_FNUC      P1_24_PWM15     // P1.24 选择 PWM-1，通道 5
```

文件 2：“main.c”

```
char    PwmArg[]    = "Mode=0 Time=10000 ActCtrl=2";

char    PwmArg2[] = "Channel=2 Time=5000 Edge=0 ActCtrl=0 OutCtrl=1";
char    PwmArg5[] = "Channel=5 Time=7000 Edge=0 ActCtrl=0 OutCtrl=1";

char    PwmArg3[] = "Channel=3 Time=8000 Edge=0 ActCtrl=0 OutCtrl=0";
char    PwmArg4[] = "Channel=4 Time=9000 Edge=1 ActCtrl=0 OutCtrl=1";

void TASK0 (void *pdata)
{
    pdata = pdata;

    PWMInit(PWM1, PwmArg, NULL);                    // 初始化，周期为 10ms

    PWMSetMode(PWM1, SET_CTRLCHN, PwmArg2);     // 单边沿输出，占空比 50%
    PWMSetMode(PWM1, SET_CTRLCHN, PwmArg5);     // 单边沿输出，占空比 70%

    // 双边沿输出，占空比 10%，正脉冲，需同时设置通道 3 和 4
    PWMSetMode(PWM1, SET_CTRLCHN, PwmArg3);
    PWMSetMode(PWM1, SET_CTRLCHN, PwmArg4);

    PWMStart(PWM1, NULL);

    while(1)
    {
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

程序全速运行，使用存储示波器测量和观察 PWM 输出波形。图 1.23 所示为通道 2 和 5 的单边沿波形，图 1.24 所示为通道 4 的双边沿正脉冲波形。

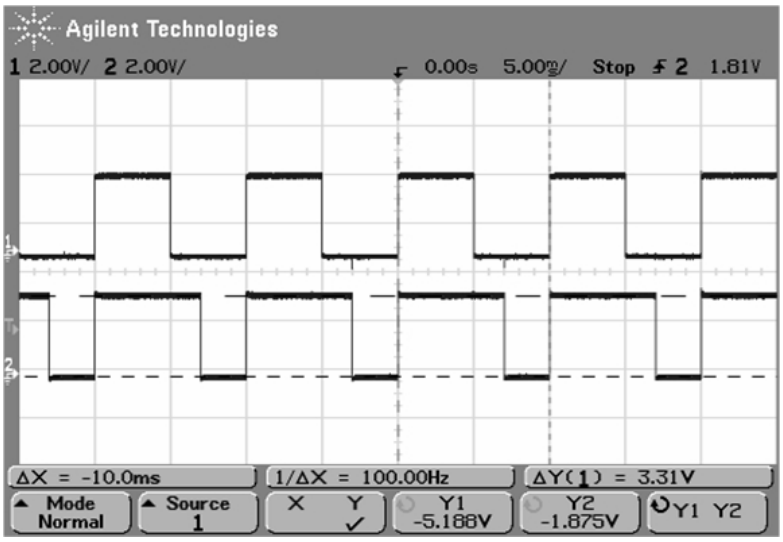

图 1.23　PWM 通道 2 和 5 单边沿输出波形

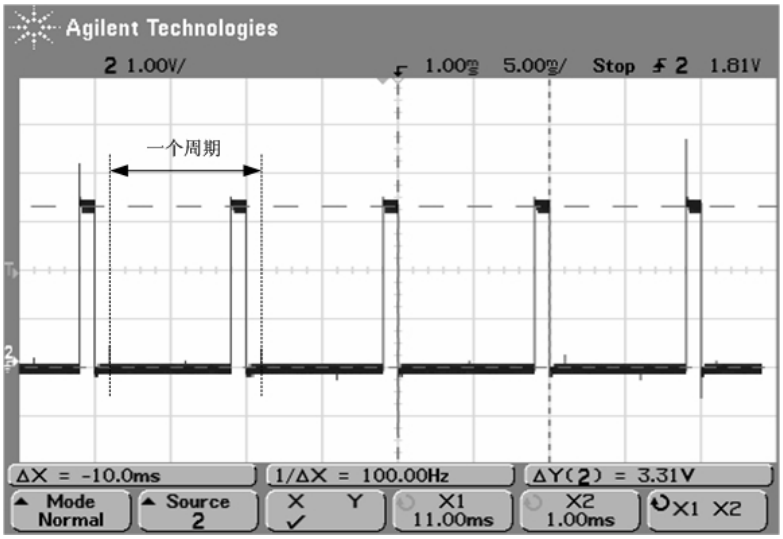

图 1.24　PWM 通道 5 双边沿（正脉冲）输出波形

2　PWM 用作定时器

虽然 LPC2300 有 4 个标准的 32 位定时器，但是在某些特定场合可能仍无法满足，此时如果不需要使用 PWM 功能，那么 PWM 部件可以作为 32 位定时器使用。本例程将演示 PWM 部件的定时器功能，并控制引脚 P1.27 电平取反（在 SmartARM2300 底板上 P1.27 为蜂鸣器控制引脚）。具体见程序清单 1.70。

程序清单 1.70　PWM 作定时器

文件 1：“LPC2300PinCFG.h”

```
#define      P1_27_FNUC            P1_27_GPIO          // P1.27 选择 GPIO
```

文件 2：“main.c”

```
void TASK0 (void *pdata)
{
    char    PwmInitArg[] = "Mode=1 Time=500000 ActCtrl=3";
      pdata = pdata;

      PWMInit(PWM1, PwmInitArg, NULL);                    // 初始化 PWM 为定时器模式，周期 500ms
      SetVICIRQ(PWM1_IRQ_CHN, 8, (uint32) PWM1_ISR);// 优先级为 8
      PWMStart(PWM1, NULL);
```

```
    while(1)
    {
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

文件 3：“PWM.h”

```
__inline void PWM1_ISR (void)
{
    // 添加用户中断处理代码
    GpioCpl(P1_27);                    // 取反蜂鸣器控制引脚

    PWMISR(PWM1);                      // 清除中断标志
    VICVectAddr = 0;                   // 通知 CPU 中断结束
}
```

全速运行程序，可听到蜂鸣器以每秒一次的频率鸣叫。也可使用示波器观察该引脚的波形。

1.17 实时时钟（RTC）

1.17.1 函数说明

RTC 驱动提供了 5 个用户可用的 API 函数，分别为 RTCInit()、RTCSetMode()、RTCRead()、RTCISR()、RTC0_ISR()来完成 RTC 的初始化、模式设置、时间值获取及中断服务处理等功能。

1.17.2 使用方法

1 初始化 RTC，设置当前时间值

首先要初始化 RTC，调用 RTCInit()函数设置当前时间，时钟开始运行。

2 设置 RTC 模式（可选）

调用 RTCSetMode()函数，设置 RTC 不同的模式。如设置秒中断如程序清单 1.71 所示。

程序清单 1.71 设置秒中断

```
RTCSetMode(RTC0, SET_SECINT, NULL);
```

3 设置中断处理函数（可选）

如果设置了中断还需要设置中断服务函数。如程序清单 1.72 所示。

程序清单 1.72 设置 RTC 中断

```
SetVICIRQ(RTC0_IRQ_CHN, 7, (uint32)RTC0_ISR);
```

4 读取时间值

调用 RTCRead()函数，可读取当前时间值或报警时间值。程序清单 1.73 所示为读取当前时间值。

程序清单 1.73 读取当前时间值

```
RTCTime    time;
RTCRead(RTC0, &time, SET_TIME, NULL);
```

1.17.3 示范例程

设置当前时间为 2007 年 8 月 28 日星期二，16 时 02 分 58 秒，时钟源选择内部预分频器的输出，外设频率使用默认值，秒增量产生中断，控制蜂鸣器鸣叫，并读取时间值通过串口发送到上位机显示。具体实现见程序清单 1.74。

程序清单 1.74 RTC 示例程序

文件 1：man.c

```
char        rtcArg[] = "Year=2007 Month=8 Day=28 Week=2 Hour=16 Min=02 Sec=58 ClkSrc=0";
char        uartArg[] = "RxBufSize=8 TxBufSize=64 BaudRate=115200";

void TASK0      (void *pdata)
{
    RTCTime   time;                                     // 保存时间值的结构体
    char        strTmp[32];                             // 保存字符串形式的时间值
    pdata = pdata;

    RTCInit(RTC0, rtcArg, NULL);                        // 初始化 RTC
    RTCSetMode(RTC0, SET_SECINT, NULL);                 // 设置秒增量中断
    SetVICIRQ(RTC0_IRQ_CHN, 7, (uint32)RTC0_ISR);       // 设置 RTC 中断
    UartInit(UART0, uartArg, NULL);                     // 初始化 UART0
    SetVICIRQ(UART0_IRQ_CHN, 8, (uint32)UART0_ISR);//  设置 UART0 中断

    while (1)
    {
        if(RTCRead(RTC0, &time, READ_TIME, NULL) == OPERATE_SUCCESS)
        {
            sprintf(strTmp, "%04d-%02d-%02d %01d %02d:%02d:%02d", \
                time.Year, time.Month, time.Day, time.Week, time.Hour, time.Minute, time.Second);
            strTmp[21] = '\r';
            strTmp[22] = '\n';
            UartWrite(UART0, (uint8 *)strTmp, 23, NULL);
            OSTimeDly(OS_TICKS_PER_SEC);
        }
    }
}
```

文件 2：RTC.h

```
__inline void RTC0_ISR(void)
{
    //添加用户程序
    GpioCpl(P1_27);                                     // 取反蜂鸣器控制引脚 P1.27

    RTCISR(RTC0);                                       // 清除中断标志
    VICVectAddr =0x00;                                  // 通知 CPU 中断结束
}
```

将程序下载到 SmartARM2300 开发板后，全速运行，并打开串口调试工具，正确设置参数，可观察到如图 1.25 所示时间显示，并每隔 1 秒可听到蜂鸣器鸣叫一次。

图 1.25 串口显示 RTC 时间

1.18 WDT 应用

1.18.1 函数说明

看门狗驱动提供了 5 个用户可用的 API 函数，分别为 WDTInit()、WDTStart()、WDTFeed()、WDT_ISR()来完成看门狗的初始化、启动、喂狗及中断服务处理等功能。

1.18.2 使用方法

1 初始化看门狗

调用 WDTInit()函数设置看门狗的时钟源、超时溢出时间和是否产生超时复位。参考程序清单 1.75。

程序清单 1.75 看门狗初始化

```
char   WdtInitArg[] = "ClkSrc=1 TimeOut=1000 RstEn=1"
                      // 看门狗初始化字符串，APB 外设时钟源，溢出时间 1000ms，溢出复位
WDTInit(WDT0, WdtInitArg, NULL);
```

2 添加中断程序（可选）

这一步可选，如需要使用看门狗定时器中断，必须先设置中断，并可在 WDT_ISR()（WDT.h 文件中）函数里添加用户执行中断处理代码，设置中断如程序清单 1.76 所示。

程序清单 1.76 设置看门狗中断

```
SetVICIRQ(WDT0_IRQ_CHN, 6, (uint32)WDT0_ISR); // 设置看门狗中断，优先级 6
```

3 定时喂狗

为防止看门狗超时溢出，要在溢出时间内调用看门狗喂狗函数 WDTFeed()。第一次正确喂狗操作时才启动 WDT。

```
WDTFeed(WDT0);                                  // 喂狗
```

1.18.3 示范例程

本实例演示的是看门狗复位功能，具体的见程序清单 1.77。

程序清单 1.77 看门狗复位

文件 1：“LPC2300PinCfg.h”

```
#define          P0_25_FNUC          P0_25_GPIO
#define          P1_27_FNUC          P1_27_GPIO
```

文件 2：“main.c”

```
void beep(void)                                 // 蜂鸣器鸣叫函数
{
    GpioClr(P1_27);
    OSTimeDly(OS_TICKS_PER_SEC);
    GpioSet(P1_27);
}

#define          KEY                 P0_25      // P0.25 做按键
#define          KEY_PRESSED         0

// 看门狗初始化字符串，选择 APB 外设时钟，定时时间为 1.2 秒，使能超时复位
char   wdtpara[] = "ClkSrc=1 TimeOut=1200 RstEn=1";
void TASK0 (void *pdata)
{
      pdata = pdata;

      beep();                                   // 蜂鸣器鸣叫两声
    WDTInit(WDT0, wdtpara, NULL);               // 看门狗定时器初始化
    WDTFeed(WDT0);                              // 第一次喂狗启动看门狗定时器
```

```
    while(1)
    {
        while(GpioGet(KEY) == KEY_PRESSED);        // 若 KEY 按下则停止喂狗
        WDTFeed(WDT0);                             // 喂狗
        OSTimeDly(OS_TICKS_PER_SEC);               // 喂狗时间约 1 秒
    }
}
```

实际操作时请将 KEY 定义为一个连接到按键的 GPIO 引脚。程序运行时，通过使用 KEY 来控制用户程序的执行。如果 KEY 没有按下，程序正常运行，间隔约 1 秒钟对看门狗“喂狗”一次。KEY 被按下后，程序将进入死循环，导致用户程序不能正常“喂狗”，看门狗内部计数器得不到有效清除就发生溢出，强制 CPU 复位。在用户程序开始运行处加入 beep()函数使蜂鸣器鸣叫一声，如果系统产生复位，则可以听到蜂鸣器鸣叫。

全速运行程序，系统在蜂鸣器鸣叫一声后开始正常运行，如果长按 KEY 不动，约 1.2 秒左右将听到蜂鸣器再次鸣叫，证明系统已经复位。

注：由于这里使用的是硬件看门狗，其溢出时间固定不变，且系统复位后，仿真不再有意义，所以不可以使用单步调试方法。应该全速运行程序，建议取下 JTAG 连接排线，直接脱机运行观察现象。

1.19 CAN 接口应用

1.19.1 函数说明

CAN驱动提供了5个用户可用的API函数，分别为CanInit()、CanSetMode()、CanRead ()、CanWrite ()、Can_ISR()来完成CAN控制器的初始化、模式设置、数据收发及中断服务处理等功能。

1.19.2 使用方法

使用本CAN驱动库进行CAN总线通信只需进行简单的配置。其基本流程如图1.26所示。

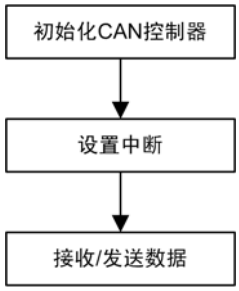

图 1.26 CAN 通信流程

1 初始化 CAN

使用CanInit()函数完成创建CAN控制器所需要的软硬件资源，并完成对应的引脚功能配置。如果初始化成功，将返回OPERATE_SUCCUSS，否则返回OPERATE_FAIL。如程序清单1.78所示。

程序清单 1.78 CAN 初始化

```
char pucCanArg[] = "BaudRate=1000000 RxBufSize=10 Mode=0";// 波特率 1M，缓冲 10 帧，正常模式
……
CanInit(CAN1, pucCanArg, NULL);                    // 初始化 CAN 控制器 0
```

2 设置中断响应函数

在初始化 CAN 控制器之后，收发数据之前，应该使用 SetVICIRQ()函数设置 CAN 中断响应函数。在 can.h 头文件中已经提供了 CAN 的中断服务函数，用户只需使用 SetVICIRQ()函数对其进行设置即可。如程序清单 1.79 所示。

程序清单 1.79 设置 CAN 中断响应函数

```
SetVICIRQ(CAN_IRQ_CHN, 7, (uint32)Can_ISR);              // 设置中断响应函数，优先级 7
```

3 发送数据

数据的发送使用 CanWrite()函数，以帧格式进行。发送数据前，需要根据 CAN 数据帧结构填写发送数据，然后使用 CanWrite()函数将数据发送到 CAN 总线，如程序清单 1.80 所示。

程序清单 1.80 发送数据

```
CANINFO    dat[1];                                        // 发送缓冲，大小为 1 帧
// 填充 CAN 发送数据，扩展帧，数据长度 8，数据 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88
dat[0].canrfs        = FILFRAME(0x08, FRAME_DATA, FRAME_EXT);
                                                        // 填充帧信息
dat[0].canid          = 0x55;                           // 帧 ID
dat[0].candat[0]    = 0x44332211;                       // 数据
dat[0].candat[1]    = 0x88776655;                       // 数据

CanWrite(CAN1, dat, 1, NULL);                           // 发送 1 帧数据
```

4 接收数据

CAN总线数据的接收通过CanRead()函数完成，如程序清单1.81所示。

程序清单 1.81 接收数据

```
CANINFO dat[1];                                         // 接收缓冲
if (1 == CanRead(CAN1, dat, 1, NULL)                    // 读一帧数据
{
    ……                                                  // 数据处理
}
```

1.19.3 验收滤波设置

CAN 的验收滤波是通过设置验收滤波表格进行的，就是将用户期望接收的帧的 ID 号按照规定放到表格里。CAN 控制器会自动过滤掉没有经过设置的帧。完整的验收滤波设置请参考后面的范例。

设置 CAN 验收滤波模式为旁路模式时将接收 CAN 总线上的任何数据，为使能模式则根据验收滤波表格选择性接收，为关闭模式时将不接收任何数据。因此要使用验收滤波功能就必须设置验收滤波模式为使能模式。

此外验收滤波模式及其表格是全局性的，即两路 CAN 控制器只能同时为同一种验收滤波模式，且共享同一个验收滤波表格设置。

1 验收表格元素定义

LPC2300 系列 ARM 的 CAN 控制器的验收滤波表格的具体定义方法请参考其用户手册等资料。为了方便用户，本驱动为验收表格元素的设置定义了宏（见文件 can.h），如程序清单 1.82 所示。

程序清单 1.82 验收滤波表格元素填充宏定义

```
enum LPC_AF_CANCH {AF_CH1=0, AF_CH2};          // CAN 验收滤波通道定义

/*******************************************************************************
**   CAN 验收滤波表格元素格式及填充宏定义
**   CH 为验收滤波通道号，取值范围为 AF_CH1 和 AF_CH2
**   SF 为要验收的标准帧 ID，如 0x08
**   LWSF 和 UPSF 为标准帧 ID 范围，如设置验收 ID 范围为：0x01～0x07，则 LWSF=0x01，UPSF=0x07
**   EF 为要验收的扩展帧 ID，如 0x80a
**   LWEF 和 UPEF 为扩展帧 ID 范围，如设置验收 ID 范围为：0x02～0x07，则 LWEF=0x02，UPEF=0x07
*******************************************************************************/
#define    AF_SSF(CH, SF)              (((CH) << 13) | ((SF) & 0x7FF))
                                                        // 标准帧验收过滤离散表格数据，16 位宽
#define    AF_GSF(CH, LWSF, UPSF)    (((AF_SSF(CH, LWSF)) << 16) | AF_SSF(CH, UPSF))
                                                        // 标准帧验收过滤范围表格数据，32 位宽
```

```
#define   AF_SEF(CH, EF)          (((CH) << 29) | ((EF) & 0x1FFFFFFF))
                                                  // 扩展帧验收过滤离散表格数据，32 位宽
#define   AF_GEF(CH, LWEF, UPEF)  ((((ulint64)AF_SEF(CH, LWEF)) << 32) | AF_SEF(CH, UPEF))
                                                  // 扩展帧验收过滤范围表格数据，64 位宽
```

要定义一个表格只需根据表格元素的位宽使用相应的数据类型定义一个数组，并使用程序清单 1.82 定义的宏来填充该数组即可。

2 验收表格定义

验收表格共有 4 个，用户可选择只设置其中的 1 个、2 个、3 个或全部，组合任意。所有的 CAN 通道均使用同一表格设置。验收表格及其说明如表 1.6 所示。

表 1.6 CAN 验收表格

表格名	说明
标准帧验收过滤离散表格	设置离散的标准帧 ID 如 0x02、0x03 等
标准帧验收过滤范围表格	设置标准帧 ID 范围如 0x09～0x1f
扩展帧验收过滤离散表格	设置离散的扩展帧 ID 如 0x02、0x03、0x805 等
扩展帧验收过滤范围表格	设置扩展帧 ID 范围如 0x09～0x1f

如要设置 CAN 通道 0 验收帧 ID 号为 0x00 和 0x03 的标准帧，CAN 通道 1 验收帧 ID 为 0x05 和 0x09 的标准帧，CAN 通道 0 验收帧 ID 在 0x801 至 0x809 范围内的扩展帧，则验收滤波表格的定义如程序清单 1.83 所示。

程序清单 1.83 验收滤波表格设置示例

```
unsigned short ssfTab[] = {                       // 标准帧验收过滤离散表格
      AF_SSF(AF_CH0, 0x00), AF_SSF(AF_CH0, 0x03),  // 通道 0 的验收表格元素
      AF_SSF(AF_CH1, 0x05), AF_SSF(AF_CH1, 0x09)   // 通道 1 的验收表格元素
};
unsigned long long gefTab [] = {                  // 扩展帧验收过滤范围表格数据
      AF_GEF (AF_CH0, 0x801, 0x809)               // 通道 0 的验收表格元素
}
```

*注：相同类型的表格数据放在同一个表格数组里，如通道 0 和通道 1 的离散验收表格数据应放在同一个数组里，顺序可任意。所有表格（最多 4 个）的大小（字节数）之和不能大于 2048 个字节。

3 验收表格的设置

用户定义的验收滤波表格要通过一个结构体来传递给 CanInit()函数。该结构体及相关数据类型的定义如程序清单 1.84 所示。该结构体的定义见 can.h 头文件。

程序清单 1.84 验收滤波表格结构体定义

```
/*****************************************************************************************
**   CAN 验收滤波相关定义
*****************************************************************************************/
typedef unsigned long long   ulint64;             // 无符号 64 位长整形数
typedef unsigned long        ulint32;             // 无符号 32 位长整形数

struct canaftab {                                 // 验收滤波表格结构体定义
      uint16       *pusSigSFTab;                  // 标准帧验收滤波离散表格指针
      uint32        uiSSFTabLen;                  // 标准帧验收滤波离散表格长度

      ulint32      *puiGrpSFTab;                  // 标准帧验收滤波范围表格指针
      uint32        uiGSFTabLen;                  // 标准帧验收滤波范围表格长度

      ulint32      *puiSigEFTab;                  // 扩展帧验收滤波离散表格指针
      uint32        uiSEFTabLen;                  // 扩展帧验收滤波离散表格长度

      ulint64      *pullGrpEFTab;                 // 扩展帧验收滤波范围表格指针
      uint32        uiGEFTabLen;                  // 扩展帧验收滤波范围表格长度
};
```

```
typedef struct canaftab        CANAFTAB;                              // 定义 CAN 验收滤波表格数据类型
```

将程序清单 1.83 定义的验收表格赋值给验收表格结构体的示意代码如程序清单 1.85 所示。

程序清单 1.85　验收表格结构体赋值示意代码

```
CANAFTAB canAfTab = 0;

canAfTab.pusSigSFTab       = ssfTab;                              // 标准帧验收滤波离散表格指针
canAfTab. uiSSFTabLen      = sizeof(ssfTab) / sizeof(ssfTab[0]);  // 标准帧验收滤波离散表格长度
canAfTab. puiGrpSFTab      = NULL;                                // 标准帧验收滤波范围表格指针
canAfTab. uiGSFTabLen      = 0;                                   // 标准帧验收滤波范围表格长度
canAfTab. puiSigEFTab      = NULL;                                // 扩展帧验收滤波离散表格指针
canAfTab. uiSEFTabLen      = 0;                                   // 扩展帧验收滤波离散表格长度
canAfTab. pullGrpEFTab     = gefTab;                              // 扩展帧验收滤波范围表格指针
canAfTab. uiGEFTabLen      = sizeof(gefTab) / sizeof(gefTab[0]);  // 扩展帧验收滤波范围表格长度
```

*注：结构体中未用到的指针元素应该赋值为 NULL，对应的表格长度元素应赋值为 0。所有 4 个验收表格的长度（以字节为单位）之和应小于等于 2048 个字节（2KB）。

最后调用 CanInit()函数来初始化验收滤波表格，如程序清单 1.86 所示。

程序清单 1.86　验收表格初始化示意代码

```
char    CanInitArg[] = "BaudRate=1000000 RxBufSize=50 Mode=0 AfMode=1";
CanInit(CAN1, CanInitArg, (void *)&canAfTab);                     // 通过 CanInit()函数设置表格
```

1.19.4　示范例程

利用 USB 转 CAN 模块（USBCAN 接口卡和 DB9 转接插座）和 ZLGCANTest 上位机工具实现下位机使用 CAN 网络与 PC 进行通信，如图 1.27 所示。

图 1.27　MiniARM 与 PC 通过 CAN 通讯

1　正常收发数据

下位机将从 CAN 总线接收到的数据原路发回，如程序清单 1.87 所示。

程序清单 1.87　CAN1 收发数据

文件 1：main.c

```
// CAN1 初始化字符串，波特率 1M，缓冲 20 帧，正常模式
char           pucCanArg[] = "BaudRate=1000000 RxBufSize=20 Mode=0";

// 任务 0 ，从 CAN1 接收数据，并将数据原路发送回去
void TASK0(void *pdata)
{
    int             iLen;
    CANINFO    dat[10];                                           // 数据缓冲区

    pdata = pdata;

    CanInit(CAN1, pucCanArg, NULL);                               // CAN 控制器 0 初始化
    SetVICIRQ(CAN_IRQ_CHN, 7, (uint32)Can_ISR);                   // 设置中断，优先级 7

    // 向上位机发送含有 7 个数据的扩展帧，数据为：0x11，0x22，0x33，0x44，0x55，0x66，0x77
    // 帧 ID 为 0x55，下面的代码示意如何填充帧信息
```

```
    dat[0].canrfs       = FILFRAME(0x07, FRAME_DATA, FRAME_EXT);
    dat[0].canid = 0x55;
    dat[0].candat[0] = 0x44332211;
    dat[0].candat[1] = 0x776655;
    CanWrite(CAN1, &dat[0], 1, NULL);                    // 发送该帧数据

    while (1)
    {
        iLen = CanRead(CAN1, dat, 10, NULL);
        if (iLen > 0)
        {
            int iTmp = 0;
            do
            {
                iTmp += CanWrite(CAN1, dat + iTmp, iLen - iTmp, NULL);
            } while (iTmp < iLen);
        }
        else
        {
            OSTimeDly(5);
        }
    }
}
```

注：Read()函数为非阻塞函数，超时时间为 100 毫秒。若需在任务中延时，为保证实时性，具体延时多少请根据所使用的波特率和实际应用来确定。

打开 ZLGCANTest 调试工具，选择菜单“设备操作\打开设备”进行相应设置（如图 1.28 所示）后启动设备。全速运行程序，在 ZLGCANTest 调试工具的数据显示区将看到下位机发送过来的一帧数据。输入要发送的数据，点击“发送”，将会在窗口看到发送出去的数据以及原路发回的数据，如图 1.29 所示。

图 1.28　设置 ZLGCANTest

图 1.29　数据的发送和接收

2　使用验收滤波方式

下面的例程带全部验收滤波表格设置。CAN 通道 1 验收帧 ID 为 0x01、0x05～0x07 及 0x09～0x0b 的标准帧和帧 ID 为 0x801、0x803 及 0x807～0x809 的扩展帧；CAN 通道 2 验收帧 ID 为 0x03、0x04 及 0x08～0x0a 的标准帧和 ID 为 0x804、0x805 及 0x80a～0x80c 的扩展帧。如程序清单 1.88 所示。

如用户要更改验收表格，只需在对应的表格数组中修改或增减表格元素，并确保所有 4 个表格字节数之和不超过 2048 字节。

程序清单 1.88　带验收滤波的 CAN 数据收发程序

文件 1：main.c

```
// 标准帧验收滤波离散 ID 数组，CAN1 通道验收具有以下帧 ID 的标准帧数据：0x01
//                                  CAN2 通道验收帧 ID 为：0x03，0x04
const uint16       ssfTab[] = {AF_SSF(AF_CH1, 0x01),
                               AF_SSF(AF_CH2, 0x03),AF_SSF(AF_CH2, 0x04)};

// 标准帧验收滤波范围 ID 数组
// CAN1 通道验收帧 ID 在以下范围内的标准帧数据：0x05~0x07，0x09~0x0b
//                                  CAN2 通道验收帧 ID 为：0x08~0x0A
const ulint32      gsfTab[] = {AF_GSF(AF_CH1, 0x05, 0x07), AF_GSF(AF_CH1, 0x09, 0x0b),
                               AF_GSF(AF_CH2, 0x0c, 0x0e)};

// 扩展帧验收滤波离散 ID 数组，CAN1 通道验收具有以下帧 ID 的扩展帧数据：0x801，0x803
//                                  CAN2 通道验收帧 ID 为：0x804，0x805
const ulint32      sefTab[] = {AF_SEF(AF_CH1, 0x801), AF_SEF(AF_CH1, 0x803),
                               AF_SEF(AF_CH2, 0x804), AF_SEF(AF_CH2, 0x805)};

// 扩展帧验收滤波范围 ID 数组，CAN1 通道验收帧 ID 在以下范围内的扩展帧数据：0x807~0x809
//                                  CAN2 通道验收帧 ID 为：0x80A~0x80C
const ulint64    gefTab[] = {AF_GEF(AF_CH1, 0x807, 0x809), AF_GEF(AF_CH2, 0x80A, 0x80C)};

const CANAFTAB canAFTab = {                                  // 验收滤波表格结构体
      (uint16 *)ssfTab,
      sizeof(ssfTab) / sizeof(ssfTab[0]),

      (ulint32 *)gsfTab,
```

```
    sizeof(gsfTab) / sizeof(gsfTab[0]),

    (ulint32 *)sefTab,
    sizeof(sefTab) / sizeof(sefTab[0]),

    (ulint64 *)gefTab,
    sizeof(gefTab) / sizeof(gefTab[0])
};

// CAN 初始化参数，波特率 1M，缓冲 50 帧，正常模式，使能验收滤波
char    pucCanArg[] = "BaudRate=1000000 RxBufSize=50 Mode=0 AfMode=1";

// TASK0，数据收发，带验收滤波设置
void TASK0    (void *pdata)
{
    int                 iLen;
    int                 iTmp = 0;
    CANINFO  dat[10];                                   // 数据缓冲区

    pdata = pdata;
    // 如果使用滤波，两路 CAN 必须都设置为滤波使能且只需要传递一次滤波表格
    CanInit(CAN1, pucCanArg, NULL);                         // 初始化 CAN 控制器 1
    CanInit(CAN2, pucCanArg, (void *)&canAFTab);        // 初始化 CAN 控制器 2
    SetVICIRQ(CAN_IRQ_CHN, 7, (uint32)Can_ISR);         // 设置中断，优先级 7

    while (1)
    {
        iLen = CanRead(CAN1, dat, 10, NULL);
        if (iLen > 0)
        {
            iTmp = 0;
            do
            {
                iTmp += CanWrite(CAN1, dat + iTmp, iLen - iTmp, NULL);
            } while (iTmp < iLen);
        }

        iLen = CanRead(CAN2, dat, 10, NULL);
        if (iLen > 0)
        {
            iTmp = 0;
            do
            {
                iTmp += CanWrite(CAN2, dat + iTmp, iLen - iTmp, NULL);
            } while (iTmp < iLen);
        }
    }
}
```

将 CAN1、CAN2 和 USB 转 CAN 模块相连。将程序下载到 SmartARM2300 开放板，全速运行程序。打开 ZLGCANTest 调试工具，设置好各项参数。发送标准帧，起始帧 ID 为 0x00，每发送一帧帧 ID 递增，一次性发送 10 帧，其收发情况如图 1.30 所示。

可以看到发送帧的 ID 范围为 0x0～0x9，但没有收到 ID 为 0x2 和 0x8 的帧。这是因为这两个 ID 号不在验收表格的定义范围之内，被 CAN 控制器过滤掉了。

图 1.30　带验收滤波的 CAN 数据收发

用户可自行实验发送标准帧或扩展帧并改变帧 ID 的范围，观察滤波的效果。

2. 高级软件资源

2.1 TCP/IP 协议栈

2.1.1 函数列表

有关 TCP/IP 协议栈的所有 API 函数（见表 2.2）所示。

表 2.1 TCP/IP 协议栈 API 函数一览表

函数名称	功能描述
ZlgipInitial	初始化 TCP/IP 协议栈
socket	创建一个 SOCKET
bind	对已创建但尚未连接的 SOCKET 绑定本地 IP 地址和服务端口
listen	将 TCP 服务器置入监听模式，并设定服务器需要监听的连接数
accept	用于 TCP 服务器确认客户机的连接
connect	用于 TCP 主动连接（一般是 TCP 客户端）
recv	TCP 通讯连接建立后，读取所得到的数据
recvfrom	用于 UDP 通讯时接收数据
send	TCP 通讯连接建立后，发送数据
sendto	UDP 通讯方式时发送数据
close	关断 TCP 连接
TCP_Abort	断开 TCP 连接
closesocket	删除已建立的 SOCKET
getlocalip	用于获取对应网络端口的 IP 地址
getpeername	在 SCOKET 建立后，获取 SOCKET 的本地绑定 IP 和端口
getsockname	功能与 getpeername()相同，在 SCOKET 建立后，获取 SOCKET 的本地绑定 IP 和端口
getsocktcpsta	用于 TCP 通讯时获取连接状态
getsockcliaddr	用于 TCP 通讯时获取对方的 IP 地址和端口
htonl	改变长整形数据的排列顺序，把 CPU 顺序变成网络顺序
htons	改变短整形数据的排列顺序，把 CPU 顺序变成网络顺序
inet_addr	把 XXX.XXX.XXX.XXX 的 IP 地址字符转换为网络顺序的长整形
inet_ntoa	改变数组型的 IP 地址数据为字符型的 XXX.XXX.XXX.XXX 地址结构
ntohl	改变长整形数据的排列顺序，把网络顺序变成 CPU 顺序
ntohs	改变短整形数据的排列顺序，把网络顺序变成 CPU 顺序

2.1.2 ZLG/IP 使用说明

SOCKET API函数从使用的方式来分有3种，一种是通用函数，就是TCP或UDP通信都使用的函数；一种是TCP专用函数，就是只在TCP通信中使用的函数；一种是UDP专用函数，就是只在UDP通信中使用的函数。在后面将分TCP和UDP两种不同的通信方式来进行介绍。

1 初始化

使用 ZLG/IP 函数库时，必须先调用函数 ZlgipInitial()，对 TCP/IP 协议栈进行初始化。如程序清单 2.1 所示。

程序清单 2.1 初始化

```
unsigned char   GucMCU_Ip [4]         = {192, 168, 100, 15};          //  本机 IP 设置
unsigned char   GucMCU_Gateway [4] = {192, 168, 100, 254};          //  本机网关设置
unsigned char   GucMCU_Mark [4]          = {255, 255, 255, 0};          //  本机子网掩码设置
unsigned char   GucMCU_Dns [4]           = {192, 168, 100, 115};        // DNS 服务器 IP 设置
ZlgipInitial(GucMCU_Ip, GucMCU_Gateway, GucMCU_Mark, GucMCU_Dns);
                                                                        // ZLG/IP 协议栈初始化
```

2 SOCKET API 函数在 TCP 通信中的使用

TCP 通信的任务分为服务器方式和客户机方式两种。服务器方式是需要监听连接，只有在与客户机建立连接后才能进行数据处理。客户机方式是主动连接服务器，它也是在连接

成功后才能进行数据处理。如图 2.1 所示。就是 TCP 通讯时服务器端和客户机端通讯的函数应用过程图。

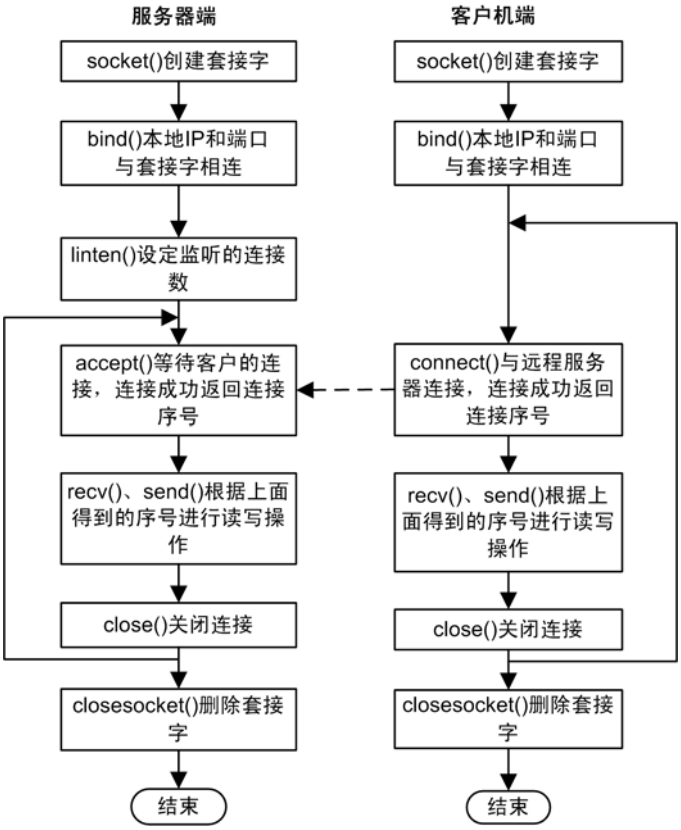

图 2.1　TCP 通讯时 SOCKET API 函数的应用

注:

1、当 MCU 与 PC 进行通信时，TCP&UDP 测试工具已完成 PC 一端的程序，只需进行相应设置；我们所编写的程序都是站在 MCU 的立场上写的，即使用函数 bind()绑定本地地址，这个地址是指 MCU 的 IP。

2、作为服务器端，当连接断开时必须先 close()关闭连接（最好再 TCP_Abor()释放资源），才能重新 accept()获得连接；当为客户端时并不一定需要此操作，为了保证程序的可靠运行也应该添加。close()与 TCP_Abort()之间的延时不应少于 100ms。

3　SOCKET API 函数在 UDP 通信中的使用

UDP 通信时没有客户机和服务器之分，如图 2.2 就是 UDP 通讯时函数应用过程图。

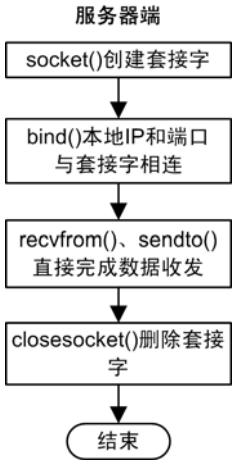

图 2.2　UDP 通讯时 SOCKET API 函数的应用

2.1.3 示范例程

1 UDP 通信例程

下面是一个简单的 UDP 通信示例。PC 机以 UDP 通信方式发送数据到 MCU，MCU 把接收到的数据回发给 PC 机，使用 TCP_UDP_Debug.exe 软件进行调试。具体的程序见程序清单 2.2。

程序清单 2.2 LPC2300 与 PC 机的 UDP 通信

```
unsigned char   GucMCU_Ip [4]          = {192, 168, 1, 240};          //  本机 IP 设置
unsigned char   GucMCU_Gateway [4] = {192, 168, 1, 253};          //  本机网关设置
unsigned char   GucMCU_Mark [4]       = {255, 255, 255, 0};  //  本机子网掩码设置
unsigned char   GucMCU_Dns [4]        = {192, 168, 100, 115};      // DNS 服务器 IP 设置
unsigned short   GusMCU_Port             = 8000;                        //  本机端口号
……
ZlgipInitial(GucMCU_Ip, GucMCU_Gateway, GucMCU_Mark, GucMCU_Dns);
                                                                       // ZLG/IP 协议栈初始化
……
#define   UDP_rece_length      1000                                    // UDP 一次接收缓存长度

void TASK0(void *pdata)
{
      SOCKET     s;
      int             ei;
      uint16        eii;
uint16        LenNet;
      uint8          UDP_rece_data[UDP_rece_length];                  // UDP 一次接收缓存
      struct sockaddr    serveraddr, clientaddr;
      pdata = pdata;

      clientaddr.sin_family    = 0;                                    //  设置客户端 IP 和端口
      clientaddr.sin_addr[0]  = GucMCU_Ip [0];
      clientaddr.sin_addr[1]  = GucMCU_Ip [1];
      clientaddr.sin_addr[2]  = GucMCU_Ip [2];
      clientaddr.sin_addr[3]  = GucMCU_Ip [3];
      clientaddr.sin_port      = GusMCU_Port;

      while(1)
      {
          s = socket( PF_INET, SOCK_DGRAM, UDP_PROTOCOL);
                                                                       //  建立 UDP 通信 SOCKET(不可重入函数)
          if(s != SOCKET_ERROR)
          {
              break;
          }
          OSTimeDly(OS_TICKS_PER_SEC/10);
    }

    while(1)
      {
          ei = bind(s, (struct sockaddr*)&clientaddr, sizeof(clientaddr));     //  绑定 MCU(客户端)端口
          if(ei != SOCKET_ERROR)
          {
              break;
          }
         OSTimeDly(OS_TICKS_PER_SEC/10);
    }

      while (1)
     {
      LenNet = recvfrom(s, UDP_rece_data, UDP_rece_length, 0, &serveraddr, &eii);
                                                                       //  接收服务器的 UDP 数据
      if (LenNet > 0)
```

```
        {
          sendto(s, UDP_rece_data, LenNet, 0, &serveraddr, sizeof(serveraddr) );
                                                    // 把接收到的 UDP 数据回发给服务器
      }
          OSTimeDly(OS_TICKS_PER_SEC / 200);
    }
}
```

程序成功下载后，LPC2300 与 PC 机通过网线连接，全速运行程序。在 PC 机上运行 TCP_UDP_Debug.exe 软件，创建连接，类型为 UDP，目标 IP 为程序中所设置的 MCU 的 IP，便可以进行数据的发送与接收了。如图 2.3 所示。

图 2.3　LPC2300 与 PC 机的 UDP 通信

2　TCP 客户端通信例程

以下用一个简单的程序来说明 TCP 客户端通信方式。LPC2300 和 PC 机建立 TCP 连接，PC 机作为服务器，LPC2300 把从 PC 机发过来的数据回发到 PC 机，使用 TCP_UDP_Debug.exe 软件进行调试。具体的程序见程序清单 2.3 所示。

程序清单 2.3　MCU 作客户端与 PC 机的 TCP 通信

```
unsigned char   GucMCU_Ip [4]         = {192, 168, 1, 240};          // 本机 IP 设置
unsigned char   GucMCU_Gateway [4] = {192, 168, 1, 254};          // 本机网关设置
unsigned char   GucMCU_Mark [4]          = {255, 255, 255, 0};          // 本机子网掩码设置
unsigned char   GucMCU_Dns [4]           = {192, 168, 100, 115};        // DNS 服务器 IP 设置
unsigned short   GusMCU_Port             = 8000;                        // 本机端口号
unsigned char   GucPeer_IP [4]           = {192, 168,   1, 89};         // 初始化 PC 的 IP 数组等信息
unsigned short   GusPeer_Port            = 2000;

……
ZlgipInitial(GucMCU_Ip, GucMCU_Gateway, GucMCU_Mark, GucMCU_Dns);
// ZLG/IP 协议栈初始化
……
```

```
#define     TCP_rece_length             1000                    // TCP 一次接收缓存长度

void TASK0(void *pdata)
{
        struct sockaddr    serveraddr, clientaddr;
        SOCKET    s;
        int             ei;
        INT8U       Temp;                                       // TCP 状态顺序号
        uint32        LenNet;
        uint8         TCP_rece_data[TCP_rece_length];           // TCP 一次接收缓存

        pdata = pdata;
        serveraddr.sin_family   = 0;                            //  设置服务器端 IP 和端口
        serveraddr.sin_addr[0] = GucPeer_IP[0];
        serveraddr.sin_addr[1] = GucPeer_IP[1];
        serveraddr.sin_addr[2] = GucPeer_IP[2];
        serveraddr.sin_addr[3] = GucPeer_IP[3];
serveraddr.sin_port      = GusPeer_Port;

        clientaddr.sin_family   = 0;                            //  设置客户端 IP 和端口
        clientaddr.sin_addr[0]  = GucMCU_Ip [0];
        clientaddr.sin_addr[1]  = GucMCU_Ip [1];
        clientaddr.sin_addr[2]  = GucMCU_Ip [2];
        clientaddr.sin_addr[3]  = GucMCU_Ip [3];
        clientaddr.sin_port       = GusMCU_Port;

        while(1)
        {
              s = socket(PF_INET, SOCK_STREAM, TCP_PROTOCOL);           //  建立 TCP 通信 SOCKET
              if(s != SOCKET_ERROR)
              {
                    break;
              }
              OSTimeDly(OS_TICKS_PER_SEC / 20);
        }

        while(1)
       {
              ei = bind(s,(struct sockaddr *)&clientaddr,sizeof(clientaddr));           //  绑定 MCU（客户端）端口
              if(ei != SOCKET_ERROR)
              {
                    break;
              }
              OSTimeDly(OS_TICKS_PER_SEC / 20);
        }

     while (1)
     {
        Temp = connect(s, (struct sockaddr*)&serveraddr, sizeof(serveraddr) );      //  连接服务器
              if(Temp != SOCKET_ERROR)
              {
              while (getsocktcpsta(Temp)==3)                //  确保服务器处于连接状态
              {
                          LenNet = recv(Temp, TCP_rece_data, TCP_rece_length, 0);
                                                                //  以 TCP 通信方式接收服务器端的数据
                          if (LenNet == SOCKET_RCV_ERROR)
                          {
                                break;
                          }
                          else if (LenNet > 0)
                          {
                          send(Temp, TCP_rece_data, LenNet, 0);//  把接收到的数据回发给服务器
                          }
```

```
                    OSTimeDly(OS_TICKS_PER_SEC/200);
                }
                close(Temp);
                OSTimeDly(OS_TICKS_PER_SEC / 10);
                TCP_Abort(Temp);
                OSTimeDly(OS_TICKS_PER_SEC / 10);
        }
        if(clientaddr.sin_port != (GusMCU_Port +100))
                clientaddr.sin_port++;
        else
                clientaddr.sin_port = GusMCU_Port;
        ei = bind(s, (struct sockaddr*)&clientaddr,sizeof(clientaddr)); // 更换端口重新绑定
    }
}
```

程序成功下载后，通过网线将 LPC2300 与 PC 机相连，全速运行程序。在 PC 机上运行 TCP_UDP_Debug.exe 软件，创建一个服务器，端口号设为 2000，启动服务器。当与客户机建立连接后便可以进行数据发送与接收了，如图 2.4 所示。

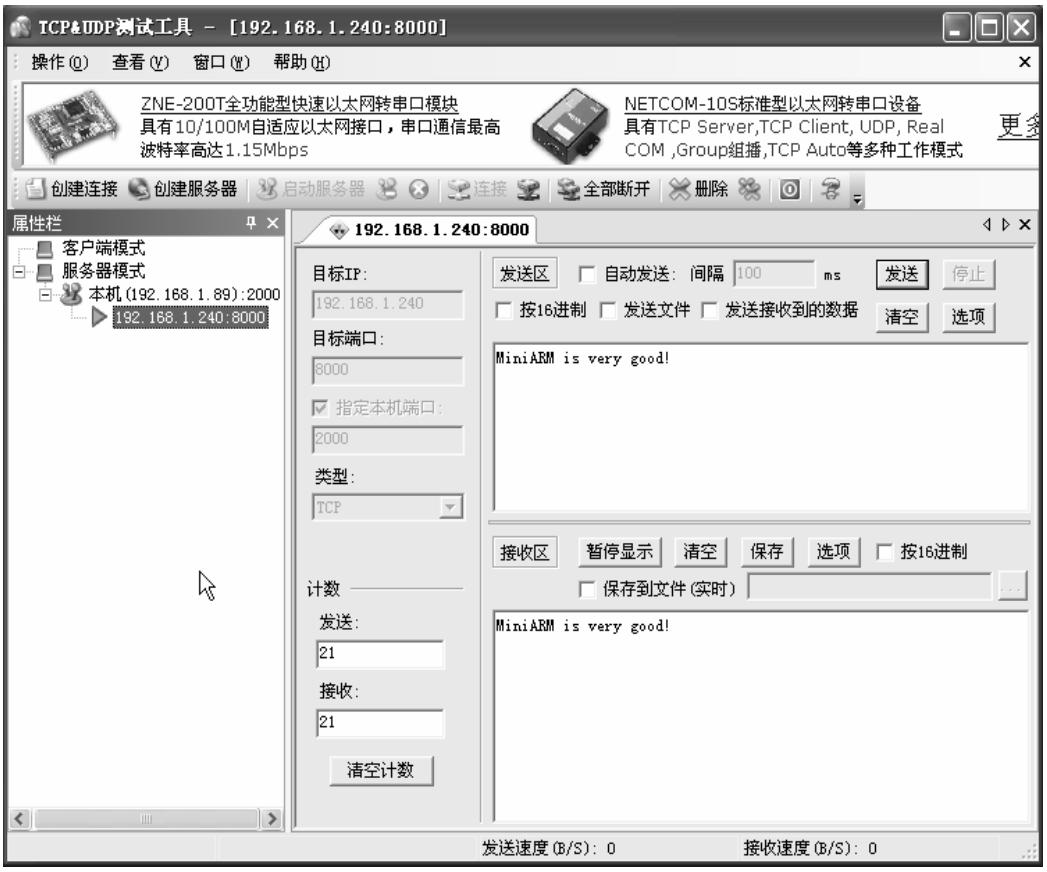

图 2.4　LPC2300 产品作客户端与 PC 机的 TCP 通信

3　TCP 服务器通信例程

LPC2300 工控系列产品与 PC 机使用 TCP 的通信方式进行连接。LPC2300 工控系列产品作为服务器，把从 PC 机发过来的数据发回到 PC 机，使用 TCP_UDP_Debug.exe 软件进行调试。具体的程序见程序清单 2.4 所示。

程序清单 2.4　LPC2300 作服务器与 PC 机的 TCP 通信

```
unsigned char   GucMCU_Ip [4]       = {192, 168, 1, 240};       // 本机 IP 设置
unsigned char   GucMCU_Gateway [4] = {192, 168, 1, 254};       // 本机网关设置
unsigned char   GucMCU_Mark [4]       = {255, 255, 255, 0};     // 本机子网掩码设置
unsigned char   GucMCU_Dns [4]        = {192, 168, 100, 115};   // DNS 服务器 IP 设置
unsigned short  GusMCU_Port           = 8000;                   // 本机端口号
```

```
……
ZlgipInitial(GucMCU_Ip, GucMCU_Gateway, GucMCU_Mark, GucMCU_Dns);
// ZLG/IP 协议栈初始化

#define    TCP_rece_length            1000                              // TCP 一次接收缓存长度

void TASK0(void *pdata)
{
      struct sockaddr    serveraddr, clientaddr;
      SOCKET      s;
      int              ei;
      uint8           Temp;                                             // TCP 状态顺序号
    uint32         LenNet;
    uint8          TCP_rece_data[TCP_rece_length];                     // TCP 一次接收缓存

      pdata = pdata;
      serveraddr.sin_family   = 0;                                      //  设置服务器端 IP 和端口
      serveraddr.sin_addr[0] = GucMCU_Ip [0];
      serveraddr.sin_addr[1] = GucMCU_Ip [1];
      serveraddr.sin_addr[2] = GucMCU_Ip [2];
      serveraddr.sin_addr[3] = GucMCU_Ip [3];
    serveraddr.sin_port = GusMCU_Port;

      while(1)
     {
           s = socket(AF_INET, SOCK_STREAM, TCP_PROTOCOL);    //  建立 TCP 通信 SOCKET
           if(s != SOCKET_ERROR)
           {
                break;
           }
           OSTimeDly(OS_TICKS_PER_SEC / 10);
      }
      while(1)
     {
           ei = bind(s,(struct sockaddr *)&serveraddr,sizeof(serveraddr));   //  绑定 MCU（客户端）端口
           if(ei != SOCKET_ERROR)
          {
                 break;
            }
            OSTimeDly(OS_TICKS_PER_SEC / 10);
      }
      while(1)
      {
         ei = listen(s, 2);                                                 //  设定监听的连接数
           if(ei != SOCKET_ERROR)
          {
                 break;
            }
      }
    while (1)
    {
         Temp = accept(s, (struct sockaddr *)&clientaddr, (int *)sizeof(clientaddr));
           if(Temp != SOCKET_ERROR)
          {
            while (getsocktcpsta(Temp)==3)
                {                                                           //  确保服务器处于连接状态
                      LenNet = recv(Temp, TCP_rece_data, TCP_rece_length, 0);
                                                                          //  以 TCP 通信方式接收客户机端的数据
                      if (LenNet == SOCKET_RCV_ERROR)
                    {
                           break;
                      }
                    else if (LenNet > 0)
```

```
                {
                        send(Temp, TCP_rece_data, LenNet, 0); // 把接收到的数据回发给客户机
                    }
                    OSTimeDly(OS_TICKS_PER_SEC);
                }
                close(Temp);
            OSTimeDly(OS_TICKS_PER_SEC / 10);
            TCP_Abort(Temp);                                    // 释放资源
            OSTimeDly(OS_TICKS_PER_SEC / 10);
        }
        OSTimeDly(OS_TICKS_PER_SEC/10);
    }
}
```

程序成功下载后，通过网线将 MiniARM 工控板与 PC 机相连，全速运行程序。在 PC 机上运行 TCP_UDP_Debug.exe 软件，创建连接，类型为 TCP，目标 IP 为程序中所设置的 MCU 的 IP，这时便可以进行数据发送与接收了。如图 2.5 所示。

图 2.5　LPC2300 工控系列产品作服务器与 PC 机的 TCP 通信

4　修改固件中的 IP

固件中固化有 MCU 的 IP 等信息，可以通过特定的接口函数来修改固件中的 IP 等信息。本示范例程使用串口发送特定的数据来修改固件中的 IP 等信息，其实也可用其他方式来改变。具体的修改程序如程序清单 2.5 所示。

程序清单 2.5　修改固件中的 IP 等信息

```
uint8 MCU_IP[4];
uint8 MCU_Mark[4];
uint8 MCU_GateWay[4];
uint16      MCU_Port;
uint8 MCU_Dns[4];
uint8 PC_IP[4];
uint16      PC_Port;
int main(void)
{
    ……
    GetIpSet(MCU_IP);                                   // 读取 IP 等信息
    GetMarkSet(MCU_Mark);
    GetGateWaySet(MCU_GateWay);
```

```
    GetDNSSet(MCU_Dns);
    MCU_Port = GetPortSet();
    GetServerIpSet(PC_IP);
    PC_Port = GetServerPortSet();
    ……
}
……
#define   Uart0_BuffLength          170                    // TCP 一次接收缓存长度
void ChangeIP(uint8 *arg,uint8 Rece_Count)                 //  改变 IP 函数
{
    int32   Err;
    int32   Cnt;

    int32   MIP_Sign        = 0;
    int32   MMark_Sign          = 0;
    int32   MGateWay_Sign       = 0;
    int32   MDns_Sign           = 0;
    int32   MPort_Sign          = 0;
    int32   PIP_Sign        = 0;
    int32   PPort_Sign          = 0;

    for (Cnt = 0; Cnt < Rece_Count; Cnt++)
    {
        if (strncmp((char *)arg+Cnt, "MCU_IP:", 7) == 0)
        {
            MIP_Sign = Cnt;
        }
        else if (strncmp((char *)arg + Cnt, "MCU_Mark:", 9) == 0)
        {
             MMark_Sign = Cnt;
        }
        else if (strncmp((char *)arg + Cnt, "MCU_GateWay:", 12) == 0)
        {
             MGateWay_Sign = Cnt;
        }
        else if (strncmp((char *)arg + Cnt, "MCU_Dns:", 8) == 0)
        {
            MDns_Sign = Cnt;
        }
        else if (strncmp((char *)arg + Cnt, "MCU_Port:", 9) == 0)
        {
            MPort_Sign = Cnt;
        }
        else if (strncmp((char *)arg + Cnt, "PC_IP:", 6) == 0)
        {
            PIP_Sign = Cnt;
        }
        else if (strncmp((char *)arg + Cnt, "PC_Port:", 8) == 0)
        {
            PPort_Sign = Cnt;
        }
    }
    if(MIP_Sign != 0)                                      //  接收处理新的 MCU IP 地址
    {
        MCU_IP[0] = (uint8)(((arg[MIP_Sign+7]    - 0x30) * 100) +
                            ((arg[MIP_Sign+8]    - 0x30) * 10) +
                            ( arg[MIP_Sign+9]    - 0x30));
        MCU_IP[1] = (uint8)(((arg[MIP_Sign+11] - 0x30) * 100) +
                            ((arg[MIP_Sign+12] - 0x30) * 10) +
                            ( arg[MIP_Sign+13] - 0x30));
        MCU_IP[2] = (uint8)(((arg[MIP_Sign+15] - 0x30) * 100) +
                            ((arg[MIP_Sign+16] - 0x30) * 10) +
                            ( arg[MIP_Sign+17] - 0x30));
        MCU_IP[3] = (uint8)(((arg[MIP_Sign+19] - 0x30) * 100) +
                            ((arg[MIP_Sign+20] - 0x30) * 10) +
```

```
                    ( arg[MIP_Sign+21] - 0x30));
}
if(MMark_Sign != 0)                                    // 接收处理新的 MCU 子网掩码
{
    MCU_Mark[0] = (uint8)(((arg[MMark_Sign+9]  - 0x30) * 100) +
                          ((arg[MMark_Sign+10] - 0x30) * 10) +
                          ( arg[MMark_Sign+11] - 0x30));
    MCU_Mark[1] = (uint8)(((arg[MMark_Sign+13] - 0x30) * 100) +
                          ((arg[MMark_Sign+14] - 0x30) * 10) +
                          ( arg[MMark_Sign+15] - 0x30));
    MCU_Mark[2] = (uint8)(((arg[MMark_Sign+17] - 0x30) * 100) +
                          ((arg[MMark_Sign+18] - 0x30) * 10) +
                          ( arg[MMark_Sign+19] - 0x30));
    MCU_Mark[3] = (uint8)(((arg[MMark_Sign+21] - 0x30) * 100) +
                          ((arg[MMark_Sign+22] - 0x30) * 10) +
                          ( arg[MMark_Sign+23] - 0x30));
}
if(MGateWay_Sign != 0)                                 // 接收处理新的 MCU 网关
{
    MCU_GateWay[0] = (uint8)(((arg[MGateWay_Sign+12] - 0x30) * 100) +
                             ((arg[MGateWay_Sign+13] - 0x30) * 10) +
                             ( arg[MGateWay_Sign+14] - 0x30));
    MCU_GateWay[1] = (uint8)(((arg[MGateWay_Sign+16] - 0x30) * 100) +
                             ((arg[MGateWay_Sign+17] - 0x30) * 10) +
                             ( arg[MGateWay_Sign+18] - 0x30));
    MCU_GateWay[2] = (uint8)(((arg[MGateWay_Sign+20] - 0x30) * 100) +
                             ((arg[MGateWay_Sign+21] - 0x30) * 10) +
                             ( arg[MGateWay_Sign+22] - 0x30));
    MCU_GateWay[3] = (uint8)(((arg[MGateWay_Sign+24] - 0x30) * 100) +
                             ((arg[MGateWay_Sign+25] - 0x30) * 10) +
                             ( arg[MGateWay_Sign+26] - 0x30));
}

if(MDns_Sign != 0)                                     // 接收处理新的 MCU 网关
{
    MCU_Dns[0] = (uint8)(((arg[MDns_Sign+8] - 0x30) * 100) +
                         ((arg[MDns_Sign+9] - 0x30) * 10) +
                         ( arg[MDns_Sign+10] - 0x30));
    MCU_Dns[1] = (uint8)(((arg[MDns_Sign+12] - 0x30) * 100) +
                         ((arg[MDns_Sign+13] - 0x30) * 10) +
                         ( arg[MDns_Sign+14] - 0x30));
    MCU_Dns[2] = (uint8)(((arg[MDns_Sign+16] - 0x30) * 100) +
                         ((arg[MDns_Sign+17] - 0x30) * 10) +
                         ( arg[MDns_Sign+18] - 0x30));
    MCU_Dns[3] = (uint8)(((arg[MDns_Sign+20] - 0x30) * 100) +
                         ((arg[MDns_Sign+21] - 0x30) * 10) +
                         ( arg[MDns_Sign+22] - 0x30));
}

if(MPort_Sign != 0)                                    // 接收处理新的 MCU 端口
{
    MCU_Port = (uint16)(((arg[MPort_Sign+9]  - 0x30) * 1000) +
                        ((arg[MPort_Sign+10] - 0x30) * 100) +
                        ((arg[MPort_Sign+11] - 0x30) * 10) +
                        ( arg[MPort_Sign+12] - 0x30));
}
if(PIP_Sign != 0)                                      // 接收处理新的 PC IP 地址
{
    PC_IP[0] = (uint8)(((arg[PIP_Sign+6] - 0x30) * 100) +
                       ((arg[PIP_Sign+7] - 0x30) * 10) +
                       ( arg[PIP_Sign+8] - 0x30));
    PC_IP[1] = (uint8)(((arg[PIP_Sign+10] - 0x30) * 100) +
                       ((arg[PIP_Sign+11] - 0x30) * 10) +
                       ( arg[PIP_Sign+12] - 0x30));
    PC_IP[2] = (uint8)(((arg[PIP_Sign+14] - 0x30) * 100) +
```

```
                        ((arg[PIP_Sign+15] - 0x30) * 10) +
                        ( arg[PIP_Sign+16] - 0x30));
            PC_IP[3] = (uint8)(((arg[PIP_Sign+18] - 0x30) * 100) +
                        ((arg[PIP_Sign+19] - 0x30) * 10) +
                        ( arg[PIP_Sign+20] - 0x30));
        }
        if(PPort_Sign != 0)                                         // 接收处理新的 PC 端口
        {
            PC_Port = (uint16)(((arg[PPort_Sign+8]    - 0x30) * 1000) +
                        ((arg[PPort_Sign+9]    - 0x30) * 100) +
                        ((arg[PPort_Sign+10] - 0x30) * 10) +
                        ( arg[PPort_Sign+11] - 0x30));
        }

        Err = SaveIpSet(MCU_IP[0],MCU_IP[1], MCU_IP[2], MCU_IP[3]);
//  存储新的 MCU IP 地址
        while (Err == FALSE)
        {
            Err = SaveIpSet(MCU_IP[0], MCU_IP[1], MCU_IP[2], MCU_IP[3]);
        }
        OSTimeDly(OS_TICKS_PER_SEC / 10);

        Err = SaveMarkSet(MCU_Mark[0], MCU_Mark[1], MCU_Mark[2], MCU_Mark[3]);
                                                                    // 存储新的 MCU  子网掩码
        while (Err == FALSE)
        {
            Err = SaveMarkSet(MCU_Mark[0], MCU_Mark[1], MCU_Mark[2], MCU_Mark[3]);
        }
        OSTimeDly(OS_TICKS_PER_SEC / 10);                           // 等待完成

        Err = SaveGateWaySet(MCU_GateWay[0],
                             MCU_GateWay[1],
                             MCU_GateWay[2],
                               MCU_GateWay[3]);                     // 存储新的 MCU  网关
        while (Err == FALSE)
        {
            Err = SaveGateWaySet( MCU_GateWay[0],
                                  MCU_GateWay[1],
                                  MCU_GateWay[2],
                                     MCU_GateWay[3]);
        }
        OSTimeDly(OS_TICKS_PER_SEC / 10);

        Err = SaveDNSSet(MCU_Dns);                                  // 存储新的 DNS
        while (Err == FALSE)
        {
            Err = SaveDNSSet(MCU_Dns);
        }
        OSTimeDly(OS_TICKS_PER_SEC / 10);

        Err = SavePortSet(MCU_Port);                                // 存储新的 MCU  端口
        while (Err == FALSE)
        {
            Err = SavePortSet(MCU_Port);
        }
        OSTimeDly(OS_TICKS_PER_SEC / 10);

        Err = SaveServerIpSet(PC_IP[0], PC_IP[1], PC_IP[2], PC_IP[3]);
//  存储新的 PC IP 地址
        while (Err == FALSE)
        {
            Err = SaveServerIpSet(PC_IP[0], PC_IP[1], PC_IP[2], PC_IP[3]);
        }
        OSTimeDly(OS_TICKS_PER_SEC / 10);
```

```
    Err = SaveServerPortSet(PC_Port);                          // 存储新的 PC 端口
    while (Err == FALSE)
    {
        Err = SaveServerPortSet(PC_Port);
    }
}

void TASK0 (void    *pdata)
{
    char       uart_para[66]="BaudRate=115200 TxBufSize=40 RxBufSize=170";
    uint32     Rece_Count    = 0;
    uint8      UART_Rece_Buff[Uart0_BuffLength];

    pdata = pdata;
    while(UartInit(UART0, uart_para, NULL) == 0);              // 初始化串口 0 参数
    SetVICIRQ(UART0_IRQ_CHN, 7, (unsigned int)UART0_ISR);
//  设置 UART0 中断，优先级为 7

    while (1)
    {
        Rece_Count += UartRead(UART0,UART_Rece_Buff+Rece_Count, Uart0_BuffLength, NULL);
        if (Rece_Count > 0)
          {
            if((UART_Rece_Buff[0] == 'A')&&(UART_Rece_Buff[Rece_Count-1] == 'Z'))
              {
                ChangeIP(UART_Rece_Buff,(uint8)Rece_Count);
                GpioClr(P1_27);
                OSTimeDly(OS_TICKS_PER_SEC / 2);
GpioSet(P1_27);
                OSTimeDly(OS_TICKS_PER_SEC / 2);
                Rece_Count = 0;
            }
            else if (UART_Rece_Buff[0] != 'A')
              {
                Rece_Count = 0;
            }
            OSTimeDly(OS_TICKS_PER_SEC / 10);
        }
          else
          {
            OSTimeDly(OS_TICKS_PER_SEC/ 10);
        }
    }
}
```

程序下载成功后，全速运行。使用串口向 LPC2300 工控系列产品上发送特定的字符串，字符串格式是：

A
MCU_IP:XXX.XXX.XXX.XXX
MCU_Mark:XXX.XXX.XXX.XXX
MCU_GateWay:XXX.XXX.XXX.XXX
MCU_Dns:XXX.XXX.XXX.XXX
MCU_Port:XXXX
PC_IP:XXX.XXX.XXX.XXX
PC_Port:XXXX
Z

可把具体的字符串保存在 TXT 文件中，使用串口发送。修改成功蜂鸣器鸣叫一声。再次使用 ADS 调试，可读出保存在固件中的 IP 信息。

2.2 USB Device 协议栈

2.2.1 函数列表

有关 USB Device 协议栈的所有 API 函数（见表 2.2）所示。

表 2.2 USB Device 协议栈 API 函数一览表

函数名称	功能描述
USB_Initialize	初始化 USB 控制器，初始化相关变量、信号量等
USB_ReadPort1	从端口 1（物理端点 2）接收字节
USB_ReadPort2	从端口 2（物理端点 4）接收字节
USB_WritePort1	用端口 1（物理端点 3）发送字节
USB_WritePort2	用端口 2（物理端点 5）发送字节

2.2.2 使用说明

1 初始化 USB

调用 USB_Initialize()函数对 USB 设备进行初始化。

2 数据收发

调用 USB_ReadPort1()，USB_WritePort1()等函数进行数据的收发。

2.2.3 示范例程

本示例欲通过 LPC2300 系列工控产品 USB 设备控制器的逻辑端点 2 收发大量数据。PC 机端应用程序首先调用 EPC2378USB.dll 中的 API 函数一次发送 3072 字节数据。下位机应用程序调用 ZLG/USB23xx 软件包的 API 函数接收完 3072 个字节后，再利用逻辑端点 2 的 IN 端点将这 3072 字节数据发回 PC。具体的程序如程序清单 2.6 所示。

程序清单 2.6 USB 收发数据

```
#define   BUFFER_SIZE          3072                  // 数据缓冲区大小

INT8U   GucDatabuffer[BUFFER_SIZE] = {0};          // USB 数据传输缓冲区

void   TASK0 (void *pdata)
{
      INT8U      ucErr = 0;

    pdata = pdata;                                  // 防止编译器报警
      while(1)
{
            ucErr = USB_ReadPort2 (BUFFER_SIZE, GucDatabuffer,200);
                                                    // 接收 PC 发送的 3072 字节数据
            if (ucErr == OS_NO_ERR)
            {                                       // 若接收正确完成
                  while(1)
                  {
                        ucErr = USB_WritePort2 (BUFFER_SIZE, GucDatabuffer, 200);
                                                    // 将收到的 3072 字节数据发回 PC
                        if(ucErr != USB_ERR_ENDP_OCCUPY)
// 获取端点失败则继续循环
                        {
                              break;
                        }
                        OSTimeDly(10);
                  }
            }
            OSTimeDly(10);
      }
}
```

程序下载成功后，将 LPC2300 系列工控产品与 PC 机用 USB 数据线相连。若为第一次

应用本 USB 设备控制器，PC 机将弹出如图 2.6 所示的“发现新硬件”提示。然后弹出使用新硬件向导对话框，选择“自动安装软件”或“从列表或指定位置安装”复选框，单击下一步；在弹出的对话框中选择“不要搜索。我要自己选择要安装的驱动程序”复选框，单击下一步；在弹出的警告对话框中单击“仍然继续”；单击浏览选择 LPC2300 系列 ARM USB 设备控制器 PC 驱动程序所在文件夹，再单击确定按扭；完成 USB 设备控制器的 PC 驱动程序的安装。

图 2.6　发现新硬件提示

这时，LPC2300 系列工控产品上的 Link 指示灯由暗变亮，表示 USB 设备枚举成功，程序运行正常。运行 PC 机端的可执行文件，文件名为 ZLG_USB23xx_Much.exe。对 USB 设备进行数据的收发，如图 2.7 所示。

图 2.7　PC 机收到的 4096 字节数据

2.3　FAT 文件管理系统

2.3.1 函数列表

有关 FAT 文件系统的所有 API 函数如表 2.3 所示。

表 2.3　文件系统 API 函数一览表

函数名称	功能描述
GetFsVer	获取文件系统版本
FileInit	初始化文件系统
FileOpen	以指定方式打开文件
FileClose	关闭指定文件
FileRead	读取文件
FileWrite	写文件
FileCloseAll	关闭所有打开的文件
FileEof	判断文件是否到读\写到文件尾
FileSeek	移动文件读\写位置
RemoveFile	删除文件
RenameFile	文件改名
GetFileSize	获得指定文件大小

续上表

GetFileDataTime	获得指定文件最后写时间
GetFileOffset	获得文件当前读写位置位置
TempDirOpen	打开临时目录，用于快速建立临时文件
TempDirClose	关闭指定临时目录
TempFileOpen	快速建立文件，用户需要自己保证文件名的唯一性及不重复打开
GetDrive	获取指定目录的驱动器
ChangeDrive	改变当前逻辑盘
ChangeDir	改变当前目录
MakeDir	建立目录
RemoveDir	删除目录
mount	加载卷（已分配盘符），允许读写
umount	卸载卷（已分配盘符），禁止读写。
AddFileDriver	增加物理盘
RemoveFileDriver	删除一个物理盘
AllCacheWriteBack	把所有已改变的扇区写回逻辑盘
FindFirst	在指定目录查看第一个文件/目录名称
FindNext	在指定目录查看下一个文件/目录名称
ChangeCodePage	改变本地编码
Encode	将本地编码字符转换成 Unicode 编码字符
Decode	将 Unicode 编码字符串转换成本地编码字符串
Strupr	把字符串转换成大写
GetLastDrive	获取最后一个逻辑盘符
CheckDrive	检查指定逻辑盘是否可用
AutoMount	自动加载逻辑盘，需要周期性调用
AutoWriteBack	自动回写，需要周期性调用
Format	格式化逻辑卷

2.3.2 使用说明

1 初始化

在使用文件系统之前首先需要调用 FileInit()函数初始化文件系统。

2 安装驱动程序

添加各种硬件驱动程序，如程序清单 2.7 所示添加 SD 卡的驱动程序。

程序清单 2.7 添加 SD 卡驱动程序

```
AddFileDriver(GetSDCommand(), GetSDInfo());            // 添加 SD 卡驱动程序
```

3 执行各种操作

这时，调用相关的函数对文件进行各种操作。比如创建目录、删除文件、读写文件等等。

2.3.3 示范例程

ZLG/FS 文件管理系统支持多种介质的文件操作，如 CF 卡、SD/MMC 卡、U 盘等。下面我们以 SD 卡为例，来介绍 ZLG/FS 文件管理系统在介质中的应用。

1 SD 卡的简单读写操作

使用 LPC2300 系列工控产品提供的模版进行操作。在 SD 卡中创建一个名称为“ReadMe.txt”的文件，并对此文件进行读写操作。

在 main 函数中初始化文件系统并安装 SD 卡的驱动程序，如程序清单 2.8 所示。

程序清单 2.8 main 函数中初始化程序

```
#ifndef   __USE_SD
#define   __USE_SD    1                                    // SD 卡功能使能宏
#endif
……
```

```
int main (void)
{
    TargetInit();
#if __USE_SD
    FileInit();
    AddFileDriver(GetSDCommand(), GetSDInfo());          // 安装 SD 卡驱动
#endif
……
}
```

我们在任务 Task0 中完成对 SD 卡的操作，如程序清单 2.9 所示。

程序清单 2.9　SD 卡的读写操作

```
uint8 read_file_data[100];
void TASK0 (void *pdata)
{
    HANDLE  fp;
    char        file_name[]={"ReadMe.TXT"};
    char        const file_data[]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    static uint32   count;
    uint32   i;
    uint32   offset = 0;
    uint8    ReadData[60];
    pdata = pdata;

    RemoveFile(file_name);                                          // 删除文件名为 ReadMe 的文件
    fp = FileOpen(file_name,"RW");                                  // 创建 ReadMe.TXT 文件
    if (fp != Not_Open_FILE)
    {
        FileSeek(fp, 0, SEEK_END);                                  // 以追加的方式写入文件信息
        count = FileWrite((uint8 *)file_data, (sizeof(file_data)-1), fp);  // 写入信息
        FileClose(fp);                                              // 文件关闭
        AllCacheWriteBack();                                        // 将缓冲区数据回写
    }
  fp = FileOpen(file_name, "R");                                    // 打开 ReadMe.TXT 文件
    if (fp != Not_Open_FILE)
    {
        do{
            FileSeek(fp, offset, SEEK_SET);                         // 设置地址偏移量
            count = FileRead(ReadData, 10, fp);                     // 读取 10 个字节的数据
        for (i=0; i<count; i++)
        {
                read_file_data[i+offset] = ReadData[i];
        }
        offset += count;

        }while (!FileEof(fp));                                      // 判断是否读到文件末尾
        FileClose(fp);                                              // 关闭文件
    }
/*******************************************************************************
**  在此设置断点，程序从 SD 卡里读出 ReadMe.TXT 文件的内容，通过变量调试窗口观察读出
**  内容的缓冲区 read_file_data 的内容是否与写入缓冲区 file_data 的内容相同
*******************************************************************************/
  while (1)
  {
        OSTimeDly(OS_TICKS_PER_SEC);
  }
}
```

程序下载成功后，将 SD 卡插入底板，在调试模式下运行程序。在如图 2.8 所示地方设置断点，在变量查看窗口查看写入和读出的内容。然后取出 SD 卡，使用读卡器在 PC 机上打开，可以观察到 ReadMe.txt 文件中的信息。

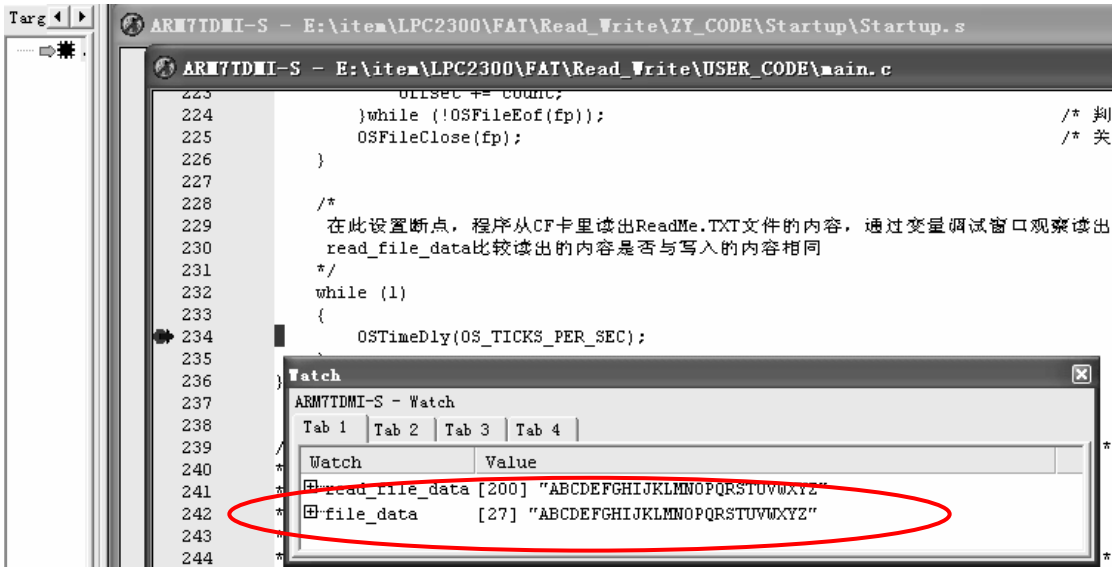

图 2.8　检查存储介质中的数据

2　创建及查找特定文件名操作

创建不同包含关系的目录，并在目录中创建文件，添加文件内容。再从所在目录查找此文件，如找到此文件再次添加文件内容。具体的程序如下所示。初始化文件系统如程序清单 2.8 所示。利用文件系统对 SD 的操作如程序清单 2.10 所示。

程序清单 2.10　SD 卡的操作

```
char  file_name[]={"ReadMe.TXT"};                    // 文件名
uint32   count;                                      // 写入文件字节个数
uint32   i;                                          // 查找文件参数
char      FileName[50];
uint8     Retvalue;                                  // 返回值
HANDLE   fp;                                         // 文件句柄
char const file_data[]="MiniARM is very good!\r\n";  // 写入文件的内容

void TASK0(void *pdata)
{
      char   Path1[4][10] = {{"A:\\One"}, {"A:\\Two"}, {"A:\\Three"}, {"A:\\Four"} };
      char   Path2[3][14] = { {"A:\\One\\One"}, {"A:\\One\\Two"}, {"A:\\One\\Three"}};

      pdata = pdata;
      Retvalue = FindFirst(FileName,"A:\\");
      if(Retvalue == FIND_DIR)
      {
          RemoveDir (FileName);                      // 删除根目录下的第一个空目录
      }
      while(1)
      {
          Retvalue = FindNext(FileName);
          if(Retvalue == FIND_DIR)
          {
              RemoveDir (FileName);                  // 删除根目录下的下一个空目录
          }
          else
          {
              break;
          }
      }
      AllCacheWriteBack();                           // 将缓冲区数据回写

      for(i=0; i<4; i++)
```

```
    {                                                   // 创建目录
        Retvalue = MakeDir(Path1[i]);
    }
  AllCacheWriteBack();                                  // 将缓冲区数据回写

    for(i=0; i<3; i++)
    {                                                   // 创建目录
        Retvalue = MakeDir(Path2[i]);
    }
    ChangeDir("A:\\One\\Two");                          // 改变当前目录为 A:\\One\\Two
  RemoveFile(file_name);                                // 删除名称为 ReadMe.TXT 文件
    fp = FileOpen(file_name,"RW");                      // 创建 ReadMe.TXT 文件并写入内容
    if (fp != Not_Open_FILE)
    {
        FileSeek(fp, 0, SEEK_END);                      // 以追加的方式写入文件信息
        count = FileWrite((uint8 *)file_data, (sizeof(file_data)-1), fp);
        FileClose(fp);                                  // 文件关闭
        AllCacheWriteBack();                            // 将缓冲区数据回写
    }

  Retvalue = FindFirst(FileName, "A:\\One\\Two");
  if(Retvalue == FIND_FILE)
  {
      if(strcmp(FileName, "ReadMe.TXT") == 0)
      {                                                 // 比较是否所找文件
          fp = FileOpen(file_name,"RW");                // 是所找文件进行写操作
          if (fp != Not_Open_FILE)
          {
              FileSeek(fp, 0, SEEK_END);                // 以追加的方式写入文件信息
              count = FileWrite((uint8 *)file_data, (sizeof(file_data)-1), fp);
              FileClose(fp);                            // 文件关闭
          }
      }
  }
  while(1)
  {
      Retvalue = FindNext(FileName);                    // 下次查找
      if(Retvalue == FIND_FILE)
      {
          if(strcmp(FileName, "ReadMe.TXT") == 0)
          {                                             // 比较是否所找文件
              fp = FileOpen(file_name, "RW");// 是所找文件进行写操作
              if (fp != Not_Open_FILE)
              {
                  FileSeek(fp, 0, SEEK_END);            // 以追加的方式写入文件信息
                  count = FileWrite((uint8 *)file_data, (sizeof(file_data)-1), fp);
                  FileClose(fp);                        // 文件关闭
                  break;
              }
          }
      }
  }
  AllCacheWriteBack();                                  // 将缓冲区数据回写
  while (1)
  {
      OSTimeDly(OS_TICKS_PER_SEC);
  }
}
```

程序下载成功后，插入 SD 卡，全速运行程序。用 SD 卡读卡器在 PC 机上可读出 SD 卡中的内容，如图 2.9 所示。

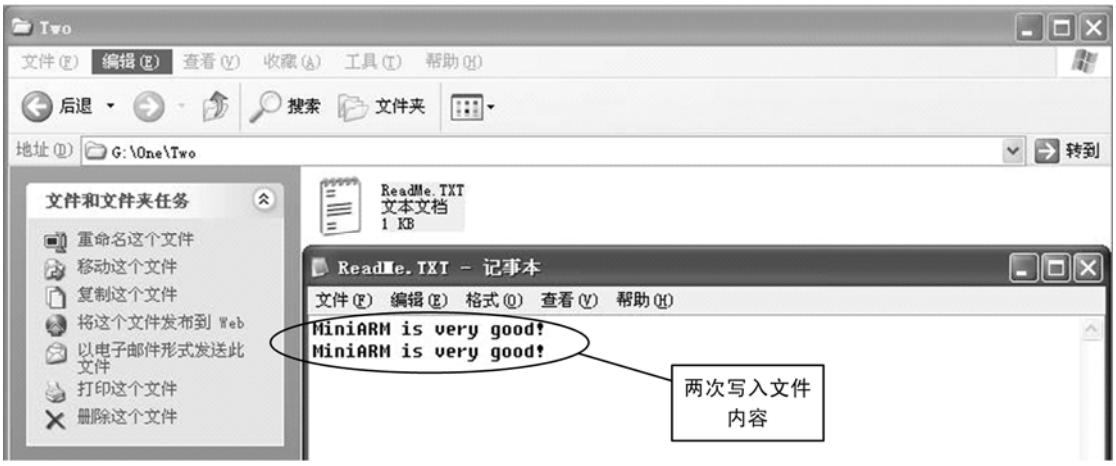

图 2.9　SD 卡中内容

3　文件的时间属性操作

在 SD 卡中创建一个文件并写入内容，查看文件的最后写入时间，前提条件是系统时间正确才能获得正确的时间。初始化文件系统的程序见程序清单 2.8 所示，获取时间的程序如程序清单 2.11 所示。

程序清单 2.11　获取文件最后写时间

```
void TASK0(void *pdata)
{
      static uint32 count;
      char    file_name[]={"ReadMe.TXT"};
      char const    file_data[]={"MiniARM is very good!"};     // 写入文件的内容
      HANDLE    fp;
    static DATE_TIME    FileTime;

      pdata = pdata;
      RemoveFile(file_name);                                   // 删除文件
      fp = FileOpen(file_name, "RW");                          // 创建 ReadMe.TXT 文件并写入内容
      if (fp != Not_Open_FILE)
      {
            FileSeek(fp, 0, SEEK_END);                         // 以追加的方式写入文件信息
            count =FileWrite((uint8 *)file_data, (sizeof(file_data)-1), fp);
            FileClose(fp);                                     // 文件关闭
            AllCacheWriteBack();                               // 将缓冲区数据回写
    }
    GetFileDataTime(&FileTime,"A:\\ReadMe.TXT");               // 获取文件最后写时间
/****************** 此处设置断点，观察 FileTime 的内容  ******************************/
      while (1)
    {
            OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

程序运行到所设置的断点处，在变量查看窗口中观察 FileTime 的内容，如图 2.10 所示。

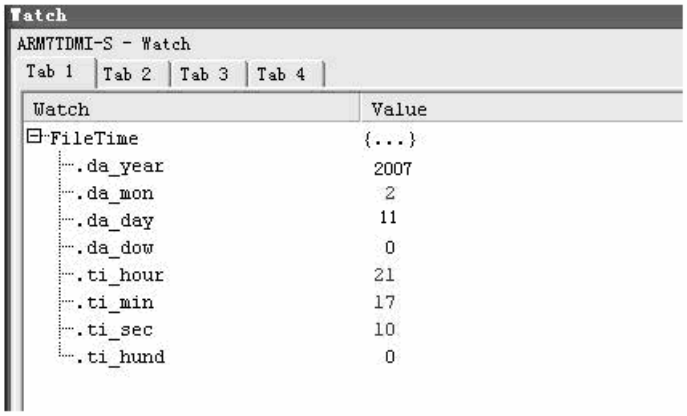

图 2.10　获取时间值

2.4　Modbus 协议栈

2.4.1 需用户编写的函数

需要用户编写的函数主要分两部分：用户的应用数据访问函数，另一部分是 UART 相关的硬件底层函数。

ZLG/Modbus 从机协议栈访问应用程序数据是通过 MB_USER_DATA.C 文件中的接口函数实现。ZLG/Modbus 从机协议栈通过调用 MB_USER_DATA.C 文件中的接口函数读出或写入线圈、输入寄存、输入离散量和保存寄存器等，这些接口函数由用户提供。

用户必须提供的函数如表 2.4 所示。表 2.4 中的函数名称并不是强制要求的，只要函数的输入参数及返回参数一样，也可以使用其它名称。这些函数通过 MB_DataHandleFunPtr( ) 函数注册。

表 2.4　访问应用数据函数

函数名称	描述
MB_GetCoils ( )	读取线圈值
MB_GetDiscrete ( )	读取离散输入量值
MB_GetRegValue ( )	读取寄存器值
MB_GetInputRegValue ( )	读取输入寄存器值
MB_SetCoil ( )	设置线圈值
MB_SetRegValue ( )	设置寄存器值

MB_GetDiscrete ( ) 函数说明如

表 2.5 所示。

表 2.5 MB_GetDiscrete ( )函数

函数名称	MB_GetDiscrete
函数原型	INT8U MB_GetDiscrete (INT8U ch, INT8U *DisInputsV, INT16U Address)
功能描述	当 Modbus 主机发送功能代码为 2 的请求时，MB_GetDiscrete ( )函数将被调用，通过该函数获得离散输入量的值
参数	ch — Modbus 通讯管道号，其取值范围 0~MB_MAX_CH-1。如果不同的通讯管道使用不同的应用数据时，可以通过该参数识别不同的管道。MB_MAX_CH 的值与处理器的 UART 个数相关，如 LPC2300 则该值为 4。 * DisInputsV — 指向保存读取离散输入量值的指针。读取到离散输入量值，保存在该指针指向的地址，离散输入量的值仅为 0 或 1。 Address — 该参数用于指出需要读取的离散输入量地址，其取值范围在 0~65535(取决于用户的应用程序)。根据该参数值，判断哪个离散输入量被访问。
返回值	执行该函数正常完成返回 MB_NO_ERR,出错返回 LLEGAL_DATA_ADDR 或 SLAVE_DEVICE_FAILURE（这些宏在 ModbusUser.h 文件中定义）
特殊说明和注意点	该函数仅被 ZLG/Modbus 从机栈调用。

示例：

假设 Modbus 从机产品中应用数据有 163 个离散输入量。其中 160 个分配在一张表 AppDiscTbl[]中，其地址为 0~159；另外 3 个离散输入量为 3 路开关输入变量 AppKey1、AppKey2 和 AppKey3，其地址分别为 200、201 和 202。实现代码如程序清单 2.12 所示。

程序清单 2.12 MB_GetDiscrete ( )

```
INT8U AppDiscTbl[20];
INT8U AppKey1;
INT8U AppKey2;
INT8U AppKey3;

INT8U MB_GetDiscrete (INT8U ch,INT8U *DisInputsV,INT16U Address)
{
    INT8U err;
    INT8U ix;
    INT8U bit_nbr;
    err = MB_NO_ERR;
    if (Address < 100 * sizeof(INT8U)) {
        ix = Address / 8;
        bit_nbr = Address % 8;
        *DisInputsV =(AppDiscTbl[ix] & (1 << bit_nbr))?1:0;

    } else  {
        switch (Address)  {
        case 200:
            *DisInputsV = AppKey1?1:0;
            break;
        case 201:
            *DisInputsV = AppKey2?1:0;
            break;
        case 202:
            *DisInputsV = AppKey3?1:0;
            break;
        default:
            err = ILLEGAL_DATA_ADDR;            // 出错返回：无效的地址
            break;
        }
    }
    return (err);
}
```

MB_GetCoils ( )函数说明如表 2.6 所示。

表 2.6 MB_GetCoils ()

函数名称	MB_GetCoils
函数原型	INT8U MB_GetCoils(INT8U ch, INT8U *CoilVPtr, INT16U Address)
功能描述	当 Modbus 主机发送功能代码为 1 的请求时，MB_GetCoils ( )函数将被调用，通过该函数获得单线圈的值
参数	ch ― Modbus 通讯管道号，其取值范围 0~MB_MAX_CH-1。如果不同的通讯管道使用不同的应用数据时，可以通过该参数识别不同的管道 *CoilVPtr ― 指向保存读取线圈值的指针。读取到线圈值，保存在该指针指向的地址，线圈的值仅为 0 或 1 Address ― 该参数用于指出需要读取的线圈地址，其取值范围在 0~65535(取决于用户的应用程序)。根据该参数值，判断哪个线圈被访问
返回值	执行该函数正常完成返回 MB_NO_ERR, 出错返回 LLEGAL_DATA_ADDR 或 SLAVE_ DEVICE_FAILURE（这些宏在 ModbusUser.h 文件中定义）
特殊说明和注意点	该函数仅被 ZLG/Modbus 从机栈调用

示例：

假如我们的 Modbus 从机产品中有 803 个线圈。其中 800 个线圈分配在一张表 AppCoilTbl[]中，其地址为 0~799；另外 3 个线圈为 3 个独立的状态，分别为 AppStatus、AppRunning 和 AppLED，其地址分别为 1000、1001 和 1002。其实现代码如程序清单 2.13 所示。

程序清单 2.13 MB_GetCoils ( )

```
INT8U AppCoilTbl[100];
INT8U AppStatus;
INT8U AppRunning;
INT8U AppLED;

INT8U MB_GetCoils (INT8U ch,INT8U *DisInputsV,INT16U Address)
{
    INT8U err;
    INT8U ix;
    INT8U bit_nbr;
    err = MB_NO_ERR;
    if (Address < 100 * sizeof(INT8U)) {
        ix = Address / 8;
        bit_nbr = Address % 8;
        *DisInputsV =(AppCoilTbl[ix] & (1 << bit_nbr))?1:0;

    } else {
        switch (Address) {
            case 1000:
                *DisInputsV = AppStatus?1:0;
                break;
            case 1001:
                *DisInputsV = AppRunning?1:0;
                break;
            case 1002:
                *DisInputsV = AppLED?1:0;
                break;
            default:
                err = ILLEGAL_DATA_ADDR;                    // 出错返回：无效的地址
                break;
        }
    }
    return (err);
}
```

MB_SetCoil ()函数说明如表 2.7 所示。

表 2.7 MB_SetCoil ( )

函数名称	MB_SetCoil
函数原型	INT8U MB_SetCoil (INT8U ch, INT16U Address, INT8U CoilValue)
功能描述	当 Modbus 主机发送功能代码为 5 或 15 的请求时，MB_SetCoil ( )函数将被调用，通过该函数向指定的线圈写入特定的值。
参数	ch ― Modbus 通讯管道号，其取值范围 0~MB_MAX_CH-1。如果不同的通讯管道使用不同的应用数据时，可以通过该参数识别不同的管道。 Address ― 该参数用于指出需要读取的线圈地址，其取值范围在 0~65535(取决于用户的应用程序)。根据该参数值，判断哪个线圈被访问。 CoilValue ―写入指定线圈的值。该值仅为 0 或 1。
返回值	执行该函数正常完成返回 MB_NO_ERR,出错返回 LLEGAL_DATA_ADDR 或 SLAVE_DEVICE_ FAILURE（这些宏在 ModbusUser.h 文件中定义）
特殊说明和注意点	该函数仅被 ZLG/Modbus 从机栈调用

示例：

假如我们的 Modbus 从机产品中有 803 个线圈，其中 800 个线圈分配在一张表 AppCoilTbl[]中，地址为 0~799；另外 3 个线圈为 3 个独立的状态，分别为 AppStatus、AppRunning 和 AppLED，地址分别为 1000、1001 和 1002，实现代码如程序清单 2.14 所示。

程序清单 2.14 MB_SetCoil ( )

```
INT8U AppCoilTbl[100];
INT8U AppStatus;
INT8U AppRunning;
INT8U AppLED;
 INT8U MB_SetCoil(INT8U ch,INT16U Address,INT8U CoilValue)
{
    INT8U err;
    INT8U ix;
    INT8U bit_nbr;
    err = MB_NO_ERR;
    if (Address < 100 * sizeof(INT8U)) {
        ix = Address / 8;
        bit_nbr = Address % 8;
        if(CoilValue) {
            AppCoilTbl[ix] |= 1 << bit_nbr;
        } else {
            AppCoilTbl[ix] &= ~(1<<bit_nbr);
        }
    } else {
switch (Address) {
            case 200:
                AppStatus = CoilValue?1:0;
                break;
            case 201:
                AppRunning = CoilValue?1:0;
                break;
            case 202:
                AppLED = CoilValue?1:0;
                break;
            default:
                err = ILLEGAL_DATA_ADDR;
                break;
        }
```

```
    }
    return (err);
}
```

MB_GetInputRegValue()函数说明如表 2.8 所示。

表 2.8　MB_GetInputRegValue ( )

函数名称	MB_GetInputRegValue
函数原型	INT8U MB_GetInputRegValue (INT8U ch, INT16U *ValuePtr, INT16U Address)
功能描述	当 Modbus 主机发送功能代码为 4 的请求时，MB_GetInputRegValue ( )函数将被调用，通过该函数读取指定输入寄存器的值
参数	ch ― Modbus 通讯管道号，其取值范围 0~MB_MAX_CH-1。 如果不同的通讯管道使用不同的应用数据时，可以通过该参数识别不同的管道 *ValuePtr ― 该指针指向输入寄存器变量值的地址。用户在读取到输入寄存器的值后，写入到该指针指向的地址。该值为 0~65536 Address ― 该参数用于指出需要读取的输入寄存器地址，其取值范围在 0~65535(取决于用户的应用程序)。 根据该参数值，判断哪个输入寄存器被访问
返回值	执行该函数正常完成返回 MB_NO_ERR, 出错返回 LLEGAL_DATA_ADDR 或 SLAVE_DEVICE_ FAILURE（这些宏在 ModbusUser.h 文件中定义）
特殊说明和注意点	该函数仅被 ZLG/Modbus 从机栈调用

示例：

假设 Modbus 从机设备有 153 个输入寄存器，其中有 150 个存放在一张 InputBuffer[]表中，其地址为 0~149；另外 3 个分别为 3 路 AD 的输入变量 AppAD0、AppAD1 和 AppAD2，其地址分别为 3000、3001 和 3002。实现代码如程序清单 2.15 所示。

程序清单 2.15　MB_GetInputRegValue ( )

```
#define END_INPUT_REG_ADDR      150
INT16U InputBuffer[END_INPUT_REG_ADDR];
INT16U AppAD0;
INT16U AppAD1;
INT16U AppAD2;
INT8U MB_GetInputRegValue(INT8U ch,INT16U *ValuePtr,INT16U Address)
{
    INT8U err;
    err = MB_NO_ERR;
    if (Address < END_INPUT_REG_ADDR)   {
        *ValuePtr = InputBuffer[Address];
} else {
    switch (Address) {
            case 3000:
                *ValuePtr = AppAD0;
                break;
            case 3001:
                *ValuePtr = AppAD1;
                break;
            case 3002:
                *ValuePtr = AppAD2;
                break;
            default:
                err = ILLEGAL_DATA_ADDR;
                break;
        }
    }
```

```
    return (err);
}
```

MB_GetRegValue ( )函数说明如表 2.9 所示。

表 2.9 MB_GetRegValue()

函数名称	MB_GetRegValue
函数原型	INT8U MB_GetRegValue(INT8U ch,INT16U *ValuePtr,INT16U Address)
功能描述	当 Modbus 主机发送功能代码为 3、22 或 23 的请求时，MB_GetRegValue ( )函数将被调用，通过该函数读取指定保持寄存器的值。该函数仅被 ZLG/Modbus 从机栈调用
参数	ch — Modbus 通讯管道号，其取值范围 0~MB_MAX_CH-1。如果不同的通讯管道使用不同的应用数据时，可以通过该参数识别不同的管道 *ValuePtr — 该指针指向保待寄存器变量值的地址。用户在读取到保持寄存器的值后，写入到该指针指向的地址。该值为 0~65535 Address — 该参数用于指出需要读取的寄存器地址，其取值范围在 0~65535(取决于用户的应用程序)。根据该参数值，判断哪个保持寄存器被访问
返回值	执行该函数正常完成返回 MB_NO_ERR, 出错返回 LLEGAL_DATA_ADDR 或 SLAVE_DEVICE_ FAILURE（这些宏在 ModbusUser.h 文件中定义）

示例：

假设 Modbus 从机设备有 150 个输入寄存器，其存放在一张 RegValueBuffer[]表中，其地址为 0~149。实现代码程序清单 2.16 所示。

程序清单 2.16 MB_GetRegValue ( )

```
#define END_HOLDING_REG_ADDR 150
INT16U RegValueBuffer[END_HOLDING_REG_ADDR];
INT8U MB_GetRegValue(INT8U ch,INT16U *ValuePtr,INT16U Address)
{
    if(Address<END_HOLDING_REG_ADDR)  {
        *ValuePtr = RegValueBuffer[Address] ;
        return MB_NO_ERR;
    } else {
        return ILLEGAL_DATA_ADDR;
   }
}
```

MB_SetRegValue()函数说明如表 2.10 所示。

表 2.10 MB_SetRegValue ( )

函数名称	MB_SetRegValue
函数原型	INT8U MB_SetRegValue(INT8U ch, INT16U Address, INT16U Value)
功能描述	当 Modbus 主机发送功能代码为 6、16、22 或 23 的请求时，MB_GetRegValue ()函数将被调用，通过该函数读取指定保持寄存器的值
参数	ch — Modbus 通讯管道号，其取值范围 0~MB_MAX_CH-1。如果不同的通讯管道使用不同的应用数据时，可以通过该参数识别不同的管道 Address — 该参数用于指出需要读取的寄存器地址，其取值范围在 0~65535(取决于用户的应用程序)。根据该参数值，判断哪个保持寄存器被访问 Value — 写入到保持寄存器指定地址的值。该值的范围为 0~65536
返回值	执行该函数正常完成返回 MB_NO_ERR, 出错返回 LLEGAL_DATA_ADDR 或 SLAVE_DEVICE_FAILURE（这些宏在 ModbusUser.h 文件中定义）。
特殊说明和注意点	该函数仅被 ZLG/Modbus 从机栈调用。

示例：

假设 Modbus 从机设备有 150 个输入寄存器，其存放在一张 RegValueBuffer[]表中，其地址为 0~149。实现代码程序清单 2.17 所示。

程序清单 2.17　MB_SetRegValue ( )

```
#define END_HOLDING_REG_ADDR 150
INT16U RegValueBuffer[END_HOLDING_REG_ADDR];
INT8U MB_SetRegValue(INT8U ch,INT16U Address,INT16U Value)
{
    if(Address<END_HOLDING_REG_ADDR)  {
        RegValueBuffer[Address] = Value;
        return MB_NO_ERR;
    } else {
        return ILLEGAL_DATA_ADDR;
    }
}
```

以上这些函数名称只是给用户参数，实际上 ZLG/Modbus 协议栈并不是直接调用这些函数，而是通过 MB_DataHandleFunPtr()函数注册这些函数。

UART 底层相关的接口函数，如表 2.11 所示。ZLG/Modbus 协议栈通过调用这些函数实现对底层的 UART 操作。表 2.11 中的函数名称并不是强制要求的，只要函数的输入参数及返回参数一样，也可以使用其它名称。

表 2.11　用户需要编写的接口函数

函数名称	描述
BSP_MB_PortCfg	协议栈在配置管道设置时，调用该函数。 该函数用于初始 UART
MB_BSP_SendData	协议栈在发送数据时，调用该函数发送数据。
CPU_ENTER_CRITICAL	关中断
CPU_EXIT_CRITICAL	开中断

底层端口配置函数 BSP_MB_PortCfg()，如表 2.12 所示。

表 2.12　BSP_MB_PortCfg()

函数名称	BSP_MB_PortCfg
函数原型	INT8U BSP_MB_PortCfg (MB_CFG *mb_ch)
功能描述	该函数被 MB_CfgCh()调用，用于配置串行通讯口， 根据 mb_ch->ch 参数判断所需配置的串口
参数	*mb_ch　一　串口设置参数指针
返回值	正常返回 MB_NO_ERR 出错返回 MB_PARAMETER_ERR(无效参数)

示例程序如程序清单 2.18 所示。

程序清单 2.18　BSP_MB_PortCfg()

```
INT8U BSP_MB_PortCfg (MB_CFG *mb_ch)
{
    char          UartArg[60];
    UART_HOOKS uarthook;
    uint8         UartIRQCh;

    /*
    *    参数检查
    */
```

```
    if (mb_ch == (void*)0) {
        return MB_PARAMETER_ERR;
    }

    if ((mb_ch->modbus_mode != MB_RTU_MODE) &&      // 用户根据管道支持的模式设置
        (mb_ch->modbus_mode != MB_ASCII_MODE)) {
        return MB_PARAMETER_ERR;
    }
    memset(&uarthook,0,sizeof(UART_HOOKS));
    uarthook.pRevice = uartReviceHook;
    sprintf (UartArg, "BaudRate=%d RxBufSize=0 TxBufSize=%d", mb_ch->BaudRate,
mb_ch->usQSendSize );
    if (OPERATE_FAIL == UartInit(mb_ch->ch, (char *)UartArg, &uarthook)) {
        while (1);
    }
    switch(mb_ch->ch)
    {
        case 0:
            UartIRQCh = UART0_IRQ_CHN;
            break;
        case 1:
            UartIRQCh = UART1_IRQ_CHN;
            break;
        case 2:
            UartIRQCh = UART2_IRQ_CHN;
            break;
        case 3:
            UartIRQCh = UART3_IRQ_CHN;
            break;
        default:
            break;
    }
    SetVICIRQ(UartIRQCh,6,(uint32)UART0_ISR);
    if((mb_ch->bits != MB_UART_8BIT)                    ||
       (mb_ch->parity != MB_NONE_PARITY)  ||
       (mb_ch->stops != MB_UART_1STOP)         ) {
       sprintf (UartArg, "DataBits=%d StopBits=%d Parity=%d", mb_ch->bits, mb_ch->stops,
mb_ch->parity);
       UartSetMode (mb_ch->ch, SET_MODE,UartArg);
    }

    /*
    *   如果使用 RS485 请执行下个函数
    */
  #if 0
    if (OPERATE_FAIL == UartSetMode(mb_ch->ch,SET_RS485,"RS485Dir=216")) {
        while (1);
    }
  #endif
    return MB_NO_ERR;
}
```

端口底层数据发送函数如表 2.13 所示。

表 2.13　MB_BSP_SendData()

函数名称	MB_BSP_SendData
函数原型	void MB_BSP_SendData (INT8U ucCh, INT8U *ucBuf, INT16U usLen)
功能描述	串行数据发送函数
参数	ucCh ― 所使用管道号 ucBuf ― 发送数据缓冲区指针 usLen ― 发送数据长度

端口底层数据发送函数示例程序如程序清单 2.19 所示。

程序清单 2.19 底层发送数据函数

```
void MB_BSP_SendData (void *pDriver,INT8U *buff,INT16U len)
{
UartWrite(ucCh, ucBuf, usLen, NULL);        // 写串口数据
}
```

ZLG/Modbus 提供了两个底层函数需要用户调用，见表 2.14。

表 2.14 用户必须需要调用的 API 函数

函数名称	描述
MB_TmrUpdate	用户必须通过定时器中断，每隔 500 微秒调用一次
ReceOneChar	每当 UART 接收到一个字符后，调用该函数处理

MB_TmrUpdate()函数如表 2.15 所示。

表 2.15 MB_TmrUpdate()

函数名称	MB_TmrUpdate
函数原型	void MB_TmrUpdate(void)
功能描述	时间处理，处理 T15、T13 的事件和协议栈状态事件。该函数需要在 500uS 被调用一次

在 LPC2300 的定时器 1 中断函数 Timer1_UsrISR 中，调用 MB_TmrUpdate()函数，如程序清单 2.20 所示对定时器 1 的配置。

程序清单 2.20 配置定时器 1 中断

```
void    BSP_MB_TmrISR_Handler (INT32U time)
{
    char TmrArg[50];

    sprintf(TmrArg, "Time=%d ActCtrl=3",time);
    TimerInit(TIMER1,"Mode=0",NULL);                    // 定时器 1 初始化

      TimerSetMode(TIMER1,SET_TIMERMODE,TmrArg);
                                                        // 设置定时器 1 的工作模式为定时模式：
                                                        // Time=500：定时时间为 500us
                                                        // ActCtrl=3：匹配后将 T1TC 复位,并产生中断
      SetVICIRQ(TIMER1_IRQ_CHN,7,(uint32)Timer1_UsrISR);
                                    // 设置中断，一定要在中断函数中添加 MB_TmrUpdate()函数
    TimerStart(TIMER1,NULL);        // 开启定时器
}
```

接收字符函数如表 2.16 所示。

表 2.16 ReceOneChar()

函数名称	ReceOneChar
函数原型	void ReceOneChar(uint8 ch,uint8 ReceCharacter,uint8 err)
功能描述	接收一个字符处理
参数	ch — 接收到数据的管道号 ReceCharacter — 接收到的字符 err — 非零值表时接收字符出错，如奇偶效验出错
特殊说明和注意点	接收一个字符处理要求:接收到一个字符后立即传入该函数处理

接收到字符处理示例，如程序清单 2.21 所示。

程序清单 2.21　接收到字符处理

```
void uartReviceHook (UART_DATA *pudInfo, INT8U ucRbr)
{
        ReceOneChar(UART0,ucRbr,0);                    // 调用 ZLG/Modbus 接收字符函数，
                                                       // 接收每一个字符
}
```

uartReviceHook 为底层 UART 驱动提供的接口函数，接收到一个字符时，该函数将被调用一次。

2.4.2 API 函数

ZLG/Modbus 的 API 函数是用户在应用层中可直接使用的函数集合，根据其使用权限可分为：主从共用、主机相关和从机相关。这些函数如表 2.17 所示。

表 2.17　ZLG/Modbus API 函数

分类	函数名称	描述
Modbus 主从共用	MB_Ini( )	Modbus 驱动初始化
	MB_CfgCh ( )	配置一个 Modbus 管道
Modbus 主机相关	OSModbusMServe( )	Modbus 主机服务任务
	ReadCoils01( )	读取线圈值功能代码请求
	ReadDisInputs02 ( )	读取离散输入量值功能代码请求
	ReadHoldReg03 ( )	读取保持寄存器值功能代码请求
	ReadInputReg04 ( )	读取输入寄存器值功能代码请求
	WriteSingleCoil05 ( )	写单个线圈值功能代码请求
	WriteSingleReg06 ( )	写单个寄存器值功能代码请求
	WriteMultipleCoils15 ( )	写多个线圈值功能代码请求
	WriteMultipleReg16 ( )	写多个寄存器值功能代码请求
	MaskWriteReg22 ( )	屏蔽写寄存器值功能代码请求
	ReadWriteMultipleReg23 ( )	读写多个寄存器值功能代码请求
	MBM_UDFCodeAdd ( )	主机添加自定义功能代码请求
	PutDataInPDU ( )	写 PDU 帧数据包
	ModbusPoll( )	提交自定义功能代码请求处理
Modbus 从机相关	OSModbusSServe ( )	Modbus 从机服务任务
	MB_DataHandleFunPtr()	注册用户数据处理函数
	MBS_UDFCodeAdd( )	从机添加自定义功能代码请求

2.4.3 使用说明

1　初始化

使用 ZLG/Modbus 协议栈需要在工程中包含 ModbusUser.h、RequestFun.h、MB_BSP_LPC2300.c、MB_USER_Data.c 和 uart.h 头文件。其中，MB_BSP_LPC2300.c、ModbusUser.h 和 uart.h 文件是必须包含的。RequestFun.h 头文件只是使用 Modbus 主机函数时才需要包含；MB_USER_Data.c 只是作从机时才需要，该文件中的函数为用户数据访问的函数。

由于使用了 UART 作为数据传输，所以少不了要初始化 UART 的中断服务函数。

另外协议栈需要一个每 500us 产生一次中断的定时器，并在中断服务程序中调用 MB_TmrUpdate()函数。如程序清单 2.22 所示。

程序清单 2.22　定时器初始化及中断

```
__inline void TIMER1ISR(void)
{
    TimerISR(TIMER1);
    //-----添加用户代码------//
    MB_TmrUpdate();
    //-----------------------------//
```

```
    VICVectAddr = 0x00;             // 中断处理结束
}
void BSP_MB_TmrInit (INT32U time)                                   // 定时器初始化函数
{
    char TmrArg[50];

    sprintf(TmrArg, "Time=%d ActCtrl=3",time);
    TimerInit(TIMER1,"Mode=0",NULL);                                // 定时器 1 初始化
     /*
     ** 设置定时器 1 的工作模式为定时模式：
   ** Time=500：定时时间为 500us
     ** ActCtrl=3：匹配后将 T1TC 复位,并产生中断
     */
     TimerSetMode(TIMER1,SET_TIMERMODE,TmrArg);
     SetVICIRQ(TIMER1_IRQ_CHN,7,(uint32)TIMER1_ISR);
// 设置中断，一定要在中断函数中添加 MB_TmrUpdate()函数
   TimerStart(TIMER1,NULL);                                         // 开启定时器
}
```

如程序清单 2.23，先通过 MB_Ini()函数初始化 ZLG/Modbus 协议栈，再初始化定时器。

程序清单 2.23　Modbus 初始化

```
MB_Ini();                                   // Modbus 协议栈初始化
BSP_MB_TmrInit(500);                        // 500uS 产生一次中断
```

使用 Modbus 主机则需要创建 Modbus 主机调度任务 OSModbusMServe()；如果使用 Modbus 从机，除了需要创建从机调度任务 OSModbusSServe()，还需要提供数据处理函数，并通过 MB_DataHandleFunPtr ()函数向 ZLG/Modbus 协议栈注册，如程序清单 2.24 所示。

程序清单 2.24　注册数据处理函数和创建任务

```
memset(&GdhfFunStr,0,sizeof(GdhfFunStr));
GdhfFunStr.GetDiscrete          = MB_GetDiscrete;
GdhfFunStr.GetCoils         = MB_GetCoils;
GdhfFunStr.SetCoil              = MB_SetCoil;
GdhfFunStr.GetInputRegValue     = MB_GetInputRegValue;
GdhfFunStr.GetRegValue          = MB_GetRegValue;
GdhfFunStr.SetRegValue          = MB_SetRegValue;
MB_DataHandleFunPtr (&GdhfFunStr);
OSTaskCreateExt(OSModbusSServe,                         // Modbus 从机服务任务
                (void *)0,
                &MBSlave_STACK[MBSlave_STACK_SIZE-1],
                MBSlave_PRIO,
                MBSlave_ID,
                &MBSlave_STACK[0],
                MBSlave_STACK_SIZE,
                (void *)0,
                OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

OSTaskCreateExt(OSModbusMServe,                         // Modbus 主机服务任务
                (void *)0,
                &MBMaster_STACK[MBMaster_STACK_SIZE-1],
                MBMaster_PRIO,
                MBMaster_ID,
                &MBMaster_STACK[0],
                MBMaster_STACK_SIZE,
                (void *)0,
                OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
```

注：GdhfFunStr 为 DATA_HANDLE_FUN 类型的全局数据结构，在初始化后，禁止改写该数据结构的任何成员。主机、从机的服务任务堆栈不能少于 100 sizeof(OS_STK)字节。

完成以上操作后，就能通过 MB_CfgCh()函数开始创建 Moudbus 主机管道或从机管道了。

2　从机配置及操作

创建一个 Modbus 从机管道时，需要使先创建两大块内存块，一个用于 ADU（数据包）处理数据，另一个用于 UART 的发送缓冲区，注意 UART 发送缓冲需要比 ADU 多 20 个字节（RTU 模式，ADU 的缓冲区大小为 256 字节；而 ASCII 模式，ADU 缓冲区的大小为 513 字节）。

另外还需创建一个全局的数据结构 MBDSTR。MBDSTR 为 Modbus 驱动管道的数据结构，该结构初始化后禁止修改。Modbus 从机管道初始化（UART0）如程序清单 2.25 所示。

程序清单 2.25　Modbus 从机管道初始化（UART0）

```
#define TEMP_BUF_SIZE     256
MB_CFG mcMBch;
INT8U     GucUart0ADUBuf[TEMP_BUF_SIZE];
MBDSTR GmbdCH0Data;
mcMBch.ch                 = MB_UART0_CH;          // 置管道
mcMBch.node_addr          = 1;                    // 置 Modbus 从机
mcMBch.modbus_mode        = MB_RTU_MODE;          // 传输模式
mcMBch.BaudRate           = 9600;                 // 波特率
mcMBch.bits               = 8;                    // 8 位字符
mcMBch.parity          = 0;                       // 无校验
mcMBch.stops              = 1;                    // 1 个停止位
mcMBch.usADUBufSize    = sizeof(GucUart0ADUBuf);
mcMBch.pucADUPtr  = GucUart0ADUBuf;
mcMBch.usQSendSize= TEMP_BUF_SIZE+20;
mcMBch.mbdPtr          = &GmbdCH0Data;
mcMBch.PortCfg            = BSP_MB_PortCfg;
mcMBch.SendData           = MB_BSP_SendData;
if (MB_NO_ERR != MB_CfgCh(&mcMBch)) {             // 调用 API 函数初始化 Modbus 管道
    while(1);                                     // Modbus 配置失败
}
```

作为 Modbus 从机，注册了数据访问函数后，只需要通过 MB_CfgCh()函数配置好从机管道，就不需要其它的函数处理了。所有的 Modbus 主机请求，都由 ZLG/Modbus 协议栈直接调用数据处理函数处理。数据类型的一般应用举例，如表 2.18 所示。

表 2.18　数据类型的一般应用

函数名称	描述	应用
MB_GetCoils ( )	读取线圈值	可输入和输出的离散值，如一些又向的 IO 口，状态标致状态位，继电器控制等。
MB_SetCoil ( )	设置线圈值	
MB_GetDiscrete ( )	读取离散输入量值	只输入的离散值，如按键，开关量等
MB_GetRegValue ( )	读取寄存器值	可输入和输出的 16 位寄存器，如 DA 输出、电机控制的参数配置等。
MB_SetRegValue ( )	设置寄存器值	
MB_GetInputRegValue ( )	读取输入寄存器值	仅输入的 16 位寄存器，输 AD 采样值，电机转速等

3　主机配置及操作

ZLG/Modbus 主机的配置与从机很类似，不同的是，配置参数中节点地址（node_addr）值为 0 时，对配置为主机，其它值则为从机。另外还有一点不同的是，只做主机时，不需要注册数据处理函数。

作为主机提供给用户使用的 API 函数较多，对从机的不同类型的数据访问都有相对应的 API 函数处理。用户可根据实际的需要调用这些函数处理。

2.4.4 示范例程

1　Modbus 从机例程

以 LPC2300 系列工控产品为例，建立一个 Modbus 从机。从机有 KEY1~KEY4 共 4 个按键输入和一个蜂鸣器输出。KEY1~KEY4 的控制引脚对应于 LPC2300 的 P0.6~P0.9，蜂鸣器对应着 P1.27。

按键为输入，所以可给它们分配离散输入量地址，当然也可以分配为线圈地址，本例程中，将它们分别分配为离散输入量的地址为 200~203；蜂鸣器只能为输出，所以只能分配为线圈地址，本例程分配的地址为 200。注意：这里的地址分配是随意列举，并没有硬性规定。

并使用 UART1 作为从机管道，使用 Modbus RTU 模式，无校位，1 个停止位。

线圈和按键访问的函数如程序清单 2.26 所示。这些函数在 MB_USER_Data.c 文件中。

程序清单 2.26　数据访问函数

```
uint8 MB_GetDiscrete (uint8 ch,uint8 *DisInputsV,uint16 Address)
{
    uint8 err;
    err = MB_NO_ERR;

    switch (Address) {
    case 200:
        *DisInputsV = (IO0PIN & KEY1) ? 1:0;          // 读取按键值
        break;
    case 201:
        *DisInputsV = (IO0PIN & KEY2) ? 1:0;          // 读取按键值
        break;
    case 202:
        *DisInputsV = (IO0PIN & KEY3) ? 1:0;          // 读取按键值
        break;
    case 203:
        *DisInputsV = (IO0PIN & KEY4) ? 1:0;          // 读取按键值
        break;
    default:
        err = ILLEGAL_DATA_ADDR;
        break;
    }
    return (err);
}

uint8 MB_SetCoil(uint8 ch,uint16 Address,uint8 CoilValue)
{
    uint8 err;
    err = MB_NO_ERR;

    if (Address == 200) {
        if ( CoilValue ) {
            BUZZER_Set();                             // 蜂鸣器响
        } else {
            BUZZER_Clr();                             // 蜂鸣器熄灭
        }
    } else {
        err = ILLEGAL_DATA_ADDR;
}
    return (err);
}
```

ZLG/Modbus 协议栈初始化和从机服务的创建，如程序清单 2.27 所示。

程序清单 2.27　初始化和从机服务的创建

```
#define MBSlave_ID              9                     // 任务的 ID
#define MBSlave_PRIO                MBSlave_ID        // 任务的优先级
#define MBSlave_STACK_SIZE      128                   // 定义用户堆栈长度
OS_STK    MBSlave_STACK[MBSlave_STACK_SIZE];

extern void keyIni(void);
extern    void    MB_CFGInit(void);
extern MB_CFG mcMBch;

DATA_HANDLE_FUN    GdhfFunStr;              // 数据处理函数的数据结构
```

```
void    TASK0 (void *pdata)
{
      keyIni();
      MB_Ini();

      memset(&GdhfFunStr,0,sizeof(GdhfFunStr));
      GdhfFunStr.GetDiscrete          = MB_GetDiscrete;
      GdhfFunStr.GetCoils             = MB_GetCoils;
      GdhfFunStr.SetCoil              = MB_SetCoil;
      GdhfFunStr.GetInputRegValue     = MB_GetInputRegValue;
      GdhfFunStr.GetRegValue          = MB_GetRegValue;
      GdhfFunStr.SetRegValue          = MB_SetRegValue;
      MB_DataHandleFunPtr (&GdhfFunStr);                    // 传递从机结构体

      OSTaskCreateExt(OSModbusSServe,                       // Modbus 从机服务任务
                      (void *)0,
                      &MBSlave_STACK[MBSlave_STACK_SIZE-1],
                      MBSlave_PRIO,
                      MBSlave_ID,
                      &MBSlave_STACK[0],
                      MBSlave_STACK_SIZE,
                      (void *)0,
                      OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

      MB_CFGInit();                                         // UART、定时器初时化，用户自己进行配置
      if(MB_NO_ERR != MB_CfgCh(&mcMBch)) {                  // 调用 API 函数初始化 Modbus 管道
          while(1);                                         // Modbus 配置失败
      }

       while(1)
       {
            OSTimeDly(OS_TICKS_PER_SEC/2);
       }
}
```

这时，从机已创建完成，只需要主机对地址为 1，波特率为 115200，1 个停止位，无校验位，以 Modbus RTU 模式就可以与刚创建的从机通讯。对线圈地址 200 写入 0 或 1 就可以控制蜂器；对离散输入地址 200~203 可读取按键 KEY1~KEY4 的状态。

2 Modbus 主机例程

创建一个 Modbus 主机，用于与以上的 Modbus 从机通讯，使用 UART0 作为通讯管道。再创建两个任务，一个不断地读取 KEY1 的按键值，当 KEY1 按下时，蜂鸣器响，松开时，蜂鸣器熄灭，另一个任务定时的对从机线圈进行写操作。

只作主机，不需要编写数据访问函数，只需要配置 node_addr 参数为 0 即为主机进行初始化，如程序清单 2.28 所示。

程序清单 2.28　Mobus 主机初始化和任务的创建

```
#define MBMaster_ID                 9               // 任务的 ID
#define MBMaster_PRIO               MBMaster_ID     // 任务的优先级
#define MBMaster_STACK_SIZE         128             // 定义用户堆栈长度
OS_STK    MBMaster_STACK[MBMaster_STACK_SIZE];

extern void keyIni(void);
extern    void    MB_CFGInit(void);
extern MB_CFG mcMBch;

void    TASK0 (void *pdata)
{
      keyIni();
      MB_Ini();
```

```
    OSTaskCreateExt(OSModbusMServe,                          // Modbus 主机服务任务
                    (void *)0,
                    &MBMaster_STACK[MBMaster_STACK_SIZE-1],
                    MBMaster_PRIO,
                    MBMaster_ID,
                    &MBMaster_STACK[0],
                    MBMaster_STACK_SIZE,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

    MB_CFGInit();
    if(MB_NO_ERR != MB_CfgCh(&mcMBch)) {              //  调用 API 函数初始化 Modbus 管道
        while(1);                                     // Modbus 配置失败
    }

     while(1)
     {
         OSTimeDly(OS_TICKS_PER_SEC);
     }
}
```

TASK1 任务不断读取 KEY1 值，当读取的值为 0 时，蜂鸣器响，值为 1 时，蜂鸣器不响。如程序清单 2.29 所示。

程序清单 2.29　读按键任务

```
void TASK1 (void *pdata)
{
    INT8U    ucKey1;

     pdata = pdata;
     while(1)
     {
         if (MB_NO_ERR == ReadDisInputs02( MB_UART0_CH, // Modbus 管道号 UART0
                                           1,           // Modbus 从机地址
                                           200,         //  读取离散输入量起始地址
                                           1,           //  读取离散输入量的个数
                                           &ucKey1)) {  //  保存离散输入量的值
             if (ucKey1) {
                 BUZZER_Clr();
             } else {
                 BUZZER_Set();
             }
          }
          OSTimeDly(OS_TICKS_PER_SEC/10);
     }
}
```

TASK2 任务定时对从机线圈进行写操作，如程序清单 2.30 所示。

程序清单 2.30　线圈写操作

```
void TASK2 (void *pdata)
{
    pdata = pdata;
     while(1)
     {
          WriteSingleCoil05(MB_UART0_CH,1, 200, COIL_OFF);
          OSTimeDly(OS_TICKS_PER_SEC/2);
          WriteSingleCoil05(MB_UART0_CH,1, 200, COIL_ON);
          OSTimeDly(OS_TICKS_PER_SEC/2);
     }
}
```

主机的配置已完成，将 Modbus 主机和从机连接上即可以进行通讯。运行程序将会听到 Modbus 从机板的蜂鸣器，一响一灭；当按下从机板的 KEY1 时，主机板的蜂鸣器将长响，

松开按键时熄灭。

以上的 Modbus 主机和从机实验是通过 RS232 连接通讯的，如果需要在 RS485 网络上使用，则在 UART 初始化后调用 UartSetMode()函数进行配置。

2.5 iCAN 协议栈

2.5.1 函数列表

有关 iCAN 协议栈的所有 API 函数（见表 2.19）所示。

表 2.19 iCAN 协议栈 API 函数一览表

函数名称	功能描述
iCAN_MasterCreate	创建一个主站对象
iCAN_MasterLoadSlvList	加载从站列表
iCAN_Master_Pool	主站论询函数
iCAN_Master_MsgDule	数据处理函数
iCAN_Master_GetMsg	数据接收函数
iCAN_Master_GetDI	读取从站 DI 资源
iCAN_Master_GetAI	读取从站 AI 资源
iCAN_Master_SetDO	写入从站 DO 功能资源数据
iCAN_Master_SetAO	写入从站 AO 功能资源数据
iCAN_Master_SetRsv	写入从站特殊功能资源数据

2.5.2 使用说明

1 在从站列表中添加从站

将 iCAN_Master_User_ADS.h 文件中 ICAN_SLV_NUM 修改成需要管理的从站数目，最大不超过 62，如程序清单 2.31 所示。

程序清单 2.31 定义从站数量

```
#ifndef ICAN_MASTER_UAER_ADS_H
#define ICAN_MASTER_UAER_ADS_H
#include "iCAN_CMD.h"
#include "iCAN_Mater_DateType.h"

//从站数量定义
#define ICAN_SLV_NUM    2
extern MSTAEROPT iCAN_Master_Opt;
extern DEVICERCSR MasterRscr;
extern iCANSLV iCANSlvList[ICAN_SLV_NUM];
#endif
```

2 添加从站

在 iCAN_Master_User_ADS.c 文件中的 iCANSlvList 数组填写添加从站，如程序清单 2.32 所示。从站 Mac ID 为 4 的访问间隔为 100ms，1 字节数字量输入，1 字节数字量输出，8 字节模拟量输入，无模拟量输出，缓冲清零；从站 Mac ID 为 3 的访问间隔为 100ms，1 字节数字量输入，1 字节数字量输出，8 字节模拟量输入，无模拟量输出，缓冲清零。

程序清单 2.32 添加从站

```
iCANSLV iCANSlvList[ICAN_SLV_NUM] = {{4,10,1,1,8,0,{0}}, {3,10,1,1,8,0,{0}}};
```

从站结构体定义如程序清单 2.33 所示。

程序清单 2.33 从站结构体定义

```
typedef struct tagiCANSLV
{
    unsigned char SlvID;        // 从站 MacId
    unsigned char CycTime;      // 访问间隔(10ms)
```

```
    unsigned char DILen;        // 从站数字量输入长度(byte)
    unsigned char DOLen;        // 从站数字量输出长度(byte)
    unsigned char AILen;        // 从站模拟量输入长度(byte)
    unsigned char AOLen;        // 从站模拟量输出长度(byte)
    unsigned char BUF[128];     // 从站数据缓冲(byte)
}iCANSLV;
```

其中，从站数字量输入所占从站缓冲为 BUF[0]～BUF[0x1F]，支持数字量输入单元的最大数目为 256 个；从站数字量输出所占从站缓冲为 BUF[0x20]～BUF[0x3F]，支持的数字量输出单元的最大数目为 256 个；从站模拟量输入所占从站缓冲为 BUF[0x40]～BUF[0x5F]，模拟量输入单元长度为 16bits，最大支持 16 路；从站模拟量输出所占从站缓冲为 BUF[0x60]～BUF[0x7F]，模拟量输出单元长度为 16bits，最大支持 16 路。

3 主站配置结构体

如程序清单 2.34，主站配置使用预定结构体，用户通常仅需要改写 MAC ID 项即可。

程序清单 2.34 主站结构体改写

```
unsigned char MASTERCFGMAP[32] =
{
    0x33,0x03,                  // Vendor ID = 0x333
    0x01,0x00,                  // Product Type = 0001
    0x01,0x00,                  // Product Code = 0001
    0xE8,0x03,                  // Hardware Version = 1000
    0xE8,0x03,                  // Firmware Version = 1000
    0x80,0x40,0x32,0x03,        // Serial Number = 20070528
    0x3f,                       // MAC ID
    0xff,                       // BaudRate
    0x31,0x00,0x1C,0x00,        // UserBaudrate Set
    0xff,                       // CyclicParameter
    0xff,                       // CyclicMaster
    0xff,                       // COS type set
    0xff,                       // Master MAC ID
    0xff,                       // IO parameter
    0xff,                       // IO configure
    0xff,0xff,0xff,0xff,0xff,0xff   // IO configure
};
```

4 初始化

主站初始化要完成 4 个步骤，如程序清单 2.35 所示。

- 初时化 CAN 驱动
- 创建两个内部使用的信号量 SemiCANRcv，SemiCANSend
- 创建主站结构体 piCANMaster
- 加载从站列表

程序清单 2.35 主站初始化

```
char        pucCanArg[] = "BaudRate=500000 RxBufSize=20 Mode=0";
OS_ENTER_CRITICAL();
CanInit(CAN1, pucCanArg, NULL);                         // CAN 控制器 0 初始化
SetVICIRQ(CAN_IRQ_CHN, 7, (uint32)Can_ISR);             //  设置中断,优先级 7
SemiCANRcv = OSSemCreate(0);
SemiCANSend = OSSemCreate(0);
piCANMaster = iCAN_MasterCreate(&MasterRscr,&iCAN_Master_Opt);
OS_EXIT_CRITICAL();
iCAN_MasterLoadSlvList(piCANMaster,iCANSlvList,ICAN_SLV_NUM);
```

5 数据操作

主站和从站连接后，标准数据（DI，DO，AI，AO）直接映射到从站的数据缓冲区中，可通过直接读取和写入方式交换数据。如程序清单 2.36 所示。

程序清单 2.36 主从站数据交换

```
unsigned char    DI;
unsigned char    DO = 0xAA;
unsigned short    AI[2];
unsigned short    AO[2] = {0x11,0x22};

iCAN_Master_GetDI(piCANMaster, 0x03, &DI, 1);        // 读取从站 3 的 DI 数据
iCAN_Master_SetDO(piCANMaster, 0x03, &DO, 1);        // 写入从站 3 的 DO 数据
iCAN_Master_GetAI(piCANMaster, 0x03, AI, 2);         // 读取从站 3 的 AI 数据
iCAN_Master_SetDO(piCANMaster, 0x03, AO, 2);         // 写入从站 3 的 DO 数据
```

如从站具有特殊输出功能，如 PWM，则使用写特殊资源的方式输出数据。

```
unsigned char    datbuf[] = {0x00,0x00,0x02,0x58,0x00,0x00,0x02,0x58};
iCAN_Master_SlvRsv (piCANMaster, 3 ,0xC0,datbuf,8);// 向特殊功能地址 0xC0 写入数据
```

2.5.3 示范例程

如程序清单 2.37 所示，每隔 500ms 将 3 号从站的 DI 数据读出并写入 2 号从站的 DO 数据中。

程序清单 2.37　iCAN 示例程序

文件 1：“iCAN_Master_User_ADS.h”——配置从站数目

```
#define ICAN_SLV_NUM      2
```

文件 2：“iCAN_Master_User_ADS.c”——添加从站，配置主站及设置 CAN 通道

```
iCANSLV iCANSlvList[ICAN_SLV_NUM] = {{2,1,1,1,8,0,{0}}, {3,1,1,1,8,0,{0}}};
……
unsigned char MASTERCFGMAP[32] =
{
        0x33,0x03,                              //Vendor ID = 0x333
        0x01,0x00,                              //Product Type = 0001
        0x01,0x00,                              //Product Code = 0001
        0xE8,0x03,                              //Hardware Version = 1000
        0xE8,0x03,                              //Firmware Version = 1000
        0x80,0x40,0x32,0x03,                    //Serial Number = 20070528
        0x3f,                                   //MAC ID
        0xff,                                   //BaudRate
        0x31,0x00,0x1C,0x00,                    //UserBaudrate Set
        0xff,                                   //CyclicParameter
        0xff,                                   //CyclicMaster
        0xff,                                   //COS type set
        0xff,                                   //Master MAC ID
        0xff,                                   //IO parameter
        0xff,                                   //IO configure
        0xff,0xff,0xff,0xff,0xff,0xff  //IO configure
};

……
unsigned char iCAN_Rcv(CANMSGFRM* pCANFrame, unsigned char nTMO, unsigned char *pErr)
{
        ……
        if(0 == CanRead(CAN1, &CANFInfo,1,0))
                return 0;
}
……
unsigned char iCAN_Send(const CANMSGFRM* pCANFrame, unsigned char nTMO,
unsigned char *pErr)
{
        ……
        if(0 != CanWrite(CAN1,&CANFInfo,1,0))
                return 1;
}
```

CAN 通道 1

CAN 通道 1

文件 3：“main.c”——初始化及各种操作

```
void *piCANMaster;
OS_EVENT* SemiCANSend;
OS_EVENT* SemiCANRcv;

void TASK0(void *pdata)
{
      char        pucCanArg[] = "BaudRate=500000 RxBufSize=20 Mode=0";
      pdata = pdata;
      CanInit(CAN1, pucCanArg, NULL);                                  // CAN 控制器 0 初始化
      SetVICIRQ (CAN_IRQ_CHN, 7, (uint32)Can_ISR);                     //  设置中断,优先级 7
      SemiCANRcv = OSSemCreate(0);
        SemiCANSend = OSSemCreate(0);
        piCANMaster = iCAN_MasterCreate(&MasterRscr,&iCAN_Master_Opt);
        iCAN_MasterLoadSlvList(piCANMaster,iCANSlvList,ICAN_SLV_NUM);
      while (1) {
       iCAN_Master_Pool(piCANMaster);
            OSTimeDly(OS_TICKS_PER_SEC/100);
      }
}

void TASK1(void *pdata)
{
      pdata = pdata;
      while (1) {
            iCAN_Master_MsgDule(piCANMaster);
      }
}

void TASK2(void *pdata)
{
      pdata = pdata;
      while (1) {
       iCAN_Master_GetMsg(piCANMaster);
      }
}

void TASK3(void *pdata)
{
        unsigned char SLV3_DI;
      pdata = pdata;
      while (1) {
             if(0 != iCAN_Master_GetDI(piCANMaster,0x03,&SLV3_DI,1))
             {
              iCAN_Master_SetDO(piCANMaster,0x02,&SLV3_DI,1);
             }
       OSTimeDly(OS_TICKS_PER_SEC/2);
      }
}
```

3. 声明

开发预备知识

MiniARM M23 系列产品将提供尽可能全面的开发模板、驱动程序及其应用说明文档以方便用户使用。但 MiniARM M23 系列产品不是教学开发平台，对于需要熟悉 ARM7 体系结构，LPC2300 系列 ARM 特性以及 ADS 开发环境的用户，建议同时购买我公司 SmartARM2300 通用教学/竞赛/工控开发平台。

修改文档的权利

广州致远电子有限公司保留任何时候在不事先声明的情况下对 MiniARM M23 系列产品相关文档的修改的权力。

ESD 静电放电保护

MiniARM M23 系列产品部分元器件已内置 ESD 保护电路，以保证产品的稳定运行。安装 MiniARM M23 系列产品时，请先将积累在身体上的静电释放，例如佩戴可靠接地的静电环，触摸接入大地的自来水管等。