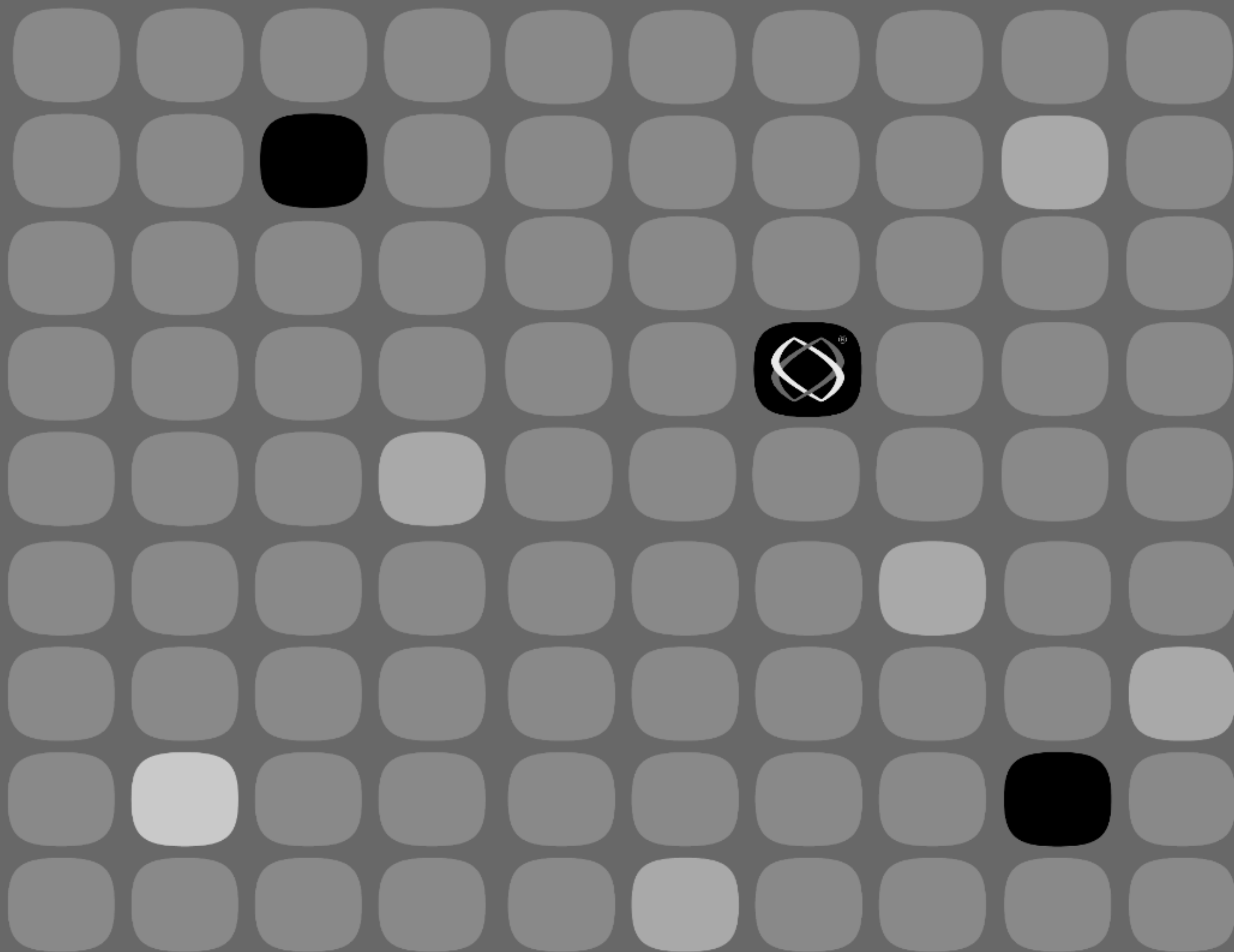


Programmer's Reference

Attachmate®

INFOConnect®

Enterprise Edition

PTR

Print and Transaction Router

User Application Programming Interface

UNISYS

Copyrights and Notices

Attachmate® INFOConnect® Enterprise Edition

© 2012 Attachmate Corporation. All Rights Reserved.

Patents

This Attachmate software is protected by U.S. patents 6252607 and 6803914.

Trademarks

Attachmate, the Attachmate logo, CryptoConnect, FileXpress, and PEPgate are either registered trademarks or trademarks of Attachmate Corporation in the USA. INFOConnect is a registered trademark of Unisys Corporation. FIPS 140-1 Validated is a certification mark of NIST, which does not imply product endorsement by NIST, the U.S. or Canadian Governments. All other trademarks, trade names, or company names referenced in product materials are used for identification only and are the property of their respective owners.

Attachmate Software License Agreement

A copy of the Attachmate software license agreement governing this product can be found in a 'license' file in the root directory of the product.

Licensor

Attachmate Corporation
1500 Dexter Avenue North
Seattle, WA 98109 USA
USA
+1.206.217.7100
http://www.attachmate.com

Third-Party Notices

Third Party Terms and notices are provided in a 'thirdpartynotices' file in the root directory of the product.

Contents

	About This Guide	vii
Chapter 1	PTR UAPI Overview	1
Chapter 2	PTR UAPI Functions	7
Appendix A	Sample Application	47
Appendix B	Troubleshooting	51
Appendix C	Native Airline Wedge Support	53
Appendix D	BPTP Wedge Support	57
Appendix E	RTE Wedge Implementation	59
Appendix F	Videcom Wedge Support	63

Glossary 65

Index 67

Contents

About This Guide

This guide is intended for C or C++ programmers who want to write custom PTR applications to send data directly to a printer from a PC.

The following sections are included in this preface:

Conventions viii

Abbreviations ix

Related Documentation x

Conventions

This guide uses the following conventions:

- Text that you type as well as messages and prompts that appear on the screen are shown `in this type style`.
- In addition to emphasizing text and highlighting terms used for the first time, *italic* indicates variables. For example, if you were asked to type *drive:\directory\filename.ext*, you would type the actual drive, directory, and file name in place of the italicized words.
- The term *printer* is used throughout this document to refer to output devices, input devices, and bi-directional devices that are compatible with PTR.

Abbreviations

The following abbreviations are used throughout this guide and are provided here for quick reference.

Abbreviation	Name
PDK	PTR development kit
PTR	Print and Transaction Router
UAPI	User application programming interface

Related Documentation

Additional information exists in the form of PTR-related guides and online help.

PTR-related documentation consists of the following guides:

- The *INFOConnect PTR API Programmer's Reference* provides you with details about how you can use the PDK (PTR development kit) to design a custom host filter.
- The *INFOConnect PTR OLE API Programmer's Reference* provides you with details on how to make an application that monitors PTR routes.
- The *INFOConnect PTR User API Programmer's Reference* provides you with details on how to make an application that prints vouchers and other specialized tickets from your PC by sending data directly to a printer.

PTR UAPI Overview

1

In This Chapter

This chapter explains what the PTR UAPI is and how to use the PTR UAPI functions in a custom application. The following sections are included:

Prerequisites 2

Using the PTR UAPI Library 3

List of PTR UAPI Functions 4

Prerequisites

To use the PTR UAPI, your system must have the following software:

- INFOConnect Connectivity Services, version 4.2 or later

Additional Requirements

To use the PTR UAPI functions, you must be able to configure a route in INFOConnect PTR, which includes configuring a host path, printer queue path, and host filter using INFOConnect Manager.

In addition, you must be familiar with the following:

- C/C++ programming language
- Microsoft Win32® API
- Microsoft Foundation Classes

PTR UAPI Files

You can find the necessary PTR UAPI library file (PTRUAPI.LIB) and header file (PTRUAPI.H), as well as a complete sample application (PTRUAPI.EXE) that uses the PTR UAPI, in the following directory on the installation CD:

\\32\PDK\PTRUAPI

This sample is documented in Appendix A, “Sample Application” beginning on page 47.

It is recommended that you run, and even modify, the sample application before you write your own application.

Using the PTR UAPI Library

The PTR UAPI library allows you to communicate with any device that works with PTR, such as printers, credit card readers, passport readers, or laser wands.

The PTR UAPI library contains functions that you can use to print vouchers and other specialized tickets from your PC by sending data directly to a printer without going through an actual host.

Linking to the PTR UAPI Library

To use the PTR UAPI functions, you must do the following:

- Write an application that links to PTRUDLL.LIB, which implements the PTR UAPI functions. If you have a legacy application that uses PTRUAPI.LIB, that will also function.
- Include PTRUDLL.LIB and PTRUAPI.H in the project for your application. PTRUAPI.H provides the PTR UAPI function prototypes.
- The file PTRUDLL.DLL must be located in a directory specified in the Windows PATH environment variable.
- If you are not using the C/C++ language in your application, set up your application to call the PTRUAPI functions directly in the PTRUDLL.DLL.

Configuring a PTR Route to Use the PTR UAPI Library

PTR requires you to specify actual host path configuration information and an actual host filter in your PTR route. Because your application uses the PTR UAPI functions to communicate directly with a printer, you must mimic the host path information and host filter by using the API—User API External Interface Library path template when you configure the host path and the APITHR32.HFF host filter when you configure a PTR route. Using this filter and template combination satisfies the PTR route. You can use any existing printer queue path.

Configuring the Host Path

In addition, when you configure the host path, the API Path Configuration dialog box prompts you for the Open Name. Open Name corresponds to the *lpName* parameter that you pass in the PTRUAPI_Open( ) function. You can specify anything for the open name as long as you use the same name for the *lpName* parameter variable. For details, see “PTRUAPI_Open( )” on page 10.

List of PTR UAPI Functions

The PTR UAPI contains the following functions, which are explained in detail in Chapter 2, "PTR UAPI Functions," beginning on page 8:

Function	Description
PTRUAPI_Init( )	This function initializes the instance of your application and associates your application with the PTR UAPI library.
PTRUAPI_Open( )	This function opens the PTR printer queue path. If this function is successful, the path you specified is opened, and a connection handle for the path becomes available. This handle must be passed to other functions.
PTRUAPI_Read( )	This function uses the connection handle returned by PTRUAPI_Open( ) to read data from the printer queue path.
PTRUAPI_ReadEx( )	This function uses the connection handle returned by PTRUAPI_Open( ) to read data from the printer queue path. It is the same as PTRUAPI_Read with the addition of chaining information in the callback.
PTRUAPI_ReadWait( )	This function is similar to PTRUAPI_Read( ) but uses a timeout integer (in seconds) instead of a callback.
PTRUAPI_ReadWaitEx( )	This function is a way of reading data synchronously from the printer queue path. It's the same as PTRUAPI_ReadWait except that it provides a parameter for chaining information.
PTRUAPI_Write( )	This function uses the connection handle returned by PTRUAPI_Open( ) to write data to the printer queue path.
PTRUAPI_WriteEx( )	This function uses the connection handle returned by PTRUAPI_Open( ) to write data to the printer queue path.
PTRUAPI_WriteWait( )	This function is similar to PTRUAPI_Write( ) but uses a timeout integer (in seconds) instead of a callback.

Function	Description
PTRUAPI_WriteWaitEx( )	This function allows you to send data synchronously to a printer queue path. It is the same as PTRUAPI_WriteWait except that it includes chaining information.
PTRUAPI_Status( )	This function uses the connection handle returned by PTRUAPI_Open( ) to read the status of the printer queue path.
PTRUAPI_StatusWait( )	This function is similar to PTRUAPI_Status( ) but uses a timeout integer (in seconds) instead of a callback.
PTRUAPI_CurrentStatus( )	This function uses the connection handle to read the status of the printer queue path.
PTRUAPI_Close( )	This function uses the connection handle provided by PTRUAPI_Open( ) to close the printer queue path.
PTRUAPI_Term( )	This function terminates the connection between the application and the PTR UAPI library.
PTRUAPI_GetBlockSize( )	This function returns the maximum block size in bytes for the particular device or route.
PTRUAPI_GetLastError( )	This function allows you to get error information if another PTRUAPI function fails.
PTRUAPI_OpenEx( )	This function and the following two functions (PTRUAPI_Lock and PTRUAPI_Unlock) facilitate device sharing between applications within a single PTR route without having to use two different PTR routes and using the Share Among setting on a single API route.
PTRUAPI_Lock( )	This function requests an API device lock by providing as input, the handle for the device, which is obtained from the PTRUAPI_OpenEx function.
PTRUAPI_UnLock( )	This function releases API device locks initiated by the function PTRUAPI_Lock.

PTR UAPI Functions

2

In This Chapter

This chapter explains each PTR UAPI function. The following sections are included:

PTRUAPI_Init( ) 9
PTRUAPI_Open( ) 10
PTRUAPI_Read( ) 12
PTRUAPI_ReadEx( ) 15
PTRUAPI_Write( ) 17
PTRUAPI_WriteEx( ) 20
PTRUAPI_Status( ) 22
PTRUAPI_Close( ) 28
PTRUAPI_Term( ) 29
PTRUAPI_CurrentStatus( ) 30
PTRUAPI_ReadWait( ) 31
PTRUAPI_ReadWaitEx( ) 33
PTRUAPI_WriteWait( ) 35
PTRUAPI_WriteWaitEx( ) 37
PTRUAPI_StatusWait( ) 39
PTRUAPI_GetBlockSize( ) 40

PTRUAPI_GetLastError( ) 41
PTRUAPI_OpenEx( ) 43
PTRUAPI_Lock( ) 44
PTRUAPI_UnLock( ) 47

PTRUAPI_Init( )

This function initializes the instance of your application and associates your application with the PTR UAPI library.

You must call this function before calling any other PTR UAPI functions.

Syntax

```
BOOL PTRUAPI_Init(HINSTANCE hInst);
```

Parameters

PTRUAPI_Init( ) requires the following parameter:

Parameter	Description
hInst	The handle that identifies the instance that PTRUAPI_Init( ) is within

Example

```
if (!PTRUAPI_Init(hInstance))
   {
     m_StatusStatic = "Unable to communicate with PTRUAPI.EIL.";
     UpdateData(FALSE);
   }
   else
   {
     m_StatusStatic = "Enter text to be printed then click the
        PRINT button.";
     UpdateData(FALSE);
     m_Edit.SetFocus( );
   }
   return FALSE;  // return TRUE  unless you set the focus to a
                 //control
```

PTRUAPI_Open( )

This function opens the PTR printer queue path. If this function is successful, the path you specified is opened, and a connection handle for the path becomes available. This handle must be passed to other functions.

You must call this function or PTRUAPI_OpenEx( ) after calling PTRUAPI_Init( ) but before calling any other function. In cases where you need exclusive device use locking, use "PTRUAPI_OpenEx( )" on page 43.

Syntax

```
HANDLE PTRUAPI_Open(LPSTR lpName);
```

Parameters

PTRUAPI_Open( ) requires the following parameter:

Parameter	Description
lpName	The Open Name of the PTR host path (see "Configuring the Host Path" on page 3 for details). Because your application appears to PTR as the host, this parameter must match the name you configured for your PTR host queue path, and the PTR host queue path must use the API—User API External Interface Library path template.

Return Value

This function returns the handle that identifies the instance that PTRUAPI_Init( ) is within.

Example

```
if(!(hConn = PTRUAPI_Open(m_PathName.GetBuffer(10))))
   {      //for non -document printer m_StatusStatic = "Unable
          //to open connection or connection already
          //open:" + m_PathName;
     m_Result = "Unable to open connection " + m_PathName;
     m_EdStatus = _T("");
   }
     else
   {
     m_Print.EnableWindow(FALSE);
                 //gray Print Button until response comes
                 //back or timeout
                 //m_StatusStatic = "Connection: " +
                 //m_PathName + " opened successfully.";
     m_Result = "Connection: " + m_PathName + " opened
```

```
        successfully.";
}
```

PTRUAPI_Read( )

This function uses the connection handle returned by PTRUAPI_Open( ) to read data from the printer queue path. It does this by posting a receive buffer (supplied by the application) to receive data from the printer. This function uses a callback routine, which is not called right away. Data arriving from the printer is posted to the application in the callback routine. See also "PTRUAPI_ReadEx( )" on page 15.

It is recommended that you call this function before calling PTRUAPI_Write( ).

Syntax

```
BOOL PTRUAPI_Read(HANDLE hConn, LPSTR lpBuf, UINT
  szbuffer, UAPICALLBACK rcb, LONG lrcbData);
```

Parameters

PTRUAPI_Read( ) requires the following parameters:

Parameter	Description
hConn	The connection handle for the printer queue path
lpBuf	The buffer to hold the data for the printer queue The callback procedure returns the actual data from the printer queue in this buffer.
szbuffer	The size of the buffer
rcb	The pointer for the callback procedure that the PTR UAPI library calls The callback procedure returns the actual data from the printer queue. For details, see "ReadCB( ) Callback" on page 13.
lrcbData	(Optional.) Additional information that you want the PTR UAPI library to return in the callback If you are using C++ to write this application, it is recommended that you use this parameter to specify the object so that it is easier to process the data in the callback routine. If you are using C, you might want to place a number in this parameter that you can use in the callback routine to determine which routine placed the original call.

Return Value

`TRUE` (Success) or `FALSE` (Failure)

Example

```
rc = PTRUAPI_Read(hConn, ReadBuf, sizeof(ReadBuf),
  &ReadCB, (long)this);
```

ReadCB( ) Callback

ReadCB( ) is the callback that the PTR UAPI library calls in response to the request by PTRUAPI_Read( ). It is recommended that you call PTRUAPI_Read( ) again in your callback routine so that a receive buffer is always posted for data arriving from the printer.

Syntax

```
void ReadCB(LONG l, LPSTR b, UINT sz)
```

Parameters

The ReadCB( ) callback requires the following parameters:

Parameter	Description
l	The additional information that you requested from the PTR UAPI library in the callback This parameter corresponds to the *lrcbData* parameter in the PTRUAPI_Read( ) function.
b	The buffer to hold the data for the printer queue
sz	The length of the data in bytes

Example

```
void ReadCB(LONG l, LPSTR b, UINT sz)
{

  CPTRSampleApplicationDlg *Dlg =
  (CPTRSampleApplicationDlg *)l;
  Dlg->OnDataRead(sz, b);  //this function
                                  //processes the data

  if (b[sz-1] == 0x03)
  {
    if (!PTRUAPI_Close(hConn))
         PTRUAPI_Term( );
    hConn = NULL;
    CloseHandle(hStatusEvent);
    CloseHandle(hEventThread);
//   Dlg->m_Print.EnableWindow(TRUE);//could turn Print
                                        //Button solid
               //here to allow printer to finishing
               //printing previous
```

```
                //accommodation coupon(s) before submitting
                //another one
    }
    else
    {
        PTRUAPI_Read(hConn, ReadBuf, sizeof(ReadBuf),
            ReadCB, 1);
    }
    return;
}
```

PTRUAPI_ReadEx( )

This function uses the connection handle returned by PTRUAPI_Open( ) to read data from the printer queue path. It is the same as PTRUAPI_Read with the addition of chaining information in the callback and uses an extended callback. When using this function, the device messages may be received in complete messages (like with PTRUAPI_Read), or the data may be received in pieces via consecutive calls to PTRUAPI_ReadEx.

The application may receive large data messages by assembling the data pieces (from the callbacks) together. The iBlockInfo value in the extended callback is what will indicate where in the data chain that piece of data is.

As with PTRUAPI_Read, the a receive buffer is supplied by the application to receive data from the device. The data arriving from the device is posted to the application in the callback routine.

If you call the function with a NULL value for the callback address (that is, the rcbx parameter), any outstanding read callback request is cancelled. It is recommended that you call one of the PTRUAPI_Read functions before calling a PTRUAPI_Write function.

Syntax

```
BOOL FAR PASCAL PTRUAPI_ReadEx(HANDLE hConn, LPSTR lpBuf,
   UINT nBytes, UAPICALLBACKEX rcbx,
                   LONG lrcbData);
```

Parameters

Same as PTRUAPI_Read except that an extended callback is used.

Return Value

`TRUE` (Success) or `FALSE` (Failure)

Comments

The functions PTRUAPI_Read and PTRUAPI_ReadWait are based on full messages. You may use those functions if you do not expect to receive large messages from the device. The data in the extended callbacks resulting from using PTRUAPI_ReadEx may be either complete messages or pieces of a larger message that needs to be assembled.

Extended Read Callback

ReadCBEx( ) is the callback that the PTR UAPI library calls in response to the request by PTRUAPI_ReadEx( ). It is recommended that you call PTRUAPI_ReadEx( ) again in your callback routine so that a receive buffer is always posted for data sent from the device.

Syntax

```
void ReadCBEx(LONG lUser, LPSTR lpBuf, UINT nBytes, INT
   iBlockInfo)
```

Parameters

The ReadCBEx( ) callback uses the same parameters as ReadCB() with the addition of iBlockInfo, which uses one of the following values:

iBlockInfo Value	Description
BLOCK_NONE	/* No Blocking/Chaining of Messages */
BLOCK_FIRST	/* First Block in Chain */
BLOCK_MIDDLE	/* Middle Block in Chain (may be multiple or none) */
BLOCK_LAST	/* Last Block in Chain */
BLOCK_COMPLETE	/* Only Block in Chain (complete message) */

PTRUAPI_Write( )

This function uses the connection handle returned by PTRUAPI_Open( ) to write data to the printer queue path.

It is recommended that you call PTRUAPI_Read( ) before calling this function.

If the return code from a PTRUAPI_Write( ) call immediately returns an error status, it is recommended that you try the same PTRUAPI_Write( ) call again at a later time. Because the application and INFOConnect communicate using Windows messages, whenever something interferes with or delays these messages, the PTRUAPI_Write( ) call may fail. Once the Windows messaging is restored (which may happen when the application returns control to the idle loop), subsequent PTRUAPI_Write( ) calls should work.

See also “PTRUAPI_WriteEx( )” on page 20.

Syntax

```
BOOL PTRUAPI_Write(HANDLE hConn, LPSTR lpBuf, UINT sz,
  UAPICALLBACK wcb,LONG lwcbData);
```

Parameters

PTRUAPI_Write( ) requires the following parameters:

Parameter	Description
hConn	The connection handle for the printer queue path
lpBuf	The message string that you want to appear on the ticket you are printing
sz	The length of the string in bytes
wcb	The pointer for the callback procedure that the PTR UAPI library calls The callback procedure determines whether or not the ticket was submitted to the printer successfully. For details, see “WriteCB( ) Callback” on page 18.

Parameter	Description
lwcbData	(Optional.) Additional information that you want the PTR UAPI library to return in the callback If you are using C++ to write this application, it is recommended that you use this parameter to specify the object so that it is easier to process the data in the callback routine. If you are using C, you might want to place a number in this parameter that you can use in the callback routine to determine which routine placed the original call.

Return Value

`TRUE` (Success) or `FALSE` (Failure)

Example

```
rc = PTRUAPI_Write(hConn, StringData, StringLength,
   &WriteCB, (long)this);
   if(!rc)
   {
     if (!PTRUAPI_Close(hConn))
          PTRUAPI_Term( );
     hConn = NULL;
     CloseHandle(hStatusEvent);
     CloseHandle(hEventThread);
     m_Result = "Unable to write buffer to printer.";
     UpdateData(FALSE);
     return;
   }
```

WriteCB( ) Callback

WriteCB( ) is the callback that the PTR UAPI library calls in response to the request by PTRUAPI_Write( ).

Syntax

```
void WriteCB(LONG l, LPSTR b, UINT szstring)
```

Parameters

The WriteCB( ) requires the following parameters:

Parameter	Description
l	The additional information that you requested from the PTR UAPI library in the callback This parameter corresponds to the *lwcbData* parameter in the PTRUAPI_Write( ) function.
b	The string written to the printer queue (ticket)

Parameter	Description
szstring	The size of the string

Example

```
void WriteCB(LONG l, LPSTR b, UINT szstring)
{
  CPTRSampleApplicationDlg *Dlg =
     (CPTRSampleApplicationDlg *)l;
  Dlg->OnWriteUpdate(szstring);
}
```

PTRUAPI_WriteEx( )

This function uses the connection handle returned by PTRUAPI_Open( ) to write data to the printer queue path. PTRUAPI_WriteEx can be used to send complete messages or, by sending consecutive calls, it can be used to send large data messages in pieces. This function is the same as PTRUAPI_Write except that it includes chaining information, which identifies the pieces.

If you call this function with a NULL callback address (that is, the wcb parameter), it will cancel any outstanding write callback request.

It is recommended that you call one of the PTRUAPI_Read functions before calling this function. If the return code from a PTRUAPI_WriteEx( ) call immediately returns an error status, it is recommended that you try the same PTRUAPI_WriteEx( ) call again at a later time. The application and INFOConnect sometimes communicate using Windows messages, which can be delayed or interfered with and cause the PTRUAPI_WriteEx( ) call to fail. After Windows messaging is restored (this may occur when the application returns control to the idle loop), subsequent PTRUAPI_WriteEx( ) calls should be successful.

Syntax

```
BOOL FAR PASCAL PTRUAPI_WriteEx(HANDLE hConn, LPSTR
  lpBuf, UINT nBytes, UAPICALLBACK wcb,
                  LONG lwcbData, INT iBlockInfo);
```

Parameters

Same as PTRUAPI_Write with the addition of iBlockInfo, which uses one of the following values:

iBlockInfo Values	Description
BLOCK_NONE	/* No Blocking/Chaining of Messages */
BLOCK_FIRST	/* First Block in Chain */
BLOCK_MIDDLE	/* Middle Block in Chain (may be multiple or none) */
BLOCK_LAST	/* Last Block in Chain */
BLOCK_COMPLETE	/* Only Block in Chain (complete message) */

Return Value

TRUE (Success) or FALSE (Failure)

Comments

The functions PTRUAPI_Write and PTRUAPI_WriteWait are based on full messages. You may use those functions if you aren't sending large messages to the device. For document printers, a message may be divided up into pieces with or without PTRUAPI message chaining. Using PTRUAPI_WriteEx with an iBlockInfo value of BLOCK_NONE is equivalent to PTRUAPI_Write.

If you send a buffer of data longer than the maximum block size allowed, this function will fail and PTRUAPI_GetLastError will indicate ERROR_BUFFER_TOO_LARGE. Use the PTRUAPI_GetBlockSize function to obtain the maximum block size.

If you supply an illogical sequence of chaining information, this function will fail and PTRUAPI_GetLastError will indicate ERROR_CHAINING_ORDER.

PTRUAPI_Status( )

This function uses the connection handle returned by PTRUAPI_Open( ) to read the status of the printer queue path.

Syntax

```
BOOL PTRUAPI_Status(HANDLE hConn, UAPICALLBACK scb,LONG
   lscbData);
```

Parameters

PTRUAPI_Status( ) requires the following parameters:

Parameter	Description
hConn	The connection handle for the printer queue path
scb	The pointer for the callback procedure that the PTR UAPI library calls The callback procedure passes in the status of the printer queue. For details, see “StatCB( ) Callback” below.
lscbData	(Optional.) Additional information that you want the PTR UAPI library to return in the callback If you are using C++ to write this application, it is recommended that you use this parameter to specify the object so that it is easier to process the data in the callback routine. If you are using C, you might want to place a number in this parameter that you can use in the callback routine to determine which routine placed the original call.

Return Value

`TRUE` (Success) or `FALSE` (Failure)

Example

```
rc = PTRUAPI_Status(hConn, &StatCB, (long)this);
```

StatCB( ) Callback

This example illustrates the StatCB( ) callback that the PTR UAPI library will call in response to the request by PTRUAPI_Status( ).

Syntax

```
void StatCB(LONG l, LPSTR b, UINT nStat)
```

Parameters

The StatCB( ) function requires the following parameters:

Parameter	Description
l	The additional information that you requested from the PTR UAPI library in the callback This parameter corresponds to the *lrcbData* parameter in the PTRUAPI_Read( ) function.
b	Always NULL
nStat	An integer that represents the status Your application must decode this integer into something meaningful. See "Decoding the Integer" below for an illustration of how to do this.

Example

```
void StatCB(LONG l, LPSTR b, UINT nStat)
{
  CPTRSampleApplicationDlg * Dlg;
  Dlg = (CPTRSampleApplicationDlg *)l;
  Dlg->OnStatusUpdate(nStat);
  return;
}
```

Decoding the Integer

In the following example, the function OnStatusUpdate( ) decodes the status integer and returns the following statuses, which correspond to letters coded in the PTR UAPI library:

Status	PTR UAPI Status Code	Description
Ready	a	The ticket printed, and the printer is ready to accept data.
Busy	b	The printer cannot print the ticket because it is busy.
Error	c	An error occurred; the printer cannot print the ticket.
Not Ready	d	The printer is not ready, but it will continue to try to print the job.

```
void CPTRSampleApplicationDlg::OnStatusUpdate(UINT s)
{
  CString str;
  CString strStatus;
  str.Format("%8.8x",s);
  strStatus = str.GetAt(4) + str.GetAt(5);

  if (strStatus.Find('a') != -1)
    strStatus = "Ticket is printed OK.";//: READY";
  else
  {
    if (strStatus.Find('b') != -1)
      strStatus = "Printer is BUSY. Ticket can not be
          printed.";//BUSY";
    else
      if (strStatus.Find('c') != -1)
      {
        if (!PTRUAPI_Close(hConn))
          PTRUAPI_Term( );
        hConn = NULL;
        CloseHandle(hStatusEvent);
        CloseHandle(hEventThread);
        m_Result = "Unable to write buffer to printer.";
        strStatus = "An ERROR occurred. Ticket can not
           be printed.";//ERROR";
      }
      else
        if (strStatus.Find('d') != -1)
          strStatus = "Printer is NOT READY. Retry
             print is in progress. Please wait...";
                                  // : NOT READY";
  }
  m_Print.EnableWindow(TRUE);
  m_EdStatus = strStatus;
  UpdateData(FALSE);
}
```

PTR Interpretations of the PTRUAPI_Status Function

The documentation of the status return value (nStat in the PTRUAPI_Status callback) is not completely correct. The integer returned is really a 32-bit INFOConnect status. To fully interpret this would require knowledge of the IDK (INFOConnect Development Kit). Only a subset really matters for PTR purposes, and that is described in this topic.

In the sample application, in the OnStatusUpdate routine, the returned integer is converted into an ASCII string hexadecimal equivalent and two of the bytes are extracted. These two bytes are concatenated and then searched for a match against one of the four device status values (`a`, `b`, `c`, or `d`). As long as we identify the table as something to reference after the conversion (the `str.Format(%8.8x)` statement in the sample code), this definition will work fine. Otherwise, the correspondence with the letters coded in the PTRUAPI library does not hold.

The details of the raw (unconverted) integer are: it can be broken down into 4 byte values. Call the highest order byte the first one, and the lowest order byte the fourth one. All PTRUAPI applications should ignore the first (highest order byte) of nStat.

The second byte of nStat should be checked for the value 0x70 or 0x01. Values of nStat without a 0x70 or a 0x01 in the second byte should be ignored.The value of 0x70 in the second byte indicates an unsolicited printer status that the application can take note of, with the third and fourth bytes interpreted as below. A value of 0x01 in the second byte might also be received (when using certain PTR libraries).

If the second byte of the PTRUAPI status has a value of 0x01, the third byte is a printer status that can be interpreted as follows:

Value	Description
0x10	Device READY
0x11	Device BUSY
0x12	Device ERROR
0x13	Device NOT READY

If the second byte is a 0x70, the third byte can be interpreted as follows:

Value	Description
0x00	Printer is ready
0x01	Printer is not ready
0x02	No device
0x03	Printer error
0x04	Acknowledge abort
0x05	Acknowledge end of document

If the third byte is 0x01, 0x02, or 0x03, the fourth byte can be interpreted as follows:

Value	Description
0x00	Offline
0x01	Out of paper
0x02	Paper jam
0x03	Busy
0x04	Locked
0x05	Powered off
0x06	Not used
0x07	Error sending
0x08	Device initialization error
0x09	Device not available
0x0A	Bad device status

Examples

XxXX 0x07 0x00 0x00	Printer is ready (fourth byte is ignored)
XxXX 0x07 0x01 0x01	Printer is not ready / Out of paper
XxXX 0x07 0x03 0x08	Printer error / Device initialization error

Note: Not all PTR printer libraries will generate all of these status values (especially the fourth byte). Some are unique to certain environments (such as common use environments), or particular printer protocol components. For a generic PTRUAPI application, it is best to continue interpreting the third byte. This is what the sample application effectively does.

PTRUAPI_Close( )

This function uses the connection handle provided by PTRUAPI_Open( ) to close the PTR printer queue path.

Syntax

```
BOOL PTRUAPI_Close(HANDLE hConn);
```

Parameters

PTRUAPI_Close( ) requires the following parameter:

Parameter	Description
hConn	The connection handle for the printer queue path

Example

```
if (!PTRUAPI_Close(hConn))
     PTRUAPI_Term( );
hConn = NULL;
```

PTRUAPI_Term( )

This function terminates the connection between the application and the PTR UAPI library.

Syntax

```
void PTRUAPI_Term( );
```

Example

```
if (!PTRUAPI_Close(hConn))
     PTRUAPI_Term( );
hConn = NULL;
```

PTRUAPI_CurrentStatus( )

This function uses the connection handle to retrieve the current device status from the printer queue path. This function explicitly retrieves (and returns) the current device status, instead of waiting for a change in status (as the PTRUAPI_Status and PTRUAPI_StatusWait functions do).

Syntax

```
LONG PTRUAPI_CurrentStatus(HANDLE hConn);
```

Comments

The PTRUAPI_CurrentStatus function takes a single parameter (the device handle) and returns the same status integer that you get from PTRUAPI_Status on status changes. This function retrieves and returns the device status immediately. This is different than the PTRUAPI_Status function which uses a callback when there is a change in the device status, or the PTRUAPI_StatusWait function which uses a timeout while waiting for a change in the device status.

Parameters

PTRUAPI_CurrentStatus( ) requires the following parameter:

Parameter	Description
hConn	The connection handle for the printer queue path

Return Value

32-bit integer containing the status. Interpret it like nStat in PTRUAPI_Status.

Example

```
nStat = PTRUAPI_CurrentStatus(hConn);
```

PTRUAPI_ReadWait( )

PTRUAPI_ReadWait is a way of reading data from the printer queue path. It works like PTRUAPI_Read except that this function uses a timeout (so it can block), whereas PTRUAPI_Read uses a callback.

See also "PTRUAPI_ReadWaitEx( )" on page 33.

Syntax

```
Int PTRUAPI_ReadWait(HANDLE hConn, LPSTR lpBuf,
UINT szbuffer, INT iTimeOut, unsigned int * lpLen)
```

Comments

PTRUAPI_ReadWait does not use callbacks. Instead, PTRUAPI_ReadWait accepts an integer timeout value (in seconds). PTRUAPI_ReadWait returns an integer (instead of the booleans used in most of the other functions).

The PTRUAPI_ReadWait function returns back to the application as soon as read data is available or when the timeout has elapsed - whichever occurs first. It is also possible to use the PTRUAPI_ReadWait function while specifying a zero value for the timeout. In that case, PTRUAPI_ReadWait will return immediately (and not block).

Parameters

PTRUAPI_ReadWait( ) requires the following parameters:

Parameter	Description
hConn	The connection handle for the printer queue path
lpBuf	The buffer to hold the data for the printer queue The callback procedure returns the actual data from the printer queue in this buffer.
szbuffer	The size of the buffer
iTimeOut	Waitout time
lpLen	The pointer to an integer, which is where it will return the length of the bytes returned in lpBuf.

Return Value

0 (FALSE) = Failure.

1 (TRUE) = Success.

-1 = Timeout Elapsed.

Example

```
rc = PTRUAPI_ReadWait(hConn, ReadBuf, sizeof(ReadBuf),
100, &resultLength);
```

PTRUAPI_ReadWaitEx( )

This function is a way of reading data synchronously from the printer queue path. It's the same as PTRUAPI_ReadWait except that it provides a parameter for chaining information. An application can use this function to receive data in complete messages or in pieces via consecutive calls to PTRUAPI_ReadWaitEx. The iBlockInfo return value indicates where a piece of data is within the data chain.

Syntax

```
INT FAR PASCAL PTRUAPI_ReadWaitEx(HANDLE hConn, LPSTR
   lpBuf, UINT nBytes, INT iTimeOut,
                  unsigned int * lpLen, INT * iBlockInfo);
```

Return Value

0 (FALSE) = Failure

1 (TRUE) = Success

-1 = Timeout Elapsed

Comments

The functions PTRUAPI_Read and PTRUAPI_ReadWait are based on full messages. You may use those functions if you do not expect to receive large messages from the device. The data returned from PTRUAPI_ReadWaitEx may be a complete message or a series of pieces which will need to be assembled to form a larger message.

Parameters

The parameters are the same as in PTRUAPI_ReadWait with the addition of iBlockInfo, which uses one of the following values:

iBlockInfo Values	Description
`BLOCK_NONE`	/* No Blocking/Chaining of Messages */
`BLOCK_FIRST`	/* First Block in Chain */
`BLOCK_MIDDLE`	/* Middle Block in Chain (may be multiple or none) */
`BLOCK_LAST`	/* Last Block in Chain */
`BLOCK_COMPLETE`	/* Only Block in Chain (complete message) */

PTRUAPI_WriteWait( )

PTRUAPI_WriteWait allows the sending of data to a printer queue path. It works like PTRUAPI_Write except that it uses a timeout (so it can block), whereas PTRUAPI_Write uses a callback.

See also "PTRUAPI_WriteWaitEx( )" on page 37.

Syntax

```
PTRUAPI_WriteWait(HANDLE hConn, LPSTR lpBuf, UINT
szbuffer, INT iTimeOut)
```

Comments

PTRUAPI_WriteWait does not use callbacks. Instead, PTRUAPI_WriteWait accepts an integer timeout value (in seconds). PTRUAPI_WriteWait returns an integer (instead of the booleans used in the other functions).

The PTRUAPI_WriteWait function returns back to the application as soon as the write has completed or when the timeout has elapsed - whichever occurs first. It is also possible to use the PTRUAPI_WriteWait function while specifying a zero value for the timeout. In that case, PTRUAPI_WriteWait will return immediately (and not block).

PTRUAPI_WriteWait( ) requires the following parameters:

Parameter	Description
hConn	The connection handle for the printer queue path
lpBuf	The buffer to hold the data for the printer queue The callback procedure returns the actual data from the printer queue in this buffer.
szbuffer	The size of the buffer
iTimeOut	Waitout time

Return Value

0 (FALSE) = Failure.

1 (TRUE) = Success.

-1 = Timeout Elapsed.

Example

```
rc = PTRUAPI_WriteWait(hConn, ReadBuf, sizeof(ReadBuf),
100);
```

PTRUAPI_WriteWaitEx( )

This function allows you to send data synchronously to a printer queue path. It is the same as PTRUAPI_WriteWait except that it includes chaining information, which allows you to to send data in a series of pieces by using consecutive calls to PTRUAPI_WriteWaitEx.

Syntax

```
INT FAR PASCAL PTRUAPI_WriteWaitEx(HANDLE hConn, LPSTR
   lpBuf, UINT nBytes, INT iTimeOut,
                   INT iBlockInfo);
```

Parameters

Same as PTRUAPI_Write except for iBlockInfo, which uses one of the following values:

iBlockInfo Values	Description
BLOCK_NONE	/* No Blocking/Chaining of Messages */
BLOCK_FIRST	/* First Block in Chain */
BLOCK_MIDDLE	/* Middle Block in Chain (may be multiple or none) */
BLOCK_LAST	/* Last Block in Chain */
BLOCK_COMPLETE	/* Only Block in Chain (complete message) */

Returns

0 (FALSE) = Failure

1 (TRUE) = Success

-1 = Timeout Elapsed

Comments

The functions PTRUAPI_Write and PTRUAPI_WriteWait are based on full messages. You may use those functions if you are not sending large messages to the device. For document printers, a message may be divided into pieces with or without PTRUAPI message chaining.

PTRUAPI_WriteWaitEx with an iBlockInfo value of BLOCK_NONE is equivalent to PTRUAPI_WriteWait.

If you send a buffer of data longer than the maximum block size allowed, this function will fail and PTRUAPI_GetLastError will indicate ERROR_BUFFER_TOO_LARGE. Use the PTRUAPI_GetBlockSize function to obtain the maximum block size value.

If you supply an illogical sequence of chaining information, this function will fail and PTRUAPI_GetLastError will indicate ERROR_CHAINING_ORDER.

PTRUAPI_StatusWait( )

PTRUAPI_StatusWait is a way to get the device status from a printer queue path. It works like the PTRUAPI_Status function except that is uses a timeout (so it can block), whereas the PTRUAPI_Status function uses a callback.

Syntax

```
PTRUAPI_StatusWait(HANDLE hConn, INT iTimeOut,
unsigned int * lpStat);
```

Comments

PTRUAPI_StatusWait does not use callbacks. Instead, PTRUAPI_StatusWait accepts an integer timeout value (in seconds). PTRUAPI_StatusWait returns an integer (instead of the booleans used in the other functions).

The PTRUAPI_StatusWait function returns back to the application as soon as the status is available or when the timeout has elapsed - whichever occurs first. It is also possible to use the PTRUAPI_StatusWait function while specifying a zero value for the timeout. In that case, PTRUAPI_StatusWait will return immediately (and not block).

PTRUAPI_StatusWait( ) requires the following parameters:

Parameter	Description
hConn	The connection handle for the printer queue path
iTimeOut	Waitout time
lpStat	The pointer to an integer, which is where it will return the requested status.

Return Value

0 (FALSE) = Failure.

1 (TRUE) = Success.

-1 = Timeout Elapsed.

Example

```
rc = PTRUAPI_StatusWait(hConn, ReadBuf, sizeof(ReadBuf),
100, &lpStat);
```

PTRUAPI_GetBlockSize( )

This function returns the maximum block size in bytes for the particular device or route. The single input parameter is the device handle (from the PTRUAPI_Open call) and the output is an integer block size (in bytes). The block size is the maximum size of the a data piece used in chained data operations (such as PTRUAPI_WriteEx and PTRUAPI_WriteWaitEx).

Syntax

```
INT FAR PASCAL PTRUAPI_GetBlockSize(HANDLE hConn);
```

Returns

Integer which is the maximum number of bytes in a single piece of a chained write message.

Parameter

Parameter	Description
hConn	The connection handle for the printer queue path

PTRUAPI_GetLastError( )

If a call to another PTRUAPI function fails, you can sometimes get more detailed error information by calling PTRUAPI_GetLastError immediately afterwards. The return value will be one of the messages in the following list.

Syntax

```
LONG PTRUAPI_GetLastError();
```

Return Value

One of the following values in the PTRUAPI.h file.

Value	Description
ERROR_NONE	/* No Error on Last Operation - Success*/
ERROR_SYSTEM_ ERROR	/* Windows System Call Failed */
ERROR_NO_ MEMORY	/* Unable to Allocate Memory */
ERROR_PTR_NOT_ UP	/* PTR32 is not running */
ERROR_NO_MATCH _FOUND	/* No Match Found for OpenName String */
ERROR_BAD_ HANDLE	/* Bad Handle passed to PTRUAPI */
ERROR_NONE_ ACTIVE	/* No Handles Active at Present */
ERROR_NOTHING_ TO_SEND	/* Send Buffer or Length Invalid */
ERROR_NO_ BUFFER	/* No User Buffer - Unable to CallBack */
ERROR_TRANSMIT _FAILED	/* PTR Transmit to Device Failed */
ERROR_SEND_ QUEUED	/* INFO - IPC Send Queued at this Time */
ERROR_SESSION_ DOWN	/* Session is Down */

Value	Description
ERROR_BUFFER_TOO_LARGE	/* Transmit Buffer is Too Large */
ERROR_DATA_TRUNCATED	/* Supplied Receive Buffer is Too Small. Data was Truncated. */
ERROR_CHAINING_ORDER	/* Unexpected/Incorrect Chaining Sequence */

PTRUAPI_OpenEx( )

This function and the following two functions (PTRUAPI_Lock and PTRUAPI_Unlock) facilitate device sharing between applications within a single PTR route without having to use two different PTR routes and using the **Share Among** setting on a single API route.

PTRUAPI_OpenEx is a subset of PTRUAPI_Open and works the same with the exception of two additional parameters. The first parameter is a Boolean to indicate whether device locking (PTRUAPI_Lock and PTRUAPI_UnLock functions) is used. The second parameter is an optional identification string to identify the application name or context. When PTRUAPI_OpenEx is used with device locking, the lock must be active for the application to receive data from the device.

The PTR User Application should use [[“must use”?]] either PTRUAPI_Open or PTRUAPI_OpenEx to open a device.

Syntax

```
HANDLE FAR PASCAL PTRUAPI_OpenEx(LPSTR lpName, BOOL
  bLocking, LPSTR lpContext);
```

Parameters

Parameter	Description
lpName	This functions the same as PTRUAPI_Open.
bLocking	When the value is TRUE, the application uses the optional device locking functions (PTRUAPI_Lock and PTRUAPI_UnLock).
lpContext	This is a text string.

Return Value

An API device handle (like PTRUAPI_Open).

PTRUAPI_Lock( )

This function requests an API device lock by providing as input, the handle for the device, which is obtained from the PTRUAPI_OpenEx function.

Syntax

```
BOOL FAR PASCAL PTRUAPI_Lock(HANDLE hConn, UAPICALLBACK
   lcb, LONG llcbData, INT iTimeOutRequest, INT
   iTimeOutGrant);
```

Comments

If PTRUAPI_Lock is called with a NULL callback address, any pending or future lock callback operations are cancelled until PTRUAPI_Lock is called again with a (non NULL) callback address.

The lock timeout values are configurable per route (specified in the API Thru Host Filter Configuration). Acquire specifies the maximum time to wait to acquire a lock on the device (the default is 20 seconds). Grant specifies the maximum amount of time the lock is in effect after it's granted (the default is 60 seconds). To use these default values, specify a timeout of -1 or 0.

If a lock has been successfully granted, you can extend the lock by calling PTRUAPI_Lock again while that lock is still in effect and before the grant timeout period is reached. An application can keep a lock for an extended period of time; however, this will block other applications from using the device during that time period.

If there are multiple PTRUAPI applications, and some are using locking, and some are not, the API lock will determine which application gets the device data. If a PTRUAPI_Lock device lock is in effect, only the application with the active lock will get the device data while the lock is in effect. If no application has an active device lock, the PTRUAPI applications that are not using locking can obtain the device via an internal PTR lock and receive the device data, or received device data may be discarded. If the API Host path has a Share Among value of two or more (such as when a reader may be used), the same conditions apply. Only the application that has an active API device lock (if any) will receive the device data while the lock is in effect. If no application has an active API device lock (and the particular API route is active), all of the PTRUAPI applications that are not using API locking (PTRUAPI_Lock and PTRUAPI_UnLock) will receive a copy of the device data.

In the case of wedge devices, where one physical device is split into two or more logical devices, the API device lock is effective for the logical device (that is, the PTRUAPI device handle).

Parameters

Parameter	Description
hConn	API device handle
lcb	Lock callback routine address
llcbData	Optional data returned with callbacks
iTimeOutRequest	Not used
iTimeOutGrant	Not used

Return Values

`TRUE` (Success) or `FALSE` (Failure)

Lock Callback

The callback routine for PTRUAPI_Lock supplies the lock status and, if requested, a callback routine address (optional). Status notifications are defined in the PTRUAPI.H file:

Status	Description
`LOCK_ACQUIRED`	`/* Lock Successfully Acquired (Granted) */`
`LOCK_EXTENSION`	`/* Lock Extension Granted (Lock Maintained) */`
`LOCK_RELEASED`	`/* Lock Released (Device now UnLocked) */`
`LOCK_EXPIRED`	`/* Lock Expired (Device now UnLocked) */`
`LOCK_NOT_ACQUIRED`	`/* Lock Acquire Failed (not achieved within timeout) */`
`LOCK_CONTEXT`	`/* Lock/UnLock operation unexpected/ unprocessed - lacking proper context */`
`LOCK_ABORTED`	`/* Lock Aborted (Device now UnLocked) */`
`LOCK_NOT_ACTIVE`	`/* UnLock unexpected without active Lock (Device now UnLocked) */`

PTRUAPI_UnLock( )

This function releases API device locks initiated by the function PTRUAPI_Lock. As input, it uses the handle for the device, obtained via the PTRUAPI_OpenEx function. Releasing a lock with PTRUAPI_UnLock triggers a callback to the PTRUAPI_Lock supplied callback routine. (This callback routine is shared by PTRUAPI_Lock and PTRUAPI_UnLock).

The PTRUAPI_UnLock call is synchronous. That is, it returns only after the unlock operation has completed, which is typically a short amount of time. If a data read is in progress when PTRUAPI_UnLock is called, that read operation will be cancelled.

Comments

When the PTRUAPI_UnLock function is called, another application (or another route) may obtain the device lock.

Syntax

```
BOOL FAR PASCAL PTRUAPI_UnLock(HANDLE hConn);
```

Parameter

Parameter	Description
hConn	API device handle

Return Value

`TRUE` (Success) or `FALSE` (Failure)

Sample Application

A

In This Appendix This appendix explains how to use the sample PTR UAPI application. The following section is included:

Using the Sample Client Application 48

Using the Sample Client Application

A sample application that uses the PTR UAPI, PTRUAPI.EXE, is provided in the following directory on the installation CD:

\\32\PDK\PTRUAPI

The following files are part of PTRUAPI.EXE:

ONFIGDIALOG.CPP
ONFIGDIALOG.H
PTR SAMPLE APPLICATION.CPP
PTR SAMPLE APPLICATION.H
PTR SAMPLE APPLICATION.RC
PTR SAMPLE APPLICATIONDLG.CPP
PTR SAMPLE APPLICATIONDLG.H
PTRUAPI.DSP
PTRUAPI.H
PTRUAPI.LIB
RESOURCE.H
STDAFX.CPP
STDAFX.H
PTR SAMPLE APPLICATION.ICO
PTR SAMPLE APPLICATION.RC2

Description

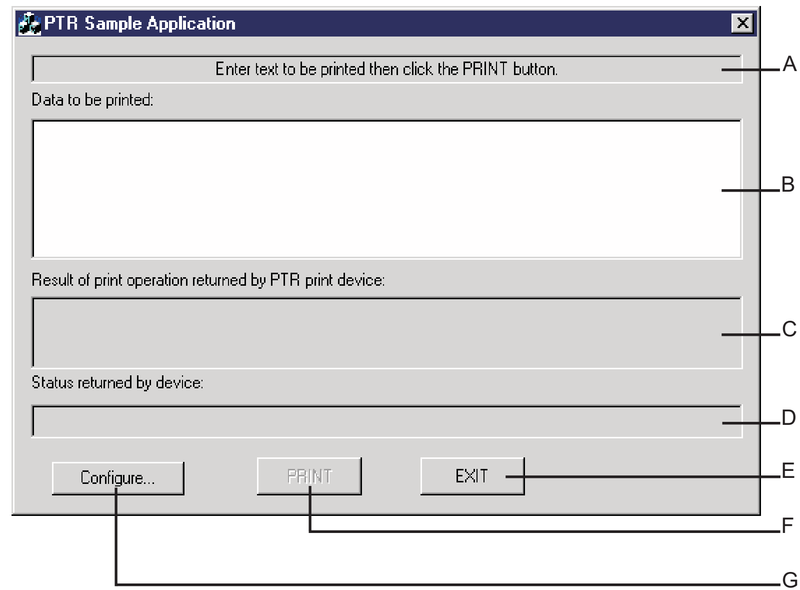
Design your PTR UAPI application to function the way you want it to. PTRUAPI.EXE is intended to be an example only.

While the PTR UAPI functions provide the means for your application to communicate with the printer, the fields and most of the buttons on the PTRUAPI.EXE dialog box (shown on the opposite page) do not correspond *directly* to any PTR UAPI functions. Exit (button E), however, invokes both PTRUAPI_Close( ) and PTRUAPI_Term( ).

In the diagram on the next page, field A in the dialog box instructs you to enter your text string into field B. When you enter the text into the field B, you must enter it as one continuous string with no paragraph returns. PTRUAPI.EXE does not send the string to the device until you click Print (button F).

Field C displays the success or failure of the call, and field D shows the status of the device. Button G, Configure, allows you to specify the Open Name that is used in the host path configuration (see “Running PTRUAPI.EXE” on page 50).

The dialog box for PTRUAPI.EXE is shown as follows:



How the Dialog Box Corresponds to the PTR UAPI Functions

Opening PTRUAPI.EXE invokes PTRUAPI_Open( ), and exiting PTRUAPI.EXE invokes both PTRUAPI_Close( ) and PTRUAPI_Term( ). Then, the application invokes either PTRUAPI_Read( ) or PTRUAPI_Write( ); if PTRUAPI_Write( ) is invoked, then PTRUAPI_Status( ) is also invoked.

PTR UAPI Functions Called

PTR SAMPLE APPLICATIONDLG.CPP shows how to call all of the functions in PTRUAPI.EXE.

Running PTRUAPI.EXE

To run PTRUAPI.EXE, do the following:

1. Configure a PTR route as described in “Configuring a PTR Route to Use the PTR UAPI Library” on page 3.

   Use the following as the Open Name in the host path configuration:

   ```
   PBS32PRINT
   ```

2. Copy PTRUAPI.EXE to a local drive from the installation CD.

3. Double-click PTRUAPI.EXE from its location on the local drive.

4. Click Configure and type the Open Name in the text line of the Printer Path Name dialog box.

5. Enter a text string in field B, and then click Print.

6. Check fields C and D of PTRUAPI.EXE to see the status of the PTR UAPI call and the status of the device.

   > **Note:** Because PTRUAPI.EXE is a generic application intended to illustrate the PTR UAPI functions, it is not designed to work specifically with *your* printer. Therefore, you may not get your intended results when you run the sample application.

Troubleshooting

B

In This Appendix

This appendix explains general troubleshooting procedures. The following sections are included:

General Troubleshooting Procedures 52

General Troubleshooting Procedures

If you have problems running PTR UAPI, complete the following steps:

1. Check to see that your system meets the minimum hardware and software requirements necessary to use the product.

2. Check your PTR configuration options. Most problems are caused by incorrect configuration. In addition, make sure of the following:

   - PTR should be started, and at least one route should be configured. The PTR Quick Status should display a green check mark (which means that both paths are working) as the status of the route that you want to use.
   - Your printer queue path is configured properly for the printer you are using.
   - You are using the APIThr32.hff host filter.
   - You are using the API—User API External Interface Library path template (for the host path).

3. Check your cable connections and make sure that they are securely attached.

4. Check your system. You may be using peripheral equipment or other software that may not be compatible with this product. Try disabling some of the other memory-resident programs.

5. If you determine that PTR is working properly, check the following things in your application:

   - The files PTRUAPI.LIB and PTRUAPI.H are present in your application’s project.
   - Your application’s project is linked to PTRUAPI.LIB.
   - Your application calls the PTR UAPI functions in the appropriate order. For example, you do not call PTRUAPI_Read( ) before PTRUAPI_Open( ).
   - If your application calls PTRUAPI_Open( ) when you open the application, be sure that it closes the device with PTRUAPI_Close( ) before exiting.

C

RTE Wedge Implementation

In This Appendix

This appendix provides a description of RTE Wedge Implementation in PTR for using devices such as credit card readers or passport readers.

Implementing the RTE Wedge 54

Implementing the RTE Wedge

The RTE wedge implementation is really a extension of the PTRUAPI. Its purpose is to allow customer applications written to the PTRUAPI to access particular devices, such as credit card readers or passport readers, attached to the PC with PTR (for example, when you check in for your flight at the airport, your frequent-flyer card might be scanned, or your passport might be read electronically). In one particular environment, Resa CREWS, the access to these devices is bundled, and the wedge code separates things by type of device.

The RTE wedge implementation in PTR is contained in the API.EIL library. You can replace the existing API.EIL file (in the INFOCN32 directory) with the wedge implementation, which is a superset of the existing library. The wedge-capable API.EIL can be used with either INFOConnect version 3.x or version 5.0.

The wedge is accessed via the PTRUAPI, and is transparent to applications. No special configuration is needed. Wedge support is activated at runtime by passing in a special device name. The format is identical to that used for the RTE wedge in the LinkUp UTS Workstation product. The API name to open an RTE wedge device is a string that consists of:

`"WEDGE" + `*`wedge number + device type + OpenDeviceName`*

Where:

"WEDGE" = A fixed prefix string, and is case insensitive.

wedge number = A single ASCII character that identifies the wedge (in case you have multiple RTEs on the workstation). This should be a digit between 1 and 9. If you have a single RTE, you can simply use 1.

device type = A single ASCII character that identifies which device on the RTE you want to access. Values, which need to match the Resa firmware, include: `C` for MSR/Credit Card Reader, `O` for Optical/Passport Reader, `A` for ATB, and so on.

OpenDeviceName = the name configured in the API Host Path as the OpenName for that PTR route.

There is a single PTR route for each RTE device. The existing APITHR32 host filter should be used, along with the existing LHResa.EIL. The host path should be created with the API path template. The route printer type should be Special. These are all typical settings for accessing readers via PTR. The application that wants to use the RTE device issues the PTRUAPI_Open commands with the expanded device name, and receives a PTRUAPI handle. All of the other PTRUAPI function calls (for example, Read, Write, Status, Close, and so on) can be made normally using that device handle. The PTR Route will be active as long as at least one wedge device is open.

Examples of PTRUAPI_Open name usage:

Example	Description
`WEDGE1CRTE`	Opens the MSR on a Route with “RTE” as the OpenName.
`WEDGE1ORTE`	Opens the OCR on a Route with "RTE" as the OpenName.
`wedge2Opassport`	Opens an OCR on a second Route with “passport” as the OpenName.

As an alternative to using the prefix on the Device Open Name, a new PTRUAPI function, PTRUAPI_OpenWedge( ), has been added. PTRUAPI_OpenWedge uses the following three parameters:

DeviceOpenName = A string, as configured in the API host path in the PTR route.

device type = A single character that identifies the particular device on the wedge.

wedge number = A single character from 1 to 9.

When using the PTRUAPI_OpenWedge call in an application, the results are the same as they would be if you used the same values for a 7-character prefix on the string passed to the PTRUAPI_Open function, except that it makes the wedge distinction clearer.

Each application developer can select his or her preferred method. To use the PTRUPAPI_OpenWedge function, you need to use the updated PTRUDLL.DLL, PTRUDLL.LIB, and PTRUAPI.H files. The updated API.EIL is needed in both cases.

Note: No device resets are necessary — they are automatically performed by the LHResa library when you specify “RTE” as the device type — so the PTRUAPI applications only need to issue “reads” to receive the Reader data. The firmware framing characters are stripped off, and the application receives normal reader data.

BPTP Wedge Support

D

In This Appendix

This appendix explains the configuration of a BPTP Wedge. The following sections are included:

Using BPTP Wedge 58

Using BPTP Wedge

To access two daisy-chained BPTP (Boarding Pass and Ticket Printer) devices individually via the PTRUAPI, you set the configuration of PTR similar to the way you would access a single device via the PTRUAPI: you configure a single PTR route, with the APITHR32.HFF host filter, with a single LHBPTP (or LHMon) printer queue pointing to the single serial port where the printers are connected, and use a single API host path.

You set the Device Open Name in the host path to whatever you like. As an example, let's call it "DAISY". The PTRUAPI application can then make one or two PTRUAPI_Open calls with a special prefix in the open device name string. The prefix is in the form:

`"LHBPTP" + `*`wedge number + device type`*

Where:

"LHBPTP" = A case-insensitive fixed string.

wedge number = A single ASCII character that identifies the wedge. This should be a digit between 1 and 9.

device type = A single ASCII character which identifies which of the two devices that you want to access: `A` is the ATB (Boarding Pass Printer); `B` is the BTP (Baggage Tag Printer).

The Device Open Name is the name configured in the API Host Path as the OpenName for that PTR route. Therefore, the whole API name that the application should use is:

`"LHBPTP" + `*`wedge number + device type + OpenDeviceName`*

In our example, `"LHBPTP1ADAISY"` is the ATB, and `"LHBPTP1BDAISY"` is the BTP.

The application then has independent PTRUAPI handles (returned by PTRUAPI_Open) to control each device's data stream separately (for both reading and writing). Changes in the serial signals will be seen in the PTRUAPI Status of both handles. The only status information which is unique to each device is the out-of-paper indication.

Airline Native Wedge Support

In This Appendix

This appendix explains the use of Airline Native Wedge Support. The following sections are included:

Using Airline Native Wedge Support 60

Using Airline Native Wedge Support

When a reader is attached to the workstation, your application can access it via the PTRUAPI if PTR is configured to control the device. Your application can then receive the whole datastream that reader outputs. In the case of airline wedge readers, there may be multiple readers (such as OCR, MSR, and LSR) bundled together in one physical unit. If want your application to use only one of the those readers, or find it advantageous to access the readers individually, set up PTR as described below.

Because PTR can be set up to separate the datastreams into the individual device-specific portions, your application can receive the reader data via different PTRUAPI device handles. In this case, no protocol framing characters (from the reader's firmware) are passed to the application (just the true datastream from the reader) when you use the PTR airline native wedge support. However, if you control a serially-attached wedge device directly, there will likely be some protocol framing characters.

The PTR configuration requirements for the airline wedge devices attached to a local serial port are:

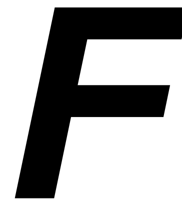
- A PTR route, which should have Host Filter “APIThr32.HFF”, and its Device Type should be “Special”.
- A Printer Queue, with the Path Template “PTR_COM”. The printer queue is configured to point to a single COMx port (i.e., COM1, COM2, etc.), where the reader is connected.
- A Host Path, with the queue Path Template “API”.
- A DeviceOpenName that you configure using any string you like, except LHWDG.

After PTR is running, the application can open, using PTRUAPI_Open, the following devices:

- `"LHWDG1M"` + *`string`* in DeviceOpenName for MSR
- `"LHWDG1O"` + *`string`* in DeviceOpenName for OCR
- `"LHWDG1L"` + *`string`* in DeviceOpenName for LSR

Each time a native wedge device is opened or closed from the PTRUAPI you will see a line in the INFOConnect trace indicating that a wedge open or close occurred. This confirms that native wedge support is being used. If you don't see a trace line indicating “wedge”, you are connected to the COM port without any special processing (i.e., device datastream separation).

Videcom Wedge Support



In This Appendix

This appendix explains the use of the Videcom Wedge device. The following sections are included:

Using Videcom Wedge 64

Using Videcom Wedge

When using the Videcom WGE (wedge) device, its datastream can be separated into relevant parts using a special PTRUAPI_Open name (in a manner similar to the way the Resa RTE Wedge is supported).

The Videcom WGE device is accessed via the PTRUAPI using the following string:

`"VCWGE" + n + t + PTRUAPIDeviceOpenName`

Where:

"VCWGE" = A case-insensitive fixed string.

n = A single ASCII character that identifies the wedge. This should be a digit between 1 and 9.

t = A single ASCII character that identifies which wedge device type you want to access: `M` for MSR/Credit Card Reader, `O` for OCR data, and `L` for LSR/SCN data.

PTRUAPIDeviceOpenName = The name configured in the API host path as the DeviceOpenName for that PTR route.

This means that a single WGE device with one PTR route can be divided into three separate PTRUAPI handles. Let's say that your PTR route has a single WGE device, with "Reader" configured as the DeviceOpenName in the host path. You could use the following examples:

Example	Description
`VCWGE1MReader`	Obtains just the MSR data on the PTR route with Reader configured as the DeviceOpenName.
`VCWGE1OReader`	Obtains just the OCR data on the PTR route with Reader configured as the DeviceOpenName.

Glossary

host filter
A link between the host and an output device that initializes the host connection, manages data for the output device, and sends the data to the output device. Host filters are specific to the host and output device.

INFOConnect Manager
The application that runs in the background each time you start an INFOConnect product and that controls interaction between accessories and transports.

You can also use this application to create, modify, and delete paths; set administrator and user-level security; control user access to configuration; and set user preferences.

path
A named set of configuration options that define an INFOConnect communication connection between a PC and a host (host path) or between PTR and an output device (printer queue path). A path consists of a path template and other configuration data associated with the library(s) listed in the path template.

route
A named set of configurations that describe a complete host to peripheral connection, consisting of the following three parts: host path, host filter, and printer queue path.

Index

A

Abbreviations, used in this guide ix
airline
 native wedge support 60-61
 wedge device 60, 61
 wedge reader 60
API path template 3
APITHRU32.HFF host filter 3

C

Callbacks
 ReadCB( ) 9-10
 StatCB( ) 14-16
 WriteCB( ) 12-13
Configuring a PTR route, using the PTR UAPI library 3
Configuring the host path 3
Conventions, used in this guide viii

D

device, airline wedge 60, 61
Documentation
 conventions viii
 related guides and online help x

F

Functions
 listed 4
 PTRUAPI_Close( ) 4, 20
 PTRUAPI_Init( ) 4, 6
 PTRUAPI_Open( ) 4, 7
 PTRUAPI_Read( ) 4, 8-10
 PTRUAPI_Status( ) 4, 14-16

Functions, *continued*
  PTRUAPI_Term( ) 4, 21
  PTRUAPI_Write( ) 4, 11-13

H

Host path, configuring 3

L

Linking your application to the PTR UAPI library 3

O

Open Name
  host path configuration 3
  lpName parameter 3
  related to the PTRUAPI_Open( ) function 3

P

Path templates
  API path template 3
  host path template 3
Prerequisites 2
PTR route, configuring 3
PTR UAPI functions
  listed 4
  PTRUAPI_Close( ) 4, 20
  PTRUAPI_Init( ) 4, 6
  PTRUAPI_Open( ) 4,10
  PTRUAPI_Read( ) 4, 8-10
  PTRUAPI_Status( ) 4, 14-16
  PTRUAPI_Term( ) 4, 21
  PTRUAPI_Write( ) 4, 11-13
PTR UAPI library, linking your application 3
PTR UAPI sample application 2
PTRUAPI.H 3
PTRUAPI.LIB 3
PTRUAPI_Close( )
  description 4, 20
PTRUAPI_Close( ), *continued*
  example 20
  illustrated 20
PTRUAPI_Init( )
  description 4, 6
  example 6
  illustrated 6
PTRUAPI_Open( )
  description 4, 10
  example 7
  illustrated 7
PTRUAPI_Read( )
  description 4, 8
  example 9, 12, 24, 25, 26
  illustrated 8
PTRUAPI_Status( )
  description 4, 14
  example 14, 22
  illustrated 14
PTRUAPI_Term( )
  description 4, 21
  example 21
  illustrated 21
PTRUAPI_Write( )
  description 4, 11
  illustrated 11

R

reader, airline wedge 60
ReadCB( ) callback 9-10
Related documentation x
Requirements to use PTR UAPI 2
Route, configuring 3
RTE wedge 54-56

S

Sample application 2
  description 28-29
  location 28
  running 30

support, wedge
  airline native 60-61

T

Troubleshooting, general procedures 32

U

Using PTR UAPI
  overview 3
  requirements 2
Using the PTR UAPI library with PTR 3

W

wedge
  airline native 60-61
  device, airline 60, 61
  implementation, RTE 54-56
  reader, airline 60
  RTE 54-56
WriteCB( ) callback 12-13