



ESMARC7000 工控主板技术参考手册

感谢您购买英创信息技术有限公司的产品：ESMARC 7000 工控主板。

ESMARC 是由英创公司发展的一套嵌入式主板与应用底板的连接规范，意为英创智能模块架构(Emtronix Smart Module Architecture，以下简称 ESMARC)，ESM7000 是符合 ESMARC 连接规范的工控主板系列产品，目前包括 ESM7000 和 ESM7100 两个型号。在本文中，除非特别说明，所有型号的主板均简称 ESM7000。

ESM7000 主板是面向工业领域的双核 Cortex-A7 高性能嵌入式主板，以 NXP 的 i.MX7D 为其硬件核心，ESM7000 通过预装完整的操作系统及接口驱动，为用户构造了可直接使用的通用嵌入式核心平台。ESM7000 可选择预装 Linux-4.1.15 或 WEC7 两种系统平台，用户应用程序开发方面，对 WEC7 平台可直接使用 Microsoft 提供的著名软件开发工具 Visual Studio 2008 进行应用开发；对 Linux 平台可采用英创公司提供的 Eclipse 集成开发环境（Linux 版本），其编译生成的程序可直接运行与 ESM7000。英创公司针对 ESM7000 提供了完整的接口底层驱动以及丰富的应用程序范例，用户可在此基础上方便、快速地开发出各种工控产品。

本手册从应用的角度，详细介绍了 ESM7000 各个接口功能的使用方法，为了兼顾 Linux 和 WEC7 两种操作系统平台，所涉及的范例代码均采用 C/C++编写。客户可从资料光盘找到其他编程语言（如 C#）的范例代码。此外，英创公司针对主板和评估底板的硬件使用，编写有《ESM7000 工控主板数据手册》和《ESM7000 开发评估底板手册》。这三个手册可相互参考。

用户还可以访问英创公司网站或直接与英创公司联系以获得 ESM7000 的其他相关资料。英创信息技术有限公司联系方式如下：

地址：成都市高新区高朋大道 5 号博士创业园 B 座 407# 邮编：610041

联系电话：028-86180660 传真：028-85141028

网址：<http://www.emtronix.com> 电子邮件：support@emtronix.com

注意：本手册的相关技术内容将会不断的完善，请客户适时从公司网站下载最新版本的数据手册，恕不另行通知。

目 录

目 录	3
1、WDT 看门狗定时器	5
1.1 LINUX 平台下 WDT 的应用	5
1.2 CE 平台下 WDT 的应用	5
2、RTC 实时时钟	7
2.1 LINUX 平台下 RTC 的应用	7
2.2 CE 平台下 RTC 的应用	9
3、USB 接口	10
4、GPIO 通用数字 IO	11
4.1 LINUX 平台下 GPIO 的应用	12
4.2 CE 平台下 GPIO 的应用	14
5、UART 异步串口	16
5.1 LINUX 平台下串口的应用	16
5.2 CE 平台下串口的应用	17
6、I2C 接口	19
6.1 LINUX 平台下 I ² C 接口的应用	19
6.2 CE 平台下 I ² C 接口的应用	20
7、SPI 同步串口	22
7.1 LINUX 平台下 SPI 接口的应用	22
7.2 CE 平台下 SPI 接口的应用	24
8、IRQ 外部中断	26
8.1 LINUX 平台下 IRQ 外部中断的应用	26
8.2 CE 平台下 IRQ 外部中断的应用	28
9、PWM 脉冲输出	31
9.1 LINUX 平台下 PWM 脉冲输出的应用	31
9.2 CE 平台下 PWM 脉冲输出的应用	33
10、精简 ISA 总线	35

10.1 LINUX 平台下精简 ISA 总线的应用.....	36
10.2 CE 平台下精简 ISA 总线的应用.....	37
11、CAN 总线接口.....	39
11.1 LINUX 平台下 CAN 的应用.....	39
11.2 CE 平台下 CAN 的应用.....	42
版本历史.....	44

1、WDT 看门狗定时器

ESM7000 直接使用了 i.MX7D 芯片内部的独立看门狗定时器，系统启动后设置看门狗的超时时间为 16 秒。ESM7000 的看门狗驱动程序是基于 CPU 内部的 WDT 硬件单元而设计的，WDT 超时时间为 16 秒，当 WDT 发生超时，将产生硬件的复位信号，ESM7000 会利用这个复位信号使整个主板完成一个断电重启过程。

ESM7000 为应用程序设计了专门的 WDT 驱动程序，应用程序可通过打开 WDT 设备文件来接管系统对看门狗的操作。应用程序接管看门狗后，需按一定的时间间隔对看门狗进行刷新操作。

1.1 Linux 平台下 WDT 的应用

Linux 下 WDT 的设备节点名称为“/dev/watchdog”，用户程序可通过 open 该设备节点来接管看门狗，“/dev/watchdog”一旦打开，Linux 内核将不再进行 WDT 刷新操作，应用程序可通过 ioctl 命令来执行对 WDT 的刷新操作，WDT 相应的 ioctl 命令定义在 ESM7000_drivers.h 中：

```
#define WATCHDOG_IOCTL_BASE 'W'  
#define WDIOC_KEEPALIVE      _IOR(WATCHDOG_IOCTL_BASE, 5, int)
```

应用程序打开 WDT 设备文件的代码为：

```
fd = open("/dev/watchdog", O_RDONLY);
```

应用程序进行 WDT 刷新操作的代码为：

```
rc = ioctl(fd, WDIOC_KEEPALIVE, 0);
```

一般来讲，应用程序应在 30 秒内进行一次 WDT 刷新操作，以保证系统的正常运行。进行刷新操作的代码，应放在应用程序的管理线程循环中，以确保应用程序不会处于无意义运行，而 WDT 又不起作用。

1.2 CE 平台下 WDT 的应用

通过标准的文件操作函数操作 WDT，参考代码如下：

```
//打开看门狗设备"WDT1:"
```

```
hWDT = CreateFile(  
    _T("WDT1:"),  
    GENERIC_READ|GENERIC_WRITE,  
    FILE_SHARE_READ|FILE_SHARE_WRITE,  
    NULL,  
    OPEN_EXISTING,  
    FILE_FLAG_RANDOM_ACCESS,  
    NULL);  
if( hWDT==INVALID_HANDLE_VALUE )  
{  
    printf( "Open WDT device fail!\n" );  
    return -1;  
}  
//得到喂狗周期  
bRet = ReadFile( hWDT, &dwWDTPeriod, sizeof(DWORD), &dwLen, NULL );  
  
//应用程序喂狗  
WriteFile( hWDT, &dwWDTPeriod, sizeof(DWORD), &dwLen, NULL );
```

完整参考代码：开发光盘\Software\Test_wdt

2、RTC 实时时钟

ESM7000 的实时时钟驱动是标准的 RTC 接口方式。

2.1 Linux 平台下 RTC 的应用

Linux 下 RTC 的设备节点名称为“/dev/rtc0”，用户程序可通过 open 该设备节点来读取或设置实时时钟。

ESM7000 对 RTC 实时时钟进行操作可以按照 Linux 标准方法进行相关的 ioctl 命令操作，相关的定义在 linux/rtc.h 文件下。参考代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <sys/ioctl.h>
#include <linux/rtc.h>
#include <linux/ioctl.h>

int main( int argc,char* argv[] )
{
    time_t          t1;
    int             ret;
    struct rtc_time  rtc_tm;
    int             rtc_fd;
    int             i;

    printf("==== RTC Test  ====\n");

    // 解析命令行参数：年-月-日 时:分:秒
    if( argc > 1 )
    {
        sscanf( argv[1], "%d-%d-%d", &rtc_tm.tm_year, &rtc_tm.tm_mon,
                &rtc_tm.tm_mday );
    }
    else
    {
        rtc_tm.tm_mday = 7;
        rtc_tm.tm_mon = 11;
        rtc_tm.tm_year = 2012;
    }
    if( argc > 2 )
```

```
{
    sscanf( argv[2], "%d:%d:%d", &rtc_tm.tm_hour, &rtc_tm.tm_min,
            &rtc_tm.tm_sec );
}
else
{
    rtc_tm.tm_hour = 15;
    rtc_tm.tm_min = 20;
    rtc_tm.tm_sec = 0;
}

rtc_fd = open("/dev/rtc0", O_RDWR, 0);
if (rtc_fd == -1)
{
    printf("/dev/rtc0 open error\n\n");
    return -1;
}

if( argc > 2 )
{
    rtc_tm.tm_mon = rtc_tm.tm_mon - 1;
    rtc_tm.tm_year = rtc_tm.tm_year - 1900;

    /* Set the system time/date */
    t1 = timelocal( (tm*)&rtc_tm );
    stime( &t1 );

    /* Set the RTC time/date */
    ret = ioctl(rtc_fd, RTC_SET_TIME, &rtc_tm);

    if (ret == -1)
    {
        printf("rtc ioctl RTC_SET_TIME error\n");
    }
    sleep( 1 );
}

//show RTC message
for(i1 = 0; ; i1++)
{
    ret = ioctl(rtc_fd, RTC_RD_TIME, &rtc_tm);
    if( ret < 0)
    {
        printf("RTC_RD_TIME failed %d", ret);
    }
}
```



```
        break;
    }
    printf("RTC:%d.%d.%d-%d:%d:%d\n", rtc_tm.tm_year + 1900,
        rtc_tm.tm_mon + 1,
        rtc_tm.tm_mday,
        rtc_tm.tm_hour,
        rtc_tm.tm_min, rtc_tm.tm_sec );

    sleep(1);
}

close(rtc_fd);

return 0;
}
```

在配套的光盘资料中有一个相应的测试程序 test_rtc.c 供客户参考。

2.2 CE 平台下 RTC 的应用

通过标准的 Windows API 操作系统 RTC。通过 SetLocalTime 设置系统时间，使用 GetLocalTime() 获取当前系统时间。

完整参考代码：开发光盘\Software\Test_Time

3、USB 接口

ESM7000 可提供 5 个 USB 端口：四路高速主控接口，和一个 USB OTG 接口。ESM7000 的 USB 主控接口可直接与标准 U 盘相连，ESM7000 会自动把 U 盘中的系统配置文件 `userinfo.txt` 拷贝到系统中，并按照 `userinfo.txt` 设置 IP 等参数，最后启动用户的应用程序。

在 CE 平台下 USB 主控口也可支持标准的键盘、鼠标等设备。ESM7000 的 USB OTG 接口，即可作为 USB 主控接口使用，也可作为 USB 设备接口使用。作为 USB 设备接口的一个典型应用，就是支持 Microsoft 的 ActiveSync 传输协议，用户可利用它方便的实现对 ESM7000 文件的管理，也可以利用 ActiveSync 来调试应用程序。另外 ActiveSync 还把 USB 设备口映射成串口，占用串口逻辑号 COM1，所以 ESM7000 真正的物理串口对应的逻辑编号从 COM2 开始。主控 USB 的供电电路很简单，布置在 ESM7000 的评估底板上，客户在设计自己的应用底板时，可参考该电路。

4、GPIO 通用数字 IO

ESM7000 的 32 位 GPIO0 – GPIO31 均为可独立设置方向的通用数字 IO，所有 GPIO 的上电初始状态均为输入状态带上拉电阻。ESM7000 为了保持模块的紧凑尺寸及机械强度，其 GPIO 与主板的其它接口功能采用了管脚复用的设计，具体复用情况如下表所示：

管脚#	复用功能	复用功能简要说明
GPIO0	CTS2#	COM2 (ttyS1) 口的 CTS#复用管脚。
GPIO1	RTS2#	COM2 (ttyS1) 口的 RTS#复用管脚。
GPIO2	RXD6	COM6 (ttyS5) 口的 RXD 复用管脚。
GPIO3	TXD6	COM6 (ttyS5) 口的 TXD 复用管脚。
GPIO4	RXD7	COM7 (ttyS6) 口的 RXD 复用管脚。
GPIO5	TXD7	COM7 (ttyS6) 口的 TXD 复用管脚。
GPIO6	PWM1	PWM1 输出，
GPIO7	PWM2	PWM2 输出
GPIO8	PWM3	PWM3 输出
GPIO9	PWM4	
GPIO10	CAN1_RXD	CAN1 口的 RXD 数据输入信号
GPIO11	CAN1_TXD	CAN1 口的 TXD 数据输出信号
GPIO12	CAN2_RXD	CAN2 口的 RXD 数据输入信号
GPIO13	CAN2_TXD	CAN2 口的 TXD 数据输出信号
GPIO14		
GPIO15		
GPIO16		
GPIO17		
GPIO18		
GPIO19		
GPIO20		
GPIO21		
GPIO22		

GPIO23		
GPIO24	IRQ1	外部中断 IRQ1 输入
GPIO25	IRQ2	外部中断 IRQ2 输入
GPIO26	I2C_SDA	I2C 总线的 SDA 数据线
GPIO27	I2C_SCL	I2C 总线的 SCL 时钟线
GPIO28	SPI_MISO	SPI 接口的数据串入 MISO
GPIO29	SPI_MOSI	SPI 接口的数据串出 MOSI
GPIO30	SPI_SCLK	SPI 接口的同步时钟 SCLK
GPIO31	SPI_CSON	SPI 接口的片选控制 CSON

在系统启动后的初始状态，所有的 GPIO 都是有效的，一旦应用程序打开某个接口的设备文件，则对应的 GPIO 功能将被禁止。注意即使应用程序关闭了设备文件，对应的 GPIO 功能同样是被禁止的。因为在嵌入式系统中，不可能存在一条管脚动态复用的情况。

4.1 Linux 平台下 GPIO 的应用

Linux 应用程序若希望操作 GPIO，首先需要打开 GPIO 的设备文件：

```
fd = open("/dev/ESM7000_gpio", O_RDWR);
```

对 GPIO 的操作可归为 5 种基本操作如下：

- 1、GPIO 输出使能：在任何时候 GPIO 的输入功能都是有效的。当执行了该项操作后，对应的 GPIO 位就为数字输出了，而应用程序仍然可以读取当前管脚的状态
- 2、GPIO 输出禁止：执行该操作后，对应 GPIO 只能作为数字输入管脚使用了
- 3、GPIO 输出置位：执行该操作后，对应的 GPIO 输出高电平
- 4、GPIO 输出清零：执行该操作后，对应的 GPIO 输出低电平
- 5、读取 GPIO 状态：执行该操作后，返回参数的 32 位分别对应各位 GPIO 当前管脚的电平状态

ESM7000 的 GPIO 驱动程序为上述 5 种功能设置了对应的命令参数，定义如下：

```
#define ESM7000_GPIO_OUTPUT_ENABLE      0
#define ESM7000_GPIO_OUTPUT_DISABLE    1
```

```
#define ESM7000_GPIO_OUTPUT_SET          2
#define ESM7000_GPIO_OUTPUT_CLEAR      3
#define ESM7000_GPIO_INPUT_STATE      5
```

然后根据 ESM7000_drivers.h 中所列的上述命令参数，利用 write() read() 函数来实现对于 GPIO 的操作。

```
struct double_pars
{
    unsigned int  par1;
    unsigned int  par2;
};
```

其中 par1 用于定义命令参数，par2 用于定义需要操作的 GPIO 位，32 位 bit 分别对应 GPIO0-GPIO31，对任意位 GPIO 设置命令，参数中对应 bit 位置 1 才有效，否则无效。

具体操作 GPIO 的典型代码为：

```
int GPIO_OutEnable(int fd, unsigned int dwEnBits)
{
    int rc;
    struct double_pars dpars;

    dpars.par1 = ESM7000_GPIO_OUTPUT_ENABLE;    // 0
    dpars.par2 = dwEnBits;

    rc = write(fd, &dpars, sizeof(struct double_pars));
    return rc;
}

int GPIO_PinState(int fd, unsigned int* pPinState)
{
    int rc;
    struct double_pars dpars;

    dpars.par1 = ESM7000_GPIO_INPUT_STATE;    // 5
    dpars.par2 = *pPinState;

    rc = read(fd, &dpars, sizeof(struct double_pars));
    if(!rc)
    {
        *pPinState = dpars.par2;
    }
    return rc;
}
```

在上述操作中，对参数中 `par2` 没有置位的 GPIO，其状态保持不变。由于 ESM7000 的部分 GPIO 管脚还复用了其他功能，如串口等。这样即使启动串口功能，驱动程序仍然可以操作其他 GPIO，而不会影响串口的功能。

4.2 CE 平台下 GPIO 的应用

ESM7000 板上已固化了面向 GPIO 的 WinCE 标准驱动程序，应用程序打开文件名为“PIO1:”的文件对象，通过标准的 `ReadFile (...)` 和 `WriteFile (...)` 函数进行 GPI 操作。为了方便用户使用，我们对操作 GPIO 的函数做了进一步封装，导出如下几个简洁易用的 API 函数。

```
// 功能描述：打开GPIO设备
// 输入参考：lpDevName 打开的设备名称，这里必须为_T("PIO1:")
// 返回值：= INVALID_HANDLE_VALUE,打开设备失败
HANDLE OpenGPIO( LPCWSTR lpDevName );

// 功能描述：将GPIO设置为输出状态
// 输入参数：hGpio 设备句柄
//           dwEnBits 其-31位对应于GPIO0-GPIO31,其中为的位对应的GPIO会被设置为输出状态
// 返回值：= TURE 操作成功
BOOL GPIO_OutEnable( HANDLE hGpio, UINT32 dwEnBits);

// 功能描述：将GPIO设置为输入状态
// 输入参数：hGpio 设备句柄
//           dwDisBits 其-31位对应于GPIO0-GPIO31,其中为的位对应的GPIO会被设置为输入状态
// 返回值：= TURE 操作成功
BOOL GPIO_OutDisable( HANDLE hGpio, UINT32 dwDisBits);

// 功能描述：设置GPIO输出高电平
// 输入参数：hGpio 设备句柄
//           dwSetBits 其-31位对应于GPIO0-GPIO31,其中为的位对应的GPIO会被设置为高电平
// 返回值：= TURE 操作成功
BOOL GPIO_OutSet( HANDLE hGpio, UINT32 dwSetBits);

// 功能描述：设置GPIO输出低电平
// 输入参数：hGpio 设备句柄
//           dwClearBits 其-31位对应于GPIO0-GPIO31,其中为的位对应的GPIO会被设置为低电平
// 返回值：= TURE 操作成功
BOOL GPIO_OutClear( HANDLE hGpio, UINT32 dwClearBits);

// 功能描述：读取GPIO的电平状态
// 输入输出参数：pPinState
```

```
//      输入时：其-31位对应于GPIO0-GPIO31,将读取其中为的位对应的GPIO电平状态
//      输出时：返回GPIO的电平状态
// 返回值：= TURE 操作成功
BOOL GPIO_PinState( HANDLE hGpio, UINT32* pPinState);

// 功能描述：关闭GPIO设备
BOOL CloseGPIO( HANDLE hGpio );
```

完整参考代码：开发光盘\Software\Test_Gpio。

5、UART 异步串口

ESM7000 物理上有 6 路串口，列表如下：

CE 名称	Linux 名称	串口速度	功能简要说明
COM2	ttyS1	高速串口	支持 RTS/CTS 硬件流控。
COM3	ttyS2	高速串口	3 线制，RS232 电平接口。
COM4	ttyS3	高速串口	3 线制，TTL 电平。
COM5	ttyS4	高速串口	3 线制，TTL 电平。
COM6	ttyS5	低速串口	3 线制，波特率不高于 19200bps。与 GPIO 复用管脚。
COM7	ttyS6	低速串口	

ESM7000 的 6 个串口按最高波特率分为高速串口 COM2 – COM5 和低速串口 COM6 – COM7。高速串口的最高波特率可达 5Mbps，而低速串口允许的波特率在 1200bps – 19200bps 之间，支持奇偶校验、MARK / SPACE 设置。因此低速串口更适合作为 RS485 端口来应用。ESM7000 在 RS485 驱动方面，除了可以采用 TXD 自动控制数据收发方向切换（具体电路请参考 ESM7000 开发评估底板电路原理图）外，还可选择一位 GPIO 作为 RTS，实现硬件方向控制。

5.1 Linux 平台下串口的应用

每个串口都有独立的中断模式，使得多个串口能够同时实时进行数据收发。各个串口的驱动已经包含在 Linux 操作系统的内核中，ESM7000 在 Linux 系统启动完成时，各个串口已作为字符设备完成了注册加载，用户的应用程序可以以操作文件的方式对串口进行读写，从而实现数据收发的功能。

在 Linux 中，所有的设备文件都位于“/dev”目录下，ESM7000 上六个串口所对应的设备名依次为：“/dev/ttyS1”、“/dev/ttyS2”、“/dev/ttyS3”、“/dev/ttyS4”、“/dev/ttyS5”、“/dev/ttyS6”。

在 Linux 下操作设备的方式和操作文件的方式是一样的，调用 open() 打开设备文件，再调用 read()、write() 对串口进行数据读写操作。这里需要注意的是打开串口除了设置普通的读写之外，还需要设置 O_NOCTTY 和 O_NDLEAY，以避免该串口成为一个控制终端，有可能会影响到用户的进程。如：

```
sprintf( portname, "/dev/ttyS%d", PortNo); //PortNo为串口端口号，从1开始
```

```
m_fd = open( portname,O_RDWR | O_NOCTTY | O_NONBLOCK);
```

作为串口通讯还需要一些通讯参数的配置，包括波特率、数据位、停止位、校验位等参数。在实际的

操作中，主要是通过设置 `struct termios` 结构体的各个成员值来实现，一般会用到的函数包括：

```
tcgetattr( );
tcflush( );
cfsetispeed( );
cfsetospeed( );
tcsetattr( );
```

在进行 RS485 通讯时，如果需要设置 RTS 控制模式，可以采用调用 `ioctl` 命令来激活一位 GPIO 作为 RTS 方向控制。

```
#define ESM7000_IOCTL_SET_RTS_PIN      _IOW('T', 0x32, int)
//config GPIO pin for RTS
unsigned int  gpio = GPIO12;
res = ioctl( m_fd, ESM7000_IOCTL_SET_RTS_PIN, (unsigned long)&gpio );
```

5.2 CE 平台下串口的应用

在应用软件方面，需要主要代码如下：

打开串口设备文件

```
HANDLE hSer;
hSer = CreateFile(_T("COM7:"),           // name of device
    GENERIC_READ|GENERIC_WRITE,        // desired access
    FILE_SHARE_READ|FILE_SHARE_WRITE,  // sharing mode
    NULL,                               // security attributes (ignored)
    OPEN_EXISTING,                     // creation disposition
    FILE_FLAG_RANDOM_ACCESS,           // flags/attributes
    NULL);                             // template file (ignored)
```

设置一位 GPIO 作为 RTS

```
DWORD dwRtsGpioPin = GPIO26;           //选择 GPIO26 作为 RTS
```

```
If (!DeviceIoControl (hSer,
    IOCTL_SET_UART_RTS_PIN,
    & dwRtsGpioPin, sizeof(DWORD),
    NULL, 0,
    NULL, NULL))
```

```
{
    // 出错处理。。
```

```
}
```

设置串口 RTS 控制模式

```
DCB SerDCB;

SerDCB.DCBlength = sizeof(DCB);
GetCommState(hSer, &SerDCB );           // 从驱动读取当前DCB
SerDCB.fRtsControl = RTS_CONTROL_TOGGLE;
SetCommState(hSer, &SerDCB );           // 再设置回驱动
```

高速串口中，只有 COM2 配置有 RTS/CTS 硬件握手功能，而其他都是常规的三线制串口。由于 RTS/CTS 硬件握手功能的应用并不是很多，同时考虑充分利用 GPIO 的功能，在打开“COM2:”时，RTS/CTS 硬件握手功能并没有激活，而对应管脚 GPIO0、GPIO1 继续保持为 GPIO 状态。应用程序需通过设置才能激活 RTS/CTS 硬件握手功能：

激活串口 RTS/CTS 硬件握手功能

```
DCB SerDCB;

SerDCB.DCBlength = sizeof(DCB);
GetCommState(hSer, &SerDCB );           // 从驱动读取当前DCB
SerDCB.fRtsControl = RTS_CONTROL_HANDSHAKE;
SetCommState(hSer, &SerDCB );           // 再设置回驱动
```

6、I2C 接口

ESM7000 的 I²C 接口为 2 线制标准 I²C 接口,信号电平为 3.3V 的 LVTTTL 电平,最高传输波特率为 400kbps。在使用 I2C 接口时,应对 SCL 和 SDA 两个信号线均加 10K 的上拉电阻。ESM7000 I2C 的 SDA 信号线与 GPIO26 复用管脚, SCL 信号线与 GPIO27 复用管脚,应用程序中一旦将 GPIO26、GPIO27 作为 i2c 的应用,就不能再作为 GPIO 进行使用了。

GPIO26	I2C_SDA	与 I2C 总线的 SDA 复用管脚。
GPIO27	I2C_SCL	与 I2C 总线的 SCL 复用管脚。

6.1 Linux 平台下 I²C 接口的应用

Linux 应用程序若希望操作 I²C,首先需要打开 I²C 的设备文件:

```
fd = open("/dev/i2c-0", O_RDWR);
```

然后可以按照 Linux 标准方法进行相关的 ioctl 命令操作,相关的定义在 linux/i2c.h linux/i2c-dev.h 文件下。

打开 i2c 设备文件:

```
// open driver of i2c
fd = open("/dev/i2c-0", O_RDWR);
```

读写数据的操作采用 i2c-dev.h 文件中定义的数据结构:

```
/* This is the structure as used in the I2C_RDWR ioctl call */
struct i2c_rdwr_ioctl_data {
    struct i2c_msg *msgs; /* pointers to i2c_msgs */
    __u32 nmsgs; /* number of i2c_msgs */
};
```

部分代码如下:

```
bool I2CWrite( int fd, pl2CParameter pl2CPar)
{
    struct i2c_rdwr_ioctl_data i2c_data;
    int rc;

    /*i2c_data.nmsgs配置为1*/
    i2c_data.nmsgs = 1;
    i2c_data.msgs = (struct i2c_msg*)malloc(i2c_data.nmsgs*sizeof(struct i2c_msg));
    if( !i2c_data.msgs )
```

```

    return -1;

    i2c_data.msgs[0].buf = (unsigned char* )malloc ( pl2CPar->iDLen + 1 );

    //write data to i2c-dev
    (i2c_data.msgs[0]).len = pl2CPar->iDLen + 1 ;           // 写入目标的地址和数据
    (i2c_data.msgs[0]).addr = pl2CPar->SlaveAddr;         // 设备地址
    (i2c_data.msgs[0]).flags= 0;                          // write
    (i2c_data.msgs[0]).buf[0]= pl2CPar->RegAddr & 0xff;// 写入目标的地址
    memcpy( &((i2c_data.msgs[0]).buf[1]), pl2CPar->pDataBuff, pl2CPar->iDLen );
    rc=ioctl( fd, I2C_RDWR,(unsigned long)&i2c_data );
    if( rc<0 )
    {
        perror("ioctl(write)");
    }
    free( i2c_data.msgs[0].buf );
    free( i2c_data.msgs );
    if( rc < 0 )
        return false;
    return true;
}

```

在配套的光盘资料中有一个相应的测试程序 test_i2c.c 供客户参考。

利用 i2c 接口我们提供了 8×8 键盘扩展模块 ETA202，以及 IO 扩展模块 ETA715，配套的资料中均有这两个模块的测试程序：

test_eta202

test_eta715

6.2 CE 平台下 I²C 接口的应用

ESM7000 板上已固化了面向 I²C 接口的 WinCE 标准驱动程序,应用程序打开文件名为“I2C1:”的文件对象,通过标准的 DeviceControl 进行 I²C 数据传输。

基本的 I²C 数据结构如下：

```

typedef struct
{
    UINT16  wAddr;           // I2C从设备地址
    UINT16  wFlags;         // I2C操作标志, 有效的定义如下:
        #define I2C_M_TEN    0x0010 //表示wAddr为10-bit地址
        #define I2C_M_RD     0x0001 // I2C读操作标志
        #define I2C_M_WR     0x0000 // I2C写操作标志
}

```

```
    UINT16  wLen;           // I2C操作的数据长度
    PBYTE   pbyBuf;        // 数据buffer
}I2C_MSGS_T, *PI2C_MSGS_T;

typedef struct
{
    I2C_MSGS_T *pi2CMsgs;
    INT32      iNumMsgs;
}I2C_MSGS_BLOCK_T, *PI2C_MSGS_BLOCK_T;
```

完整参考代码：开发光盘\Software\ETA202Test。ETA202Test是ESM7000的I2C接口，连接TCA8418 I2C键盘的测试程序。

7、SPI 同步串口

ESM7000 的 SPI 接口为 4 线制标准 SPI 接口，信号电平为 3.3V 的 TTL 电平，最高传输波特率为 20Mbps。主要应用于设备内部各功能单元之间的短距离高速传输。

ESM7000 提供的 SPI 驱动支持 master 模式，该 SPI 接口为四线制 SPI，包括：时钟 CLK；数据 MISO (master in, slave out)；数据 MOSI (master out, slave in)；片选 CS，SPI 管脚分别和 GPIO28- GPIO31 复用。

GPIO28	SPI_MISO	与 SPI 接口的数据串入 MISO 复用管脚。
GPIO29	SPI_MOSI	与 SPI 接口的数据串出 MOSI 复用管脚。
GPIO30	SPI_SCLK	与 SPI 接口的同步时钟 SCLK 复用管脚。
GPIO31	SPI_CSON	与 SPI 接口的片选控制 CSON 复用管脚。

SPI 常用四种数据传输模式，主要差别在于：输出串行同步时钟极性（CPOL）和相位（CPHA）可以进行配置。如果 CPOL= 0，串行同步时钟的空闲状态为低电平；如果 CPOL= 1，串行同步时钟的空闲状态为高电平。如果 CPHA= 0，在串行同步时钟的前沿（上升或下降）数据被采样；如果 CPHA = 1，在串行同步时钟的后沿（上升或下降）数据被采样。对于 SPI 模式的定义如下表一：

SPI Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

7.1 Linux 平台下 SPI 接口的应用

ESM7000 在系统启动完成后，SPI 所对应的设备节点为：“/dev/spidev1.0”。

应用程序可以通过 read()、write()、ioctl()函数使用 spi-dev 驱动，在 ESM7000 中 SPI 是全双工模式，最高波特率为 12Mbps， 所以选择调用 ioctl()函数进行数据通讯以及 SPI 通讯参数的设置。如：

```
static const char *device = "/dev/spidev1.0";
static uint8_t mode = 3;
```

```
static uint8_t bits = 8;
static uint32_t speed = 1000000;

struct spi_ioc_transfer tr[2];

void transfer(int fd)
{
    int ret;
    int i1;

    printf("transfer\n");
    tr[0].tx_buf = (unsigned long)tx;
    tr[0].rx_buf = (unsigned long)rx;
    tr[0].len = ARRAY_SIZE(tx);
    tr[0].delay_usecs = delay;
    tr[0].speed_hz = speed;
    tr[0].bits_per_word = bits;

    ret = ioctl(fd, SPI_IOC_MESSAGE(1), tr);
    if (ret < ARRAY_SIZE(tx))
        pabort("can't send spi message");
    printf("ret=%d\n", ret);
    for (i1=0; i1<ret; i1++)
        printf("0x%.2X\n", rx[i1]);
}

int main( int argc, char *argv[] )
{
    int i, fd;
    int ret = 0;

    fd = open(device, O_RDWR);
    if (fd < 0)
        pabort("can't open device");

    // 设置 spi mode, 其定义参见表一
    ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
    if (ret == -1)
        pabort("can't set spi mode");
    ret = ioctl(fd, SPI_IOC_RD_MODE, &mode);
    if (ret == -1)
        pabort("can't get spi mode");

    // 设置数据bit位
```

```

ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
if (ret == -1)
    pabort("can't set bits per word");
ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
if (ret == -1)
    pabort("can't get bits per word");

// 设置SPI通讯波特率
ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
if (ret == -1)
    pabort("can't set max speed hz");

ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
if (ret == -1)
    pabort("can't get max speed hz");

printf("spi mode: %d\n", mode);
printf("bits per word: %d\n", bits);
printf("max speed: %d Hz (%d KHz)\n", speed, speed/1000);

transfer( fd );

    close(fd);
}

```

光盘资料中有 spi 的测试代码。

7.2 CE 平台下 SPI 接口的应用

ESM7000 板上已固化了面向 SPI 接口的 WinCE 标准驱动程序, 应用程序只需要打开文件名为“SPI1:”的文件对象, 就可以通过标准的 DevicloControl 进行 SPI 数据传输了。基本的 SPI 数据结构如下:

```

typedef struct
{
    DWORD    dwMode;           // = 0, 1, 2, 3 -> CLKPOL / CLKPHA
    DWORD    dwClockHz;       // 设置SPI通讯波特率
    DWORD    dwBitsPerWord;   // SPI通讯位长度8, 12, 14, 16, 32
    DWORD    dwMaxClockHz;    // 返回CPU支持的SPI最高波特率
} SPI_CONFIG_T, *PSPI_CONFIG_T;

typedef struct
{
    LPVOID   pTxBuf;          // SPI发送数据缓存

```



```
LPVOID    pRxBuf;           // SPI接收数据缓存
DWORD     dwLen;           // SPI 数据读写的字节长度
}SPI_MSGS_T, *PSPI_MSGS_T;
```

如果传入SPI_MSGS_T结构体的pTxBuf为空时(NULL)，驱动程序自动进行SPI半双工读操作，当pRxBuf为空时为SPI半双工写操作。pTxBuf和pRxBuf不能现时空。

完整参考代码：开发光盘\Software\Test_Spi。

8、IRQ 外部中断

ESM7000 共有 2 路外部中断输入 IRQ1 和 IRQ2，中断信号的缺省配置为上升沿有效，即触发中断。分别与 GPIO24 及 GPIO25 复用管脚。

GPIO24	IRQ1	与外部中断 IRQ1 复用管脚。
GPIO25	IRQ2	与外部中断 IRQ2 复用管脚。

8.1 Linux 平台下 IRQ 外部中断的应用

IRQ1 对应设备文件“/dev/ESM7000_irq1”，IRQ2 对应设备文件“/dev/ESM7000_irq2”。ESM7000 驱动程序采用了 Linux 的异步通知的机制，即外部中断信号一旦触发中断驱动程序，驱动程序会主动向应用程序发送 SIGIO 信号（该信号为 Linux 系统预定义的信号），这样应用程序就不需要查询设备的状态，只需要简单响应 SIGIO 进行相关操作即可。在应用程序的相应函数中，可通过操作其他驱动程序的 API 函数来实现对硬件的操作，如精简 ISA 驱动或 GPIO 驱动等。有关应用程序响应中断驱动程序发出的 SIGIO 的方法，Linux 操作系统已提供了成熟的方法，即通过在应用程序初始化阶段调用：

```
signal(SIGIO, ESM7000_irq_handler);
```

把响应函数 ESM7000_irq_handler(int signum)与 Linux 信号 SIGIO 绑定。应用程序对硬件中断的具体响应操作代码则放在 ESM7000_irq_handler 函数中。此外 ESM7000 的中断驱动程序还在内部设置了一个中断次数的计数器，应用程序可以通过 read()函数来读取计数值，读取后内部计数值自动清零。

以下范例程序是 GPIO0 产生一个正脉冲，GPIO0 信号连接到 IRQ1 上，利用 GPIO0 的上升沿触发中断。在中断响应函数进行中断计数。相关的主要代码如下：

```
//接收到异步读信号后的动作
void ESM7000_irq_handler( int signum )
{
    nIrqCounter++;
    printf( "there is a IRQ!!!\n" );
}

int main(int argc, char** argv)
{
    int          i1, nNum;
    int          irq_no, irq_fd;
    int          oflags;
```

```
unsigned int NumOfIrqs;
unsigned int SumNumOfIrqs;
char device[32];
int sec;

struct gpio_pulse gp;

printf("Test IRQ Async Signation on ESM7000\n");

irq_no = 1;
nNum = 100;
if(argc > 1)
{
    irq_no = atoi(argv[1]);
}
if( argc > 2 )
{
    nNum = atoi(argv[2]);
}

gp.fd = 0;
gp.gpio = GPIO0;
gp.low_ms = 10;
gp.high_ms = 10;
gp.number = nNum;

sec = nNum * ( gp.low_ms + gp.high_ms ) /1000 + 5;
i1 = ConfigGPIO( (void*)&gp );
printf( "Open GPIO %d\n", gp.gpio );

sprintf( device, "/dev/ESM7000_irq%d", irq_no );
irq_fd = open(device, O_RDWR, S_IRUSR | S_IWUSR);
if (irq_fd < 0)
{
    printf("can not open /dev/ESM7000\_irq1 device file!\n");
    return -1;
}
printf( "Open %s sec:%d\n", device, sec );

//启动信号驱动机制
signal(SIGIO, ESM7000_irq_handler); // 让irq_handler()处理SIGIO信号
fcntl(irq_fd, F_SETOWN, getpid( ) );
oflags = fcntl(irq_fd, F_GETFL);
fcntl(irq_fd, F_SETFL, oflags | FASYNC);
```

```
StartPulseThread( (void*)&gp );

nIrqCounter = 0;
SumNumOfIrqs = 0;
for( i1=0; i1<sec; i1++ )
{
    sleep( 1 );
    read( irq_fd, (void*)&NumOfIrqs, sizeof(int) );
    SumNumOfIrqs += NumOfIrqs;
    printf("%d -- ISRcount = %d, SignalCount = %d\n", (i1 + 1), SumNumOfIrqs,
nIrqCounter);
}

close(irq_fd);
printf("close file\n");
return 0;
}
```

在上面的程序中，通过中断驱动内部计数值 `SumNumOfIrqs` 与应用程序响应计数值 `nIrqCounter` 的比较，可以判断中断是否有丢失。

具体代码可参见光盘资料。

8.2 CE 平台下 IRQ 外部中断的应用

当应用程序打开 IRQ 驱动程序对应的设备文件“IRQ1:”-“IRQ2:”后，外部中断输入上升沿正脉冲，脉冲宽度大于 50ns，驱动程序将响应该下降沿中断，并产生事件通知处于等待中的应用线程。典型代码包括：

打开 IRQ 文件

```
HANDLE hIrq;
hIrq = CreateFile(L"IRQ1:",
    0,
    0,
    NULL,
    OPEN_EXISTING,
    FILE_FLAG_RANDOM_ACCESS,
    NULL);
```

等待 IRQ 时间子程序

```
DWORD WaitIRQEvent ( HANDLE hIrq, DWORD dwTimeout )
{
```

```

        DWORD dwRet = 0;

    If(! bRet = DeviceIoControl(hIrq,                // file handle to the driver
        IOCTL_WAIT_FOR_IRQ,                        // I/O control code
        &dwTimeout,                                // in buffer
        sizeof(DWORD),                            // in buffer size
        &dwRet,                                    // out buffer
        sizeof(DWORD),                            // out buffer size
        NULL,                                      // pointer to number of bytes returned
        NULL) )
    {
        //出错
        dwRet = WAIT_FAILED;
    }
    Return dwRet;
}

```

注：dwTimeout 为等待超时时间，如果为 INFINITE 则一直等待，直到 IRQ 事件产生或 IRQ 关闭。

应用线程等待中断事件

```

DWORD dwTimeoutMS = 5000;    //超时时间设置为 5 秒
DWORD dwReturn;

```

```

dwReturn = WaitIRQEvent (hIRQ, dwTimeoutMS) ;
if (dwReturn == WAIT_OBJECT_0)
{
    //外部中断发生，进行中断处理
    //... ..
}
else if (dwReturn == WAIT_TIMEOUT)
{
    //超时处理
    //... ..
}
else
{
    //出错处理
    //... ..
}

```

计算中断产生次数

```

DWORD dwCount;
bRet = DeviceIoControl(hIrq,
    IOCTL_GET_COUNT,

```

```
NULL,  
0,  
&dwCount,  
sizeof(DWORD),  
NULL,  
NULL);
```

有的时候需要统计自打开 IRQ 之后，一共产生了多少次中断，可用此代码得到中断数 dwCount。

应用程序还可以通过 DeviceIoControl 设置中断的触发方式，ESM7000 IRQ 支持电平和边沿中断触发，支持的触发方法包括：

- 0: 低电平触发
- 1: 高电平触发
- 2: 上升沿触发
- 3: 下降沿触发

配置为高电平触发方式的示例代码如下：

```
DWORD dwInterruptLogic = 1;  
DeviceIoControl(  
    g_hIRQ, // file handle to the driver  
    (DWORD)IOCTL_GENERIC_SETUP, // I/O control code  
    &dwInterruptLogic, // in buffer  
    sizeof(DWORD), // in buffer size  
    NULL, // out buffer  
    0, // out buffer size  
    NULL, // pointer to number of bytes returned  
    NULL);
```

完整参考代码：开发光盘\Software\Test_Irq。

9、PWM 脉冲输出

ESM7000 共有 3 路 PWM 输出，其最高输出频率可达 30MHz，但如果希望保证一定精度的占空比（1% 的精度），则输出最高频率只能到 1MHz。这 3 路 PWM 分别与分别与 GPIO6 – GPIO8 复用管脚。

GPIO6	PWM1	PWM1 输出，
GPIO7	PWM2	PWM2 输出
GPIO8	PWM3	PWM3 输出

9.1 Linux 平台下 PWM 脉冲输出的应用

ESM7000 板卡在 Linux 平台下 PWM 脉冲输出所对应的设备节点名称为：

脉冲输出	设备节点名称
PWM1	"/dev/ESM7000_pwm1"
PWM2	"/dev/ESM7000_pwm2"
PWM3	"/dev/ESM7000_pwm3"

对 PWM 的操作可归为 2 种基本操作如下：

- 1、PWM 脉冲输出使能，按照设置的频率和占空比参数输出 PWM 脉冲。
- 2、PWM 脉冲输出停止。

所对应的命令参数，定义如下：

```
#define ESM7000_PWM_START          10
#define ESM7000_PWM_STOP          11
```

在 ESM7000_drivers.h 文件中还定义了 PWM 的数据结构，包括频率、占空比以及极性参数：

```
struct pwm_config_info
{
    unsigned int    cmd;           // = 10, 11,...
    unsigned int    freq;         /* in Hz */
    unsigned int    duty;        /* in % */
    unsigned int    polarity;
};
```

其中：

freq 表示输出的脉冲频率，单位为 Hz。Freq 的取值范围 10Hz – 1MHz。

duty 表示输出脉冲的占空比，单位为%。Duty 的取值范围：1 – 99。

Polarity 表示输出脉冲的极性，选择 0 或者 1。

进行 PWM 操作时，首先打开相应的设备节点文件，然后再调用 write()函数进行 pwm 的设置、启动以及停止操作，以下为相关的应用代码：

```
fd = open("/dev/ESM7000_pwm1", O_RDWR);

#include "ESM7000_drivers.h"
#include "pwm_api.h"

#define POLARITY          PWM_POLARITY_INVERTED;
//#define POLARITY          PWM_POLARITY_NORMAL;

int PWM_Start(int fd, int freq, int duty )
{
    int rc;
    struct pwm_config_info conf;

    conf.cmd = ESM7000_PWM_START;
    conf.freq = freq;
    conf.duty = duty;
    conf.polarity = POLARITY;

    rc = write(fd, &conf, sizeof(struct pwm_config_info));
    return rc;
}

int PWM_Stop(int fd )
{
    int rc;
    struct pwm_config_info conf;

    memset( &conf, 0, sizeof(struct pwm_config_info));
    conf.cmd = ESM7000_PWM_STOP;

    rc = write(fd, &conf, sizeof(struct pwm_config_info));
    return rc;
}
```


另外，如果关闭设备文件，也将停止 PWM 脉冲输出。

9.2 CE 平台下 PWM 脉冲输出的应用

ESM7000 板上已固化了面向 PWM 接口的 WinCE 标准驱动程序，应用程序只需打开文件名为“PWM1:”-“PWM3:”的文件对象，再通过 WriteFile 设置启动 PWM 脉冲的参数（频率和占空比）即可，应用程序也可通过 WriteFile 随时停止 PWM 的输出。典型的 PWM 应用，包括为红外串口提供调制信号（38.5KHz，50%占空比）、为 ISO7816 提供时钟信号（3.5712MHz，9600bps 波特率）。

基本的 PWM 数据结构如下：

```
typedef struct
{
    DWORD    dwFreq;           // PWM频率，单位为Hz，= 0: 停止PWM输出
    DWORD    dwDuty;          // 占空比，取值范围1-1000，分辨率0.1%
    DWORD    dwPol;           // PWM极性 =1: PWM信号反向
    DWORD    dwNum;           // 输出PWM脉冲个数（暂不支持）
} PWM_INFO, *PPWM_INFO;
```

上述结构中，参数 dwFreq 表示输出的脉冲频率，单位为 Hz。DwFreq 的取值范围 1Hz – 25MHz，当 dwFreq = 0 时，表示 PWM 停止输出。dwDuty 的分辨率为 0.1%，取值范围 1-1000。

PWM 操作的主要代码如下：

```
//打开 ESM7000 PWM4
HANDLE hPWM4 = CreateFile( TEXT("PWM4:"),
    GENERIC_READ|GENERIC_WRITE,
    FILE_SHARE_READ|FILE_SHARE_WRITE,
    NULL,
    OPEN_EXISTING,
    FILE_FLAG_RANDOM_ACCESS,
    NULL);

// PWM参数设置
pwm_info.dwFreq = 100000; // 100KHz
pwm_info.dwDuty = 300;     // 30%, uint is 0.1%
pwm_info.dwPol = 0;
// 输出PWM信号
bRc = WriteFile( hPWM4, &pwm_info, sizeof( PWM_INFO), &dwBytes, NULL );

//关闭设备文件，停止 PWM 脉冲输出。

CloseHandle( hPWM4 );
```

完整参考代码：开发光盘\Software\Test_pwm

10、精简 ISA 总线

ESM7000 的 8-bit 数据宽度的精简 ISA 扩展总线，主要用于支持高速数据采集、多路串口扩展以及其他的高级扩展应用。精简 ISA 总线共有 12 条基本信号线，其定义如下：

ISA 总线信号	信号功能说明
ISA_ADV#	地址锁存，低电平有效，上升沿锁存。
ISA_CS#	片选信号，低电平有效，总线周期 190ns
ISA_RD#	读取 ISA 设备数据，低电平有效，脉冲宽度 110ns
ISA_WE#	写数据至 ISA 设备，低电平有效，脉冲宽度 110ns
ISA_D0 (LSB)	8-bit 双向数据总线，D0 为最低有效位，D7 为最高有效位。在 ISA_ADV#有效时，数据总线输出 8-bit 地址 A0 – A7，ISA 设备则利用 ISA_ADV#上升沿锁存地址；在读写脉冲有效时，数据总线分别为设备提供的数据或 ESM7000 输出的数据。
ISA_D1	
ISA_D2	
ISA_D3	
ISA_D4	
ISA_D5	
ISA_D6	
ISA_D7 (MSB)	
IRQ/DMA	可选功能。外部中断或 DMA 请求，上升沿有效。

当应用程序打开 ISA 驱动的设备文件时，管脚会自动切换至 ISA 总线的信号状态。ESM7000 的精简 ISA 总线周期为 190ns，支持最高 5MB/s 的数据传输速率；尽管只有 8-bit 数据宽度，但对应用程序的接口函数却支持一次读写 16-bit 的操作，底层硬件能自动分解成 2 个总线读写周期，ISA 扩展硬件可通过 A0 的电平来分辨低字节（A0 = 0）或高字节（A0 = 1）操作。ESM7000 的数据总线实际是分时复用的，在总线周期的前 70ns 输出 8-bit 地址，通过 ADV#上升沿锁存，而总线周期的后半部分才是实际的读或写操作。可见 ESM7000 的精简 ISA 总线具有 8-bit 的地址寻址能力，控制一般的 FIFO、AD、UART 等芯片都没有任何问题。

利用精简 ISA 总线的硬件中断功能和硬件 DMA 请求功能，可有效降低数据采集中对 AD 数据缓冲电路的复杂性，从而降低整个设备的成本。有关这方面的应用方法，将在专门的技术应用文章中详细说明，这里不作更多介绍。

10.1 Linux 平台下精简 ISA 总线的应用

在用户应用层，操作精简 ISA 总线的第一步，就是打开设备驱动文件：

```
#include "ESM7000_drivers.h"
int    fd;
```

```
fd = open("/dev/ESM7000_isa", O_RDWR);
```

在 Linux-3.12 环境，提供有 4 个基本的读写函数：

```
unsigned char isa_read(int fd, unsigned int offset);
```

字节读函数，fd 为设备文件句柄，offset 为地址偏移量，有效值 0x00 – 0Xff。返回值为从 ISA 扩展设备指定寄存器读取的数据值。

```
void isa_write(int fd, unsigned int offset, unsigned char val_b);
```

字节写函数，fd 为设备文件句柄，offset 为地址偏移量，有效值 0x00 – 0Xff。Val_b 为需要写入至 ISA 设备指定寄存器的数据值。

```
unsigned short int isa_read16(int fd, unsigned int offset);
```

双字节读函数，fd 为设备文件句柄，offset 为地址偏移量，有效值 0x00 – 0Xff，注意地址偏移量必须是偶数地址对齐的。返回值为从 ISA 扩展设备指定寄存器读取的数据值。

```
void isa_write16(int fd, unsigned int offset, unsigned short int val_w);
```

双字节写函数，fd 为设备文件句柄，offset 为地址偏移量，有效值 0x00 – 0Xff，注意地址偏移量必须是偶数地址对齐的。Val_w 为需要写入至 ISA 设备指定寄存器的数据值。

上述函数的具体实现，均可在精简 ISA 总线读写范例程序 test_isa.c 中查看其源码。

10.2 CE 平台下精简 ISA 总线的应用

WEC7 环境中，应用程序打开 ISA 驱动程序对应的设备文件“ISA1:”后，通过标准的读写 API 函数就可实现 ISA 总线的基本操作。典型代码包括：

打开 ISA 文件

```
HANDLE Hisa;
Hisa = CreateFile(_T("ISA1:"),           // name of device
    GENERIC_READ|GENERIC_WRITE,        // desired access
    FILE_SHARE_READ|FILE_SHARE_WRITE,  // sharing mode
    NULL,                               // security attributes (ignored)
    OPEN_EXISTING,                     // creation disposition
    FILE_FLAG_RANDOM_ACCESS,           // flags/attributes
    NULL);                              // template file (ignored)
```

ISA 总线的基本操作包括：单字节数据读写，双字节（WORD，16-bit）数据的读写，分别说明如下：

单字节读

```
BOOL Isa_ReadUchar(HANDLE Hisa, DWORD dwPortOffset, PBYTE pBuf)
{
    DWORD dwBufSize = sizeof(BYTE);
    DWORD dwNbBytesRead = 0;

    *pBuf = (BYTE)(dwPortOffset & 0xff);
    ReadFile(Hisa, pBuf, dwBufSize, &dwNbBytesRead, NULL);

    return TRUE;
}
```

在上面函数中，Hisa 为 CreateFile 返回的句柄，dwPortOffset 为需要读取数据的寄存器地址偏移量，范围 0 – 255；pBuf 用于保存读取的总线数据，注意指针类型为字节型。

单字节写

```
BOOL Isa_WriteUchar(HANDLE Hisa, DWORD dwPortOffset, BYTE ucValue)
{
    WORD wValue;
    DWORD dwBufSize = sizeof(WORD);
    DWORD dwNbBytesWritten = 0;

    wValue = (WORD)(dwPortOffset & 0xff);
    wValue = (wValue << 8) | ucValue;
    WriteFile(Hisa, &wValue, dwBufSize, &dwNbBytesWritten, NULL);
}
```

```
    return TRUE;
}
```

在上面函数中，Hisa 为 CreateFile 返回的句柄，dwPortOffset 为需要写入数据的寄存器地址偏移量，范围 0 – 255；ucValue 为写入总线寄存器的数据值。

双字节读

```
BOOL Isa_ReadWord(HANDLE Hisa, DWORD dwPortOffset, WORD* pBuf)
{
    DWORD dwBufSize = sizeof(WORD);
    DWORD dwNbBytesRead = 0;

    *pBuf = (WORD)(dwPortOffset & 0Xfe);
    ReadFile(Hisa, pBuf, dwBufSize, &dwNbBytesRead, NULL);

    return TRUE;
}
```

在上面函数中，Hisa 为 CreateFile 返回的句柄，dwPortOffset 为需要读取数据的寄存器地址偏移量，注意偏移量必须是偶数（A0 = 0）；pBuf 用于保存读取的总线数据，注意指针类型为 16-bit 的 WORD 类型。

双字节写

```
BOOL Isa_WriteWord(HANDLE Hisa, DWORD dwPortOffset, WORD wValue)
{
    DWORD dwValue;
    DWORD dwBufSize = sizeof(DWORD);
    DWORD dwNbBytesWritten = 0;

    dwValue = (dwPortOffset << 16) | wValue;
    WriteFile(Hisa, &dwValue, dwBufSize, &dwNbBytesWritten, NULL);

    return TRUE;
}
```

在上面函数中，Hisa 为 CreateFile 返回的句柄，dwPortOffset 为需要写入数据的寄存器地址偏移量，注意偏移量必须是偶数（A0 = 0）；wValue 为写入总线寄存器的数据值。

此外为了提高数据读写的效率，ISA 总线驱动还支持数据块的读写操作，其实现代码可在产品光盘中看到（isa_test1）。

完整参考代码：开发光盘\Software\Test_isa

11、CAN 总线接口

ESM7000 CAN 总线接口支持 CAN2.0B 协议，支持从 10KBit/s 到 1MBit/s 的位速率设置。

11.1 Linux 平台下 CAN 的应用

ESM7000 主板中 CAN 的通讯实现的是 Socket CAN 方式，Socket CAN 使用了 socket 接口和 Linux 网络协议栈，这种方法使得 CAN 设备驱动可以通过网络接口函数来调用。这样大大方便了熟悉 Linux 网络编程的程序员，由于调用的都是标准的 socket 函数，也使得应用程序便于移植，而不会因为硬件的调整而修改应用程序，这样加强了应用程序的可维护性。

使用 CAN 接口通讯，首先需要使用 IP 命令来配置 CAN0 接口：

```
// 关闭can0接口，以便进行配置
ifconfig can0 down
// 方法一：配置can0的波特率为250Kbps
ip link set can0 type can bitrate 250000
// 方法二：配置can0的波特率为250Kbps
ip link set can0 type can tp 250 prog-seg 5 phase-seg1 8 phase-seg2 2 sjw 2
// 启动can0接口
ifconfig can0 up
```

ESM7000 的 CAN 模块时钟选用的是 24MHz 的外部晶体振荡时钟。为了适应各种不同的采样率，我们采用方法二来对 can 的波特率进行设置，以 CiA 推荐的采样点在 bit 的 87.5%处，作为基准来计算：

波特率	PRES DIV -> fTq	TSEG1	TSEG2	TQ	采样点
		PROPSEG+PSEG1+2	PSEG2+1		
1000	1 -> 12MHz	(0 + 7 + 2) = 9	(1+1) = 2	12	83.3%
800	1 -> 12MHz	(3 + 7 + 2) = 12	(1+1) = 2	15	86.6%
500	2 -> 8MHz	(4 + 7 + 2) = 13	(1+1) = 2	16	87.5%
250	5 -> 4MHz	(4 + 7 + 2) = 13	(1+1) = 2	16	87.5%
125	11 -> 2MHz	(4 + 7 + 2) = 13	(1+1) = 2	16	87.5%
100	14 -> 1.6MHz	(4 + 7 + 2) = 13	(1+1) = 2	16	87.5%
60	24 -> 960KHz	(4 + 7 + 2) = 13	(1+1) = 2	16	87.5%
50	29 -> 800KHz	(4 + 7 + 2) = 13	(1+1) = 2	16	87.5%
20	74 -> 320KHz	(4 + 7 + 2) = 13	(1+1) = 2	16	87.5%
10	149 -> 160KHz	(4 + 7 + 2) = 13	(1+1) = 2	16	87.5%

就像 TCP/IP 协议一样，在使用 CAN 网络之前首先需要打开一个套接字。CAN 的套接字使用到了一个新

的协议族 PF_CAN，所以在调用 `socket()` 这个系统函数的时候需要将 PF_CAN 作为第一个参数。当前有两个 CAN 的协议可以选择，一个是原始套接字协议（raw socket protocol），另一个是广播管理协议 BCM（broadcast manager）。作为一般的工业应用我们选用原始套接字协议：

```
s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
printf("SOCK_RAW can sockfd:%d\n", s);
if(s < 0)
{
    return -1;
}
```

基本的 CAN 帧结构体和套接字地址结构体定义在 `include/linux/can.h` 中：

```
/*
 * 扩展格式识别符由 29 位组成。其格式包含两个部分：11 位基本 ID、18 位扩展 ID。
 * Controller Area Network Identifier structure
 *
 * bit 0-28 : CAN识别符 (11/29 bit)
 * bit 29 : 错误帧标志 (0 = data frame, 1 = error frame)
 * bit 30 : 远程发送请求标志 (1 = rtr frame)
 * bit 31 : 帧格式标志 (0 = standard 11 bit, 1 = extended 29 bit)
 */
typedef __u32 canid_t;
struct can_frame {
    canid_t can_id; /* 32 bit CAN_ID + EFF/RTR/ERR flags */
    __u8 can_dlc; /* 数据长度: 0 .. 8 */
    __u8 data[8] __attribute__((aligned(8)));
};
```

过滤规则（过滤器）的定义同样在 `include/linux/can.h` 中：

```
struct can_filter {
    canid_t can_id;
    canid_t can_mask;
};
```

过滤规则的匹配：

```
<received_can_id> & mask == can_id & mask
```

在成功创建一个套接字之后，通常需要使用 `bind()` 函数将套接字绑定在某个 CAN 接口上。在绑定 (CAN_RAW)套接字之后，就可以在套接字上使用 `read()/write()` 进行数据收发的操作。

如果不是用滤波器，可以直接设置并绑定套接字到我们刚才设置好的 CAN 接口上：


```

struct sockaddr_can addr;
struct ifreq      ifr;
int loopback = 0; /* 0 = disabled, 1 = enabled (default) */
setsockopt(s, SOL_CAN_RAW, CAN_RAW_LOOPBACK, &loopback, sizeof(loopback));

strcpy(ifr.ifr_name, "can0");
ret = ioctl(s, SIOCGIFINDEX, &ifr);
if( ret < 0 )
{
    return -1;
}

addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;

bind(s, (struct sockaddr *)&addr, sizeof(addr));

```

如果需要使用过滤器，采用原始套接字选项 `CAN_RAW_FILTER`，`CAN_RAW` 套接字的接收就可使用 `CAN_RAW_FILTER` 套接字选项指定的多个过滤规则（过滤器）来过滤。

过滤器能接收的数据要求满足 $\langle \text{received_can_id} \rangle \& \text{mask} == \text{can_id} \& \text{mask}$ ，也就是收数据的 `can_id` 和滤波器设定的 `can_id` 分别于滤波器的 `mask` 相与以后相等，才能够被接收，否则直接被硬件过滤掉。在下面的例程中，两组滤波器 $0x123 \& \text{CAN_SFF_MASK} = 0x123$ ， $0x200 \& 0x700 = 0x200$ ，所以当接收数据的 `can_id` 和滤波器的 `mask` 相与以后，需要等于 `0x123` 或者 `0x200`，也就是接收数据的 `can_id` 等于 `0x123` 或者 `0x200-0x2ff` 这个区间才能够被接收，否则直接被硬件过滤掉，如下面两个等式：

```

<received_can_id>& CAN_SFF_MASK==0x123 & CAN_SFF_MASK
<received_can_id>>=0x123
<received_can_id>& 0x700=0x200 & 0x700
<received_can_id>>=0x200-0x2ff

```

设置套接字，启动滤波器，并绑定 `CAN0` 接口：

```

struct sockaddr_can addr;
struct ifreq      ifr;
struct can_filter  filter[2]; //定义过滤器
filter[0].can_id  = 0x123;
filter[0].can_mask = CAN_SFF_MASK;
filter[1].can_id  = 0x200;
filter[1].can_mask = 0x700;

setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &filter, sizeof(filter)); //采用
原始套接字选项 CAN_RAW_FILTER

```

```
strcpy(ifr.ifr_name, "can0" );
ret = ioctl(s, SIOCGIFINDEX, &ifr);
if( ret < 0 )
{
    return -1;
}

addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;

bind(s, (struct sockaddr *)&addr, sizeof(addr));
```

发送数据的实现代码:

```
struct can_frame frame;
frame.can_id = 0x08 | CAN_EFF_FLAG; //定义为扩展帧
frame.can_dlc = 8; //数据长度
memset( frame.data, 0x32, frame.can_dlc );
nbytes = write(s, &frame, sizeof(struct can_frame)); //发送数据
if(nbytes!=sizeof(struct can_frame))
{
    perror("can raw socket write");
    return 1;
}
```

接收数据的实现代码:

```
struct can_frame frame;
nbytes = read(s, &frame, sizeof(struct can_frame)); //接收数据
if (nbytes < 0) {
    perror("can raw socket read");
    return 1;
}
if( nbytes < (int)sizeof(struct can_frame))
{
    fprintf(stderr, "read: incomplete CAN frame\n");
    return 1;
}
```

完整的代码请参考开发光盘中的: \应用开发软件\驱动模块测试\test_socketcan。

11.2 CE 平台下 CAN 的应用

ESM7000 CAN总线使用英创统一的标准API函数进行操作,操作方法和详细说明可参考:

[《英创WinCE主板CAN驱动使用说明》](#)

[《英创CAN驱动接口调用方法及说明》](#)

[《关于CAN驱动的错误帧定义》](#)

完整参考代码：开发光盘\Software\Test_Can

版本历史

手册版本	适用主板	简要描述	日期
V1.0	ESM7000 V2.1	ESM7000 工控主板技术参考手册	2017-1