第三部分　内部处理实例

应用程序并不仅仅等于用户界面。从读取、写入文件到定时事件、多任务，有大量的处理工作都在后台进行着。虽然 MFC被人们熟知为界面开发系统，但是在 MFC类库中同样具备一些类，它们可提供对应用程序中的非界面部分的支持。

这部分中的例子涉及大量应用程序内部的处理工作。这些例子包括程序内的多任务和发送、接收信息，还包括演奏声音和制作定时器。

第13章　消息和通信

这一章中的实例涉及应用程序内部和外部发送数据和消息。通常有两种方法处理消息队列中的消息。这一章中还给出了除使用窗口消息之外的其他三种与外部世界通信的方式，包括Windows socket(套接字)和低级串行I/O。

第14章　多任务

这一章中的实例涉及应用程序的处理过程，包括多任务、后台处理和其他应用程序的执行。

第15章　其他

最后的章节中收集了内部过程处理的一些其他实例。其中将介绍定时器、二进制字符串和VC++的宏指令等内容。

第13章　消息和通信

应用程序之间除了发送窗口消息外，还有一些与外部环境进行通信的其他方式，包括使用串行I/O方式以及在网络上使用套接字方式通信等等。对以上主题的深入讨论在第 1章和第3章中已涉及，而在本章中将给出这三种通信方式的实际应用的例子，这些例子包括：

实例47　等待消息，在本例中将看到如何使用消息处理函数在另一个消息到来之前暂停应用程序的运行。

实例48　清除消息，在本例中将看到如何清空应用程序的消息队列，以便使后续消息成为最新消息。

实例49　向其他应用程序发送消息，本例中将创建一条唯一的消息并把它发送给系统中的其他应用程序。

实例50　与其他应用程序共享数据，在本例中将看到如何向其他应用程序发送大量的数据。

实例51 使用套接字与应用程序通信，本例中将使用 Window套接字和其他Windows应用程序或任何支持套接字的应用程序 (例如 UNIX 应用程序)进行通信。

实例52 使用串行I/O，本例中将实现MFC应用程序与串行设备之间的通信。

13.1 实例47：等待消息

1. 目标

在一个函数中暂停应用程序，直到用户单击鼠标或者按键为止。

2. 策略

当应用程序看起来是空闲时，实际上它在运行函数： CWinApp::Run()。该函数不仅具有查询新消息的逻辑功能，并且完成了大量包括更新用户界面状态和清除临时内存对象在内的应用程序维护工作。因此，如果需要在应用程序中的某个地方停止运行并等待消息，也需要进行这种应用程序维护工作。由于微软提供了 CWinApp()::Run()函数的源代码，因此本例将创建该函数的一个新版本，可用于应用程序中的任何地方。

为什么要在除CWinApp函数以外的其他地方停顿呢？请参阅本实例结尾有关移植的说明，在此可发现该实例可能的用武之地。

3. 步骤

等待windows消息

找到需要应用程序暂停并等待特定消息的位置，并添加以下代码：

```
// wait till user clicks on status bar before proceeding
MSG msg;
BOOL bIdle = TRUE;
LONG lIdleCount = 0;
CWinApp* pApp = AfxGetApp();

AfxMessageBox( "Into wait loop." );
m_bWait = TRUE;
while ( m_bWait )
{
   // idle loop waiting for messages
   while ( bIdle && !::PeekMessage( &msg, NULL, NULL, NULL,
      PM_NOREMOVE) )
   {
      if ( !pApp -> OnIdle( lIdleCount++ ) ) bIdle = FALSE;
   }

   // process new messages
   do {
      // pump messages
      pApp -> PumpMessage();

      // if we're done, let's go...
      if ( !m_bWait )
         break;

      // otherwise keep looping
      if ( pApp -> IsIdleMessage( &msg ) )
```

```
            {
                bIdle = TRUE;
                lIdleCount = 0;
            }
        } while ( ::PeekMessage( &msg, NULL, NULL, NULL,
            PM_NOREMOVE ) );
    }
```

以上例程将在此等待处理消息和进行应用程序维护，直到 m_bWait变为TRUE为止。(如WM_LBUTTONDOWN消息)。

使用ClassWizard给正在等待的消息添加消息处理函数。在其中需设置m_bWait为TRUE。

4. 注意

■ 该等待例程不仅将消息转发到它们要发送的窗口，还进行后台处理工作。如果省略后台处理，会发现此时仍然可以改变应用程序窗口的尺寸并移动它们，但是这些窗口无法完全重新绘制，而且还在屏幕上留下无用的信息。同时工具栏和状态栏也将停止更新其状态，这是因为以上这些工作都是在后台处理时进行的。

■ 除了等待m_bWait变成TURE以外，还可以在等待过程中检查通过消息泵 (message pump)的消息。要注意的是：在此只能截取已寄送的 (posted)消息。使用SendMessage ()发送的消息将直接到达指定的窗口。而已寄送的消息通常只包括鼠标和键盘的行为消息 —— 其他任何消息都是使用发送方式。 (如果希望截取由 SengMessage()发送的消息，可试一试使用SetWindowsHookEx()及WH_CALLWNDPROC。)

■ 有关消息和消息泵的更多内容，请参阅第 1章。

5. 使用光盘时注意

在随书附带光盘上运行工程时，首先单击 Test，然后单击Wzd1菜单命令。注意应用程序将一直在OnWzd1Test()函数处暂停，直到单击Wzd2菜单命令才会退出该函数并继续运行。

6. 有关移植事项

该实例理想地实现了从DOS 应用程序到Windows的移植。Windows 应用程序是事件驱动的。函数仅在鼠标或者键盘按下时调用，当这些操作结束时，函数就释放控制权直到下一个事件发生。而DOS应用程序则占用诸如键盘和鼠标的系统资源，直到它们的功能完成后才释放这些资源。

DOS应用程序由用户启动之后，一个的典型功能是询问一系列问题，每一次都产生停顿并等待回答，在用户做出响应之前应用程序一直控制系统。为将这种功能移植到 Windows中，可以使用本例中的代码，允许系统在用户回答以前继续处理消息。然后在其他的消息处理函数中处理用户的回答，并将 m_bWait设置为Ture ，这样可以由应用程序继续下一次询问或者终止该函数。

13.2 实例48：清除消息

1. 目标

在继续运行之前,清除特定窗口中所有的鼠标单击和键盘敲击消息队列。

2. 策略

基于上一个实例编写一个小型消息泵 ,它可以处理当前应用程序消息队列中所有的消息

—— 除了被选定窗口的消息之外，该窗口的消息将被删除。

3. 步骤

为某一特定窗口在需要清除消息队列的地方，添加以下代码:

```
MSG msg;
CWinApp* pApp = AfxGetApp();
while ( ::PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
{
   // kill any mouse messages for this window
   if ( ( msg.hwnd != m_hWnd || ( msg.message < WM_MOUSEFIRST ||
      msg.message > WM_MOUSELAST ) ) && !pApp ->
      PreTranslateMessage( &msg ) )
   {
      ::TranslateMessage( &msg );
      ::DispatchMessage( &msg );
   }
}
```

在此将清除所有的鼠标消息。为了清除所有的键盘消息，还要对 WM_KEYFIRST 和WM_KEYLAST进行一次范围检查(range check)。为了清除已寄送到某一个窗口的消息，注意不要进行任意范围的检查 —— 只需检查这条消息是否到该窗口即可。为了清除已寄送到所有窗口的消息，只需删除while语句中的内容即可。

4. 注意

■ 当使用弹出菜单时，该实例特别有用。由于当用户在它的范围之外单击鼠标时弹出菜单将立即关闭，剩余的鼠标单击动作可能导致在弹出菜单尚未出现之前就关闭了弹出菜单。为了消除错误的单击，可在打开弹出菜单前添加本例中的这段程序。

■ 参阅第1章可以了解关于MFC消息发送更多的知识。

5. 使用光盘时注意

当运行随书附送的光盘时，注意在从视中弹出菜单之前，有关该视的鼠标消息从OnTestWzd ()的消息队列中被删除。

13.3 实例49：向其他应用程序发送消息

1. 目标

向其他应用程序发送消息。

2. 策略

本例将使用Windows API::SendMessage()函数向目标应用程序的某个窗口的句柄发送消息。其中的技巧在于获取该窗口的句柄。同时使用 Windows API ::RegisterWindowMessage()函数创建一个唯一的消息，并且两个应用程序相互都了解这条消息的含义。

本例中还将看到另一个Windows API函数：::BrodcastSystemMessage()函数，它可以向系统中的每个应用程序的主窗口发送消息。这样便可以避免出现获取另一个应用程序窗口句柄的问题。

3. 步骤

首先注册自己的窗口消息。相比较于前一个例子中使用的 WM_USER+1的技术，注册窗口消息的好处是不必费心考虑 WM_USER加上某个数之后，所表示的消息标识符是否超出工

程的允许范围。本例在两个工程中都使用文本字符串来注册消息。由于这个文本字符串在整个系统中应当是唯一的，因此将使用一种称为 GUID的COM技术来命名消息。GUID名字生成器程序可以在MFC的\BIN目录下找到，其可执行文件名为 GUIDGEN.EXE。该程序将生成在应用程序已知范围内认为是唯一的文本字符串，这对应用程序来说当然是最好不过的。

1) 注册一个唯一的窗口消息

使用GUIDGEN.EXE生成一个GUID。

在应用程序中把GUID定义为窗口消息文本字符串:

```
#define HELLO_MSG "{6047CCB1-E4E7-11 d 1 - 9 B 7 E - 0 0 A A 0 0 3 D 8 6 9 5 }"
```

使用::RegisterWindowsMessage()注册该窗口消息文本字符串:

```
idHelloMsg = ::RegisterWindowMessage( HELLO_MSG );
```

保存消息标识符idHelloMsg，便于以后使用。

2) 向其他应用程序发送消息

使用::RegisterWindowsMessage()返回的消息标识符发送消息，可使用以下代码:

```
::SendMessage(
   hWnd,                  // handle of a window belonging to destination app
   idHelloMsg,            // registered message id
   wParam,                // as usual
   lParam                 // as usual
   );
```

以上代码假定事先可以通过某种方式获取目标应用程序的某个窗口的句柄。一个指向CWnd类的指针不能在程序范围之外而发挥作用。但是可以在 CWnd类中封装已获取的窗口句柄，并如下所示来发送消息：

```
CWnd wnd;
wnd.Attach( hWnd );
wnd.SendMessage( idHelloMsg,wParam,lParam );
```

3) 接收已注册的窗口消息

为接收已注册的窗口消息，需要在接收窗口类，一般为 CMainFrame中手工添加 ON_REGISTERED_MESSAGE消息宏到消息映射中:

```
BEGIN_MESSAGE_MAP( CMainFrame, CMDIFrameWnd )
   // {{AFX_MSG_MAP( CMainFrame )
   // }}AFX_MSG_MAP
   ON_REGISTERED_MESSAGE( idHelloMsg,OnHelloMsg )
END_MESSAGE_MAP()
```

有关已注册消息的消息处理函数的代码如下：

```
LRESULT CMainFrame::OnHelloMsg( WPARAM wParam,LPARAM lParam )
{
   // process message
   return 0;
}
```

该实例到目前为止，一直假定事先可以通过某种方式取得目标应用程序的某个窗口的句柄。但这是一个困难的任务。简单的方法是向每个应用程序广播一条消息，并且希望目标程序正在监听。由于在系统中注册了一条唯一的消息，因此只有目标程序会响应这条消息。应

用程序广播的消息可能是它自己的窗口句柄，于是接收程序可以使用 ::SendMessage()来发送应答，也可能是用窗口句柄来结束循环。

4) 广播窗口消息

使用下面的代码广播窗口消息：

```
WPARAM wParam = xxx;              // your definition
LPARAM lParam = xxx;              // your definition
DWORD dwRecipients = BSM_APPLICATIONS;
::BroadcastSystemMessage( BSF_IGNORECURRENTTASK,&dwRecipients,
   idHelloMsg,                    // registered window message
   wParam,lParam );               // user defined parameters
```

4. 注意

■ ::BroadcastSystemMessage()函数提供了附加的标志 BSF_LPARAMPOINTER，可以将写入参数lParam的指针转化为可以被目标程序用来访问程序空间的指针，但是这个标志可能尚未进行文档标准化。

■ 进程间消息最广泛的使用是在外壳程序的子进程之间。一般情况下，外壳程序使用唯一的名字创建普通且不可见的窗口，子进程将通过这个窗口通信。此时子进程可以使用CWnd::FindWindow()获得使用该唯一窗口名的通信窗口的句柄，然后子窗口可以使用自己的窗口句柄向外壳程序发送消息I ’m here，如此完成连接。创建自己的普通窗口请参考实例 38。

5. 使用光盘时注意

运行两个工程中的实例，其中的一个处于调试模式，另一个处于发布模式。在一个应用程序中单击Test和Wzd菜单命令，注意到它与另一个应用程序交换信息。

13.4 实例50：与其他应用程序共享数据

1. 目标

与另一个应用程序共享数据。

2. 策略

使用::CreateFileMapping() Windows API建立一段交换文件(swap file)，与其他应用程序共享。

3. 步骤

1) 创建共享内存

为共享内存打开一段交换文件，可使用以下代码:

```
m_hMap = ::CreateFileMapping(
   ( HANDLE )0xffffffff,        // or can be an open file handle
   0,                           // security
   PAGE_READWRITE,              // or PAGE_READONLY or PAGE_WRITECOPY
   0,                           // size - - high order
                                //     (required if no file handle)
   0x1000,                      // size - - low order
                                //     (required if no file handle)
   MAP_ID                       // unique id - - required if no file handle
   );
```

如果希望通过自己的文件而不是交换文件来共享数据，可以提供一个已打开的文件句柄，

该文件句柄使用与在此使用的文件属性 (只读或者可写，等等)相同的文件属性。为创建唯一的MAP_ID，可使用VC++的\BIN目录中的GUIDGEN.EXE应用程序。

注意 使用MAP_ID第一次调用::CreateFileMapping() 实际上将生成一个映射文件。所有以后使用相同MAP_ID的调用将只返回现有文件的句柄。

使用以下的代码获得映射文件的内存指针:

```
m_pSharedData = ::MapViewOfFile(
   m_hMap,
   FILE_MAP_WRITE,          // or FILE_MAP_READ, FILE_MAP_COPY
                            //    (FILE_MAP_WRITE is read/write)
   0,                       // offset - - high order
   0,                       // offset - - low order
   0                        // number of bytes (zero maps entire file)
   );
```

2) 使用共享内存

现在可使用::MapViewOfFile()返回的指针进行数据存取:

```
// writing to shared memory
memcpy( ( LPBYTE )m_pSharedData,pWrite,10 );

// reading from shared memory
memcpy( pRead, ( LPBYTE )m_pSharedData,10 );
```

如果希望使用文件函数来访问数据，可以使用 CMwmFile类封装该内存指针，如下所示:

```
CMemFile file;
file.Attach( ( LPBYTE )m_pSharedData,size );
file.Write( pBuffer,100 );       // write 100 bytes to shared memory
```

3) 关闭共享内存

关闭共享内存时，首先取消视与它之间的映射关系，然后关闭句柄，如下所示。

```
::UnmapViewOfFile( m_pSharedData );
::CloseHandle( m_hMap );
```

4. 注意

■ 系统将同步所有对::CreateFileMapping()创建的共享内存的访问，这意味着不必担心应用程序A和应用程序B同时向相同的共享内存写入数据。但是当应用程序 A和应用程序B通过网络共享数据时，便可能发生这种问题。这是因为数据在传送到共享内存之前首先被写入本地缓存。在这种情况下，需要提供自己的文件同步。

■ 当在::MapViewOfFile()中使用非零的文件偏移量时，必须是系统内存分配单位值的倍数。为确定该单位值，可使用 ::GetSystemInfo()取得SYSTEM_INFO的结构。该单位值存放于结构的dwAllocationGranularity成员中。过去该值一般固定设置为 64K，将来可能会改变。

13.5 实例51：使用套接字与任意的应用程序通信

1. 目标

使用套接字与另一个 Windows应用程序或者与任何支持套接字的应用程序通信，例如与

UNIX应用程序通信。

2. 策略

本例将使用MFC的CSocket类在两个或者多个应用程序之间建立通信。通过套接字，服务器应用程序创建一个特殊的套接字，用于监听客户应用程序的连接请求，然后服务器应用程序创建新的套接字来完成连接。从客户和服务器两端读取该连接，直到一个需要处理的报文到来为止。请参阅第3章了解Windows套接字的详细内容。

本例将封装这些功能，这样所有应用程序需要完成的只是创建一个套接字连接，然后处理到来的报文。这将包括一个新的服务器 socket类、新客户端socket类和新的报文队列类。报文队列类允许使用同步方式向报文队列 (通常为COblist)中添加消息，这样在一个多任务环境中的两个消息不会产生冲突。之所以需要采取这种保护措施，是因为每个套接字将在它自己的线程中进行同步读操作:

3. 步骤

1) 创建新的服务器套接字类

使用ClassWizard创建一个从CSocket派生的新类。可以在新类名的某个位置加上“服务器(Server)”这个词。

首先在该类中创建一个名为 Open()的新函数，它将使用 CSocket的函数Creat()来创建套接字。本例将使用该函数而不是使用 CSocket::Creat()直接完成打开套接字 (类似于打开一个端口或文件)的操作:

```
BOOL CWzdServer::Open( UINT nPort )
{
   return Create( nPort );
}
```

现在目标是编写一组函数，用户监听意图连接的客户应用程序，并且通过创建一个新的套接字来建立连接。由于一些客户应用程序可以建立连接，因此需要在对象的映射表中跟踪这些新的套接字对象。一旦套接字建立，立刻开始从中读取内容。

在本例中，完成上述功能需要三个函数和一个结构。第一个函数 ListenEx(),用于通知套接字开始监听客户应用程序。第二个函数 OnAccept()在接收到连接请求时被调用。在其中创建新的套接字，并立刻设置它开始从客户应用程序读取报文。这些是通过调用第三个函数RecvThread()来完成的，该函数位于它自己的线程中。

下面首先编写函数ListenEx()。

添加ListenEx()函数，它通过调用 CSocket的Listen()函数监听来自客户应用程序的连接请求。ListenEx()同时在结构中设置其调用参数，这些参数最终被传递到 RecvThread()函数以实现读操作:

```
void CWzdServer::ListenEx( int hdrSz, int bodyPos, CWzdQueue *pQueue,
   CWnd *pWnd, UINT id )
{
   // initialize receive data
   m_RecvData.hdrSz = hdrSz;
   m_RecvData.bodyPos = bodyPos;
   m_RecvData.pQueue = pQueue;
   m_RecvData.pWnd = pWnd;
   m_id = id;                              // starting id
```

```
    // start listening
    Listen();
}
```

使用文本编辑器(Text Editor)重载CSocket的OnAccept()函数。在其中将创建新的套接字，以此来建立与客户应用程序的连接，同时使用由用户定义的标识符作为关键字将该套接字保存到对象映射表中。然后，设置套接字进入同步模式，并创建一个线程从套接字中读取数据：

```
void CWzdServer::OnAccept ( int nErrorCode )
{
    if ( nErrorCode == 0 )
    {
        // create a new socket and add to map
        CSocket *pSocket = new CSocket;
        m_mapSockets[m_id] = pSocket;

        // use this new socket to connect to client
        Accept( ( CAsyncSocket& )*pSocket );

        // put socket into synchronous mode
        DWORD arg = 0;
        pSocket -> AsyncSelect( 0 );
        pSocket -> IOCtl( FIONBIO, &arg );

        // setup this socket to listen for client messages
        m_RecvData.pSocket = pSocket;
        m_RecvData.id = m_id++;

        // start the thread
        AfxBeginThread( RecvThread,&m_RecvData );
    }
}
```

最后添加线程例程。RecvThread()使用CSocket的Receive()函数等待，直到通过套接字接收到新的报文。该线程假定每一个报文包含固定字长的报头和可变长度的报文体。对于每一个新的套接字报文，RecvThread()还向应用程序发送WM_NEW_MESSAGE消息，通知新的报文等待处理。如果套接字关闭，线程将在终止前向应用程序发送WM_DONE_MESSAGE消息:

```
UINT RecvThread( LPVOID pParam )
{
    // get data from thread creator
    RECVDATA *pRecv = ( RECVDATA * )pParam;

    int len = 1;
    int error = 0;
    char *pBody = NULL;
    char *pHdr = NULL;
    // while both sockets are open
    while (TRUE)
    {
        // read the header
```

```
        int res;
        pBody = NULL;
        pHdr = new char[pRecv -> hdrSz];
        if ( ( res = pRecv -> pSocket ->
            CAsyncSocket::Receive( pHdr, pRecv -> hdrSz ) )
            == SOCKET_ERROR )
            error = ::GetLastError();
        else
            len = res;

        // if closing down, exit thread
        if ( len == 0 || error == WSAECONNRESET ||
            error == WSAECONNABORTED ) break;

        // read the body???
        if ( !error && len && pRecv -> bodyPos != -1 )
        {
            int bodyLen = *( ( short * )pHdr+pRecv -> bodyPos );
            pBody = new char[bodyLen];
            if ( ( res = pRecv -> pSocket ->
                CAsyncSocket::Receive( pBody, bodyLen ) ) ==
                SOCKET_ERROR )
                error = ::GetLastError();
            else
                len+ = res;

            // if closing down, exit thread
            if ( len == 0 || error == WSAECONNRESET ||
                error == WSAECONNABORTED ) break;
        }

        // put message in queue
        pRecv -> pQueue ->
            Add( new CWzdMsg( pRecv -> id,pHdr,pBody,len,error ) );

        // post message to window to process this new message
        pRecv -> pWnd -> PostMessage( WM_NEW_MESSAGE );

    }

    // cleanup anything we started
    delete []pHdr;
    delete []pBody;

    // tell somebody we stopped
    pRecv -> pWnd -> SendMessage( WM_DONE_MESSAGE,
        ( WPARAM )pRecv -> id,( LPARAM )error );

    return 0;
}
```

接下来添加函数 SendEx()向客户应用程序发回报文，该函数将根据用户定义的标识符从

对象映射表中取出套接字对象，然后调用线程函数向该套接字发送报文:

```
void CWzdServer::SendEx( int id, LPSTR lpBuf, int len )
{
   // locate the socket for this id
   CSocket *pSocket = m_mapSockets[id];
   if ( pSocket )
   {
      m_SendData.pSocket = pSocket;
      m_SendData.lpBuf = lpBuf;
      m_SendData.len = len;

      // start the thread
      AfxBeginThread( SendThread,&m_SendData );
   }
}
```

SendThread使用CSocket类的Send()函数将报文数据发送出去:

```
UINT SendThread( LPVOID pParam )
{
   // get data from thread creator
   SENDDATA *pSend = ( SENDDATA * )pParam;

   // do the write
   pSend -> pSocket -> Send( pSend -> lpBuf, pSend -> len );

   return 0;
}
```

该类中的最后需要创建关闭函数，这个函数不仅将关闭监听套接字，而且将关闭创建的所有与客户端连接的套接字:

```
void CWzdServer::CloseEx()
{
   int id;
   CSocket *pSocket;
   for ( POSITION pos = m_mapSockets.GetStartPosition(); pos; )
   {
      m_mapSockets.GetNextAssoc( pos,id,pSocket );
      pSocket -> Close();
   }
   Close();

}
```

2) 创建客户套接字类

本例实际上将把该类添加到刚才创建服务器套接字类时所使用的同一个 .cpp和.h头文件中。因为在前面创建的两个线程函数 SendThread()和RecvThread()在客户套接字类中可以重用。

首先添加Open()函数。这个函数使用CSocket类的Open()和Connect()函数创建套接字并与服务器应用程序连接:

```
BOOL CWzdClient::Open( LPCTSTR lpszHostAddress, UINT nHostPort )
{
   if ( Create() && Connect( lpszHostAddress, nHostPort ) )
   {
      // put socket into synchronous mode
      DWORD arg = 0;
      AsyncSelect( 0 );
      IOCtl( FIONBIO, &arg );
      return TRUE;
   }
   return FALSE;
}
```

接下来添加一个发送函数。由于只有一个服务器，所以该发送函数不需要标识符。向发送数据结构中加入相应的数据，并调用 SendThread()函数完成实际的发送任务:

```
void CWzdClient::SendEx( LPSTR lpBuf, int len )
{
   // initialize the structure we will pass to thread
   m_SendData.pSocket = this;
   m_SendData.lpBuf = lpBuf;
   m_SendData.len = len;

   // start the thread
   AfxBeginThread( SendThread,&m_SendData );
}
```

注意 SendThread()函数与前面服务器套接字中使用的函数是同一个函数。

添加ListenEx()函数。这个函数将使客户套接字开始监听报文。与服务器套接字不同的是它不监听连接。ListenEx()实际上将调用RecvThread()来完成实际的监听任务:

```
void CWzdClient::ListenEx( int hdrSz, int bodyPos, CWzdQueue *pQueue,
   CWnd *pWnd, UINT id )
{
   // initialize receive data
   m_RecvData.pSocket = this;
   m_RecvData.hdrSz = hdrSz;
   m_RecvData.bodyPos = bodyPos;
   m_RecvData.pQueue = pQueue;
   m_RecvData.pWnd = pWnd;
   m_RecvData.id = id;

   // start the thread
   AfxBeginThread( RecvThread,&m_RecvData );
}
```

注意 RecvThread()函数与前面服务器套接字中使用的函数是同一个函数。

3) 创建报文队列类

报文队列类负责维护从一个或者多个套接字来的报文，直到它们被处理。报文队列类不仅包含用于容纳报文信息的通用报文类，而且使用同步过程防止线程取出报文时其他线程向

队列添加报文。

使用ClassWizard创建从CObList派生的新类。

在新类中嵌入CMutex类变量:

```
CMutex    m_mutex;
```

在新类中添加Add()函数，它使用CObList的AddTail()函数在队列中添加报文对象。使用m_mutex和CSingleLock类以同步对该函数的访问:

```
void CWzdQueue::Add( CWzdMsg *pMsg )
{
   CSingleLock slock( &m_mutex );

   if ( slock.Lock( 1000 ) )          // timeout in milliseconds,
                                      //     default = INFINITE
   {
      AddTail( pMsg );                //   fifo
   }
}
```

在新类中添加Remove()函数，它使用CObList的RemoveHead()函数从队列中清除报文对象。再次使用m_mutex和CSingleLock类保护对该函数的访问。

```
CWzdMsg *CWzdQueue::Remove()
{
   CSingleLock slock( &m_mutex) ;

   if ( slock.Lock( 1000 ) )    // timeout in milliseconds,
                                //     default = INFINITE
   {
      if ( !IsEmpty() )
         return ( CWzdMsg* )RemoveHead();
   }
   return NULL;
}
```

参考本实例结尾的程序清单 —— 报文队列类以查看浏览这个类的完整代码列表。

4) 使用新的服务器套接字类

首先，在应用程序类的InitInstance()函数调用AfxSocketInit()初始化套接字动态链接库。

在应用程序窗口类(例如CMainFrame)中嵌入新的套接字类和报文队列类对象:

```
CWzdServer    m_server;
CWzdQueue     m_queue;
```

然后打开套接字，开始监听:

```
if ( m_server.Open(
   1032                           // between 1025 and 0xffffffff set by you
                                  //     to identify this server to your other apps
   ) )
{
   m_server.ListenEx(
   10,                            // size of message header
   -1,                            // position of size of message body in header
                                  //     -1 means all message lengths are fixed
```

```
    &m_queue,                   // CWzdQueue to store new messages
    this,                       // pWnd of window to send messages
    32                          // starting id—as new connections are made
                                //     this number increases
    );
}
```

当套接字接收到新报文，使用 WM_NEW_MESSAGE消息通知用户。对该消息的处理可如下所示：

```
LRESULT CMainFrame::OnNewMessage( WPARAM,LPARAM )
{
    CWzdMsg *pMsg = NULL;
    while ( pMsg = m_queue.Remove() )
    {
        // pMsg contains:
        // m_nID    - - the user defined id of which port
        //                sent the message
        // m_pHdr   - - the message header
        // m_pBody  - - the message body
        // m_len    - - the total message length
        // m_error  - - any errors

        // make sure to delete the message after processing!
        delete pMsg;
    }
    return 0L;
}
```

如果连接终止，将接收到WM_DONE_MESSAGE消息:

```
LRESULT CMainFrame::OnDoneMessage( WPARAM id,LPARAM error )
{
    // gets here if a socket receive thread returns
    // id == id of client port that terminated
    // error == any error that caused the socket to close
    return 0L;
}
```

注意CWzdServer将跟踪已建立的新客户端连接，用户所要知道的是：当新的连接建立时，该连接将得到一个标识符号码，该标识符号码可从任何来自该连接的报文消息中返回，它可以设置为任意值，每建立一个新的连接，对应的标识符号码便递增一个值。在应用程序中可以使用一个列表来保存标识符号码和实际客户端连接的关联。当连接关闭时，可从列表中删去标识符号。

为发送报文到客户应用程序，使用：

```
m_server.SendEx(
    32,             // id of client port
    hello,          // buffer to send
    10              // length of buffer to send
    );
```

如果应用程序终止，还必须调用服务器套接字类的函数 CloseEx()。可在窗口类中添加

WM_CLOSE消息处理函数，并在其中调用该函数：

```
void CMainFrame::OnClose()
{
   m_server.CloseEx();

   CMDIFrameWnd::OnClose();
}
```

5) 使用新的客户套接字类

在应用程序类的InitInstance()函数中调用AfxSocketInit()初始化套接字动态链接库。

在应用程序窗口类(例如CMainFrame)中嵌入新的套接字类和报文队列类对象。

```
CWzdClient    m_client;
CWzdQueue     m_queue;
```

打开套接字，开始监听服务器应用程序:

```
if ( m_client.Open(
   "localhost",              // system address of server specified as:
                             //    "ftp.myhost.com" or "128.23.1.22" or
                             //    "localhost" for the same machine
   1032                      // the server's port number
   ) )
{
   m_client.ListenEx(
      10,                    // size of message header
      -1,                    // position of size of message body in header
                             //    -1 means all message lengths are fixed
      &m_queue,              // CWzdQue to store new messages
      this,                  // pWnd of window to send messages
      0                      // the user defined id of which client socket
                             //     sent the message
      );
}
```

虽然以上步骤允许应用程序监听来自服务器应用程序的报文，但是这些报文只是在服务器应用程序接收到客户发送报文后才响应返回。为此，需首先向服务器应用程序发送报文，使用下列代码：

```
m_client.SendEx(
   hello,       // buffer to send
   10           // length of buffer to send
   );
```

就像使用服务器应用程序一样，当新报文出现在队列中时，客户应用程序将接收到WM_NEW_MESSAGE消息，当连接终止时，则接收到WM_DONE_MESSAGE消息。可以按照类似于前面介绍的服务器应用程序消息处理方式来处理这些消息。

使用ClassWizard向需要关闭套接字的窗口类中添加WM_CLOSE消息处理函数:

```
void CMainFrame::OnClose()
{
   m_client.Close();
```

```
    CMDIFrameWnd::OnClose();
}
```

4. 注意

■ 受本实例篇幅所限，一些细节在上面的步骤中省略了，例如报文类的确切构成和传递给线程函数的数据结构的具体格式。在本例结尾的程序清单中可以看到这些细节代码实现。

■ 报文的头和主体被当作两个实体来处理，所以报文可以是变长度的，在本例中，报文体的长度以指定的固定偏移量保存在报头中，该偏移值在调用 ListenEx()函数时指定。它是一个16位的变量。对不同的大小，只需在 RecvThread()函数中改变重载的短整型 (short*)的类型即可。如果报文为固定长度，则可以删去本例中对报文体的处理代码，或者在指定报文长度变量的位置时使用-1。

■ 本例使用套接字的同步模式，意味着在读取或写入完整的报文之前，读写操作不会返回。如果应用程序完成其他的事情，例如响应用户输入，这样显然会出现问题。因此需要通过各自的线程完成读写操作。套接字虽然可以实现同步读写，但实现十分复杂，而且不会比使用线程的方式完成更多的功能。

■ 套接字允许在应用程序之间使用串行化 (Serialization)通信，因此可以容易地传输可变长度的类对象，而不是具有报文头和报文体的数据结构。串行化考虑了 Intel 和Motorola计算机之间的大小终端，但是通信双方的应用程序都必须使用 MFC创建(例如：在Windows主机和苹果机之间)。为使用串行化来实现套接字，需要使用下面的代码重写 SendThread()和RecvThread()函数:

```
// create a socket file class object
CSocketFile file(
    &sock                   // either the client or server socket class
    );
// construct an archive
CArchive ar(
    &file,                  // the file from above
    CArchive::load          // for RecvThread()
    // CArchive:: store     // for SendThread()
    );
// for RecvThread() to read a message
ar >> object;
// for SendThread() to write a message
ar << object;
```

通过使用多态性，甚至可以发送可变长度的报文。有关套接字的更多知识，请参阅第 3 章。

5. 使用光盘时注意

运行两个工程实例，一个处于调试模式，另一个处于发布模式。单击每个程序上的 Test菜单。可以指定任一个应用程序作为服务器应用程序或者是客户应用程序。程序中还有用于在两个应用程序之间发送报文的菜单命令，可以在 CMainFrame中中断并监视这些菜单命令。

6. 程序清单 —— 报文队列类

```
// WzdQue.h: interface for the CWzdQueue class.
//
//////////////////////////////////////////////////////////////////////
```

```
#if !defined( AFX_QUEUE_H__81CE0F22_8C16_11D2_A18D_99620BDF6820__INCLUDED_ )
#define AFX_QUEUE_H__81CE0F22_8C16_11D2_A18D_99620BDF6820__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif    // _MSC_VER > 1000

#include <afxmt.h>

class CWzdMsg : public CObject
{
public:
    CWzdMsg( int id,LPSTR pHdr,LPSTR pBody,int len,int error );
    virtual ~CWzdMsg();

    int m_nID;
    LPSTR m_pHdr;
    LPSTR m_pBody;
    int m_len;
    int m_error;
};

class CWzdQueue : public CObList
{
public:
    void Add( CWzdMsg *pMsg );
    CWzdMsg *Remove();

private:
    CMutex m_mutex;
};

#endif
    // !defined( AFX_QUEUE_H__81CE0F22_8C16_11D2_A18D_99620BDF6820__INCLUDED_ )
// WzdQue.cpp: implementation of the CWzdQueue class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "WzdQue.h"

//////////////////////////////////////////////////////////////////////
// Construct Message
//////////////////////////////////////////////////////////////////////

CWzdMsg::CWzdMsg( int id,LPSTR pHdr,LPSTR pBody,int len,int error )
{
    m_nID = id;
    m_pHdr = pHdr;
    m_pBody = pBody;
```

```
    m_len = len;
    m_error = error;
}


CWzdMsg::~CWzdMsg()
{
    delete []m_pHdr;
    delete []m_pBody;
}


//////////////////////////////////////////////////////////////////////
// Add Message
//////////////////////////////////////////////////////////////////////

void CWzdQueue::Add( CWzdMsg *pMsg )
{
    CSingleLock slock( &m_mutex );

    if ( slock.Lock( 1000 ) )    // timeout in milliseconds, default = INFINITE
    {
        AddTail(pMsg);          // fifo
    }
}

//////////////////////////////////////////////////////////////////////
// Remove Message
//////////////////////////////////////////////////////////////////////

CWzdMsg *CWzdQueue::Remove()
{
    CSingleLock slock( &m_mutex );

    if ( slock.Lock( 1000 ) )    // timeout in milliseconds, default = INFINITE
    {
        if ( !IsEmpty() )
            return ( CWzdMsg* )RemoveHead();
    }
    return NULL;
}
```

7. 程序清单 —— 套接字类

```
// WzdSock.h: interface for the CWzdServer and CWzdClient class.
//
//////////////////////////////////////////////////////////////////////

#if !defined( AFX_WZDSOCK_H__81CE0F20_8C16_11D2_A18D_99620BDF6820__INCLUDED_ )
#define AFX_WZDSOCK_H__81CE0F20_8C16_11D2_A18D_99620BDF6820__INCLUDED_

#if _MSC_VER > 1000
#pragma once
```

```
#endif    // _MSC_VER > 1000

#include <afxsock.h>
#include <afxtempl.h>
#include "WzdQue.h"

#define WM_NEW_MESSAGE WM_USER+1
#define WM_DONE_MESSAGE WM_USER+2

/////////////////////////////////////////////////////////////////////////
// Thread data
/////////////////////////////////////////////////////////////////////////
typedef struct t_SENDDATA
{
    CSocket *pSocket;
    LPSTR lpBuf;
    int len;
} SENDDATA;

typedef CMap<int,int,CSocket *,CSocket *> SOCKMAP;
typedef struct t_RECVDATA
{
    CSocket *pSocket;
    int hdrSz;
    int bodyPos;
    CWzdQueue *pQueue;
    CWnd *pWnd;
    UINT id;
} RECVDATA;

/////////////////////////////////////////////////////////////////////////
// CWzdServer
/////////////////////////////////////////////////////////////////////////


class CWzdServer : public CSocket
{
public:
    CWzdServer(){};
    virtual ~CWzdServer();

    BOOL Open( UINT nPort );
    void CloseEx();
    void SendEx( int id, LPSTR lpBuf, int len );
    void ListenEx( int hdrSz, int bodyPos, CWzdQueue *pQueue, CWnd *pWnd,
        UINT id );

// Overrides
    virtual void OnAccept ( int nErrorCode );

private:
    int m_id;
```

```
    SENDDATA m_SendData;
    RECVDATA m_RecvData;
    SOCKMAP m_mapSockets;
};

//////////////////////////////////////////////////////////////////////
// CWzdClient
//////////////////////////////////////////////////////////////////////

class CWzdClient : public CSocket
{
public:
    CWzdClient() {};
    virtual ~CWzdClient() {};

    BOOL Open( LPCTSTR lpszHostAddress, UINT nHostPort );
    void SendEx( LPSTR lpBuf, int len );
    void ListenEx( int hdrSz, int bodyPos, CWzdQueue *pQueue, CWnd *pWnd,
        UINT id );
private:
    SENDDATA m_SendData;
    RECVDATA m_RecvData;
};

//////////////////////////////////////////////////////////////////////
// Threads
//////////////////////////////////////////////////////////////////////
UINT SendThread( LPVOID pParam );
UINT RecvThread( LPVOID pParam );

#endif
    // !defined( AFX_WZDSOCK_H__81CE0F20_8C16_11D2_A18D_99620BDF6820__INCLUDED_ )
// WzdSock.cpp: implementation of the CWzdServer and CWzdClient classes.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "WzdSock.h"

//////////////////////////////////////////////////////////////////////
// CWzdServer
//////////////////////////////////////////////////////////////////////

////////////////////////
// Cleanup
////////////////////////

CWzdServer::~CWzdServer()
{
    // cleanup all created sockets
    int id;
    CSocket *pSocket;
```

```
    for ( POSITION pos = m_mapSockets.GetStartPosition(); pos; )
    {
        m_mapSockets.GetNextAssoc( pos,id,pSocket );
        delete pSocket;
    }
}

//////////////////////////
// Open Socket
//////////////////////////

BOOL CWzdServer::Open( UINT nPort )
{
    return Create( nPort );
}

//////////////////////////
// Send to Socket
//////////////////////////

void CWzdServer::SendEx( int id, LPSTR lpBuf, int len )
{
    // locate the socket for this id
    CSocket *pSocket = m_mapSockets[id];
    if ( pSocket )
    {
        m_SendData.pSocket = pSocket;
        m_SendData.lpBuf = lpBuf;
        m_SendData.len = len;

        // start the thread
        AfxBeginThread(SendThread,&m_SendData);
    }
}

//////////////////////////
// Listen to Socket
//////////////////////////

void CWzdServer::ListenEx( int hdrSz, int bodyPos, CWzdQueue *pQueue,
    CWnd *pWnd, UINT id )
{
    // initialize receive data
    m_RecvData.hdrSz = hdrSz;
    m_RecvData.bodyPos = bodyPos;
    m_RecvData.pQueue = pQueue;
    m_RecvData.pWnd = pWnd;
    m_id = id;                          // starting id

    // start listening
    Listen();
}
```

```
// Listen() calls OnAccept() when a new client is attempting to connect
void CWzdServer::OnAccept ( int nErrorCode )
{
   if ( nErrorCode == 0 )
   {
      // create a new socket and add to map
      CSocket *pSocket = new CSocket;
      m_mapSockets[m_id] = pSocket;

      // use this new socket to connect to client
      Accept( ( CAsyncSocket& )*pSocket );

      // put socket into synchronous mode
      DWORD arg = 0;
      pSocket -> AsyncSelect( 0 );
      pSocket -> IOCtl( FIONBIO, &arg );

      // setup this socket to listen for client messages
      m_RecvData.pSocket = pSocket;
      m_RecvData.id = m_id++;

      // start the thread
      AfxBeginThread( RecvThread,&m_RecvData );
   }
}

//////////////////////////
// Close Sockets
//////////////////////////

void CWzdServer::CloseEx()
{
   int id;
   CSocket *pSocket;
   for ( POSITION pos = m_mapSockets.GetStartPosition(); pos; )
   {
      m_mapSockets.GetNextAssoc( pos,id,pSocket );
      pSocket -> Close();
   }
   Close();

}

////////////////////////////////////////////////////////////////////////////
// CWzdClient
////////////////////////////////////////////////////////////////////////////

//////////////////////////
// Open Socket
//////////////////////////
```

```
BOOL CWzdClient::Open( LPCTSTR lpszHostAddress, UINT nHostPort )
{
    if ( Create() && Connect( lpszHostAddress, nHostPort ) )
    {
        // put socket into synchronous mode
        DWORD arg = 0;
        AsyncSelect( 0 );
        IOCtl( FIONBIO, &arg );
        return TRUE;
    }
    return FALSE;
}

//////////////////////////
// Send to Socket
//////////////////////////

void CWzdClient::SendEx( LPSTR lpBuf, int len )
{
    // initialize the structure we will pass to thread
    m_SendData.pSocket = this;
    m_SendData.lpBuf = lpBuf;
    m_SendData.len = len;

    // start the thread
    AfxBeginThread( SendThread,&m_SendData );
}


//////////////////////////
// Listen to Socket
//////////////////////////

void CWzdClient::ListenEx( int hdrSz, int bodyPos, CWzdQueue *pQueue,
    CWnd *pWnd, UINT id )
{
    // initialize receive data
    m_RecvData.pSocket = this;
    m_RecvData.hdrSz = hdrSz;
    m_RecvData.bodyPos = bodyPos;
    m_RecvData.pQueue = pQueue;
    m_RecvData.pWnd = pWnd;
    m_RecvData.id = id;

    // start the thread
    AfxBeginThread( RecvThread,&m_RecvData );
}


////////////////////////////////////////////////////////////////////////
// Threads
```

```
//////////////////////////////////////////////////////////////////////

UINT SendThread( LPVOID pParam )
{
   // get data from thread creator
   SENDDATA *pSend = ( SENDDATA * )pParam;

   // do the write
   pSend -> pSocket -> Send( pSend -> lpBuf, pSend -> len );

   return 0;
}


UINT RecvThread( LPVOID pParam )
{
   // get data from thread creator
   RECVDATA *pRecv = ( RECVDATA * )pParam;

   int len = 1;
   int error = 0;
   char *pBody = NULL;
   char *pHdr = NULL;
   // while both sockets are open
   while ( TRUE )
   {
      // read the header
      int res;
      pBody = NULL;
      pHdr = new char[pRecv -> hdrSz];
      if ( ( res = pRecv -> pSocket ->
         CAsyncSocket::Receive( pHdr, pRecv -> hdrSz)) == SOCKET_ERROR )
         error = ::GetLastError();
      else
         len = res;

      // if closing down, exit thread
      if ( len == 0 || error == WSAECONNRESET || error == WSAECONNABORTED )
         break;

      // read the body???
      if ( !error && len && pRecv -> bodyPos != -1 )
      {
         int bodyLen = *( ( short * )pHdr+pRecv -> bodyPos );
         pBody = new char[bodyLen];
         if ( ( res = pRecv -> pSocket ->
            CAsyncSocket::Receive( pBody, bodyLen ) ) == SOCKET_ERROR )
            error = ::GetLastError();
         else
            len += res;

         // if closing down, exit thread
```

```
            if (len == 0 || error == WSAECONNRESET || error == WSAECONNABORTED)
                break;
        }

        // put message in queue
        pRecv -> pQueue ->
            Add(new CWzdMsg( pRecv -> id,pHdr,pBody,len,error ) );

        // post message to window to process this new message
        pRecv -> pWnd -> PostMessage( WM_NEW_MESSAGE );

    }

    // cleanup anything we started
    delete []pHdr;
    delete []pBody;

    // tell somebody we stopped
    pRecv -> pWnd ->
        SendMessage( WM_DONE_MESSAGE,( WPARAM )pRecv -> id,( LPARAM )error );

    return 0;
}
// WzdQue.h: interface for the CWzdQueue class.
//
//////////////////////////////////////////////////////////////////////

#if !defined( AFX_QUEUE_H__81CE0F22_8C16_11D2_A18D_99620BDF6820__INCLUDED_ )
#define AFX_QUEUE_H__81CE0F22_8C16_11D2_A18D_99620BDF6820__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif    // _MSC_VER > 1000

#include <afxmt.h>

class CWzdMsg : public CObject
{
public:
    CWzdMsg( int id,LPSTR pHdr,LPSTR pBody,int len,int error );
    virtual ~CWzdMsg();

    int m_nID;
    LPSTR m_pHdr;
    LPSTR m_pBody;
    int m_len;
    int m_error;
};

class CWzdQueue : public CObList
{
public:
    void Add( CWzdMsg *pMsg );
    CWzdMsg *Remove();
```

```
private:
    CMutex m_mutex;
};

#endif
    // !defined( AFX_QUEUE_H__81CE0F22_8C16_11D2_A18D_99620BDF6820__INCLUDED_ )
// WzdQue.cpp: implementation of the CWzdQueue class.
//
//////////////////////////////////////////////////////////////////////

#include “stdafx.h”
#include “WzdQue.h”

//////////////////////////////////////////////////////////////////////
// Construct Message
//////////////////////////////////////////////////////////////////////

CWzdMsg::CWzdMsg( int id,LPSTR pHdr,LPSTR pBody,int len,int error )
{
    m_nID = id;
    m_pHdr = pHdr;
    m_pBody = pBody;
    m_len = len;
    m_error = error;
}


CWzdMsg::~CWzdMsg()
{
    delete []m_pHdr;
    delete []m_pBody;
}


//////////////////////////////////////////////////////////////////////
// Add Message
//////////////////////////////////////////////////////////////////////

void CWzdQueue::Add( CWzdMsg *pMsg )
{
    CSingleLock slock( &m_mutex );

    if ( slock.Lock( 1000 ) )    // timeout in milliseconds, default = INFINITE
    {
        AddTail(pMsg);          // fifo
    }
}

//////////////////////////////////////////////////////////////////////
// Remove Message
//////////////////////////////////////////////////////////////////////
```

```
CWzdMsg *CWzdQueue::Remove()
{
   CSingleLock slock( &m_mutex );

   if ( slock.Lock( 1000 ) )    // timeout in milliseconds, default = INFINITE
   {
      if ( !IsEmpty() )
         return (CWzdMsg*)RemoveHead();
   }
   return NULL;
}
```

13.6 实例52：使用串行或并行I/O

1. 目标

在MFC的应用程序中支持存取串行或者并行设备。

2. 策略

与串行或并行设备通信实际上和与磁盘文件交互没有什么区别。在本例中将使用 MFC的CFile类通过指定COM1: 或者LPT2:来实现与这些设备的通信。将读写操作转换为与串行或者并行设备的实际转换过程由硬件驱动程序完成。本例在此方式上实现了更进一步的功能，即允许应用程序同时与几个串行或者并行设备通信，其中使用了多任务以避免某一设备过于繁忙而堵塞。本例中实际上借用了在前面套接字实例中创建的报文队列类和一些术语，以实现将接收到的报文引导到一个中央消息处理程序。与前面例子不同的是，串行和并行通信是如此相似以至于可以将这些功能封装在一个 CFile的派生类中。

3. 步骤

1) 创建新的端口I/O类

使用ClassWizard创建从CFile派生的新类。

使用文本编辑器在新类中添加 OpenLPT()函数，用于打开并行设备。这里仅使用 CFile::ModeReadWrite模式调用Open()函数:

```
BOOL CWzdPortIO::OpenLPT( int n,CFileException *e )
{
   CString portName;
   portName.Format( "LPT%d:", n );
   return Open( portName, CFile::modeReadWrite, e );
}
```

创建OpenCOM()函数，用于打开串行设备。实际上，串行设备也必须使用::SetCommState ()设置波特率和停止位，如下所示:

```
BOOL CWzdPortIO::OpenCOM( int n, CFileException *e, int baud,
   int parity, int databits, int stopbits )
{
   CString portName;
   portName.Format( "COM%d:", n );
   if ( Open( portName, CFile::modeReadWrite, e ) )
   {
      DCB dcb;
```

```
        ::GetCommState( ( HANDLE )m_hFile, &dcb );
        if ( baud != -1 )     dcb.BaudRate = baud;
        if ( databits != -1 ) dcb.ByteSize = databits;
        if ( stopbits != -1 ) dcb.StopBits = stopbits;
        if ( parity != -1 )   dcb.Parity = parity;
        ::SetCommState( ( HANDLE )m_hFile, &dcb );
        return( TRUE );
    }
    return( FALSE );
}
```

为向端口设备发送报文。只需使用 CFile的Write()函数，但是在线程中而不是在应用程序中调用它，这样可以避免应用程序陷入困境:

```
void CWzdPortIO::Send( LPSTR lpBuf, int len )
{
    // initialize the structure we will pass to thread
    m_SendData.pFile = this;
    m_SendData.lpBuf = lpBuf;
    m_SendData.len = len;

    // start the thread
    AfxBeginThread( SendThread,&m_SendData );
}
```

如前面所述，发送线程只使用 CFile的Write()函数:

```
UINT SendThread( LPVOID pParam )
{
    // get data from thread creator
    SENDDATA *pSend = ( SENDDATA * )pParam;

    // do the write
    pSend -> pFile -> Write( pSend -> lpBuf, pSend -> len );

    return 0;
}
```

为从端口读取数据，在此采用了类似 Windows套接字中监听的概念，创建一个叫做Listen()的函数，它在线程中使用一个 CFile的Read()函数。该函数在接收到字节之前一直进行读取，当接收到字节时，一个报文对象将创建并加入到报文队列中，同时一个 WM_NEW_MESSAGE消息将发送到主应用程序并在此处理，实际上在这里使用的报文队列类来自上一个套接字实例。

使用文本编辑器添加一个 Listen()函数，在其中首先使用 ::SetCommTimeout()函数避免该端口在进行读取操作时出现超时，然后调用 RecvThread()函数来进行实际的读取操作:

```
void CWzdPortIO::Listen( int hdrSz, int bodyPos, CWzdQueue *pQueue,
    CWnd *pWnd, UINT msg, UINT id )
{
    // cancel timeouts! we want to wait forever until
    //     next message comes in
    COMMTIMEOUTS cto;
    ::GetCommTimeouts( ( HANDLE )m_hFile, &cto );
```

```
    cto.ReadIntervalTimeout = 0;
    cto.WriteTotalTimeoutMultiplier = 0;
    cto.WriteTotalTimeoutConstant = 0;
    ::SetCommTimeouts( ( HANDLE )m_hFile, &cto );

    // initialize the structure we will pass to thread
    m_RecvData.pFile = this;
    m_RecvData.hdrSz = hdrSz;
    m_RecvData.bodyPos = bodyPos;
    m_RecvData.pQueue = pQueue;
    m_RecvData.pWnd = pWnd;
    m_RecvData.msg = msg;
    m_RecvData.id = id;

    // start the thread
    AfxBeginThread( RecvThread,&m_RecvData );
}
```

RecvThread()函数使用了报头和报文体的概念，因此报文可以是可变长度的。报头一般是固定长度的，其中包含了报文体的长度。 Read()函数可以通过这种方式确切地知道在返回前需读取多少个字节。

使用文本编辑器添加 RecvThread()函数到该类，在此设置循环直到出错。如果出错，则停止读操作并向应用程序发送 WM_DONE_MESSAGE消息。使用 CFile的Read()函数读取报文的报头和报文体，并将它们放入报文队列，然后使用 WM_NEW_MESSAGE消息通知应用程序:

```
UINT RecvThread( LPVOID pParam )
{
    // get data from thread creator
    RECVDATA *pRecv = ( RECVDATA * )pParam;

    while ( TRUE )    // forever
    {
        // read the header
        int len;
        int error = 0;
        char *pHdr = new char[pRecv -> hdrSz];
        try
        {
            len = pRecv -> pFile -> Read( pHdr, pRecv -> hdrSz );
        }
        catch ( CFileException *e )
        {
            error = e -> m_cause;
            e -> Delete();
        }

        // read the body???
        char *pBody = NULL;
        if ( !error && pRecv -> bodyPos != -1 )
```

```
        {
            int bodyLen = *( ( short * )pHdr+pRecv -> bodyPos );
            pBody = new char[bodyLen];
            try
            {
                len += pRecv -> pFile -> Read( pBody, bodyLen );
            }
            catch ( CFileException *e )
            {
                error = e -> m_cause;
                e -> Delete();
            }
        }

        // put message in queue
        pRecv -> pQueue ->
            Add( new CWzdMsg( pRecv -> id,pHdr,pBody,len,error ) );

        // post message to window to process this new message
        pRecv -> pWnd -> PostMessage( pRecv -> msg );
    }

    return 0;
}
```

2) 对并行通信使用新的端口I/O类

在窗口类例如CMainFrame中，嵌入CWzdQueue和CWzdPortIO。

```
CWzdPortIO m_parallel;
CWzdQueue m_queue;
```

打开连接，并使用以下代码开始监听:

```
CFileException e;
if ( m_parallel.OpenLPT(
    1,                          // LPT number (1,2,etc.)
    &e                          // exception errors (defaults to NULL)
    ) )
{
    m_parallel.Listen(
        10,                     // size of message header
        -1,                     // position of size of message body
                                //     in header
                                //     -1 means all message lengths
                                //     are fixed
        &m_queue,               // CWzdQue to store new messages
        this,                   // pWnd of window to send
                                //     “new message” message
        WM_NEW_MESSAGE,         // “new message” to send
        0                       // the user defined id of which port
                                //     sent the message
        );
}
```

为在接收到新报文时对它们进行处理，在窗口类中手工添加 WN_NEW_MESSAGE处理函数:

```
ON_MESSAGE( WM_NEW_MESSAGE,OnNewMessage )
:  :  :

LRESULT CMainFrame::OnNewMessage( WPARAM,LPARAM )
{
   CWzdMsg *pMsg = NULL;
   while ( pMsg = m_queue.Remove() )
   {
      // pMsg contains:
      // m_nID      - - the user defined id of which port sent
      //                the message
      // m_pHdr     - - the message header
      // m_pBody    - - the message body
      // m_len      - - the total message length
      // m_error    - - any errors

      // make sure to delete the message after processing!
      delete pMsg;
   }
   return OL;
}
```

为发回报文到端口，使用：

```
m_parallel.Send(
   hello,              // buffer to send
   7                   // length of buffer to send
   );
```

3) 对串行通信使用新的端口I/O类

在窗口类例如CMainFrame中，嵌入CWzdQueue和CWzdPortIO:

```
CWzdPortIO m_serial;
CWzdQueue m_queue;
```

打开连接，并使用以下代码开始监听:

```
CFileException e;
if ( m_serial.OpenCOM(
   1,                          // COM number (1,2,etc.)
   &e,                         // exception errors (defaults to NULL)
   CBR_19200,                  // baud rate, also CBR_1200, CBR_2400, etc.
   NOPARITY,                   // parity, also EVENPARITY, ODDPARITY,
                               //    MARKPARITY, SPACEPARITY
   8,                          // number of bits in a byte
   ONESTOPBIT                  // stopbits, also ONE5STOPBITS, TWOSTOPBITS
   ) )

{
   m_serial.Listen(
   10,                         // size of message header
   -1,                         // position of size of message body in header
```

```
                                    //    -1 means all message lengths are fixed
    &m_queue,                       // CWzdQue to store new messages
    this,                           // pWnd of window to send "new message" message
    WM_NEW_MESSAGE,                 // message to send
    1                               // the user defined id of which port sent
                                    //    the message
    );
}
```

处理和发送报文的过程与前面在并行端口中的方式相同。

4. 注意

本例中使用的方案允许应用程序使用自己的标识符来区分哪一个报文来自哪一个设备，以此实现同时监控几个设备。

5. 使用光盘时注意

运行工程的两个实例，其中一个处于调试模式，另一个处于发布模式。单击 Test菜单可以试验串行或者并行通信，但是需要能够协同工作的设备。

6. 程序清单 —— 端口I/O类

```
// PortIO.h: interface for the CPortIO class.
//
//////////////////////////////////////////////////////////////////////

#if !defined( AFX_PORTIO_H__81CE0F20_8C16_11D2_A18D_99620BDF6820__INCLUDED_ )
#define AFX_PORTIO_H__81CE0F20_8C16_11D2_A18D_99620BDF6820__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif    // _MSC_VER > 1000

#include "WzdQue.h"

// define the send and receive threads
typedef struct t_SENDDATA
{
    CFile *pFile;
    LPSTR lpBuf;
    int len;
} SENDDATA;

typedef struct t_RECVDATA
{
    CFile *pFile;
    int hdrSz;
    int bodyPos;
    CWzdQueue *pQueue;
    CWnd *pWnd;
    UINT msg;
    UINT id;
} RECVDATA;
```

```
class CWzdPortIO : public CFile
{
public:
    CWzdPortIO() {};
    virtual ~CWzdPortIO() {};

    BOOL OpenLPT( int n,CFileException *e = NULL );
    BOOL OpenCOM( int n,CFileException *e = NULL, int baud = -1,
        int parity = -1, int databits = -1, int stopbits = -1);

    void Send( LPSTR lpBuf, int len );
    void Listen( int hdrSz, int bodyPos, CWzdQueue *pQueue, CWnd *pWnd,
        UINT msg, UINT id);

private:
    SENDDATA m_SendData;
    RECVDATA m_RecvData;


};
UINT SendThread( LPVOID pParam );
UINT RecvThread( LPVOID pParam );

#endif
    // !defined( AFX_PORTIO_H__81CE0F20_8C16_11D2_A18D_99620BDF6820__INCLUDED_ )
// PortIO.cpp: implementation of the CPortIO class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "WzdPrtIO.h"

//////////////////////////////////////////////////////////////////////
// Open Printer Port
//////////////////////////////////////////////////////////////////////

BOOL CWzdPortIO::OpenLPT( int n,CFileException *e )
{
    CString portName;
    portName.Format( "LPT%d:", n );
    return Open( portName, CFile::modeReadWrite, e );
}

//////////////////////////////////////////////////////////////////////
// Open Serial Port
//////////////////////////////////////////////////////////////////////

BOOL CWzdPortIO::OpenCOM( int n, CFileException *e, int baud, int parity,
    int databits, int stopbits )
{
    CString portName;
```

```
    portName.Format( "COM%d:", n );
    if ( Open( portName, CFile::modeReadWrite, e) )
    {
        DCB dcb;
        ::GetCommState( ( HANDLE )m_hFile, &dcb );
        if ( baud != -1 )     dcb.BaudRate = baud;
        if ( databits != -1 ) dcb.ByteSize = databits;
        if ( stopbits != -1 ) dcb.StopBits = stopbits;
        if ( parity != -1 )   dcb.Parity = parity;
        ::SetCommState( ( HANDLE )m_hFile, &dcb );
        return( TRUE );
    }
    return( FALSE );
}

//////////////////////////////////////////////////////////////////////
// Send to Port
//////////////////////////////////////////////////////////////////////

void CWzdPortIO::Send( LPSTR lpBuf, int len )
{
    // initialize the structure we will pass to thread
    m_SendData.pFile = this;
    m_SendData.lpBuf = lpBuf;
    m_SendData.len = len;

    // start the thread
    AfxBeginThread( SendThread,&m_SendData );
}

UINT SendThread( LPVOID pParam )
{
    // get data from thread creator
    SENDDATA *pSend = ( SENDDATA * )pParam;

    // do the write
    pSend -> pFile -> Write( pSend -> lpBuf, pSend -> len );

    return 0;
}


//////////////////////////////////////////////////////////////////////
// Listen to Port
//////////////////////////////////////////////////////////////////////

void CWzdPortIO::Listen( int hdrSz, int bodyPos, CWzdQueue *pQueue,
    CWnd *pWnd, UINT msg, UINT id )
{
    // cancel timeouts! we want to wait forever until next message comes in
    COMMTIMEOUTS cto;
```

```
    ::GetCommTimeouts( ( HANDLE )m_hFile, &cto );
    cto.ReadIntervalTimeout = 0;
    cto.WriteTotalTimeoutMultiplier = 0;
    cto.WriteTotalTimeoutConstant = 0;
    ::SetCommTimeouts( ( HANDLE )m_hFile, &cto );

    // initialize the structure we will pass to thread
    m_RecvData.pFile = this;
    m_RecvData.hdrSz = hdrSz;
    m_RecvData.bodyPos = bodyPos;
    m_RecvData.pQueue = pQueue;
    m_RecvData.pWnd = pWnd;
    m_RecvData.msg = msg;
    m_RecvData.id = id;

    // start the thread
    AfxBeginThread( RecvThread,&m_RecvData );
}

UINT RecvThread( LPVOID pParam )
{
    // get data from thread creator
    RECVDATA *pRecv = ( RECVDATA * )pParam;

    while ( TRUE ) //forever
    {
        // read the header
        int len;
        int error = 0;
        char *pHdr = new char[pRecv -> hdrSz];
        try
        {
            len = pRecv -> pFile -> Read( pHdr, pRecv -> hdrSz );
        }
        catch ( CFileException *e )
        {
            error = e -> m_cause;
            e -> Delete();
        }

        // read the body???
        char *pBody = NULL;
        if ( !error && pRecv -> bodyPos != -1 )
        {
            int bodyLen = *( ( short * )pHdr+pRecv -> bodyPos );
            pBody = new char[bodyLen];
            try
            {
                len += pRecv -> pFile -> Read( pBody, bodyLen );
            }
            catch ( CFileException *e )
```

```
            {
                error = e -> m_cause;
                e -> Delete();
            }
        }

        // put message in queue
        pRecv -> pQueue ->
            Add( new CWzdMsg( pRecv -> id,pHdr,pBody,len,error ) );

        // post message to window to process this new message
        pRecv -> pWnd -> PostMessage( pRecv -> msg );
    }

    return 0;
}
```