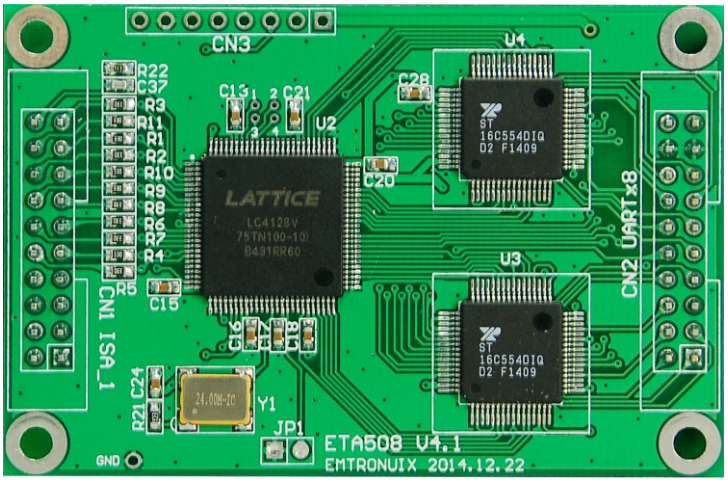
Emtronix



ETA508 串口扩展模块应用手册

感谢您购买英创信息技术有限公司的产品：**ETA508串口扩展模块。**

您可以访问英创公司网站或直接与英创公司联系以获得ETA508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701# 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

目　　录

1.ETA508 V1.0 简介.....3
2.硬件接口说明.....3
3.应用说明.....6
3.1 ETA508 在 WinCE 操作系统中的应用.....6
3.1.1 打开和关闭串口.....6
3.1.2 配置串口.....7
3.1.3.读写串口.....9
3.1.4．设置端口读写超时.....10
3.1.5．CCESerial 类.....11
3.2 ETA508 在 Linux 操作系统中的应用.....13
3.2.1. CSerial 类介绍.....14
3.2.2. CSerial 类的调用.....17

1.ETA508 V1.0简介

ETA508 是基于英创公司嵌入式工控主板所特有的精简 ISA 总线，扩展 4 个串口的扩展电路板。该模块可以通过 ISA 总线在英创公司的所有嵌入式主板（X86 系列及 ARM 系列）中使用。ETA508 多串口扩展单元由包括 2 片 16C554 和一片逻辑控制器组成，英创公司提供针对 ETA508 的驱动及应用程序范例。本文将介绍 ETA508 的使用、各个接口的信号定义等。在英创公司网站文章《串口通信应用方案》中，可以得到更多串口扩展应用的信息。

2.硬件接口说明

ETA508 的硬件设计使得用户既能快速方便的对它进行评估，又能很好的融入用户自己的产品设计中。用户对 ETA508 进行评估时，可通过带线与英创嵌入式主板的精简 ISA 总线相连，以方便进行功能评估。在用户自己做应用底板时，ETA508 可以作为一个“器件”背插在用户的应用底板上，以获得最佳的数据传输性能。我们提供 ETA508 protel 形式的器件 PCB 封装，以方便用户 Layout。图 1 是 ETA508 的外形图。

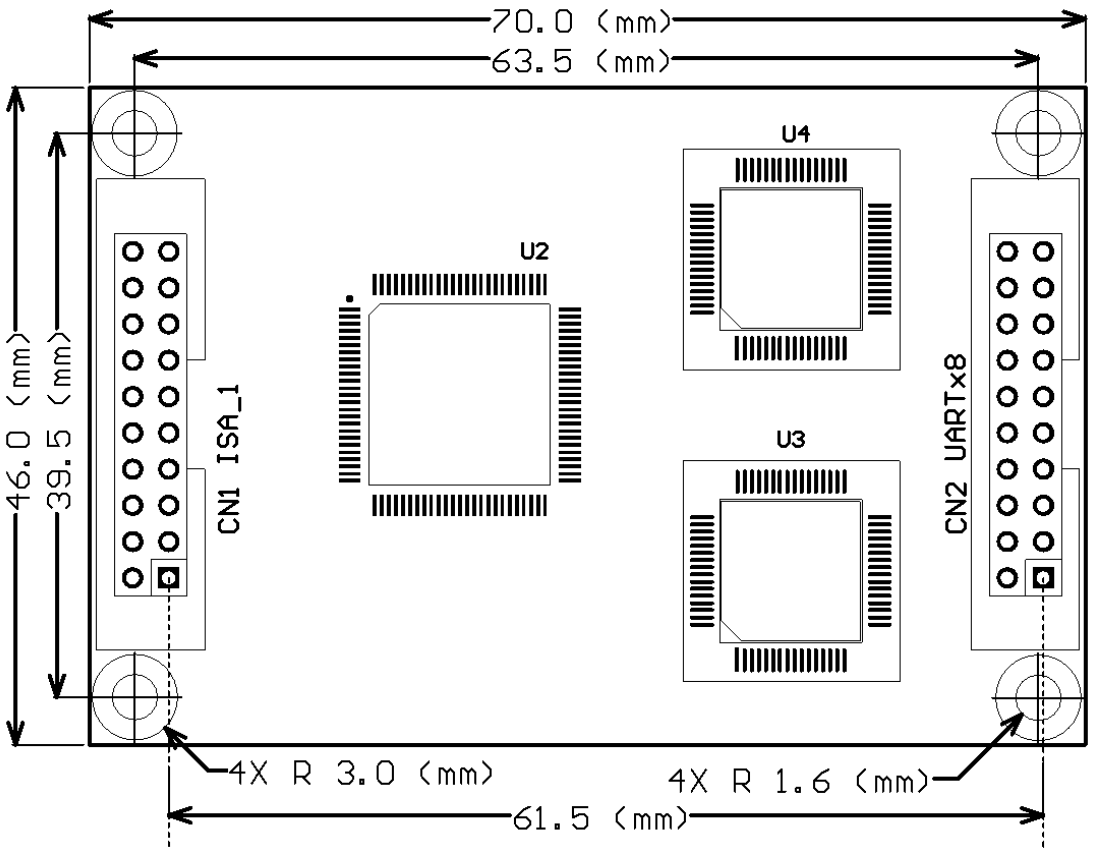


图 1：ETA508 示意图

（标注尺寸：inch（1 inch = 1000mils = 2.54cm））

CN1 为精简 ISA 总线接口，采用 20 芯 IDC 插针，可直接用带线与英创公司各嵌入式网络模块的评估底板相连接。在以下的信号定义表中，信号名称带#尾缀的，表示低电平有效。

CN1 具体信号定义如下：

信号名称及简要描述	CN1		信号名称及简要描述
	PIN#	PIN#	
SD0，数据总线 LSB	1	2	SD1，数据总线
SD2，数据总线	3	4	SD3，数据总线
SD4，数据总线	5	6	SD5，数据总线
SD6，数据总线	7	8	SD7，数据总线 MSB
SA0，地址总线	9	10	SA1，地址总线
SA2，地址总线	11	12	SA3，地址总线
SA4，地址总线	13	14	SA5，地址总线
SA6，地址总线	15	16	RD#，读信号
WE#，写信号	17	18	RESET#，外部复位输入
CS#，片选信号	19	20	IRQ，中断请求输出

对于 EM335x 系列的主板，由于主板管脚的限制，ISA 总线采用了地址数据复用的方式，所以 CN1 具体信号定义如下：

信号名称及简要描述	CN1		信号名称及简要描述
	PIN#	PIN#	
SD0，数据总线 LSB	1	2	SD1，数据总线
SD2，数据总线	3	4	SD3，数据总线
SD4，数据总线	5	6	SD5，数据总线
SD6，数据总线	7	8	SD7，数据总线 MSB
ALE，地址锁存信号	9	10	SD4，地址总线
SD5，地址总线	11	12	SD6，地址总线

SD7，地址总线	13	14	
	15	16	RD#，读信号
WE#，写信号	17	18	RESET#，外部复位输入
CS#，片选信号	19	20	IRQ，中断请求输出

CN2 是 8 路串口信号输出，LVTTL(3.3V)电平，采用 20 芯 IDC 插针。

信号名称及简要描述	CN2		信号名称及简要描述
	PIN#	PIN#	
TXD1，串口 1 输出	1	2	RXD1，串口 1 输入
TXD2，串口 2 输出	3	4	RXD2，串口 2 输入
TXD3，串口 3 输出	5	6	RXD3，串口 3 输入
TXD4，串口 4 输出	7	8	RXD4，串口 4 输入
TXD5，串口 5 输出	9	10	RXD5，串口 5 输入
TXD6，串口 6 输出	11	12	RXD6，串口 6 输入
TXD7，串口 7 输出	13	14	RXD7，串口 7 输入
TXD8，串口 8 输出	15	16	RXD8，串口 8 输入
GND	17	18	GND
VCC，+5V 电源输入	19	20	VCC，+5V 电源输入

从上表可知，ETA508 扩展的是 8 路 3 线制的串口，不支持硬件流控功能。

3.应用说明

ETA508 可以在英创的 WinCE 系统和 Linux 系统中使用。在 WinCE 系统和 Linux 系统下，操作 ETA508 扩展串口的方法与操作板上自带串口的方法一致，使用标准的流式文件操作接口函数，下面就来做详细的介绍。

需要注意，ETA508 不支持流控！

3.1 ETA508 在 WinCE 操作系统中的应用

ETA508 在 WinCE 平台中使用，当硬件配置好之后，客户需要进行一次软件配置，以让系统启动后知道扩展串口的具体配置。为此，我们设置了专门的内部命令 ETA508Set。客户可通过 Telnet 登录进主板，通过内部命令 ETA508Set 实现 ETA508 配置。运行 ETA508Set 实现的配置由命令参数决定如下：

命令	参数	实现配置
ETA508Set	0	禁止 ETA508 串口扩展
	1	扩展 8 串口

运行 ETA508Set 后，重启系统使设置生效。下面将介绍在 WinCE 下操作串口的 API 函数。这些函数的更详细信息，可以查阅微软的在线帮助。

3.1.1 打开和关闭串口

ETA508 驱动程序采用了标准的流式设备驱动结构，和所有流式设备驱动程序一样，ETA508 扩展的串口也使用 CreatFile 函数打开，需要注意的是，在串口号名之后必需加一个冒号（:），例如，下面的代码将调用 CreatFile 函数以读写的方式打开串口 8：

```
HANDLE hComm = CreateFile(
    _T(“COM8:”),
    GENERIC_READ | GENERIC_WRITE,        //允许读和写
    0，                                  //独占方式
    NULL,
    OPEN_EXISTING,                       //打开而不是创建
    0，
```

```
        NULL
        );
```

在此需要说明的是：在 CreateFile 函数的参数中，共享参数必需设置为 0，表示独占方式，安全参数必需设置为 NULL 值，模板文件参数也必需被设置成 NULL。由于在 CE 中不支持重叠 I/O 模式，因此不能在参数 dwFlagsAndAttributes 中传递 FILE_FLAG_OVERLAPPED。如果打开串口成功，将返回打开串口的句柄，否则将返回 INVALID_HANDLE_VALUE。需要注意的是，在打开串口号大于 9 的串口时，需要使用“\\$device\\COMxx”，而不是通常的“COMx:”。打开串口后，串口就已经被独占了，因此当我们不再使用打开的串口时，应及时关闭串口，此时可以 CloseHandle 函数关闭串口，例如，可以使用以下代码来关闭上面打开的串口：

```
BOOL bResult = CloseHandle(hComm);
```

3.1.2 配置串口

在实际使用串口用时，还必需配置好串口的波特率、奇偶校验和数据位等参数。CE 中提供了 GetCommState 和 SetCommState 函数，分别用于获取串口的当前参数和设置串口的参数，它们的定义如下：

```
BOOL GetCommState(
        HANDLE hFile,
        LPDCCB lpDCB );
BOOL SetCommState(
        HANDLE hFile,
        LPCDB lpDCB );
```

这两个函数都包含了相同的参数，其中参数 hFile 是输入参数，指向已打开的串口句柄；参数 lpDCB 指向 DCB 结构的指针，在 GetCommState 函数中，它属于输出参数，在 SetCommState 函数中，它属于输入参数。DBC 结构完全描述了串口的使用参数，其定如下：

```
typedef struct _DCB{
        DWORD DCBlength;              //DCB 结构大小
```

```
    DWORD BaudRate;             //波特率
    DWORD fBinary:1;            //二进制模式
    DWORD fParity:1             //进行奇偶较验
    DWORD fOutxCtsFlow:1;       //使 CTS 信号进行输出流量控制
    DWORD fOutxDSRfLOW:1;       //使 DSR 信号进行输入流量控制
    DWORD fDtrDsrFlow:1;        //DTR 流量控制
    DWORD fDsrSensitivity:1;    //DSR 敏感度
    DWORD fTXContinueOnXoff:1;//XOFF 后是否继续发送
    DWORD fOutX:1;              //使得输出 XON/XOFF 有效
    DWORD fInx:1                //使得输入 XON/XOFF 有效
    DWORD fErrorChar:1;         //允许奇偶错误替换
    DWORD fNull:1;              //允许删除 NULL
    DWORD fRtsControl:2;        //RTS 流量控制
    DWORD fAbortOnError:1       //出错时是否终止读写操作
    DWORD fDummy2:17;           //保留
    DWORD wReserved;            //当前未用，必须置为 0
    DWORD XonLim;               //XON 阀值
    DWORD XoffLim;              //XOFF 阀值
    BYTE   ByteSize;            //字符位数，4～8
    BYTE   Parity;          //奇偶校验位，0～4 分别为 no,odd,even, mark, space
    BYTE   StopBits;        //停止位，0，1，2 分别为 1，1.5，2
    Char   XonChar;         //XON 字符
    Char   XoffChar;        //XOFF 字符
    Char   ErrorChar;       //奇偶错误替换字符
    Char   EofChar;         //结束字符
    Char   EvtChar;         //事件字符
    WORD wReservedl;        //保留，未用
} DCB;
```

3.1.3.读写串口

正如使用 CreateFile 函数打开串口一样，可以使用 ReadFile 和 WriteFile 函数读取串口或向串口中写入。需要注意的是，由于从串口中读写数据的速度比较慢，因此最好的方法是用单独的线程来读写数据。虽然 CE 中不支持重叠 I/O 操作，但还是可以分别用单独的线程去地读写串口。同时 CE 还提供了 WaitCommEvent 函数，该函数将阻塞线程，直到预先设置的串口事件中的某一事件发生，在我们封装的 CCESerial 类中即是使用在一个线程中 WaitCommEvent 函数来等待数据接收。在使用串口事件之前，还需要了解如下三个函数：

BOOL GetCommMack(HANDLE hFile, LPDWORD lpEvtMadsk);

BOOL SetCommMask(HANDLE hFile, DWORD dwEvtMask);

BOOL WaitCommEvent(HANDLE hFile, LPDWORD lpEvtMask, LPOVERLAPPED lpoverlapped);

GetCommMask 函数用于得到串口已经设置了的串口事件，参数 hFile 指定已打开的串口句柄，参数 lpEvtMask 用于存取得到的串口事件集。

SetCommMask 函数的功能与 GetCommMask 函数相反，用于设置串口事件集。

WairCommEvent 函数用于等待预先设置的串口事件中的某一事件发生。参数 lpEvtMask 用于存储已经发生的事件;参数 lpOverlaped 必须设置为 NULL,因此在 CE 中不支持重叠结构。上面三个函数中的第二个参数，也就是串口事件集，它可以是下表中的某个值或其中几个值的组合。

EV_BREAK	检测到中断发生
EV_CTS	CTS 改变了状态
EV_DSR	DSR 信号改变了状态
EV_ERR	串口驱动程序检测到了错误
EV_RING	检测到振铃
EV_RLSD	RLSD 行改变了状态
EV_RXCHAR	接收到一个字符
EV_RXFLAG	接收到一个事件字符
EV_TXEMPTY	在输出缓冲区中的最后一个字符被发生

3.1.4．设置端口读写超时

在调用 ReadFile 和 WrireFile 函数从串口读取数据和写入数据时，WINCE 提供了超时机制，也就是设置了等待它们返回的时间长度。设置串口超时函数 SetCommTimeouts 的定义如下：

BOOL SetCommTimeouts(HANDLE hFile, LPCOMMTIMEOUTS lpCommTimeouts)l

参数 hFile 指向已经打开的串口句柄。参数 lpCommTimeouts 指向 COMMTIMEOUTS 结构，设置新的超时值。COMMTIMEOUTS 结构定义如下：

```
type struct _COMMTIMEOUTS{
 DWORD ReadIntervalTimeout;
DWORD ReadTotalTimeoutMultiplier;
DWORD ReadTotalTimeoutConstant;
DWORD WriteTotalTimeoutMultiploer;
DWORD WriteTotalTimeoutConstant;
}COMMTIMEOUTS, *LPCOMMTIMEOUTS;
```

读超时的计算方法有两种：一种超时是 ReadIntervalTimeout 指定了在接收字符间的最大时间间隔，如果超过了这个时间，ReadFile 函数立刻批回;另一种超时是基于要接收的字符数量，ReadTotalTimeoutMultiploer 表示平均读一字节的时间上限，ReadTotalTimeoutConstant 表示读数据总超时常量。

第二种读数据时可以用如下式子表示：读数据总超时=ReadTotalTimeoutConstant + (ReadTotalTimeoutMultiploer*要读的字节数)

写超时计算方法与读超时的第二种计算方法相同，WriteTotalTimeoutMultiploer 表示平均写一字节的时间上限，WriteTotalTimeoutConstant 表示写数据超时常量，总超时计算方法如下：写数据总超时= WriteTalTimeoutConstant + (WriteTotalTimeoutMultiploer*要写的字节数)

对于读数据超时，第一种超时（间隔超时）和第二种超时（总超时）同时有效，当出现任何一种超时，都将返回。下面介绍确切的超时设置：

- 有读间隔超时、读总超时和写总超时：将 COMMTIMEOUTS 结构中的五个成员设置相应值。
- 有读总超时和写总超时，但没有读间隔超时：将 ReadIntervalTimeout 设置为 0，将其它字段设置相应值。
- 不管是否有数据读取，ReadFile 立刻返回：将 ReadIntervalTimeout 设置成 MAX_DWORD,将 ReadTotalTimeoutMultiploer 和 ReadTotalTimeoutConstant 都设置成 0。
- ReadFile 没有超时设置，直到有适当的字符数返回或错误发生，该函数才返回：将 ReadIntervalTimeout,ReadTotalTimeoutMultiplier 和 ReadTotalTimeoutConstant 值都设置为 0.
- WriteFile 没有超时设置：将 WriteTotalTimeoutMultiploer 和 WriteTotalTimeoutConstant 都设置成 0 。

对于串口读写，以上所介绍的超时操作是至关重要的。用户可以根据实际情况考虑采用何种超时操作。如果串口读取和写入数据都采用超时，最好采用单独的统一线程负责读取和写入，以便不会阻塞主线程。

3.1.5．CCESerial 类

为了进一步简化用户对串口的操作，我们将串口操作的 API 函数和一系列的相关设置函数封装成一个 CCESerial 类，最后导出 3 个接口函数来完成对串口的操作。

（1）BOOL CCESerial:: OpenPort(UINT PortNo, UINT Baud, UCHAR Parity, UINT Databits, UINT Stopbits);

功能描述：打开指定串口并做相应配置

输入参数：

UINT PortNo 要打开的串口号（“COMx:”）,当串口号大于 9 时，使用“\\$device\\COMxx”

UINT Baud 串口波特率

UCHAR Parity 奇偶较验设置

UINT Databits 数据位

UINT Stopbits 停止位

返回值：

TRUE：串口打开成功

FALSE：串口打开失败

(2) DWORD CCESerial:: WritePort(char* Buf, int len);

功能描述：通过串口发送数据

输入参数：

char* Buf 发送数据缓存

int len 发送的字节数

返回值：

返回实际发送的字节数

(3) BOOL CCESerial:: ClosePort( );

功能描述：关闭打开的串口

在 CCESerial 类中，创建了单独的线程负责串口数据的接收，当通过 WaitCommEvent 等待串口事件集，如果是合法的串口数据，将调用回调函数对接收到的数据进行处理。串口数据接收线程如下：

```
DWORD WINAPI CCESerial::ReceiveThreadFunc(LPVOID lparam)
{
    CCESerial *lpSerial = (CCESerial*)lparam;
    DWORD       dwEvtMask, dwReadError;
    COMSTAT       cmStat;
    ULONG       nWillLen;

    SetCommMask( lpSerial->m_hSer, EV_RXCHAR|EV_ERR   );
    for( ; ; )
    {
        if( WaitCommEvent( lpSerial->m_hSer, &dwEvtMask, NULL ) )
        {
            SetCommMask( lpSerial->m_hSer, EV_RXCHAR|EV_ERR );
            // get how many data available in receive buffer
            if( dwEvtMask & EV_RXCHAR )
            {
                //取接收数据长度信息
                ClearCommError( lpSerial->m_hSer, &dwReadError, &cmStat );
                    nWillLen = cmStat.cbInQue;
                if( nWillLen <=0 )
                    continue;

                lpSerial->m_lDatLen = 0;
                ReadFile(       lpSerial->m_hSer,       lpSerial->DatBuf,       nWillLen,
&lpSerial->m_lDatLen, 0 );

                if( lpSerial->m_lDatLen>0 )
                {
                    // 调用回调函数处理接收到的数据
```

```
                    lpSerial->OnReceive( );
                }
            }
            else if( dwEvtMask & EV_ERR )
            {
                // 清错误标志
                ClearCommError( lpSerial->m_hSer, &dwReadError, &cmStat );
                lpSerial->OnError( );
            }

        }

        if(  WaitForSingleObject(  lpSerial->m_hKillRxThreadEvent,  0  )  ==
WAIT_OBJECT_0)
        {
            SetEvent( lpSerial->m_hReceiveCloseEvent );
            break;
        }
    }
    return 0;
}
```

3.2 ETA508 在 Linux 操作系统中的应用

ETA508 在 Linux 平台中使用，当硬件配置好之后，客户需要进行一次驱动加载，让系统识别出扩展的串口。驱动程序已经编译成 ko 文件，以模块的形式放在文件系统中，所以用户只需要执行 insmod 命令加载 ko 文件即可：

insmod /lib/modules/(Linux versions)/eta503_serial.ko

加载完成后，系统会将 503 扩展的 4 路串口识别为标准的串口，操作方法就和标准的串口相同了。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( );

3.2.1. CSerial 类介绍

其中各个函数的具体使用方法这里就不一一介绍了，用户可以参考 Linux 应用程序开发的相关书籍，也可参看 Step2_SerialTest 中 Serial.cpp 模块中 set_port( )函数代码。为了进一步简化用户对串口的操作，我们将串口操作的 API 函数和一系列的相关设置函数封装成一个 CSerial类，CSerial 类中提供了 4 个公共函数、一个串口数据接收线程以及数据接收用到的数据 Buffer。

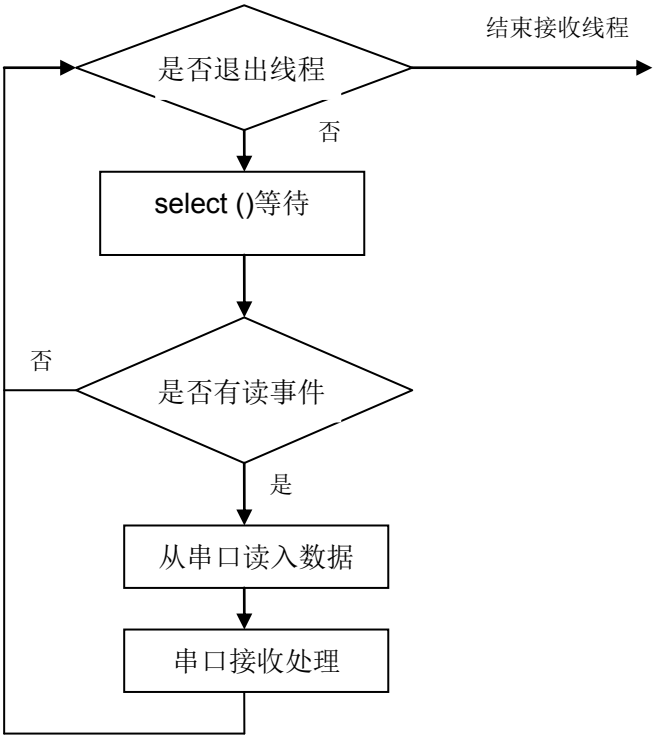
```
class CSerial
{
private:
    //通讯线程标识符ID
    pthread_t     m_thread;
    // 串口数据接收线程
    static int ReceiveThreadFunc( void* lparam );
public:
    CSerial();
    virtual ~CSerial();

    int       m_fd;                 // 已打开的串口文件描述符
    int       m_DatLen;
    char      DatBuf[1500];
    int       m_ExitThreadFlag;

    // 按照指定的串口参数打开串口，并创建串口接收线程
    int OpenPort( int PortNo, int baudrate, char databits, char stopbits, char parity );
    // 关闭串口并释放相关资源
    int ClosePort( );
    // 向串口写数据
    int WritePort( char* Buf, int len );
    // 接收串口数据处理函数
    virtual int PackagePro( char* Buf, int len );
};
```

OpenPort 函数用于根据输入串口参数打开串口，并创建串口数据接收线程。在 Linux 环境中是通过函数 pthread_create( )创建线程，通过函数 pthread_exit( )退出线程。Linux 线程属性存在有非分离（缺省）和分离两种，在非分离情况下，当一个线程结束时，它所占用的系统资源并没有被释放，也就是没有真正的终止；只有调用 pthread_join( )函数返回时，

创建的线程才能释放自己占有的资源。在分离属性下，一个线程结束时立即释放所占用的系统资源。基于这个原因，在我们提供的例程中通过相关函数将数据接收线程的属性设置为分离属性。如：

```
// 设置线程绑定属性
res = pthread_attr_setscope( &attr, PTHREAD_SCOPE_SYSTEM );
// 设置线程分离属性
res += pthread_attr_setdetachstate( &attr, THREAD_CREATE_DETACHED );
```

ReceiveThreadFunc 函数是串口数据接收和处理的主要核心代码，在该函数中调用 select( )，阻塞等待串口数据的到来。对于接收到的数据处理也是在该函数中实现，在本例程中处理为简单的数据回发，用户可结合实际的应用修改此处代码，修改 PackagePro( )函数即可。流程如下：



```
int CSerial::ReceiveThreadFunc(void* lparam)
{
    CSerial *pSer = (CSerial*)lparam;

    //定义读事件集合
    fd_set fdRead;
    int ret;
    struct timeval aTime;

    while( 1 )
    {
        //收到退出事件，结束线程
        if( pSer->m_ExitThreadFlag )
        {
            break;
        }
        FD_ZERO(&fdRead);
        FD_SET(pSer->m_fd,&fdRead);
        aTime.tv_sec = 0;
        aTime.tv_usec = 300000;
        ret = select( pSer->m_fd+1,&fdRead,NULL,NULL,&aTime );
        if (ret < 0 )
        {
            //关闭串口
            pSer->ClosePort( );
            break;
        }
        if (ret > 0)
        {
            //判断是否读事件
            if (FD_ISSET(pSer->m_fd,&fdRead))
            {
                //data available, so get it!
                pSer->m_DatLen = read( pSer->m_fd, pSer->DatBuf, 1500 );
                // 对接收的数据进行处理，这里为简单的数据回发
                if( pSer->m_DatLen > 0 )
                {
                    pSer->PackagePro( pSer->DatBuf, pSer->m_DatLen);
                }
                // 处理完毕
            }
        }
    }
    printf( "ReceiveThreadFunc finished\n");
```

```
        pthread_exit( NULL );
        return 0;
    }
```

需要注意的是，select( )函数中的时间参数在 Linux 下，每次都需要重新赋值，否则会自动归 0。

CSerial 类的实现代码请参见 Serial.CPP 文件。

3.2.2. CSerial 类的调用

CSerial 类的具体使用也比较简单，主要是对于类中定义的 4 个公共函数的调用，以下为 Step2_SerialTest.cpp 中相关代码。

```
class CSerial    m_Serial;
int main( int argc,char* argv[] )
{
   int      i1;
   int         portno,      baudRate;
   char       cmdline[256];

   printf( "Step2_SerialTest V1.0\n" );
   // 解析命令行参数：串口号    波特率
   if( argc > 1 )          strcpy( cmdline, argv[1] );
   else                      portno = 1;
   if( argc > 2 )
   {
        strcat( cmdline, " " );
        strcat( cmdline, argv[2] );
        scanf( cmdline, "%d %d", &portno, &baudRate );
   }
   else
   {
        baudRate = 115200;
   }
   printf( "port:%d baudrate:%d\n", portno, baudRate);
   //打开串口相应地启动了串口数据接收线程
   i1 = m_Serial.OpenPort( portno, baudRate, '8', '1', 'N');
   if( i1<0 )
   {
```

```
            printf( "serial open fail\n");
            return -1;
        }
        //进入主循环，这里每隔1s输出一个提示信息
        for( i1=0; i1<10000;i1++)
        {
            sleep(1);
            printf( "%d \n", i1+1);
        }
        m_Serial.ClosePort( );
        return 0;
    }
```

从上面的代码可以看出，程序的主循环只需要实现一些管理性的功能，在本例程中仅仅是每隔 1s 输出一个提示信息，在实际的应用中，可以把一些定时查询状态的操作、看门狗的喂狗等操作放在主循环中，这样充分利用了 Linux 多任务的编程优势，利用内核的任务调度机制，将各个应用功能模块化，以便于程序的设计和管理。这里顺便再提一下，在进行多个串口编程时，也可以利用本例程中的 CSerial 类为基类，根据应用需求派生多个 CSerial 派生类实例，每一个派生类只是重新实现虚函数 PackagePro(…)，这样每个串口都具有一个独立的串口数据处理线程，利用 Linux 内核的任务调度机制以实现多串口通讯功能。