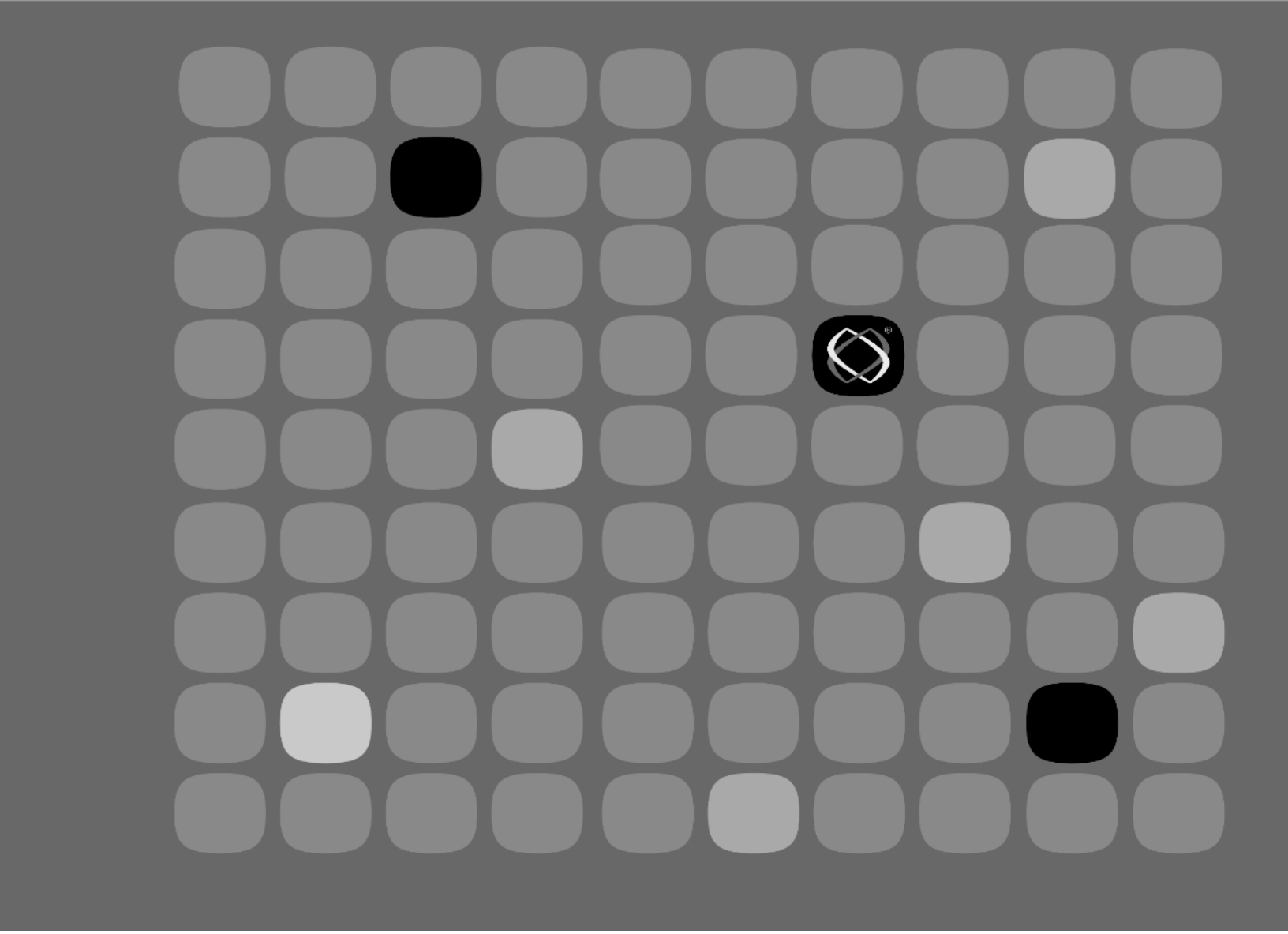


Programmer's Reference

Attachmate®

INFOCONNECT®

Enterprise Edition

PTR

Print and Transaction Router

Host Filter Development Kit

Copyrights and Notices

Attachmate® INFOConnect® Enterprise Edition

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

This guide contains general information about Print and Transaction Router API host filters, including descriptions of each API call and its callback mechanism. This guide and the sample source code form the Host Filter Development Kit.

The following sections are included in this preface:

Prerequisites

To use the Host Filter Development Kit you must have the following:

- Fluency in the C Programming Language.
- Windows® DLL development experience.
- INFOConnect™ Development Kit (IDK) version 3.0 or newer.
- A C compiler package that contains a Microsoft® Windows Development Kit. We recommend Microsoft Visual C++™.

Conventions

This guide uses the following documentation conventions:

- All text that you type on a screen or messages and prompts that appear on the screen are displayed in `this type style`.
- References to selections, parameters, dialog boxes, window names, and menus are displayed with initial-letter capitalization. For example, "use the Save On Cancel option to save part of a file already downloaded."
- Keys that have names consisting of more than one character appear within angle brackets as in <Tab> or <Space bar>.
- If a word or character stands for something other than its apparent meaning (for example, variable names), it appears in italic characters. For example, you might see the following:

  `delete` *`filename`*

  In this example, *filename* represents the name of a file.

Installing the PTR Host Filter Development Kit

1

In This Chapter

This chapter provides an overview of the PTR Host Filter Development Kit and installation instructions. The following sections are included:

PTR Host Filter Development Kit: Overview

The PTR Host Filter Development Kit contains the sets of source code for development based on the language used. These kits were developed using Microsoft Visual C++ version 1.52. Version 1.50 or newer is recommended.

Compilers

Use one of the following compilers with the PTR Host Filter Development Kit:

- ANSI-compatible C compiler, where the features of C++ are not used
- Microsoft Visual C++ with the Microsoft Foundation Classes (MFCs)

Prerequisites

The following software is required before you install the PTR Host Filter Development Kit:

- For Windows 95 or Windows NT, these kits were developed using Microsoft Visual C++ version 2.2.
- INFOConnect and the INFOConnect Development Kit (IDK)

The provided source code is intended as a tool base from which you can build your own host filter. It is not complete until you add your processing instructions. Select the options that make the resulting host filter compatible with your development tools.

The .MAK files generated by Microsoft Visual C++ are included, as well as external makefiles that make multiple targets and move the resulting executables into the destination directory where they'll be invoked. These executables can be modified, renamed, or merged for use with external make utilities.

> **Note:** Some runtime library calls in the source code may also require modification to reflect specific nuances of your compiler, runtime library, and tool system.

Installing the PTR Host Filter Development Kit

To install the PTR Host Filter Development Kit (PDK) for the Windows 98 SE, Windows NT, Windows 2000, or Windows XP environments, complete the following steps:

1. Close all open applications.

   If any INFOConnect applications are open when you perform the installation, errors might occur. In addition, if you're using Windows 98 SE and a DOS window is open, the Setup Utility will not be able to restart the PC if any system files are upgraded.

2. Insert the CD into a CD-ROM drive.

   If you're running Windows 98 SE, the Setup Utility runs automatically. If you're running Windows NT or newer, run SETUP.EXE from the CD's root.

   > **Note:** If you're installing from a network CD-ROM drive, use My Computer or File Manager to access that CD-ROM drive and then run SETUP.EXE from the CD's root.

   If you're installing from a CD image on a file server, use My Computer or File Manager to access the folder on the file server where the CD image was created and then run SETUP.EXE from that folder.

3. When prompted for your Product Access Code, type the number exactly as it appears on the sticker on your package and click Next. The Product Access Code begins with 98-.

4. When prompted about the number of users, click Next.

   If you previously installed an INFOConnect product, you cannot change your installation mode. Your current installation mode is automatically selected.

5. On the Destination Directory dialog box, click Install to use the default folder (C:\INFOCN32), or type the name of the folder where you want to install INFOConnect Connectivity Services (up to 28 characters), and then click Install.

If you previously installed a 32-bit INFOConnect product, the destination folder appears on the screen, but you cannot change it. Any new products are automatically installed in the existing folder.

> **Note:** If you previously installed a 16-bit version of these products, or if you're running multiple operating systems on the same PC, be sure to install these products in a different folder than the one you used previously.

If the group window that appears during installation obscures your view of the status bar, you can minimize or close it.

If system files had to be upgraded, you are prompted to restart the PC. Click Yes. If any open applications cannot be closed automatically, you are prompted to close them.

If you click No at this prompt, a message box appears, indicating that the installation is complete. Click OK. You must restart the PC manually before you run Accessory Manager.

If the restart prompt does not appear, a message appears, indicating that the installation is complete. Click OK.

If you installed an emulator, you are prompted to run Accessory Manager and create a session. Respond to the prompt.

Directory Structure

Each version of the Host Filter Development Kit contains four subdirectories:

- INCLUDE
- SOURCE
- LIB
- SAMPLE

INCLUDE This subdirectory contains header (.H) files used when compiling host filter sources. Add this directory as a parameter to your C compiler and resource compilers, or add it to your INCLUDE environment.

SOURCE This subdirectory contains runtime (.C or .CPP) sources used to build the host filter libraries. Using this source code with debugging tools, you can step through the instructions of each host filter library call. Debug information in HOSTFLTD.LIB references these source files.

LIB This subdirectory contains library files built from the base code sources. Specify this directory on the command line of your linker or add it to your LIB environment. Example makefiles are included so that you can rebuild the libraries with debug information appropriate for your tools.

SAMPLE This subdirectory contains sample sources, headers, and makefiles unique to the NULL and THRU host filters. The NULL host filter has all the defined entry points, but is not operational. Assigning NULL.HFF to a route is functionally equivalent to disabling it in the PTR configuration. The THRU host filter passes data from host to queue (and conditionally, from queue to host as well) without modification. In addition to being functional in its own right, it includes the base code necessary to design and produce a working host filter. Modifying the THRU source code is the fastest way to develop your own unique host filter.

The Libraries

The Host Filter Development Kit includes three libraries:

- HOSTFLTD.LIB
- HOSTFLTT.LIB
- HOSTFILT.LIB

HOSTFLTD.LIB This library is a full debugging version. It is not optimized and contains debugging information. It also produces a trace log file (C:\DEBUGLOG.TXT).

HOSTFLTT.LIB This library is optimized and free of debugging information, but contains the code to produce a trace log file (_DEBUGLOG is defined).

HOSTFILT.LIB This library is the release version; it is optimized and free of debug information. It is not capable of producing debug logging information.

Library Source Code

The sample source code is built with the following module organization.

Source Modules

The following modules are included in the PDK:

- DEBUGLOG
- LIST (ANSI C version only)
- PTRAPP
- PTRDLL
- PTRENTRY
- PTRSESS
- PTRTIMER
- STDAFX (Visual C++ version only)
- TASK
- THRU

> **Note:** Each module includes a C or C++ source file (.C or .CPP) and a header file (.H) required for compilation.

Additional Header Files

The following header files are included in the PDK:

- COMMON.H
- ICPTRSTS.H
- ICMEM.H
- ICLIB.H
- CERROR.H
- CSTATUS.H
- CCONFIG.H
- 16OR32.H (Windows 95 and Windows NT)

Windows Build Files

The following Windows build files are included in the PDK:

- THRU.DEF
- THRU.RC

Accessing the Guide

In addition to the printed copy of the Print and Transaction Router API Programmer Reference, an electronic (.PDF) file of this guide is provided. To view or print the guide, you must install and run Adobe® Acrobat® Reader, which is installed with all INFOConnect products.

Installing Acrobat Reader

To install Acrobat Reader, complete the following steps:

1. Go to the ACROREAD subdirectory in the INFOConnect directory.
2. Double-click RS405ENG.EXE.
3. Respond to the prompts on the screen.

Running Acrobat Reader

To run Acrobat Reader, follow these steps:

1. Click the Start button, point to Programs, point to Adobe Acrobat, and click Acrobat Reader 4.0.
2. From the File menu, click Open.

   On the Open dialog box, go to the ACROBAT subdirectory in the INFOConnect directory and open the PDK40PR.PDF file.

Understanding Print and Transaction Router

2

In This Chapter

This chapter explains Print and Transaction Router and how it works. The following sections are included:

Print and Transaction Router: Overview

The Host Filter Development Kit is a tool to create customized host filters for Print and Transaction Router (PTR). To develop a custom host filter, you should understand how PTR works, be able to read and interpret sample source code, assess your site requirements, and write or adapt source code to meet those requirements. In most cases, only minor modification of the sample source code can produce a suitable custom host filter.

Following are explanations of some of the concepts with which you should be familiar before writing a customized host filter.

DLL Structure

An abbreviated diagram of the Print and Transaction Router DLL structure appears below. One route is shown.

<table>
<tr><td colspan="3">PTR.EXE</td></tr>
<tr><td colspan="3">PTRAPI.DLL</td></tr>
<tr><td colspan="2">INFOConnect Manager</td><td>Host Filter (.HFF)</td></tr>
<tr><td>INFOConnect Host SL</td><td>INFOConnect Queue SL</td><td rowspan="3"></td></tr>
<tr><td>:</td><td>:</td></tr>
<tr><td>INFOConnect Host EIL</td><td>INFOConnect Queue EIL</td></tr>
</table>

Linking

All entry points in the API interface use the FAR PASCAL calling convention (the CALLBACK macro) to support dynamic linking. The API consists of two types of calls:

- Exported host filter functions are functions required within the host filter and called by PTR. If the host filter does not export *all* of these calls, PTR does not initialize the host filter. These must also be defined as `EXPORT` or be exported *by name* in the module's definition (.DEF) file. Refer to Chapter 3, "Exported Host Filter Functions," beginning on page 30 for more information.
- Imported API functions are functions contained within PTR for support of the communications routes and are called by the host filter. You will need to import only the calls required by the exported host filter functions. Refer to Chapter 4, "Imported API Functions," beginning on page 49 for more information.

Routes

The basic connection unit within PTR is the route. Every active route must be configured in the PTR Quick Configuration. From the INFOConnect Manager, select the Configure Packages menu option, then highlight the PTR package and click the Quick Config button.

The route specifies which host filter is used, names for the host and queue INFOConnect paths, a timeout value, and translation options. PTR handles the data translation. The other configuration options are passed to the host filter via the InitializeRoute call. The host filter receives a separate InitializeRoute call for each route that specifies that host filter.

Initialization

During InitializeRoute, the host filter determines the measures appropriate to initialize each host and queue session. It usually allocates any internal route structures, then attempts to open the default host session. It may attempt to open the default queue and initialize the timer.

Sessions

Each host filter processes data from zero to one *host session* and zero or more *queue sessions*. The host session is typically the source of the printer or transaction data, and the queue session is the potentially shared destination of the filtered data. Each session within a route is one of these two types, and each type of session uses different API calls for handling initialization or data transfer.

For example, the host could be an input device connected to the workstation, and the queue could refer to a shared mainframe connection where the input data from several such devices is to be sent. The purpose of the host filter in this case would be to format a transaction from the input device "host" before it is forwarded to the mainframe "queue."

Establishing Sessions

Opening Host Sessions

Host sessions are opened with the HostRequest call. HostRequest returns a handle to the host session, but this handle should not be used until the open is complete. Using a session handle before the host session is open could damage your data.

Host session establishment errors are indicated immediately by a NULL session handle, or subsequently via the HostClosed call. After either of these, the HostRequest call can be retried.

If successful, the PTR API responds with a call to the host filter's HostOpen function. At this point, all transitional states have concluded, the host session is open and it is safe to use the host session handle for sending or receiving data.

Opening Queue Sessions

Queue sessions are opened with the QueueRequest call. The major difference is that multiple queues can be opened by the same route. QueueRequest returns a handle to the queue session, but this handle should not be used until the open is complete. Using a session handle before the queue session is open could damage your data.

Queue session establishment errors are indicated immediately by a NULL session handle, or subsequently via the QueueClosed call. After either of these, the QueueRequest call can be retried.

If successful, the PTR API responds with a call to the host filter's QueueOpen function. At this point, all transitional states have concluded, the queue session is open and it is safe to use the queue session handle for sending or receiving data.

PTR Queue Statuses

For the purposes of this program, a complete set of data sent to a queue session is called a *document*. A document can represent a physical document or a single transaction or message.

Queues require special status transactions to prevent interleaving of documents from different sources on the same output session. These statuses are labeled *Start of Document (SOD)* and *End of Document (EOD)*. If data is not enclosed with an SOD and an EOD, it generates an error.

Similar requirements apply to receiving data from a queue session. PTR must have an indication of which route should handle unsolicited queue data. An attempt to receive data on a queue session that is neither locked nor processing a document results in an error.

Start of Document (SOD)

The host filter indicates the start of a document or message by sending the IC_PTR_SOD status (see sample source code) to the queue session. The PTR API responds with a call to StatusFromQueue with the appropriate status value.

- IC_PTR_PRINTERREADY indicates that the queue session is available for data transfer.
- IC_PTR_BUSY indicates that another route is currently accessing the queue. The host filter must determine how to handle this contingency. The SOD status can be reissued to determine whether the queue is again available. See the QueueAllocate() call for more information.
- Any other status indicates that the queue is not able to receive data for the indicated reason.

End of Document (EOD)

The host filter indicates the end of a document by sending the IC_PTR_EOD status (see sample source code) to the queue session. The host filter must then issue another IC_PTR_SOD and wait for the IC_PTR_PRINTERREADY status before sending the next document or message to the queue session. The PTR API responds with a call to StatusFromQueue with the IC_PTR_ACKEOD status value.

QueueLock

QueueLock is an optional call. Its purpose is to ensure that other routes cannot access the queue between documents. All other routes receive IC_PTR_BUSY in response to QueueLock or an IC_PTR_SOD status. It also allows a route to receive and process data from the queue session outside of a specific document context.

The lock is released by calling QueueUnlock after the end of the final document. QueueUnlock implies EOD, if a status of IC_PTR_EOD has not already been explicitly issued.

QueueAllocate

QueueAllocate is an intelligent queue locking mechanism for queues which are expected to be shared most of the time. It functions similarly to QueueLock, but allows the caller to specify additional parameters for *priority* and *maximum wait*.

If the queue is available or the maximum wait is set to zero, then QueueAllocate operates identically to QueueLock. Calling QueueLock and calling QueueAllocate with *priority* and *maximum wait* values of zero are functionally equivalent.

If the queue is busy and the *maximum wait* is nonzero, QueueAllocate holds the request within PTR, and returns a value of IC_ERROR_QUEUEWAITING. When the route using the queue completes its printing by sending an EOD status or calling QueueUnlock, then PTR allocates the queue session to the waiting route which has specified the highest priority. The route is then notified with the IC_PTR_ACKLOCK status.

If the *maximum wait* (in seconds) elapses before the printer becomes available, then a pending wait is terminated and removed. The route is notified with the IC_PTR_LOCKTIMEOUT status. The host filter may then take corrective action. We suggest you increment the priority and reissue the QueueAllocate request or take abortive action when the priority reaches a reasonable ceiling.

If the *maximum wait* is set to -1, PTR will never timeout the waiting request.

The host filter can cancel a pending QueueAllocate request by calling QueueUnlock.

Transferring Data

The host filter typically transfers data in three operations:

- Receives data from the host session
- Processes the data
- Sends the data to the queue session

The host filter can also process acknowledgments, statuses, and other data traffic from the recipient (queue) back to its origin (host). However, due to the shared nature of the queue session(s), the host filter can only attempt to receive data from a queue during a document transfer (between IC_PTR_PRINTERREADY and IC_PTR_EOD statuses) or while the queue is locked by the route.

Other sources, such as spooled files and additional dynamic links established via another API, can also be used for input or output, but those operations are beyond the scope of the PTR API.

Receiving Host Data

To initialize data receipt from an open host session, call RcvFromHost. Data is returned via the DataFromHost entry point, which can be called from within RcvFromHost.

Only one outstanding RcvFromHost call is permitted per session; subsequent calls result in an error.

In absence of data, the RcvFromHost call returns as soon as receipt is enabled; it does not wait for data to arrive. Nor is it a polling type of function, since it returns no indication of whether data is actually available on the session.

Receive errors are indicated immediately by an error value returned from the RcvFromHost call or subsequently via the RcvErrorFromHost function. After either of these, the RcvFromHost call can be retried.

RcvFromHost is terminated without indication when the host session is closed.

DataFromHost

DataFromHost passes received host session data to the host filter. The host filter normally processes this data and sends it to the queue session, then reissues a RcvFromHost call to collect the next host message.

PTR reuses the provided buffer for subsequent receives from the host connection, so the host filter must copy or complete processing of this data before returning from DataFromHost or calling RcvFromHost again.

Receiving Queue Data

Due to the sharing nature of the queue connection, there are special requirements to receive data from a queue session. The queue must be locked or have an IC_PTR_PRINTERREADY status due to SOD. See "PTR Queue Statuses" on page 13 for more information.

To initialize data receipt from a locked queue session or during a document transfer, call RcvFromQueue. Data is returned via the DataFromQueue entry point, which can be called from within RcvFromQueue.

Only one outstanding RcvFromQueue call is permitted per session; subsequent calls result in an error.

In absence of data, the RcvFromQueue call returns as soon as receipt is enabled. It does not wait for data to arrive and does not act as a polling type of function, since it returns no indication of whether data is actually available on the session.

Receive errors are indicated immediately by an error value returned from the RcvFromQueue call or subsequently via the RcvErrorFromQueue function. After either of these, the RcvFromHost call can be retried.

RcvFromQueue is terminated without indication when QueueUnlock is called, when an IC_PTR_EOD status is sent to an unlocked queue, or when the queue session is closed.

DataFromQueue

DataFromQueue passes received queue session data to the host filter. The host filter normally processes this data (perhaps sending it to the host session) and then reissues a RcvFromQueue call to collect the next Queue message.

PTR reuses the provided buffer for subsequent receives from the queue connection, so the host filter must copy or complete processing of this data before returning from DataFromQueue or calling RcvFromQueue again.

Sending Data to the Host

SendToHost

SendToHost sends data to an open host connection. Successful completion of the call is indicated by the SendToHostDone entry point, which can occur from within SendToHost. PTR copies the data before returning from SendToHost, so the host filter's provided buffer can be reused or freed immediately upon return.

Only one outstanding SendToHost call is permitted per session; subsequent calls result in an error.

SendToHost does not wait for the data to be sent. It returns as soon as the data is queued to be sent.

Transmit errors are indicated immediately by an error value returned from the SendToHost call or subsequently via the DataErrorFromHost function. After either of these, the SendToHost call can be retried.

SendToHost is terminated without indication when the host session is closed.

Sending Data to the Queue

SendToQueue

Due to the sharing nature of the queue connection, there are special requirements to send data to a queue session. The data must be enveloped in SOD/EOD statuses. See "PTR Queue Statuses" on page 13.

SendToQueue sends data to an open queue connection with an IC_PTR_PRINTERREADY status. Successful completion of the call is indicated by the SendToQueueDone entry point, which can occur from within SendToQueue. PTR copies the data before returning from SendToQueue, so the host filter's provided buffer can be reused or freed immediately upon return.

Only one outstanding SendToQueue call is permitted per session; subsequent calls result in an error.

SendToQueue does not wait for the data to be sent. It returns as soon as the data is queued to be sent.

Transmit errors are indicated immediately by an error value returned from the SendToQueue call or subsequently via the DataErrorFromQueue function. After either of these, the SendToQueue call can be retried.

SendToQueue is terminated without indication when QueueUnlock is called, when an IC_PTR_EOD status is sent to an unlocked queue, or when the queue session is closed.

Terminating PTR

PTR is terminated by selecting the "Close" option from the menu that appears when you click the PTR icon. PTR calls TerminateRoute for each active route. PTR closes all sessions and no other PTR API calls are permitted at that point. The host filter should free all resources associated with each route before the host filter and PTR are removed from memory.

> **Note:** The "Close" option can be disabled in the PTR Quick Configuration.

Terminating a Host Session

To terminate a host session, call HostRelease. PTR calls HostClosed when the session is closed.

At this point, the host session handle is invalid and no further calls to PTR should be made with that handle. Using a session handle while the session is closing could damage your data. Also, trying to reopen the session before HostClosed has been returned can damage your data.

HostClosed is called if the host path terminates, whether or not an error has been indicated or the host filter has called HostRelease. At that point, the host session handle is invalid and no further calls to PTR should be made with that handle.

If TerminateRoute has not been called, the host filter can attempt to reopen the host session by calling HostRequest during or after HostClosed.

Terminating a Queue Session

To terminate a queue session, call QueueRelease. PTR calls QueueClosed when the session is closed.

At this point, the queue session handle is invalid and no further calls to PTR should be made with that handle. Using a session handle while the session is closing could damage your data. Trying to reopen the session before QueueClosed has been returned can also damage your data.

QueueClosed is called if the queue path terminates, whether or not an error has been indicated or the host filter has called QueueRelease. At that point, the queue session handle is invalid and no further calls to PTR should be made with that handle.

If TerminateRoute has not been called, the host filter can attempt to reopen the queue session by calling QueueRequest during or after QueueClosed.

Timer

PTR contains a one-second resolution timer, which it multiplexes among all routes that request this service.

This service is requested by the SetTimeOut and SetTimeOutms calls. PTR subsequently calls the host filter's TimeOut entry point at an interval indicated by SetTimeOut or SetTimeOutms.

One TimeOut call is received for each call to SetTimeOut or SetTimeOutms. For a continuous timer, you need to call SetTimeOut or SetTimeOutms once upon initialization and once during each call to TimeOut.

The sample source code uses a continuous timer, and further multiplexes it among several possible timed events on each route. This type of implementation also accommodates removal of timeout activity tracking once the awaited event occurs. Most host filters will need to track timed events in a similar fashion.

Long Information Types: STATUS and RESULT

There are two types of long information indicated in the PTR API: STATUS and RESULT. Each of these is essentially the INFOConnect IC_RESULT value.

The PTR API uses two values because of the difference in the encoding of the IC_RESULT_TYPE field. For errors, this contains an error severity. For statuses, it contains an indicator of the status type, such as IC_STATUS_LINESTATE, IC_STATUS_CONNECT, or IC_STATUS_PTR. Separate LogStatus and LogError functions identify each one.

- The only imported API function that returns a STATUS by the PTR API is QueueLock. It is not sent to or generated by the INFOConnect stack. All other imported API functions that return a long value return a RESULT.
- The only exported host filter functions that receive a STATUS are StatusFromHost and StatusFromQueue. PTR passes INFOConnect statuses from INFOConnect as StatusFromHost or StatusFromQueue and also generates PTR statuses of its own. The ErrorFrom functions all receive a RESULT directly from the INFOConnect stack.
- IC_RESULT values are the results passed with an IC_STATUS_CONNECT. IC_RESULT values from INFOConnect message types are handled as indicated, depending on the session type.

INFOConnect Message Type	API Handling
IC_STATUS	StatusFromHost, StatusFromQueue (STATUS)
IC_XMTERROR	SendErrorFromHost, SendErrorFromQueue (RESULT)
IC_RCVERROR	DataErrorFromHost, DataErrorFromQueue (RESULT)
IC_ERROR (during open)	HostClosed, QueueClosed (not passed)
IC_STATUSRESULT	StatusFromHost, StatusFromQueue (STATUS)
IC_CONNECT_OPEN	StatusFromHost, StatusFromQueue (STATUS)
IC_CONNECT_CLOSED	StatusFromHost, StatusFromQueue (STATUS)

In a connection-oriented host or queue path, IC_CONNECT_OPEN and IC_CONNECT_CLOSED messages indicate whether or not you are physically connected to the host. The HostOpen and QueueOpen calls indicate that the communication path is open for use.

Custom host filters will need to interpret STATUS values based on the specific IC_RESULT_TYPE indicated. In particular, some IC_STATUS_PTR events must be handled to determine the success of SOD and EOD events.

RESULT values are usually processed based on severity alone. Otherwise, a particular INFOConnect module's RESULT values must be imported into the host filter itself. This can introduce version dependencies that are not normally favorable for host filters.

Data Type Definitions

The data type definitions used by PTR API include ASESSION, RESULT, and STATUS. Refer to the INFOConnect Development Kit (IDK) documentation for more information.

ASESSION

The structure definition ASESSION is used by several host filter and API functions. Following are the session records that are passed to each of the API calls.

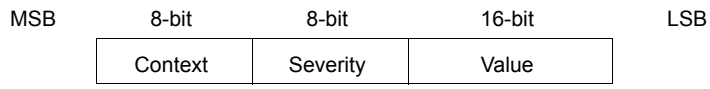
```
typedef struct
{
     HSESSIONh   Session
     char _far   *pBuffer
     unsigned    uBufSize;
} ASESSION, _far *LPASESSION;
```

Where	Is
hSession	A handle to the host path or printer queue. The value of the handle is NULL until the host path or printer queue has been established. The handle is unique and needs to be passed to the HostRelease or QueueRelease function when the queue or path is released by the host filter.
pBuffer	A pointer to a data buffer.
uBufSize	The size of the buffer. On HostOpen and QueueOpen, the *uBufSize* returned in *aSession* contains the maximum buffer size that the host or queue will accept.

RESULT

A longword RESULT is a 32-bit value with the following format:

```
typedef LONG RESULT;
```



Context

The Context byte is assigned to each INFOConnect Accessory and Service Library at runtime.

Severity

The Severity byte is the seriousness of the error. PTR-specific messages are in this category. Severity is in the following categories:

Category	Description
IC_ERROR_INFO	Informational message only, not an error.
IC_ERROR_WARNING	Warning. The request and a suggested result are displayed.
IC_ERROR_SEVERE	Severe. The request did not succeed.
IC_ERROR_TERMINATE	Terminal. The request failed and all future requests will fail.

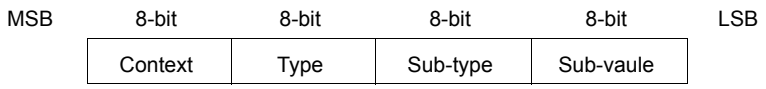
Value

The Value is the actual error. Refer to Appendix A, "Status and Error Messages," beginning on page 135 for PTR specific errors.

STATUS

A longword STATUS is a 32-bit value with the following format:

```
typedef LONG STATUS;
```



Context

The Context byte is assigned to each INFOConnect Accessory and Service Library at runtime. Through INFOConnect functions, the Context can allow you to determine which module initiated the Status message. For PTR statuses, this byte can be ignored.

Type

The Type byte is the generic class of the message. All PTR status messages will have a type of IC_STATUS_PTR, which has a value of 0x60.

Sub-type

The Sub-type field contains the category of the status indicated, such as device selection, function key events, or connection state changes.

Sub-value

The Sub-value field contains the specific status value for the indicated Sub-type, such as the device selected, the function key pressed, or the new connection state.

Exported Host Filter Functions

3

In This Chapter

This chapter provides an overview of each exported host filter function. The following sections are included:

ConfigProc

ConfigProc implements host filter configuration. Each host filter can use this function to implement host filter configuration items not present in the global PTR configuration. This is the only optional exported function in the PTR API; it is an exception to the "all exported functions required" rule.

Prototype

```
UINT CALLBACK ConfigProc(HWND hWnd, BOOL
   bSupported, LPSTR lpszRoute)
```

Parameters

Parameter	Description
hWnd	Parent Window handle that should be used for Application Modal configuration dialog boxes.
bSupported	Flag to query if there are any Host Filter specific configuration options. If bSupported is TRUE, the call queries to determine whether the host filter implements advanced configuration. PTR uses this to determine whether the Advanced button should be enabled in the PTR Route configuration dialog box. Configuration should not be performed at this time, but the return value should be set to indicate whether this capability is present. If bSupported is FALSE, then Host Filter configuration should be performed. PTR calls ConfigProc in this manner when the user clicks on the Advanced button in the PTR Route configuration dialog.
lpszRoute	The name of the route being configured.

Returned Value

If bSupported is TRUE, return one of the following flag mask values:

Value	Description
0	Indicates that advanced configuration is not supported.
1	Indicates that advanced configuration is supported and that the route with the supplied name has already been configured.
3	Indicates that advanced configuration is supported, but the route with the supplied name is not configured in the host filter. PTR will requires you to configure this route within the host filter before exiting the Route Configuration dialog box.

If *bSupported* is FALSE, the return value is ignored.

DataErrorFromHost

DataErrorFromHost informs the host filter that the previous RcvFromHost call on the session handle *hSession* has failed.

Since RcvFromHost is a non-reentrant call, either DataFromHost or DataErrorFromHost must be received before RcvFromHost can be called again for this host session.

Prototype

```
BOOL DataErrorFromHost(PROUTE pRoute, HSESSION
  hHost, RESULT Error);
```

Parameters

Parameter	Description
pRoute	Window handle that uniquely identifies the route.
hSession	Pointer to the session record.
Error	Error value in RESULT format.

Returned Value

TRUE to indicate the error was handled. FALSE to have PTR handle the error in a default manner.

See Also

DataFromHost, RcvFromHost

DataErrorFromQueue

DataErrorFromQueue informs the host filter that the previous RcvFromQueue call for the session handle *hSession* has failed.

Since RcvFromQueue is a non-reentrant call, either DataFromQueue or DataErrorFromQueue must be received before RcvFromQueue can be called again for this queue session.

Prototype

```
BOOL DataErrorFromQueue(PROUTE pRoute, HSESSION
  hQueue, RESULT Error);
```

Parameters

Parameter	Description
pRoute	Window handle that uniquely identifies the route.
hQueue	Pointer to the queue record.
Error	Error value in RESULT format.

Returned Value

TRUE to indicate the error was handled. FALSE to have PTR handle the error in a default manner.

See Also

DataFromQueue, RcvFromQueue

DataFromHost

DataFromHost is called upon the completion of RcvFromHost, when data has arrived on the host session and is ready for processing. The buffer pointer *pBuffer* points to the incoming data, which has a length of *uBufSize*.

Since RcvFromHost is a non-reentrant call, either DataFromHost or DataErrorFromHost must be received before RcvFromHost can be called again for this host session.

If more data is needed before processing can continue, call RcvFromHost again. Once more data has arrived, DataFromHost is called.

Any data that arrives before HostOpen or after HostClosed should be discarded.

> **Note:** The host filter must complete processing the entire data buffer before calling RcvFromHost again and before returning from DataFromHost. Typically, the host filter allocates a buffer and copies the received data into it before continuing its processing.

Prototype

```
void EXPORT CALLBACK DataFromHost(HWND hRoute,
  LPASESSION aSession)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record.

Returned Value

None.

See Also

RcvFromHost, HostOpen, HostClosed, DataErrorFromHost

DataFromQueue

DataFromQueue is called upon the completion of RcvFromQueue, when text data has arrived from the queue session and is ready for processing. The buffer pointer *pBuffer* points to the incoming data, which has a length of *uBufSize*.

Since RcvFromQueue is a non-reentrant call, either DataFromQueue or DataErrorFromQueue must be received before RcvFromQueue can be called again for this queue session.

If more data is needed before processing can continue, call RcvFromQueue again. Once more data has arrived, DataFromHost is called.

Any data arriving before QueueOpen or after QueueClosed should be discarded.

> **Note:** The host filter must complete processing the entire data buffer before calling RcvFromQueue again and before returning from DataFromQueue. Typically, the host filter allocates a buffer and copies the received data into it before continuing its processing.

Prototype

```
void EXPORT CALLBACK DataFromQueue(HWND hRoute,
  LPASESSION aSession)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record.

Returned Value

None.

See Also

RcvFromQueue, QueueOpen, QueueClosed, DataErrorFromQueue

HostClosed

HostClosed is called in response to a HostRelease call, if an unsolicited closure occurs on the host session or if a HostRequest call fails. The session handle is not valid for use by the host filter after this call is received.

Prototype

```
void EXPORT CALLBACK HostClosed(HWND hRoute, LPASESSION
   aSession)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record.

Returned Value

None.

See Also

HostRelease, HostOpen, HostRequest

HostOpen

HostOpen is called in response to HostRequest. After HostOpen, the session handle is valid for use by the host filter for all API calls. If the host session could not be opened, the HostClosed function is called instead.

The behavior of Host Open generally depends on what type of path you are using and its requirements. HostOpen does not indicate a complete connection to the host. A receive (RcvFromHost) should be issued.

An IC_CONNECT_OPEN status must be received before a SendToHost is accepted by the communication path. If your host is not connection-oriented, such as the UDP FRAD, you will not receive an IC_CONNECT_OPEN.

Prototype

```
void EXPORT CALLBACK HostOpen(HWND hRoute,
   LPASESSION aSession)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record. The *uBufSize* returned in *aSession* contains the maximum buffer size that the host will return.

Returned Value

None.

See Also

HostRequest, HostClosed, HostRelease

InitializeRoute

InitializeRoute is the first function call on a route. It passes the unique session handle and configuration data to the host filter. Any route-specific local data structures can be allocated within this function. All future communication to the route is through the session handle, which is passed with every call from the API.

The QueueRequest and HostRequest API calls can be made during or after the InitializeRoute function.

Prototype

```
BOOL EXPORT CALLBACK InitializeRoute (HWND hRoute,
   LPSTR HostPath, LPSTR PrinterQueue, UINT TimeOut,
   LPSTR QueueType, LPSTR Route)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies this route. It needs to be passed back to the API on all calls for this route.
HostPath	Pointer to a string containing the default name of the host path. If HostRequest is called with a NULL pointer for the path name, this path is opened.
PrinterQueue	Pointer to a string containing the default printer queue name. If QueueRequest is called with a NULL pointer for the queue name, this queue is opened.
TimeOut	A time out value passed to the host filter.
QueueType	Pointer to a string containing the queue type. This can be used to customize host filter operation for specific printer types. The current queue types are Document, ATB, ATB II, Bag/Tag, Boarding, and Special.
Route	Pointer to a string containing the name of the route. This can be used to identify the configuration parameters edited by ConfigProc or to log debugging information.

Returned Value

TRUE if the function is able to initialize the route. FALSE if the initialization fails.

See Also

TerminateRoute, QueueRequest, HostRequest

QueueClosed

QueueClosed is called in response to QueueRelease, if an unsolicited closure occurs on the queue session or if a QueueRequest call fails. The session handle is not valid for use by the host filter after this call is received.

Prototype

```
void EXPORT CALLBACK QueueClosed(HWND hRoute,
   LPASESSION aSession)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record.

Returned Value

None.

See Also

QueueRelease, QueueOpen, QueueRequest

QueueOpen

QueueOpen is called in response to QueueRequest. After QueueOpen, the session handle is valid for use by the host filter for all API calls. If the queue could not be opened, the QueueClosed function is called instead.

A Queue may not be used until it is locked or an IC_PTR_PRINTERREADY is received. Refer to IC_PTR_SOD status or QueueAllocate for more information.

Prototype

```
void EXPORT CALLBACK QueueOpen(HWND hRoute,
   LPASESSION aSession)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record. The *uBufSize* returned in *aSession* contains the maximum buffer size that the queue will accept and return.

Returned Value

None.

See Also

QueueRequest, QueueClosed, QueueRelease

SendErrorFromHost

SendErrorFromHost informs the host filter that the previous SendToHost call on the session handle has failed.

Since SendToHost is a non-reentrant call, either SendToHostDone or SendErrorFromHost must be received before SendToHost can be called again for this host session.

Prototype

```
BOOL EXPORT CALLBACK SendErrorFromHost(PROUTE
  pRoute, HSESSION hHost, RESULT Error);
```

Parameters

Parameter	Description
pRoute	Window handle that uniquely identifies the route.
hHost	Pointer to the session record.
Error	Error value in RESULT format.

Returned Value

TRUE to indicate the error was handled. FALSE to have PTR handle the error in a default manner.

See Also

SendToHost, SendToHostDone

SendErrorFromQueue

SendErrorFromQueue informs the host filter that the previous SendToQueue call has failed.

Since SendToQueue is a non-reentrant call, either SendToQueueDone or SendErrorFromQueue must be received before SendToQueue can be called again for this queue session.

Prototype

```
BOOL EXPORT CALLBACK SendErrorFromQueue(PROUTE
  pRoute, HSESSION hQueue, RESULT Error);
```

Parameters

Parameter	Description
pRoute	Window handle that uniquely identifies the route.
hQueue	Pointer to the queue record.
Error	Error value in RESULT format.

Returned Value

TRUE to indicate the error was handled. FALSE to have PTR handle the error in a default manner.

See Also

SendToQueue, SendErrorFromQueue

SendToHostDone

SendToHostDone is the callback function for SendToHost. SendToHostDone informs the host filter that the previous SendToHost call on the session handle has completed successfully.

Since SendToHost is a non-reentrant call, either SendToHostDone or SendErrorFromHost must be received before SendToHost can be called again for this host session.

Prototype

```
void EXPORT CALLBACK SendToHostDone(HWND hRoute,
   LPASESSION aSession)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record.

Returned Value

None.

See Also

SendToHost, SendErrorFromHost

SendToQueueDone

SendToQueueDone is the callback function for SendToQueue. SendToQueueDone informs the host filter that the previous SendToQueue call has completed successfully.

Since SendToQueue is a non-reentrant call, either SendToQueueDone or SendErrorFromQueue must be received before SendToQueue can be called again for this queue session.

Prototype

```
void EXPORT CALLBACK SendToQueueDone(HWND hRoute,
   LPASESSION aSession)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record.

Returned Value

None.

See Also

SendToQueue, SendToQueueError

StatusFromHost

StatusFromHost receives all status messages from the host session. The types of statuses passed to this function depend on the type of host connection. Unknown status values should be discarded by the host filter.

Prototype

```
void EXPORT CALLBACK StatusFromHost(HWND hRoute,
   LPASESSION aSession, STATUS sStatus)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record.
sStatus	The status message.

Returned Value

None.

See Also

SendHostStatus

StatusFromQueue

StatusFromQueue receives all status messages from the queue session. Refer to Appendix A, "Status and Error Messages," beginning on page 135 for a list of PTR-specific messages. Additional status messages passed to this function depend on the type of printer queue connection. Unknown status values should be discarded by the host filter.

Prototype

```
void EXPORT CALLBACK StatusFromQueue(HWND hRoute,
  LPASESSION aSession, STATUS sStatus)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record.
sStatus	The status message.

Returned Value

None.

See Also

SendQueueStatus

TerminateRoute

TerminateRoute is called when PTR is closing. It is the last call made on a route. All route-specific local structures should be freed at this time. Any queue sessions or host sessions opened by QueueRequest or HostRequest are closed.

Prototype

```
void EXPORT CALLBACK TerminateRoute(HWND hRoute)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.

Returned Value

None.

See Also

InitializeRoute, QueueRequest, HostRequest

TimeOut

TimeOut is the callback function from SetTimeOut or SetTimeOutms. Each call to SetTimeOut or SetTimeOutms causes only one TimeOut callback. For a continuous timer, call SetTimeOut or SetTimeOutms once during initialization and once during each TimeOut callback.

Prototype

```
void EXPORT CALLBACK TimeOut(HWND hRoute, UINT
   uKey)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
uKey	The value that was passed to SetTimeOut or SetTimeOutms.

Returned Value

None.

See Also

SetTimeOut, SetTimeOutms

4 Imported API Functions

In This Chapter

This chapter provides an overview of each imported API function. The following sections are included:

Host Release

HostRelease closes a host session. HostClosed is called once closure is complete.

The handle is not valid once the HostClosed call is made to the host filter. It is possible to get data and status messages from the host session after the HostClosed call is made. This data must be discarded.

Prototype

```
void CALLBACK HostRelease (HWND hRoute, HANDLE
  hSession)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
hSession	Session handle of the host to be released.

Returned Value

None.

See Also

HostOpen, HostClosed, HostRequest

HostRequest

HostRequest opens a host session. If the *lpszHostPath* string is a NULL pointer, the default host path for this route is opened.

The return value for HostRequest is the handle to the host session. This handle is not valid until a HostOpen call is made to the host filter. It is possible to get data and status messages from the host session before the HostOpen call is made. This data must be discarded until the HostOpen call is received.

If the return value is NULL, the request has failed. Either the HostPath name does not exist, the host path is in use by another host filter, or a host session has already been opened by this route.

If a host session cannot be opened after a handle has been returned, HostClosed is called.

Prototype

```
HANDLE CALLBACK HostRequest (HWND hRoute, LPSTR
   lpszHostPath)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
lpszHostPath	Pointer to a string that is the name of the host path.

Returned Value

If successful, it returns a handle to the host session. It returns NULL if unsuccessful.

See Also

HostOpen, HostClosed, HostRelease

QueueAllocate

QueueAllocate is an intelligent, queue-locking mechanism for queues expected to be shared most of the time. It functions similarly to QueueLock, but allows the caller to specify additional parameters for priority and maximum wait.

Prototype

```
RESULT CALLBACK QueueAllocate(HWND hRoute, HANDLE
   hQueue, int Timeout, UINT Priority)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
hQueue	handle of the queue session to be locked.
Timeout	Maximum time in seconds to wait for the queue session to become available. A value of -1 indicates that timeout is disabled and the route waits forever or until the queue is available. A value of zero indicates that QueueAllocate must return an error immediately if the queue is not available.
Priority	A priority specifier. Routes with a higher priority are given access to the queue first. Routes with the same priority are given access to the queue in the order in which they requested access. QueueAllocate does not revoke or interrupt the document or lock status from the current document or print job, regardless of the priority setting.

Returned Value

A longword zero if the queue is available immediately. Otherwise, a nonzero PTR STATUS describing the state of the request. Refer to Appendix A, "Status and Error Messages," beginning on page 135 for more information.

If the *Timeout* value was zero and the queue is not available, IC_ERROR_QUEUEBUSY is returned.

If the QueueAllocate request is pending, IC_ERROR_QUEUEWAITING is returned. PTR sends an IC_PTR_ACKLOCK status when the queue becomes available or an IC_PTR_LOCKTIMEOUT status when the timeout period expires.

See Also

QueueLock, QueueUnlock, StatusFromQueue, SendQueueStatus, GetDataFromQueue

QueueLock

QueueLock manually locks a queue session. To release the queue you must call QueueUnlock.

The queue is automatically locked by a Start of Document status message to a queue and released by an End of Document. QueueLock needs to be called only if multiple jobs must be sent to the queue without intervention or if status or data are required from the queue session.

QueueLock is functionally equivalent to QueueAllocate with *Timeout* and *Priority* values of zero.

Prototype

```
RESULT CALLBACK QueueLock(HWND hRoute, HANDLE
  hQueue)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
hQueue	Handle of the queue session to be locked.

Returned Value

A longword zero if the function is successful. Otherwise, a non-zero PTR STATUS value describing the reason for failure. Refer to Appendix A, "Status and Error Messages," beginning on page 135 for more information.

See Also

QueueAllocate, QueueUnlock, SendQueueStatus, GetDataFromQueue

QueueRelease

QueueRelease closes a queue. QueueClosed is called once closure is complete.

The queue session handle is not valid once the QueueClosed call is made to the host filter. It is possible to get data and status messages from the queue session after the QueueClosed call is made. This data must be discarded.

Prototype

```
void CALLBACK QueueRelease (HWND hRoute, HANDLE
  hSession)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
hSession	Session handle of the queue to be released.

Returned Value

None.

See Also

QueueOpen, QueueClosed, QueueRequest

QueueRequest

QueueRequest opens a queue. If the *lpszPrinterQueue* string is a NULL pointer, the default queue for this route is opened.

The return value for QueueRequest is the handle to the queue. This handle is not valid until a QueueOpen call is made to the host filter. It is possible to get data and status messages from the queue before the QueueOpen call is made. This data must be discarded until the QueueOpen call is received.

If the return value is NULL, the request has failed. Either the *lpszPrinterQueue* name does not exist or the name is not a valid PTR path.

If a queue cannot be opened after a non-NULL handle has been returned, QueueClosed is called.

Prototype

```
HANDLE CALLBACK QueueRequest (HWND hRoute, LPSTR
   lpszPrinterQueue)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
lpszPrinterQueue	Session handle of the host to be released.

Returned Value

If successful, it returns a handle to the queue. It returns NULL if unsuccessful.

See Also

QueueOpen, QueueClosed, QueueRelease

QueueUnlock

QueueUnlock manually unlocks a queue. The queue is automatically locked by a Start of Document status message to a queue and released by an End of Document. To manually lock a queue you must call QueueLock.

You can only unlock a queue that has been locked by QueueLock. An automatic lock initiated by a Start of Document status message can only be ended by an End of Document status.

Prototype

```
void CALLBACK QueueUnlock (HWND hRoute, LPASESSION
   aSession)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record.

Returned Value

None.

See Also

QueueLock, SendQueueStatus

RcvFromHost

RcvFromHost begins receiving data from the host path. DataFromHost is called when data has been successfully received.

Since RcvFromHost is non-reentrant, a DataFromHost or DataErrorFromHost call must be received before attempting to receive more data from the host.

The return value of this function informs the host filter that a receive request has been accepted. An unsuccessful response from this function can occur if the host has not completed opening, if a receive is already pending, or if there is an error in the host path.

Prototype

```
RESULT CALLBACK RcvFromHost(HWND hRoute, LPASESSION
   aSession)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record. The *uBufSize* returned in *aSession* contains the amount of data requested. In addition, the *pBuffer* is not used.

Returned Value

A longword zero if the request was initiated correctly. Otherwise, a RESULT value indicating why the request could not be initiated.

See Also

DataFromHost, DataErrorFromHost, HostOpen

RcvFromQueue

RcvFromQueue begins receiving data from the queue. DataFromQueue is called when data has been successfully received.

Since RcvFromQueue is non-reentrant, a DataFromQueue or DataErrorFromQueue call must be received before attempting to receive more data from the queue.

The return value of this function informs the host filter that a receive request has been accepted. An unsuccessful response from this function can occur if the queue path has not completed opening, if a receive is already pending, or if there is an error in the queue session.

Prototype

```
RESULT CALLBACK RcvFromQueue(HWND hRoute,
   LPASESSION aSession)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record. The *uBufSize* returned in *aSession* contains the maximum amount of data requested. In addition, the *pBuffer* is not used.

Returned Value

A longword zero if the request was initiated correctly. Otherwise, a RESULT value indicating why the request could not be initiated.

See Also

DataFromQueue, DataErrorFromQueue, QueueOpen, QueueLock, SendQueueStatus

SendHostStatus

SendHostStatus sends status messages to the host.

The return value of SendHostStatus informs the host filter that a send request has been accepted. An unsuccessful response from this function can occur if the host has not completed opening or if there is an error in the host session.

Prototype

```
RESULT CALLBACK SendHostStatus(HWND hRoute,
   LPASESSION aSession, STATUS sStatus)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record.
sStatus	The status message.

Returned Value

A longword zero if the request was initiated correctly. Otherwise, a RESULT value indicating why the status could not be sent.

See Also

StatusFromHost

SendQueueStatus

SendQueueStatus sends status messages to the queue.

The return value of SendQueueStatus informs the host filter that a send request has been accepted. An unsuccessful response from this function can occur if the queue has not completed opening or if there is an error in the queue.

Prototype

```
RESULT CALLBACK SendQueueStatus(HWND hRoute,
   LPASESSION aSession, STATUS sStatus)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record.
sStatus	The status message.

Returned Value

A longword zero if the request was initiated correctly. Otherwise, a RESULT value indicating why the status could not be sent.

See Also

StatusFromQueue

SendToHost

SendToHost sends data to the host. SendToHostDone is called when the data has been successfully sent to the host.

Since SendToHost is non-reentrant, a SendToHostDone or SendErrorFromHost must be received before any more data can be sent to the host.

The return value of this function informs the host filter that a send request has been accepted. An unsuccessful response from this function can occur if the host has not completed opening, if a send is pending, or if there is an error in the host session.

Prototype

```
RESULT CALLBACK SendToHost(HWND hRoute, LPASESSION
   aSession)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record.

Returned Value

A longword zero if the request was initiated correctly. Otherwise, a RESULT value indicating why the request could not be initiated.

See Also

SendToHostDone, SendErrorFromHost, HostOpen

SendToQueue

SendToQueue sends data to the queue. SendToQueueDone is called when the data is successfully sent to the queue.

Since SendToQueue is non-reentrant, a SendToQueueDone or SendErrorFromQueue must be received before any more data can be sent to the queue.

The return value of this function informs the host filter that a send request has been accepted. An unsuccessful response from this function can occur if the queue has not completed opening, if a send is pending, or if there is an error in the queue.

Prototype

```
RESULT CALLBACK SendToQueue(HWND hRoute, LPASESSION
   aSession)
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
aSession	Pointer to the session record.

Returned Value

A longword zero if the request was initiated correctly. Otherwise, a RESULT value indicating why the request could not be initiated.

See Also

SendToQueueDone, QueueOpen, SendErrorFromQueue, QueueLock, SendQueueStatus

SetTimeOut

SetTimeOut works with the TimeOut function. Each call to SetTimeOut causes only one TimeOut callback. For a continuous timer, call SetTimeOut once during initialization and once during each TimeOut callback.

Prototype

```
void CALLBACK SetTimeOut(HROUTE hRoute, int
   iTimeOutLen, UINT uKey, BOOL bSetTimer);
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
iTimeOutLen	The time, in seconds, before the callback is made. Setting the value to one-second will actually generate a timeout between 1 and 1.2 seconds. A zero value will generate a timeout between 0 and 200 milliseconds on the next timer tick.
uKey	A value passed to the TimeOut function.
bSetTimer	Controls whether or not the timer is set. If the value is SETTIMER, the timeout is in effect. If the value is RESETTIMER, the pending timeout is canceled.

Returned Value

None.

See Also

TimeOut, SetTimeOutms

SetTimeOutms

SetTimeOutms works with the TimeOut function. Each call to SetTimeOutms causes only one TimeOut callback. For a continuous timer, call SetTimeOutms once during initialization and once during each TimeOut callback.

Prototype

```
void CALLBACK SetTimeOutms(HROUTE hRoute, long
   iTimeOutLen, UINT uKey, BOOL bSetTimer);
```

Parameters

Parameter	Description
hRoute	Window handle that uniquely identifies the route.
iTimeOutLen	The time, in milliseconds, before the callback is made. Setting the value to one millisecond will actually generate a timeout between 200 and 400 milliseconds. A zero value will be called back on the next timer tick.
uKey	A value passed to the TimeOut function.
bSetTimer	Controls whether or not the timer is set. If the value is SETTIMER, the timeout is in effect. If the value is RESETTIMER, the pending timeout is canceled.

Returned Value

None.

See Also

TimeOut, SetTimeOut

5 PDK Functions for ANSI C

In This Chapter

This chapter provides an overview of each PDK function for ANSI C. The following sections are included:

DBUGLOG Module

Description

This module contains code that supports debug logging to a file. The header file defines a number of macros used throughout the source code to document time-tagged event tracing. There is also a switch (_DEBUGLOG) to enable or disable the tracing capability after the debug portion of the development cycle.

Notes

The macro _DEBUGLOG enables and disables the logging function. It is passed on the compiler command line.

If _DEBUGLOG is undefined, the following macros are defined as null statements and are undefined. This enables you to permanently disable debug logging for the final implementation after debugging is complete.

Macros

The following macros are included in the DEBUGLOG module:

`LOG(LPSTR s, LPSTR e)`

This macro adds a line to the trace log containing the session name (Session) and event name (Event) only. If the first parameter is NULL it is cast (LPSTR)(LPCSTR) by the macro. If _DEBUGLOG is defined, this macro calls Log with NULL and zero as the last two arguments, respectively.

`LOG1(LPSTR s, LPSTR e, LPSTR b, UINT u)`

This macro adds lines to the trace log containing the session name (Session), event name (Event), and a hexadecimal memory dump (buffer pointer b and length u). The first and third parameters may be NULL or a CString; they are cast (LPSTR)(LPCSTR) by the macro. If _DEBUGLOG is defined, this macro calls Log with all parameters passed.

Caution: Be sure to pass the address of diagnostic data (b) to this function, not a data value. Passing a value can result in a protection exception, as in the following example:

```
RESULT x;
LOG1("Sample", "This dumps the RESULT value", &x,
sizeof(x) );
LOG1("Sample", "This causes a protection fault",
x, sizeof(x) );
```

```
LOGSTATUS(STATUS s)
```

This macro adds a line to the trace log indicating the interpretation of a STATUS value(s). This is intended to be called immediately after LOG1, where LOG1 describes the source of the STATUS and dumps the STATUS in binary form. If _DEBUGLOG is defined, this macro calls LogStatus.

```
LOGERROR(HSESSION h, RESULT e)
```

If _DEBUGLOG is defined, this macro adds a line to the trace log indicating the interpretation of a RESULT value (*e*). This is intended to be called immediately after LOG1, where LOG1 indicates the source of the RESULT and dumps the RESULT in binary form. A session handle (*h*) is necessary because session-specific INFOConnect calls are necessary to determine the context and meaning of the RESULT. If _DEBUGLOG is defined, this macro calls LogError.

```
LOGINIT()
```

This macro conditionally initializes the log file. If _DEBUGLOG is defined, this macro calls InitLog.

Functions

The following functions are included in the PDK DEBUGLOG module:

```
void InitLog();
```

This function initializes the debug log. This truncates the output file, if it exists, to zero length or creates the output file.

```
void Log(LPSTR lpszSessionName, LPSTR szEvent,
   LPSTR lpBuffer, UINT uBufSize);
```

This function is called by the LOG and LOG1 macros, LogStatus and LogError. It opens the log file, appends lines of text to the trace file, and closes the trace file again. Depending on the parameters passed, this may or may not have time tags affixed. Any Log call with a non-NULL session name or nonzero data length is time-tagged. Others are indented so as to appear to be associated with the previous output.

```
char *DumpHex(char ch);
```

This function is called by Log. It converts a character to a hexadecimal dump string suitable for use as part of a dump output. It returns a far pointer to a null-terminated space-delimited string.

```
char *DumpChar(char ch);
```

This function is called by Log. It converts a character to a printable character suitable for use as a part of dump output. Characters less than 0x20 or greater than 0x80 are converted to periods. It returns a far pointer to a null-terminated single-character string.

```
void TimeString(time_t tTime, LPSTR lpsBuffer);
```

This function is called by Log. It converts the current time to a string suitable for appending to a log output.

```
void LogStatus(STATUS Status);
```

This function is called by the LOGSTATUS macro. It interprets a STATUS value, and calls Log to display the interpretation in the log file.

```
LogError(HSESSION hSession, RESULT Error);
```

This function is called by the LOGERROR macro. It interprets a RESULT value and calls Log to display the interpretation in the log file. A session handle (*hSession*) is necessary because INFOConnect calls are made to determine the context and meaning of the RESULT.

LIST Module

Description

This module contains list processing functions for a variety of object types. It replaces the list processing functions in the CObjList class of the Microsoft Foundation Classes (MFCs).

Notes

All structures placed on a LIST must meet certain criteria. Specifically, they must contain a pointer to a like item (a PITEM equivalent) as their first data item. The pointer to a linkable structure can then be cast as a PITEM and passed to AddListHead or AddListTail. The PITEM link should be initialized to NULL and maintained exclusively by the routines in the LIST module. Do not attempt to move or remove the PITEM links from the ROUTE, SESSION, PTRTIMER, or TASK structures defined in the host filter source code.

Functions

The following functions are included in the PDK LIST module:

```
PLIST ListInit();
```

This function allocates and initializes a LIST structure. It returns a pointer to the new LIST or NULL if the list was not properly allocated and initialized.

```
void AddListHead(PLIST pList, PITEM pItem);
```

This function adds an ITEM referenced by pItem to the head (start) of the LIST referenced by *pList*. A NULL pointer cannot be added to a list.

```
void AddListTail(PLIST pList, PITEM pItem);
```

This function adds an item referenced by pItem to the tail (end) of the list referenced by *pList*. A NULL pointer cannot be added to a list.

```
PITEM GetListTail(PLIST pList);
```

This function returns a pointer to the last entry on the list or NULL if the list is empty.

```
PITEM RemoveListHead(PLIST pList);
```

This function delinks the head (first) item from the list referenced by *pList* and returns a pointer to the delinked item. A NULL pointer is returned if the list was empty or if *pList* was NULL.

```
void RemoveList(PLIST pList, PITEM pItem);
```

This function delinks an item referenced by pItem from the list referenced by *pList*.

PTRAPP Module

Description

This module contains the default core code of the host filter interfaces. It contains routines that manage each route's data components, handle multiplexed timer events, and handle task management for each session.

Notes

This module should not require extensive modification for development purposes.

Initialization Functions

The following initialization functions are included in the PDK PTRAPP module:

```
PROUTE InitApp();
```

This function is called by InitRoute to allocate and initialize a ROUTE structure. It returns a pointer to the ROUTE structure or a NULL pointer if the structure could not be allocated or initialized. Any values initialized here may be changed by the caller or as a result of subsequent realtime initialization or route operation.

```
BOOL AppInitializeRoute(PROUTE pRoute, HWND hRoute,
   LPSTR lpszRoute, LPSTR lpszHost, LPSzTR
   lpszQueue, UINT uTimeout, LPSTR lpszQueueType);
```

This function is called by RouteInitializeRoute to perform realtime initialization of the route structure after all required memory has been allocated. The route's configuration parameters (the call parameters) are copied into the appropriate members of the ROUTE structure, and the default host session is opened.

Event Handler Functions

The following event handler functions are included in the PDK PTRAPP module:

```
void AppHostOpen(PROUTE pRoute, HSESSION hHost);
```

This function is invoked by RouteHostOpen when PTR calls the HostOpen entry point. It stops the host open timer, logs the event, marks the host session as open, begins receipt of data from the host connection, and processes any pending host tasks.

```
void AppQueueOpen(PROUTE pRoute, HSESSION hQueue);
```

This function is invoked by RouteQueueOpen when PTR calls the QueueOpen entry point. It stops the queue open timer, logs the event, marks the queue session as open, initializes the queue status as ACKEOD, and processes any pending queue tasks.

```
void AppHostClosed(PROUTE pRoute, HSESSION hHost);
```

This function is invoked by RouteHostClosed when PTR calls the HostClosed entry point. It logs the event and removes the host session from memory.

```
void AppQueueClosed(PROUTE pRoute, HSESSION
   hQueue);
```

This function is invoked by RouteQueueClosed when PTR calls the QueueClosed entry point. It logs the event, delinks the queue session from the route's queue list, and removes the queue session from memory.

```
void AppSendToHostDone(PROUTE pRoute, HSESSION
   hHost);
```

This function is invoked by RouteSendToHostDone when PTR calls the SendToHostDone entry point. It stops the host send timer, logs the event, clears the *m_bSending* flag in the host session, and processes pending host tasks.

```
void AppSendToQueueDone(PROUTE pRoute, HSESSION
   hQueue);
```

This function is invoked by RouteSendToQueueDone when PTR calls the SendToQueueDone entry point. It stops the queue send timer, logs the event, clears the *m_bSending* flag in the queue session, and processes pending queue tasks.

```
BOOL AppDataErrorFromHost(PROUTE pRoute, HSESSION
   hHost, RESULT Error);
```

This function is invoked by RouteDataErrorFromHost when PTR calls the DataErrorFromHost entry point. It stops the host receive timer and logs the event. It returns TRUE if the passed parameters are valid or FALSE if an invalid parameter is passed.

```
BOOL AppDataErrorFromQueue(PROUTE pRoute, HSESSION
   hQueue, RESULT Error);
```

This function is invoked by RouteDataErrorFromQueue when PTR calls the DataErrorFromQueue entry point. It stops the queue receive timer and logs the event. It returns TRUE if the passed parameters are valid or returns FALSE if an invalid parameter is passed.

```
BOOL AppSendErrorFromHost(PROUTE pRoute, HSESSION
   hHost, RESULT Error);
```

This function is invoked by RouteSendErrorFromHost when PTR calls the SendErrorFromHost entry point. It stops the host send timer, clears the *m_bSending* flag in the host session, and logs the event. It returns TRUE if the passed parameters are valid or returns FALSE if an invalid parameter is passed.

```
BOOL AppSendErrorFromQueue(PROUTE pRoute, HSESSION
   hQueue, RESULT Error);
```

This function is invoked by RouteSendErrorFromQueue when PTR calls the SendErrorFromQueue entry point. It stops the queue receive timer, clears the *m_bSending* flag in the queue session, and logs the event. It returns TRUE if the passed parameters are valid or returns FALSE if an invalid parameter is passed.

```
void AppStatusFromHost(PROUTE pRoute, HSESSION
   hHost, STATUS Status);
```

This function is called by RouteStatusFromHost when PTR calls the StatusFromHost entry point. It logs the event, stores the new status in the host session, and processes pending host tasks.

```
void AppStatusFromQueue(PROUTE pRoute, HSESSION
   hQueue, STATUS Status);
```

This function is called by RouteStatusFromQueue when PTR calls the StatusFromQueue entry point. It logs the event, stores the new status in the queue session, calls HandleStatus, and processes any pending queue tasks.

```
void AppTimeOut(PROUTE pRoute, int iKey);
```

This function is called by TimeOut when PTR calls the TimeOut entry point. If the *m_bHoldTimers* flag for the route is cleared, it ticks each timer on the route's *m_Timers* list. If a timer expires, it calls MultiTimeOut to process the timeout event.

```
void AppTerminateRoute(PROUTE pRoute);
```

This function is called by RouteTerminateRoute when PTR calls the TerminateRoute entry point. It logs the event, sets the *m_bTerminating* flag, closes all sessions, resets the PTR timer, and invalidates the route handle.

Internal Management Functions

The following internal management functions are included in the PDK PTRAPP module:

```
PSESSION OpenDefaultHost(PROUTE pRoute, HSESSION
   hRequisite);
```

This function opens a session to the default host. It invokes OpenHost with the host path name passed to InitializeRoute. The optional requisite queue session handle (*hRequisite*) is stored as *m_hRequisite* in the new or existing host session.

```
PSESSION OpenHost(PROUTE pRoute, LPSTR
   lpszHostName, HSESSION hRequisite);
```

This function opens a host session with a specific path name. If a host session is already active on this route, this function returns a pointer to it. Otherwise, it allocates a new host session, logs the event, and calls HostRequest. If HostRequest returns a valid session handle, the host open timer is started and the host session pointer is stored in the route.

The optional requisite queue session handle (*hRequisite*) is stored as *m_hRequisite* in the existing or new host session.

This function normally returns a pointer to the host session. If this function returns NULL, there was not enough memory to allocate a new host session or HostRequest returned a NULL session handle.

```
PSESSION OpenDefaultQueue(PROUTE pRoute, HSESSION
   hRequisite);
```

This function opens a session to the default queue. It invokes OpenQueue with the queue path name passed to InitializeRoute. The returned session pointer is stored in the route as *m_pDefaultQueue*. The optional requisite host session handle (*hRequisite*) is stored as *m_hRequisite* in the new or existing queue session.

```
PSESSION OpenQueue(PROUTE pRoute, LPSTR
   lpszQueueName, HSESSION hRequisite);
```

This function opens a queue session with a specific path name. If there is already a queue session with this name active on this route, this function returns a pointer to it. Otherwise, it allocates a new queue session, logs the event, and calls QueueRequest. If QueueRequest returns a valid session handle, the queue open timer is started and the queue session pointer is added to the route's *m_Queues* list.

The optional requisite host session handle (*hRequisite*) is stored as *m_hRequisite* in the existing or new queue session.

This function normally returns a pointer to the queue session. If this function returns NULL, then there was not enough memory to allocate a new queue session or QueueRequest returned a NULL session handle.

```
void CloseHost(PROUTE pRoute, BOOL bNow);
```

This function closes the host session. The passed parameter *bNow* indicates whether or not the session should be closed without regard to pending data transfers. If *bNow* is TRUE or there are no pending tasks on the host session, the *m_bOpen* flag for the session is cleared, the task list is cleared, the event is logged, the host close timer is started, and HostRelease is called. Otherwise, a T_CLOSE event is added to the task list and pending host tasks are processed.

```
void CloseQueue(PROUTE pRoute, PSESSION pQueue,
   BOOL bNow);
```

This function closes a queue session. The passed parameter *bNow* indicates whether or not the session should be closed without regard to pending data transfers. If appropriate, an EOD status is sent to the queue and/or UnlockQueue is called. If *bNow* is TRUE or there are no pending tasks on the queue session, then the *m_bOpen* flag for the session is cleared, the task list is cleared, the event is logged, the queue close timer is started, and QueueRelease is called. Otherwise, a T_CLOSE event is added to the task list and pending queue tasks are processed.

```
void CloseAllQueues(PROUTE pRoute, BOOL bNow);
```

This function closes all queues on the route. It calls CloseQueue for each queue on the route. See CloseQueue for the interpretation of *bNow*.

```
void RemoveHost(PROUTE pRoute);
```

This function deallocates a closed or abandoned host session. It removes all tasks from the session, removes all timers associated with the session, and deallocates the host session. The route's host session pointer is set to NULL.

```
void RemoveQueue(PROUTE pRoute, PSESSION pQueue);
```

This function deallocates a closed or abandoned queue session. It removes all tasks from the session, removes all timers associated with the session, delinks the session from the route's *m_Queues* list, and deallocates the queue session. If the session was the route's default queue session, the default queue pointer is set to NULL.

```
void RemoveAllQueues(PROUTE pRoute);
```

This function deallocates all queues on a closed or abandoned route. It calls RemoveQueue for each queue on the route.

```
LPSTR PrepDataFromHost(PROUTE pRoute, HSESSION
   hHost, LPSTR pBuffer, UINT uBufSize);
```

This function is invoked by RouteDataFromHost when PTR calls the DataFromHost entry point. It stops the host receive timer, logs the event, allocates a buffer, copies the data into the new buffer, and returns a pointer to the new buffer.

```
LPSTR PrepDataFromQueue(PROUTE pRoute, HSESSION
   hQueue, LPSTR pBuffer, UINT uBufSize);
```

This function is invoked by RouteDataFromQueue when PTR calls the DataFromQueue entry point. It stops the queue receive timer, logs the event, allocates a buffer, copies the data into the new buffer, and returns a pointer to the new buffer.

```
void SendDataToHost(PROUTE pRoute, LPSTR pBuffer,
   UINT uBufSize, BOOL bRequeue);
```

This function sends data to the host. If the host session is not initialized, the buffer is freed without action. If the host session is not yet open or busy sending previous data, a T_DATA task is added to the session. Otherwise, the event is logged, the *m_bSending* flag is set, the host send timer is started, SendToHost is called, and the buffer is freed. If SendToHost returns an error, RouteSendErrorFromHost is invoked to handle the error.

```
void SendDataToQueue(PROUTE pRoute, PSESSION
   pQueue, LPSTR pBuffer, UINT uBufSize, BOOL
   bRequeue);
```

This function sends data to a queue. If the queue session is not initialized, OpenDefaultQueue is invoked to open a default destination session. If this fails, the buffer is freed and the route aborted.

If the queue session is not yet open or busy sending previous data, a T_DATA task is added to the session. If the queue session has not sent an SOD status, one is sent. If a PTR_PRINTERREADY status has not been received, a T_DATA task is added to the session. Otherwise, the event is logged, the *m_bSending* flag is set, the queue send timer is started, SendToQueue is called, and the buffer is freed. If SendToQueue returned an error, RouteSendErrorFromQueue is invoked to handle the error.

```
void SendStatusToHost(PROUTE pRoute, STATUS Status,
   BOOL bRequeue);
```

This function sends a STATUS to the host. The STATUS value must contain valid entries in all fields except Context. Host statuses depend only on the INFOConnect transport, though PTR supplies the appropriate context value. If the host session is not open, a T_STATUS task is added to the host session; otherwise, SendHostStatus is called immediately. Errors returned from SendHostStatus are logged, but are otherwise ignored.

```
void SendStatusToQueue(PROUTE pRoute, PSESSION
   pQueue, STATUS Status, BOOL bRequeue);
```

This function sends a STATUS to a queue. The STATUS value must contain valid entries in all fields except Context. Queue statuses include those dependent upon the host INFOConnect transport as well as those processed by PTR. If the queue session is not yet open, a T_STATUS task is added to the queue session; otherwise, SendQueueStatus is called. Errors returned from SendQueueStatus are logged, but are otherwise ignored.

```
void GetDataFromHost(PROUTE pRoute);
```

This function enables receipt of data from the host. The event is logged, the host receive timer is started, and RcvFromHost is called. If RcvFromHost returns an error, RouteDataErrorFromHost is invoked to handle the error.

> **Note:** To receive data from a host session, the host session must be open. Otherwise, this call has no effect.

```
void GetDataFromQueue(PROUTE pRoute, PSESSION
   pQueue);
```

This function enables receipt of data from a queue. If the queue is not locked and has not yet received a PTR_PRINTERREADY status from an SOD, LockQueue is called to lock the queue session. The event is logged, the queue receive timer is started, and RcvFromQueue is called. If RcvFromQueue returns an error, RouteDataErrorFromQueue is called to handle the error.

> **Note:** To receive data from a queue session, the queue session must be open. It must also be locked via QueueLock or have an SOD status acknowledged with PTR_PRINTERREADY.

```
BOOL AllocateQueue(PROUTE pRoute, PSESSION pQueue,
   int iTimeout, UINT uPriority);
```

This function conditionally locks the queue. The event is logged, and QueueAllocate is called with the passed priority and timeout values. A return value of TRUE indicates that an error other than IC_ERROR_QUEUEWAITING has occurred.

If QueueAllocate returns a zero status, the queue has been locked immediately. The session's *m_bLocked* flag is set, and GetDataFromQueue is called. If QueueAllocate returns IC_ERROR_QUEUEWAITING, then the host filter must wait for a StatusFromQueue response of IC_PTR_ACKLOCK or IC_PTR_LOCKTIMEOUT.

```
BOOL LockQueue(PROUTE pRoute, PSESSION pQueue);
```

This function locks the queue. The event is logged, and QueueLock is called. If QueueLock returns an error, the error is logged. RouteStatusFromQueue is called to indicate the result with either IC_PTR_ACKLOCK (successful) or IC_PTR_LOCKTIMEOUT (failed).

This function returns TRUE if there was an error or FALSE if the queue was successfully locked. The *m_bLocked* flag is also set to indicate the lock condition of the session.

```
void UnlockQueue(PROUTE pRoute, PSESSION pQueue);
```

This function unlocks a queue. The event is logged, QueueUnlock is called, and the queue's *m_bLocked* flag is cleared.

```
void StartOfDocument(PROUTE pRoute, PSESSION
   pQueue);
```

As a special case of SendStatusToQueue, this function sends an SOD status to the queue session. If *pQueue* is NULL, it tries to open the default queue. If an SOD is already pending on this session, a second SOD status is inhibited. Otherwise, the SOD timer is started, the *m_bSODPending* flag is set, and SendStatusToQueue is called with the appropriate SOD status value.

```
void EndOfDocument(PROUTE pRoute, PSESSION pQueue);
```

As a special case of SendStatusToQueue, this function sends an EOD status to the queue session. If there are tasks pending on the queue session, a T_STATUS task with EOD status is added to the queue's task list. Duplicate adjacent EOD statuses on the same queue are inhibited. Otherwise, SendStatusToQueue is called with the appropriate status value, and the *m_bSODPending* and *m_bDocument* flags are cleared.

```
void Abort(PROUTE pRoute);
```

This function aborts processing on a route. It removes all timers from the route and deallocates all sessions. It should only be invoked when other termination methods fail. It is designed to deallocate as much memory as possible and leave the route in an inoperative state, so that it does not interfere with the operation of other routes using this host filter.

Task Management Functions

The following task management functions are included in the PDK PTRAPP module:

```
void AddPending(PROUTE pRoute, PSESSION pSession,
   UINT uTask, LPSTR pBuffer, UINT uBufSize, STATUS
   Status, BOOL bRequeue)
```

This function adds a task to the session's task list. All passed parameters are stored in the task and recalled by the Process*Pending routines. If *bRequeue* is TRUE, the task is placed on the head (start) of the task list; otherwise, it is placed on the tail (end) of the task list. It generally indicates that this task has been delinked from the head of the list by Process*Pending, but has not been performed, so it must be relinked at the head to maintain the correct task order.

```
void RemovePending(PLIST pList);
```

This function removes all pending tasks from a session's task list. Buffers attached to a task are freed at this time.

```
void ProcessHostPending(PROUTE pRoute);
```

This function processes tasks pending on a host session. If the host session is open and there are tasks on the host session's task list, this function delinks the head (first) pending task on the list and passes it to DoHostTask.

```
void ProcessQueuePending(PROUTE pRoute, PSESSION
   pQueue);
```

This function processes tasks pending on a queue session. If the queue session is open and there are tasks on the queue session's task list, this function delinks the head (first) pending task on the list and passes it to DoQueueTask.

```
void DoHostTask(PROUTE pRoute, PTASK pTask);
```

This function performs a single host task. It copies the task parameters to local variables, then frees the task. It then calls a function appropriate for the task's indicated type. User defined task types are handled by a call to DoUserHostTask.

```
void DoQueueTask(PROUTE pRoute, PSESSION pQueue,
   PTASK pTask);
```

This function performs a single queue task. It copies the task parameters to local variables, then frees the task. It then calls a function appropriate for the task's indicated type. User-defined task types are handled by a call to DoUserQueueTask.

Multiplexed Timer Functions

The following multiplexed timer functions are included in the PDK PTRAPP module:

```
void StartTimeOut(PROUTE pRoute, UINT uSeconds,
   UINT uType, HSESSION hSession);
```

This function allocates a timer of type *uType* associated with *hSession*, and adds it to the route's *m_Timers* list. If the *uSeconds* value is zero, no timer is started. Otherwise, the timer is processed by the TimeOut entry point until it times out or until StopTimeOut is called with the same *uType* and *hSession* values.

```
void StopTimeOut(PROUTE pRoute, UINT uType,
  HSESSION hSession);
```

This function finds and delinks all timers of type *uType* associated with from the route's *m_Timers* list and frees them. Subsequent calls to the TimeOut entry point will not process *uType* timers on *hSession* unless a subsequent call to StartTimeOut occurs with the values of *uType* and *hSession* passed.

```
void MultiTimeOut(PROUTE pRoute, UINT uType,
  HSESSION hSession);
```

This function is called when a timer of type *uType* on *hSession* has expired. The *uType* value is interpreted, the event is logged, and the appropriate On<SessionType><TimeoutType>Timeout routine is called to handle the timeout event. Undefined timer types are passed to OnUserTimeOut.

```
void RemoveSessionTimers(PROUTE pRoute, HSESSION
  hSession);
```

This function removes all timers on a route's *m_pTimer* list that are associated with *hSession*, regardless of its *uType*. It is used to clean up a terminated, abandoned, or disabled session. Each timer associated with *hSession* is delinked and freed.

```
void RemoveAllTimers(PROUTE pRoute);
```

This function removes all timers on a route, regardless of the *uType* or *hSession* values. It is used to clean up a terminated, abandoned, or disabled route. Each timer on the *m_Timers* list is delinked and freed.

Utility Functions

The following utility functions are included in the PDK PTRAPP module:

```
PSESSION HandleToQueue(PROUTE pRoute, HSESSION
  hQueue);
```

This function scans the *m_Queues* list on *pRoute*, looking for a queue session with an *m_hSession* matching *hQueue*. It returns a pointer to the queue session or NULL if no queue session is found.

```
void includer();
```

This function should never be called. It ensures that the modules containing exported functions are included from the PDK library, since they are not otherwise referenced from inside the DLL.

Status Handling Functions

The following status display functions are included in the PDK PTRAPP module:

```
BOOL HandleStatus(PROUTE pRoute);
```

This function is called by StatusFromQueue or RouteStatusFromQueue to perform two functions. First, it handles the current PTR API queue status, which must be copied into pQueue->m_Status. Second, it determines whether or not *Status* is a PTR API error status. It returns TRUE if the status is a PTR status indicative of an error. It returns FALSE if the status is indicative of normal operation or if it is not an IC_STATUS_PTR status type.

If *Status* is an error, this function optionally displays the status to the user in a "route modal" message box (*hRoute* is the parent window handle of the route). The display portion of the code is normally dormant, but can be activated for developer use in interactively debugging host filters. To restore the error display function, define _DISPLAYERROR in the compiler command line.

```
int AppMessageBox(PROUTE pRoute, HWND hRoute, LPSTR
   Content,  LPSTR Caption, UINT uType);
```

This function inhibits timer expiration on the route and displays a message box with the passed characteristics. Timer activity is reinstated when the user exits the message box.

```
void AddHex(LPSTR String, char ch);
```

This function appends two hexadecimal digits representing the value of the character to the end of a string. It is called by conditional code in HandleStatus.

PTRDLL Module

Description

This module handles DLL initialization and termination. There is normally no special processing required when a host filter initializes or terminates.

Notes

This module is different for Visual C++ than for ANSI C. The ANSI C code provides a LibMain function. Each contains the minimum code necessary to load as a Windows DLL.

Functions

The following functions are included in the PDK PTRDLL module:

```
int CALLBACK LibMain(HINSTANCE hInstance, WORD
   wDataSeg, WORD cbHeapSize, LPSTR lpCmdLine);
```

This function initializes an instance of the host filter. It copies the instance handle to *hMyInstance* and returns 1.

PTRENTRY Module

Description

This module contains all entry points called directly from the PTR API. It converts the passed parameter information into a set of parameters suitable for each call. For example, it obtains the address of the route structure referenced by the passed handle.

Notes

Due to the nature of the PTR API, each entry point has the same ANSI C style prototype in both ANSI C and Visual C++.

This module's summary reflects half of the PTR API. The other half, those functions exported by PTR and imported by the host filter, are defined in the PTRPROTO.H header file.

Functions

The following functions are included in the PDK PTRENTRY module:

```
BOOL EXPORT CALLBACK InitializeRoute(HWND hRoute,
   LPSTR lpszHost, LPSTR lpszQueue, UINT uTimeout,
   LPSTR lpszQueueType, LPSTR lpszRoute);
```

This function allocates and initializes a route.

```
void EXPORT CALLBACK HostOpen(HWND hRoute,
   LPASESSION aSession);
```

This function indicates that HostRequest has completed successfully.

```
void EXPORT CALLBACK QueueOpen(HWND hRoute,
   LPASESSION aSession);
```

This function indicates that QueueRequest has completed successfully.

```
void EXPORT CALLBACK QueueClosed(HWND hRoute,
   LPASESSION aSession);
```

This function indicates that QueueRequest has failed or QueueRelease has completed successfully.

```
void EXPORT CALLBACK HostClosed(HWND hRoute,
   LPASESSION aSession);
```

This function indicates that HostRequest has failed or HostRelease has completed successfully.

```
void EXPORT CALLBACK DataFromHost(HWND hRoute,
  LPASESSION aSession);
```

This function indicates that data has been received from the host session.

```
void EXPORT CALLBACK DataFromQueue(HWND hRoute,
  LPASESSION aSession);
```

This function indicates that data has been received from the queue session.

```
void EXPORT CALLBACK SendToHostDone(HWND hRoute,
  LPASESSION aSession);
```

This function indicates that SendToHost has completed successfully.

```
void EXPORT CALLBACK SendToQueueDone(HWND hRoute,
  LPASESSION aSession);
```

This function indicates that SendToQueue has completed successfully.

```
BOOL EXPORT CALLBACK RcvErrorFromHost(HWND hRoute,
  LPASESSION aSession, RESULT Status);
```

This function indicates that RcvFromHost has failed. A return value of TRUE indicates that the error was handled. A FALSE value causes PTR to handle the error in a default manner.

```
BOOL EXPORT CALLBACK RcvErrorFromQueue(HWND hRoute,
  LPASESSION aSession, RESULT Status);
```

This function indicates that RcvFromQueue has failed. A return value of TRUE indicates that the error was handled. A FALSE value causes PTR to handle the error in a default manner.

```
BOOL EXPORT CALLBACK SendErrorFromHost(HWND hRoute,
  LPASESSION aSession, RESULT Status);
```

This function indicates that SendToHost has failed. A return value of TRUE indicates that the error was handled. A FALSE value causes PTR to handle the error in a default manner.

```
BOOL EXPORT CALLBACK SendErrorFromQueue(HWND
  hRoute, LPASESSION aSession, RESULT Status);
```

This function indicates that SendToQueue has failed. A return value of TRUE indicates that the error was handled. A FALSE value causes PTR to handle the error in a default manner.

```
void EXPORT CALLBACK StatusFromHost(HWND hRoute,
  LPASESSION aSession, STATUS Status);
```

This function indicates a status change on the host session.

```
void EXPORT CALLBACK StatusFromQueue(HWND hRoute,
  LPASESSION aSession, STATUS Status);
```

This function indicates a status change on the queue session.

```
void EXPORT CALLBACK TimeOut(HWND hRoute, int
  iKey);
```

This function indicates that the SetTimeOut time has expired.

```
void EXPORT CALLBACK TerminateRoute(HWND hRoute);
```

This function closes and deallocates a route.

```
BOOL EXPORT ConfigProc(HWND hWnd, BOOL bSupported,
  LPSTR lpszRoute);
```

This function supports the "Advanced" configuration button in the PTR Quick Configuration dialog box. It calls UserConfigProc, which should be defined in the project source file.

```
CPtrRoute *HandleToObject(HWND hRoute);
```

This function converts a route handle to a pointer to a ROUTE structure. It searches the RouteList for an entry with the same route handle. When found, it returns the item's pointer. It returns NULL if the list is empty or if *hRoute* was not found in any of the entries on the list.

PTRSESS Module

Description

This module handles the session class or structure. It defines the session data elements and contains code to allocate, free, and initialize each host or queue session.

Notes

The same session class or structure is used for both host and queue sessions. All data items are publicly accessed from the object or structure pointer.

Function

The following function is included in the PDK PTRSESS module:

```
PSESSION SessionInit(LPSTR lpszSessionName);
```

This function allocates and initializes a CSession object or SESSION structure. The passed name is stored in *m_SessionName*.

PTRTIMER Module

Description

This module handles the multiplexed timer structure. It defines the data elements and contains code to allocate and initialize each multiplexed timer.

Functions

The following functions are included in the PDK PTRTIMER module:

```
PPTRTIMER TimerInit(UINT iTicks, UINT uType,
  HSESSION hSession);
```

This function allocates and initializes a CPtrTimer object or PTRTIMER structure. The passed parameters are stored in the data elements of the new timer.

```
BOOL TimerTick(PPTRTIMER pTimer);
```

This function is called for each timer by the TimeOut multiplexed timer entry point. Decrements *m_TicksRemaining*. It returns TRUE if the timer count has reached zero or returns FALSE if the timer has not yet expired.

TASK Module

Description

This module handles the task structure. It defines the data elements and contains code to allocate and initialize each pending task.

Functions

The following functions are included in the PDK TASK module:

```
PTASK TaskInit(UINT uTask, LPSTR pBuffer, UINT
   uBufSize, STATUS Status);
```

This function is called by AddPending to allocate and initialize a Ctask object or TASK structure. The passed parameters are stored in the data elements of the new task.

THRU and NULL Modules

Description

Each of these modules contains sample source code for a host filter. THRU is the primary module intended for developer modification. Add your processing here to customize THRU.HFF into your host filter. You can copy this file and create several different host filters, each with only a single unique module.

Initialization and Termination Functions

The following initialization and termination functions are included in the PDK THRU and NULL modules:

```
PROUTE InitRoute();
```

This function allocates and initializes a ROUTE structure. It returns a pointer to the new route or returns NULL if allocation and initialization failed.

```
void RouteTerminateRoute(PROUTE pRoute);
```

This function deallocates any user structures appended to the ROUTE structure and calls AppTerminateRoute.

```
BOOL RouteInitializeRoute(PROUTE pRoute, HWND
   hRoute, LPSTR lpszRoute, LPSTR lpszHost, LPSTR
   lpszQueue, UINT uTimeout, LPSTR lpszQueueType);
```

This function performs runtime startup initialization of a route.

Event Handler Functions

The following event handler functions are included in the PDK THRU and NULL modules:

```
void RouteHostOpen(PROUTE pRoute, HSESSION hHost);
```

This function is called when the host session opens. It calls the default AppHostOpen routine.

```
void RouteQueueOpen(PROUTE pRoute, HSESSION
   hQueue);
```

This function is called when the queue session opens. It calls the default AppQueueOpen routine.

```
void RouteHostClosed(PROUTE pRoute, HSESSION
   hHost);
```

This function is called when HostRequest fails or HostRelease completes successfully. It calls the default AppHostClosed function to clean up the session structure. It then determines the post-cleanup action to be taken when a host session is closed. This varies according to the requirements of each host filter. The sample THRU host filter starts a TO_HOST_RECONN timer.

> **Note:** Since HostClosed can be called from within HostRequest, reopening a host session immediately from this function can cause recursion.

```
void RouteQueueClosed(PROUTE pRoute, HSESSION
   hQueue);
```

This function is called when QueueRequest fails or when QueueRelease completes successfully. It calls the default AppQueueClosed function to clean up the session structure. It then determines the post-cleanup action to be taken when a queue session is closed. This varies according to the requirements of each host filter. The sample THRU host filter starts a TO_QUEUE_RECONN timer.

> **Note:** Since QueueClosed can be called from within QueueRequest, reopening a queue session immediately from this function can cause recursion.

```
void RouteDataFromHost(PROUTE pRoute, HSESSION
   hHost, LPSTR pBuffer, UINT uBufSize);
```

This function is called when data arrives on the host session. The host filter may not access this data buffer after returning from this call or after calling RcvFromHost again. The data is copied to a locally allocated buffer in the PrepDataFromHost routine. The processing of the received host data varies according to the requirements of each host filter.

> **Note:** GetDataFromHost may result in an immediate call to DataFromHost. To maintain correct buffer ordering, GetDataFromHost should be avoided until after the received data is processed and queued for output.

```
void RouteDataFromQueue(PROUTE pRoute, HSESSION
   hQueue, LPSTR pBuffer, UINT uBufSize);
```

This function is called when data arrives on the queue session. The host filter may not access this data buffer after returning from this call or after calling RcvFromQueue again. The data is copied to a locally allocated buffer in the PrepDataFromHost routine. The processing of the received queue data varies according to the requirements of each host filter.

> **Note:** GetDataFromQueue may result in an immediate call to DataFromQueue. To maintain correct buffer ordering, GetDataFromQueue should be avoided until after the received data is processed and queued for output.

```
void RouteSendToHostDone(PROUTE pRoute, HSESSION
   hHost);
```

This function is called when SendToHost completes without error. It calls the default AppSendToHostDone routine.

```
void RouteSendToQueueDone(PROUTE pRoute, HSESSION
   hQueue);
```

This function is called when SendToQueue completes without error. It calls the default AppSendToQueueDone routine.

```
BOOL RouteDataErrorFromHost(PROUTE pRoute, HSESSION
   hHost, RESULT Error);
```

This function is called when RcvFromHost fails. It determines the action to be taken when a receive error occurs on a host session. This varies according to the requirements of each host filter. The sample THRU host filter calls CloseHost if the error is severe or terminal or calls GetDataFromHost if the error is only a warning or informational. It returns TRUE to indicate that the error was handled or returns FALSE to have PTR handle the error in a default manner.

```
BOOL RouteDataErrorFromQueue(PROUTE pRoute,
  HSESSION hQueue, RESULT Error);
```

This function is called when RcvFromQueue fails. It determines the action to be taken when a receive error occurs on a queue session. This varies according to the requirements of each host filter. The sample THRU host filter calls CloseQueue if the error is severe or terminal or calls GetDataFromQueue if the error is only a warning or informational. It returns TRUE to indicate that the error was handled or returns FALSE to have PTR handle the error in a default manner.

```
BOOL RouteSendErrorFromHost(PROUTE pRoute, HSESSION
  hHost, RESULT Error);
```

This function is called when SendToHost fails. It determines the action to be taken when a transmit error occurs on a host session. This varies according to the requirements of each host filter. The sample THRU host filter calls CloseHost if the error is severe or terminal or calls ProcessHostPending if the error is only a warning or informational. It returns TRUE to indicate that the error was handled or returns FALSE to have PTR handle the error in a default manner.

```
BOOL RouteSendErrorFromQueue(PROUTE pRoute,
  HSESSION hQueue, RESULT Error);
```

This function is called when SendToQueue fails. It determines the action to be taken when a transmit error occurs on a queue session. This varies according to the requirements of each host filter. The sample THRU host filter calls CloseQueue if the error is severe or terminal or calls ProcessQueuePending if the error is only a warning or informational. It returns TRUE to indicate that the error was handled or returns FALSE to have PTR handle the error in a default manner.

```
void RouteStatusFromHost(PROUTE pRoute, HSESSION
  hHost, STATUS Status);
```

This function is called when the status of the host session changes. It calls the default AppStatusFromHost routine.

```
void RouteStatusFromQueue(PROUTE pRoute, HSESSION
   hQueue, STATUS Status);
```

This function is called when the status of the queue session changes. It calls the default AppStatusFromQueue routine.

Internal Queuing Functions

The following internal queuing functions are included in the PDK THRU and NULL modules:

```
void DoUserHostTask(PROUTE pRoute, UINT uTask,
   LPSTR pBuffer, UINT uBufSize, STATUS Status);
```

This function is called by DoHostTask when a user defined task type is delinked from a host session's task list. The action to be taken varies according to the requirements of each host filter and user-defined task type. The sample THRU host filter calls ProcessHostPending, thus ignoring undefined task types.

```
void DoUserQueueTask(PROUTE pRoute, PSESSION
   pQueue, UINT uTask, LPSTR pBuffer, UINT uBufSize,
   STATUS Status);
```

This function is called by DoQueueTask when a user-defined task type is delinked from a queue session's task list. The action to be taken varies according to the requirements of each host filter and user defined task type. The sample THRU host filter calls ProcessQueuePending, thus ignoring undefined task types.

Timeout Handler Functions

Each of these functions is a default handler for a specific type of timeout on a specific type of session.

```
void OnHostOpenTimeOut(PROUTE pRoute);
```

This function performs a host open timeout. The action to be taken varies according to the requirements of each host filter. The sample THRU host filter closes the host session.

```
void OnHostCloseTimeOut(PROUTE pRoute);
```

This function performs a host close timeout. The action to be taken varies according to the requirements of each host filter. The sample THRU host filter deallocates the host session.

```
void OnHostRecvTimeOut(PROUTE pRoute);
```

This function performs a host receive timeout. The action to be taken varies according to the requirements of each host filter. The sample THRU host filter sends an EOD to all queues.

```
void OnHostSendTimeOut(PROUTE pRoute);
```

This function performs a host send timeout. The action to be taken varies according to the requirements of each host filter. The sample THRU host filter closes the host session.

```
void OnHostReconnTimeOut(PROUTE pRoute);
```

This function performs a host reconnect timeout. The action to be taken varies according to the requirements of each host filter. The sample THRU host filter opens the default host session.

```
void OnQueueOpenTimeOut(PROUTE pRoute, HSESSION
   hQueue);
```

This function performs a queue open timeout. The action to be taken varies according to the requirements of each host filter. The sample THRU host filter closes the queue session.

```
void OnQueueCloseTimeOut(PROUTE pRoute, HSESSION
   hQueue);
```

This function performs a queue close timeout. The action to be taken varies according to the requirements of each host filter. The sample THRU host filter deallocates the queue session.

```
void OnQueueRecvTimeOut(PROUTE pRoute, HSESSION
   hQueue);
```

This function performs a queue receive timeout. The action to be taken varies according to the requirements of each host filter. The sample THRU host filter sends an EOD to the queue session and unlocks it.

```
void OnQueueSendTimeOut(PROUTE pRoute, HSESSION
   hQueue);
```

This function performs a queue send timeout. The action to be taken varies according to the requirements of each host filter. The sample THRU host filter closes the queue session.

```
void OnQueueReconnTimeOut(PROUTE pRoute);
```

This function performs a queue reconnect timeout. The action to be taken varies according to the requirements of each host filter. The sample THRU host filter opens the default queue session.

```
void OnQueueDocumentTimeOut(PROUTE pRoute, HSESSION
   hQueue);
```

This function performs a queue PTR_PRINTERREADY timeout after SOD. The action to be taken varies according to the requirements of each host filter. The sample THRU host filter closes the queue session if no response was received or reissues SOD if the status was busy.

```
void OnUserTimeOut(PROUTE pRoute, UINT uType,
   HSESSION hSession);
```

This function performs a user timeout. The action to be taken varies according to the requirements of each host filter and user-defined timer type. The sample THRU host filter performs no action, thus ignoring undefined timer types.

```
BOOL OnQueueBusy(PROUTE pRoute, PSESSION pQueue);
```

This function is called when an attempt to initialize output to the queue has resulted in a response indicating that the printer is not currently available. It returns TRUE if this is an unexpected error condition or returns FALSE if the busy condition has been handled.

The default implementation performs two actions. First, it disables receipt of additional data from the host session by setting the *m_pLocalQueue* pointer to the busy queue. This inhibits GetDataFromHost until the busy queue closes or processes all of its queued data and events. Second, it implements the AutoAllocate feature by setting the queue's *m_bAutoAllocate* flag and calling QueueAllocate to wait for the queue to become available.

If the *m_bAutoAllocate* flag was already set, OnQueueBusy returns TRUE, indicating that this queue session has timed out.

AutoAllocation uses the *m_bDocumentTimeout* value of the route to time queue availability. You may want to allow the user to configure this value, alter the 45 second default value of *m_bDocumentTimeout* after calling AppInitializeRoute, or alter the OnQueueTimeout routine to use a different timeout value or algorithm.

Configuration Function

The following configuration function is included in the PDK THRU and NULL modules:

```
UINT UserConfigProc(HWND hWnd, BOOL bSupported,
   LPSTR lpszRoute);
```

This function is an optional advanced configuration support. If advanced host filter configuration is not implemented, returning FALSE all the time is recommended.

If bSupported is TRUE, return one of the following flag mask values:

0—Indicates that advanced configuration is not supported.

1—Indicated that advanced configuration is supported and that the route with the supplied name has already been configured.

3—Indicates that advanced configuration is supported, but the route with the supplied name is not configured in the host filter. PTR will require you to configure this route within the host filter before exiting the Route Configuration dialog box.

If *bSupported* is FALSE, the user has pressed the Config button in the PTR Configuration edit dialog box. The value of *hWnd* is the PTR Configuration's parent window handle for use with application-modal dialog boxes. The route name currently entered in the PTR Configuration edit dialog box is provided in *lpszRoute* for use by the host filter's configuration. The return value is ignored.

The host filter temporarily takes over the configuration process at this point, displaying dialog boxes to the user and manipulating a separate configuration database keyed by the indicated route name. The following standards are recommended for host filters that support multiple routes:

- If *lpszRoute* is NULL or points to an empty string, ask the user to enter a route name in PTR before performing host filter configuration and exit without further action.
- If *lpszRoute* is already in the host filter configuration, load and display the configuration for this route and allow the user to edit it.
- If *lpszRoute* is not already in the host filter configuration, display a list of route names currently in the host filter configuration and allow the user to either add a new entry or update the route name of an existing entry.
- You should provide a method of deleting old or unused route entries. This could be done when the user adds a new route. One way of forcing this issue is to establish a maximum number of routes configured and disable the New option when the configuration table is full.

PDK Functions for Visual C++

6

In This Chapter

This chapter provides an overview of each PDK function for C++. The following sections are included:

DEBUGLOG Module

Description

This module contains code that supports debug logging to a file. The header file defines a number of macros used throughout the source code to document time-tagged event tracing. There is also a switch (_DEBUGLOG) to enable or disable the tracing capability after the debug portion of the development cycle.

Notes

The macro _DEBUGLOG enables and disables the logging function. It is passed on the compiler command line.

If _DEBUGLOG is undefined, the following macros are defined as null statements and are undefined. This enables you to permanently disable debug logging for the final implementation after debugging is complete.

Macros

The following macros are included in the PDK DEBUGLOG module:

`LOG(LPSTR s, LPSTR e)`

This macro adds a line to the trace log containing the session name (Session) and event name (Event) only. The first parameter may be NULL or a CString; it is cast (LPSTR)(LPCSTR) by the macro. If _DEBUGLOG is defined, this macro calls CDebugLog::Log with NULL and zero as the last two arguments, respectively.

`LOG1(LPSTR s, LPSTR e, LPSTR b, UINT u)`

This macro adds lines to the trace log containing the session name (Session), event name (Event), and a hexadecimal memory dump (buffer pointer *b* and length *u*). The first and third parameters may be NULL or a CString, they are cast (LPSTR)(LPCSTR) by the macro. If _DEBUGLOG is defined, this macro calls CDebugLog::Log with all parameters passed.

Caution: Be sure to pass the *address* of diagnostic data (*b*) to this function, not a data *value*. Passing a *value* can result in a protection exception, as in the following example:

```
RESULT x;
LOG1("Sample", "This dumps the RESULT value", &x,
sizeof(x) );
LOG1("Sample", "This causes a protection fault",
x, sizeof(x) );
```

```
LOGSTATUS(STATUS s)
```

This macro adds a line to the trace log indicating the interpretation of a STATUS value(s). This is intended to be called immediately after LOG1, where LOG1 describes the source of the STATUS and dumps the STATUS in binary form. If _DEBUGLOG is defined, this macro calls CDebugLog::LogStatus.

```
LOGERROR(HSESSION h, RESULT e)
```

If _DEBUGLOG is defined, this macro adds a line to the trace log indicating the interpretation of a RESULT value (e). This is intended to be called immediately after LOG1, where LOG1 indicates the source of the RESULT and dumps the RESULT in binary form. A session handle (h) is necessary because session-specific INFOConnect calls are necessary to determine the context and meaning of the RESULT. If _DEBUGLOG is defined, this macro calls CDebugLog::LogError.

Functions

The following functions are included in the PDK DEBUGLOG module:

```
CDebugLog::CDebugLog();
```

This function initializes the debug log. This truncates the output file, if it exists, to zero length or creates the output file.

```
void CDebugLog::Log(LPSTR lpszSessionName, LPSTR
   szEvent, LPSTR lpBuffer, UINT uBufSize);
```

This function is called by the LOG and LOG1 macros, LogStatus and LogError. It opens the log file, appends lines of text to the trace file, and closes the trace file again. Depending on the parameters passed, this may or may not have time tags affixed. Any Log call with a non-NULL session name or nonzero data length are time-tagged. Others are indented so as to appear to be associated with the previous output.

```
LPSTR CDebugLog::DumpHex(char ch);
```

This function is called by Log. It converts a character to a hexadecimal dump string suitable for use as part of a dump output. It returns a far pointer to a null-terminated space-delimited string.

```
char CDebugLog::DumpChar(char ch);
```

This function is called by Log. It converts a character to a printable character suitable for use as a part of dump output. Characters less than 0x20 or greater than 0x80 are converted to periods. It returns the printable character.

```
void TimeString(time_t tTime, LPSTR lpsBuffer);
```

This function is called by Log. It converts the current time to a string suitable for appending to a log output.

```
void CDebugLog::LogStatus(STATUS Status);
```

This function is called by the LOGSTATUS macro. It interprets a STATUS value, and calls Log to display the interpretation in the log file.

```
void CDebugLog::LogError(HSESSION hSession, RESULT
   Error);
```

This function is called by the LOGERROR macro. It interprets a RESULT value, and calls Log to display the interpretation in the log file. A session handle (*hSession*) is necessary because INFOConnect calls are made to determine the context and meaning of the RESULT.

PTRAPP Module

Description

This module contains the default core code of the host filter interfaces. These routines manage each route's data components, handle multiplexed timer events, and handle task management for each session. This is implemented as the CPtrApp base class.

Notes

This module should not require extensive modification for development purposes.

Initialization Functions

The following initialization functions are included in the PDK PTRAPP module:

```
CPtrApp::CPtrApp();
```

This function is called by CPtrRoute::CPtrRoute to construct a CPtrApp object. Any values initialized here may be changed by the caller or as a result of subsequent realtime initialization or route operation.

```
BOOL CPtrApp::InitializeRoute(HWND hRoute, LPSTR
   lpszRoute, LPSTR lpszHost, LPSTR lpszQueue, UINT
   uTimeout, LPSTR lpszQueueType);
```

This function is called by CPtrRoute::InitializeRoute to perform realtime initialization of the route object after all required memory has been allocated. The route's configuration parameters (the call parameters) are copied into the appropriate data members of the CPtrApp object, and the default host session is opened.

Event Handler Functions

The following event handler functions are included in the PDK PTRAPP module:

```
void CPtrApp::HostOpen(HSESSION hHost);
```

This function is invoked by CPtrRoute::HostOpen when PTR calls the HostOpen entry point. It stops the host open timer, logs the event, marks the host session as open, begins receipt of data from the host connection, and processes any pending host tasks.

```
void CPtrApp::QueueOpen(HSESSION hQueue);
```

This function is invoked by CPtrRoute::QueueOpen when PTR calls the QueueOpen entry point. It stops the queue open timer, logs the event, marks the queue session as open, initializes the queue status as ACKEOD, and processes any pending queue tasks.

```
void CPtrApp::HostClosed(HSESSION hHost);
```

This function is invoked by CPtrRoute::HostClosed when PTR calls the HostClosed entry point. It logs the event and removes the host session from memory.

```
void CPtrApp::QueueClosed(HSESSION hQueue);
```

This function is invoked by CPtrRoute::QueueClosed when PTR calls the QueueClosed entry point. It logs the event, delinks the queue session from the route's queue list, and removes the queue session from memory.

```
void CPtrApp::DataFromHost(HSESSION hHost, LPSTR
  pBuffer, UINT uBufSize);
```

This function is the default data handler in the CPtrApp base class. It calls PrepDataFromHost, passes the resulting buffer to SendDataToQueue, and calls GetDataFromHost to continue receiving data. It is only called if the user removes the CPtrRoute::DataFromHost function from the CPtrRoute class.

```
void CPtrApp::DataFromQueue(HSESSION hQueue, LPSTR
  pBuffer, UINT uBufSize);
```

This function is the default data handler in the CPtrApp base class. It calls PrepDataFromQueue, passes the resulting buffer to SendDataToHost, and calls GetDataFromQueue to continue receiving data. It is only called if the user removes the CPtrRoute::DataFromQueue function from the CPtrRoute class.

```
void CPtrApp::SendToHostDone(HSESSION hHost);
```

This function is invoked by CPtrRoute::SendToHostDone when PTR calls the SendToHostDone entry point. It stops the host send timer, logs the event, clears the *m_bSending* flag in the host session, and processes pending host tasks.

```
void CPtrApp::SendToQueueDone(HSESSION hQueue);
```

This function is invoked by CPtrRoute::SendToQueueDone when PTR calls the SendToQueueDone entry point. It stops the queue send timer, logs the event, clears the *m_bSending* flag in the queue session, and processes pending queue tasks.

```
BOOL CPtrApp::DataErrorFromHost(HSESSION hHost,
   RESULT Error);
```

This function is invoked by CPtrRoute::DataErrorFromHost when PTR calls the DataErrorFromHost entry point. It stops the host receive timer and logs the event. It returns TRUE if the passed parameters are valid or returns FALSE if an invalid parameter is passed.

```
BOOL CPtrApp::DataErrorFromQueue(HSESSION hQueue,
   RESULT Error);
```

This function is invoked by CPtrRoute::DataErrorFromQueue when PTR calls the DataErrorFromQueue entry point. It stops the queue receive timer and logs the event. It returns TRUE if the passed parameters are valid or returns FALSE if an invalid parameter is passed.

```
BOOL CPtrApp::SendErrorFromHost(HSESSION hHost,
   RESULT Error);
```

This function is invoked by CPtrRoute::SendErrorFromHost when PTR calls the SendErrorFromHost entry point. It stops the host send timer, clears the *m_bSending* flag in the host session, and logs the event. It returns TRUE if the passed parameters are valid or returns FALSE if an invalid parameter is passed.

```
BOOL CPtrApp::SendErrorFromQueue(HSESSION hQueue,
   RESULT Error);
```

This function is invoked by CPtrRoute::SendErrorFromQueue when PTR calls the SendErrorFromQueue entry point. It stops the queue receive timer, clears the *m_bSending* flag in the queue session, and logs the event. It returns TRUE if the passed parameters are valid or returns FALSE if an invalid parameter is passed.

```
void CPtrApp::StatusFromHost(HSESSION hHost, STATUS
   Status);
```

This function is called by CPtrRoute::StatusFromHost when PTR calls the StatusFromHost entry point. It logs the event, stores the new status in the host session, and processes pending host tasks.

```
void CPtrApp::StatusFromQueue(HSESSION hQueue,
   STATUS Status);
```

This function is called by CPtrRoute::StatusFromQueue when PTR calls the StatusFromQueue entry point. It logs the event, stores the new status in the queue session, calls HandleStatus, and processes pending queue tasks.

```
void CPtrApp::TimeOut(int iKey);
```

This function is called by TimeOut when PTR calls the TimeOut entry point. If the *m_bHoldTimers* flag for the route is cleared, it ticks each timer on the route's m_Timers list. If a timer expires, it calls MultiTimeOut to process the timeout event.

```
void CPtrApp::TerminateRoute();
```

This function is called by CPtrRoute::TerminateRoute when PTR calls the TerminateRoute entry point. It logs the event, sets the *m_bTerminating* flag, closes all sessions, resets the PTR timer, and invalidates the route handle.

List Search Function

The following list search function is included in the PDK PTRAPP module:

```
HWND CPtrApp::GetHandle();
```

This function is called by HandleToObject in PTRENTRY.C to obtain the (private) route handle associated with a route object. It returns the *m_hRoute* stored in the route object during CPtrApp::InitializeRoute.

Internal Management Functions

The following internal management functions are included in the PDK PTRAPP module:

```
PSESSION CPtrApp::OpenDefaultHost(HSESSION
   hRequisite);
```

This function opens a session to the default. It invokes OpenHost with the host path name passed to InitializeRoute. The optional requisite queue session handle (hRequisite) are stored as m_hRequisite in the new or existing host session.

```
PSESSION CPtrApp::OpenHost(LPSTR lpszHostName,
   HSESSION hRequisite);
```

This function opens a host session with a specific path name. If a host session is already active on this route, this function returns a pointer to it. Otherwise, it allocates a new host session, logs the event, and calls HostRequest. If HostRequest returns a valid session handle, the host open timer is started and the host session pointer is stored in the route.

The optional requisite queue session handle (*hRequisite*) is stored as *m_hRequisite* in the existing or new host session.

This function normally returns a pointer to the host session. If this function returns NULL, there was not enough memory to allocate a new host session, or HostRequest returned a NULL session handle.

```
PSESSION CPtrApp::OpenDefaultQueue(HSESSION
   hRequisite);
```

This function opens a session to the default queue. It invokes OpenQueue with the queue path name passed to InitializeRoute. The returned session pointer is stored in the route as *m_pDefaultQueue*. The optional requisite host session handle (*hRequisite*) is stored as *m_hRequisite* in the new or existing session.

```
PSESSION CPtrApp::OpenQueue(LPSTR lpszQueueName,
  HSESSION hRequisite);
```

This function opens a queue session with a specific path name. If there is already a queue session with this name active on this route, this function returns a pointer to it. Otherwise, it allocates a new queue session, logs the event, and calls QueueRequest. If QueueRequest returns a valid session handle, the queue open timer is started and the queue session pointer is added to the route's *m_Queues* list.

The optional requisite host session handle (hRequisite) is stored as *m_hRequisite* in the existing or new queue session.

This function normally returns a pointer to the queue session. If this function returns NULL, then there was not enough memory to allocate a new queue session, or QueueRequest returned a NULL session handle.

```
void CPtrApp::CloseHost(BOOL bNow);
```

This function closes the host session. The passed parameter bNow indicates whether or not the session should be closed without regard to pending data transfers. If *bNow* is TRUE or there are no pending tasks on the host session, the *m_bOpen* flag for the session is cleared, the task list is cleared, the event is logged, the host close timer is started, and HostRelease is called. Otherwise, a T_CLOSE event is added to the task list and pending host tasks are processed.

```
void CPtrApp::CloseQueue(PSESSION pQueue, BOOL
  bNow);
```

This function closes a queue session. The passed parameter *bNow* indicates whether or not the session should be closed without regard to pending data transfers. If appropriate, an EOD status is sent to the queue and/or UnlockQueue is called. If *bNow* is TRUE or there are no pending tasks on the queue session, then the *m_bOpen* flag for the session is cleared, the task list is cleared, the event is logged, the queue close timer is started, and QueueRelease is called. Otherwise, a T_CLOSE event is added to the task list and pending queue tasks are processed.

```
void CPtrApp::CloseAllQueues(BOOL bNow);
```

This function closes all queues on the route. It calls CloseQueue for each queue on the route. See CloseQueue for the interpretation of *bNow*.

```
void CPtrApp::RemoveHost();
```

This function deallocates a closed or abandoned host session. It removes all tasks from the session, removes all timers associated with the session, and deallocates the host session. The route's host session pointer is set to NULL.

```
void CPtrApp::RemoveQueue(PSESSION pQueue);
```

This function deallocates a closed or abandoned queue session. It removes all tasks from the session, removes all timers associated with the session, delinks the session from the route's *m_Queues* list, and deallocates the queue session. If the session was the route's default queue session, the default queue pointer is set to NULL.

```
void CPtrApp::RemoveAllQueues();
```

This function deallocates all queues on a closed or abandoned route. It calls RemoveQueue for each queue on the route.

```
LPSTR CPtrApp::PrepDataFromHost(HSESSION hHost,
   LPSTR pBuffer, UINT uBufSize);
```

This function is invoked by CPtrRoute::DataFromHost when PTR calls the DataFromHost entry point. It stops the host receive timer, logs the event, allocates a buffer, copies the data into the new buffer, and returns a pointer to the new buffer.

```
LPSTR CPtrApp::PrepDataFromQueue(HSESSION hQueue,
   LPSTR pBuffer, UINT uBufSize);
```

This function is invoked by CPtrRoute::DataFromQueue when PTR calls the DataFromQueue entry point. It stops the queue receive timer, logs the event, allocates a buffer, copies the data into the new buffer, and returns a pointer to the new buffer.

```
void CPtrApp::SendDataToHost(LPSTR pBuffer, UINT
   uBufSize, BOOL bRequeue);
```

This function sends data to the host. If the host session is not initialized, the buffer is freed without action. If the host session is not yet open or is busy sending previous data, a T_DATA task is added to the session. Otherwise, the event is logged, the *m_bSending* flag is set, the host send timer is started, SendToHost is called, and the buffer is freed. If SendToHost returned an error, CPtrRoute::SendErrorFromHost is invoked to handle the error.

```
void CPtrApp::SendDataToQueue(PSESSION pQueue,
   LPSTR pBuffer, UINT uBufSize, BOOL bRequeue);
```

This function sends data to a queue. If the queue session is not initialized, OpenDefaultQueue is invoked to open a default destination session. If this fails, the buffer is freed and the route aborted.

If the queue session is not yet open or is busy sending previous data, a T_DATA task is added to the session. If the queue session has not sent an SOD status, one is sent. If a PTR_PRINTERREADY status has not been received, a T_DATA task is added to the session. Otherwise, the event is logged, the *m_bSending* flag is set, the queue send timer is started, SendToQueue is called, and the buffer is freed. If SendToQueue returned an error, CPtrRoute::SendErrorFromQueue is invoked to handle the error.

```
void CPtrApp::SendStatusToHost(STATUS Status, BOOL
   bRequeue);
```

This function sends a STATUS to the host. The STATUS value must contain valid entries in all fields except Context. Host statuses are dependent solely upon the host INFOConnect transport, though PTR supplies the appropriate context value. If the host session is not yet open, a T_STATUS task is added to the host session. Otherwise, SendHostStatus is called immediately. Errors returned from SendHostStatus are logged, but are otherwise ignored.

```
void CPtrApp::SendStatusToQueue(PSESSION pQueue,
   STATUS Status, BOOL bRequeue);
```

This function sends a STATUS to a queue. The STATUS value must contain valid entries in all fields except Context. Queue statuses include those dependent upon the host INFOConnect transport as well as those processed by PTR. If the queue session is not yet open, a T_STATUS task is added to the queue session. Otherwise, SendQueueStatus is called. Errors returned from SendQueueStatus are logged, but otherwise ignored.

```
void CPtrApp::GetDataFromHost();
```

This function enables receipt of data from the host. The event is logged, the host receive timer is started, and RcvFromHost is called. If RcvFromHost returns an error, CPtrRoute::DataErrorFromHost is invoked to handle the error.

> **Note:** To receive data from a host session, the host session must be open. Otherwise, this call has no effect.

```
void CPtrApp::GetDataFromQueue(PSESSION pQueue);
```

This function enables receipt of data from a queue. If the queue is not locked and has not yet received a PTR_PRINTERREADY status from an SOD, LockQueue is called to lock the queue session. The event is logged, the queue receive timer is started, and RcvFromQueue is called. If RcvFromQueue returns an error, CPtrRoute::DataErrorFromQueue is called to handle the error.

> **Note:** To receive data from a queue session, the queue session must be open. It must also be locked via QueueLock or have an SOD status acknowledged with PTR_PRINTERREADY.

```
BOOL CPtrApp::AllocateQueue(PSESSION pQueue, int
   iTimeout, UINT uPriority);
```

This function conditionally locks the queue. The event is logged, and QueueAllocate is called with the passed priority and timeout values. A return value of TRUE indicates that an error other than IC_ERROR_QUEUEWAITING has occurred.

If QueueAllocate returns without error, the queue has been locked immediately. The session's *m_bLocked* flag is set, and GetDataFromQueue is called. If QueueAllocate returns IC_ERROR_QUEUEWAITING, the host filter must wait for a StatusFromQueue response of IC_PTR_ACKLOCK or IC_PTR_LOCKTIMEOUT.

```
BOOL CPtrApp::LockQueue(PSESSION pQueue);
```

This function locks the queue. The event is logged and QueueLock is called. If QueueLock returns an error, the error is logged. CPtrRoute::StatusFromQueue is called to indicate the result with either IC_PTR_ACKLOCK (successful) or IC_PTR_LOCKTIMEOUT (failed).

This function returns TRUE if there was an error or FALSE if the queue was successfully locked. The *m_bLocked* flag is also set to indicate the lock condition of the session.

```
void CPtrApp::UnlockQueue(PSESSION pQueue);
```

This function unlocks a queue. The event is logged, QueueUnlock is called, and the queue's *m_bLocked* flag is cleared.

```
void CPtrApp::StartOfDocument(PSESSION pQueue);
```

As a special case of SendStatusToQueue, this function sends an SOD status to the queue session. If *pQueue* is NULL, it tries to open the default queue. If an SOD is already pending on this session, a second SOD status is inhibited. Otherwise, the SOD timer is started, the *m_bSODPending* flag is set, and SendStatusToQueue is called with the appropriate SOD status value.

```
void CPtrApp::EndOfDocument(PSESSION pQueue);
```

As a special case of SendStatusToQueue, this function sends an EOD status to the queue session. If there are tasks pending on the queue session, a T_STATUS task with EOD status is added to the queue's task list. Duplicate adjacent EOD statuses on the same queue are inhibited. Otherwise, SendStatusToQueue is called with the appropriate status value, and the *m_bSODPending* and *m_bDocument* flags are cleared.

```
void CPtrApp::Abort();
```

This function aborts processing on a route. It removes all timers from the route and deallocates all sessions. It should only be invoked when other termination methods fail. It is designed to deallocate as much memory as possible and leave the route in an inoperative state so that it does not interfere with the operation of other routes using this host filter.

Task Management Functions

The following task management functions are included in the PDK PTRAPP module:

```
void CPtrApp::AddPending(PSESSION pSession, UINT
   uTask, LPSTR pBuffer, UINT uBufSize, STATUS
   Status, BOOL bRequeue);
```

This function adds a task to the session's task list. All passed parameters are stored in the task and recalled by the Process*Pending routines. If *bRequeue* is TRUE, the task is placed on the head (start) of the task list; otherwise, it is placed on the tail (end) of the task list. It generally indicates that this task has been delinked from the head of the list by Process*Pending but has not been performed, so it must be relinked at the head to maintain the correct task order.

```
void CPtrApp::RemovePending(CObList &List);
```

This function removes all pending tasks from a session's task list. Buffers attached to a task are freed at this time.

```
void CPtrApp::ProcessHostPending();
```

This function processes tasks pending on a host session. If the host session is open and there are tasks on the host session's task list, this function delinks the head (first) pending task on the list and passes it to DoHostTask.

```
void CPtrApp::ProcessQueuePending(PSESSION pQueue);
```

This function processes tasks pending on a queue session. If the queue session is open and there are tasks on the queue session's task list, this function delinks the head (first) pending task on the list and passes it to DoQueueTask.

```
void CPtrApp::DoHostTask(PTASK pTask);
```

This function performs a single host task. It copies the task parameters to local variables and then frees the task. It then calls a function appropriate for the task's indicated type. User-defined task types are handled by a call to DoUserHostTask.

```
void CPtrApp::DoQueueTask(PSESSION pQueue, PTASK
  pTask);
```

This function performs a single queue task. It copies the task parameters to local variables and then frees the task. It then calls a function appropriate for the task's indicated type. User-defined task types are handled by a call to DoUserQueueTask.

```
void CPtrApp::DoUserHostTask(UINT uTask, LPSTR
  pBuffer, UINT uBufSize, STATUS Status);
```

This function performs no task in the CPtrApp class. To perform user-defined host task types, override this function as CPtrRoute::DoUserHostTask.

```
void CPtrApp::DoUserQueueTask(PSESSION pSession,
  UINT uTask, LPSTR pBuffer, UINT uBufSize, STATUS
  Status)
```

This function performs no task in the CPtrApp class. To perform user-defined host task types, override this function as CPtrRoute::DoUserQueueTask.

Multiplexed Timer Functions

The following multiplexed timer functions are included in the PDK PTRAPP module:

```
void CPtrApp::StartTimeOut(UINT uSeconds, UINT
  uType, HSESSION hSession);
```

This function allocates a timer of type *uType* associated with *hSession*, and adds it to the route's *m_Timers* list. If the *uSeconds* value is zero, no timer is started. Otherwise, the timer is processed by the TimeOut entry point until it times out or until StopTimeOut is called with the same *uType* and *hSession* values.

```
void CPtrApp::StopTimeOut(UINT uType, HSESSION
   hSession);
```

This function finds and delinks all timers of type *uType* associated with from the route's *m_Timers* list, and frees them. Subsequent calls to the TimeOut entry point will not process *uType* timers on *hSession* unless a subsequent call to StartTimeOut occurs with the values of *uType* and *hSession* passed.

```
void CPtrApp::MultiTimeOut(UINT uType, HSESSION
   hSession);
```

This function is called when a timer of type *uType* on *hSession* has expired. The *uType* value is interpreted, the event logged, and the appropriate On<SessionType><TimeoutType>Timeout routine is called to handle the timeout event. Undefined timer types are passed to OnUserTimeOut.

```
void CPtrApp::RemoveSessionTimers(HSESSION
   hSession);
```

This function removes all timers on a route's *m_pTimer* list that are associated with *hSession*, regardless of its uType. It is used to clean up a terminated, abandoned, or disabled session. Each timer associated with *hSession* is delinked and freed.

```
void CPtrApp::RemoveAllTimers();
```

This function removes all timers on a route, regardless of the *uType* or *hSession* values. It is used to clean up a terminated, abandoned, or disabled route. Each timer on the *m_Timers* list is delinked and freed.

Timeout Handler Functions

Each of these functions is a default handler for a specific type of timeout on a specific type of session. You can override these functions by providing replacement functions of the same name in the CPtrRoute class, located in the file THRU.CPP.

```
void CPtrApp::OnHostOpenTimeOut();
```

This function performs a host open timeout. The action to be taken varies according to the requirements of each host filter. The default handler closes the host session.

```
void CPtrApp::OnHostCloseTimeOut();
```

This function performs a host close timeout. The action to be taken varies according to the requirements of each host filter. The default handler deallocates the host session.

```
void CPtrApp::OnHostRecvTimeOut();
```

This function performs a host receive timeout. The action to be taken varies according to the requirements of each host filter. The default handler sends an EOD to all queues.

```
void CPtrApp::OnHostSendTimeOut();
```

This function performs a host send timeout. The action to be taken varies according to the requirements of each host filter. The default handler closes the host session.

```
void CPtrApp::OnHostReconnTimeOut();
```

This function performs a host reconnect timeout. The action to be taken varies according to the requirements of each host filter. The default handler opens the default host session.

```
void CPtrApp::OnQueueOpenTimeOut(HSESSION hQueue);
```

This function performs a queue open timeout. The action to be taken varies according to the requirements of each host filter. The default handler closes the queue session.

```
void CPtrApp::OnQueueCloseTimeOut(HSESSION hQueue);
```

This function performs a queue close timeout. The action to be taken varies according to the requirements of each host filter. The default handler deallocates the queue session.

```
void CPtrApp::OnQueueRecvTimeOut(HSESSION hQueue);
```

This function performs a queue receive timeout. The action to be taken varies according to the requirements of each host filter. The default handler sends an EOD to the queue session and unlocks it.

```
void CPtrApp::OnQueueSendTimeOut(HSESSION hQueue);
```

This function performs a queue send timeout. The action to be taken varies according to the requirements of each host filter. The default handler closes the queue session.

```
void CPtrApp::OnQueueReconnTimeOut();
```

This function performs a queue reconnect timeout. The action to be taken varies according to the requirements of each host filter. The default handler opens the default queue session.

```
void CPtrApp::OnQueueDocumentTimeOut(HSESSION
   hQueue);
```

This function performs a queue PTR_PRINTERREADY timeout after an SOD. The action to be taken varies according to the requirements of each host filter and user-defined task type. The default handler closes the queue session if no response was received or reissues an SOD if the status was BUSY.

```
void CPtrApp::OnUserTimeOut(UINT uType, HSESSION
   hSession);
```

This function performs no action in the CPtrApp class. To handle user-defined timer types, override this function as CPtrRoute::OnUserTimeOut.

```
BOOL CPtrApp::OnQueueBusy(PSESSION pQueue);
```

This function is called when an attempt to initialize output to the queue has resulted in a response indicating that the printer is not currently available. It returns TRUE if this is an unexpected error condition or returns FALSE if the busy condition has been otherwise handled.

The default implementation performs two actions: First, it disables receipt of additional data from the host session by setting the *m_pLocalQueue* pointer to the busy queue. This inhibits GetDataFromHost until the busy queue closes or processes all of its queued data and events. Second, it implements the AutoAllocate feature by setting the queue's *m_bAutoAllocate* flag and calling QueueAllocate to wait for the queue to become available.

If the *m_bAutoAllocate* flag was already set, OnQueueBusy returns TRUE, indicating that this queue session has timed out.

AutoAllocation uses the *m_bDocumentTimeout* value of the route to time queue availability. You may want to allow the user to configure this value, alter the 45 second default value of *m_bDocumentTimeout* after calling CPtrApp::InitializeRoute, or alter the OnQueueTimeout routine to use a different timeout value or algorithm.

Utility Functions

The following utility functions are included in the PDK PTRAPP module:

```
PSESSION CPtrApp::HandleToQueue(HSESSION hQueue);
```

This function scans the *m_Queues* list on *pRoute*, looking for a queue session with an *m_hSession* matching *hQueue*. It returns a pointer to the queue session or NULL if no queue session is found.

```
void includer();
```

This function should never be called. It ensures that the modules containing exported functions are included from the PDK library, since they are not otherwise referenced from inside the DLL.

Status Display Functions

The following status display functions are included in the PDK PTRAPP module:

```
BOOL CPtrApp::HandleStatus();
```

This function is called by StatusFromQueue or RouteStatusFromQueue to perform two functions. First, it handles the current PTR API queue status, which must be copied into pQueue->m_Status. Second, it determines whether *Status* is a PTR API error status. It returns TRUE if the status is a PTR status error. It returns FALSE if the status is normal operation or if it is not an IC_STATUS_PTR status type.

If *Status* is an error, this function optionally displays the status to the user in a route modal message box (*hRoute* is the parent window handle of the route). The display portion of the code is normally dormant, but can be activated for developer use in interactively debugging host filters. To restore the error display function, define _DISPLAYERROR in the compiler command line.

```
int CPtrApp::MessageBox(HWND hRoute, CString
   &Content, CString &Caption, UINT uType);
```

This function inhibits timer expiration on the route and displays a message box with the passed characteristics. Timer activity is reinstated when the user exits the message box.

```
void AddHex(CString &String, char ch);
```

This function appends two hexadecimal digits representing the value of the character to the end of a string. It is called by conditional code in HandleStatus.

PTRDLL Module

Description

This module handles DLL initialization and termination. There is normally no special processing required when a host filter initializes or terminates.

Notes

The Visual C++ code derives a CPtrDll class from the MFC CWinApp class and provides the InitInstance and ExitInstance functions. It contains the minimum code necessary to load as a Windows DLL.

Functions

The following functions are included in the PDK PTRDLL module:

```
BOOL CPtrDll::InitInstance();
```

This function initializes an instance of the host filter. It copies the instance handle to hMyInstance and returns TRUE.

```
int CPtrDll::ExitInstance();
```

This function terminates an instance. It calls CWinApp::ExitInstance.

PTRENTRY Module

Description

This module contains all entry points called directly from the PTR API. It converts the passed parameter information into a set of parameters suitable for each call. For example, it obtains the address of the route object referenced by the passed handle.

Notes

Due to the nature of the PTR API, each entry point has the same ANSI C style prototype in both the ANSI C and Visual C++ versions. In the Visual C++ code, all the entry points are defined as external "C."

This module's summary reflects half of the PTR API. The other half, those functions exported by PTR and imported by the host filter, are defined in the PTRPROTO.H header file.

Functions

The following functions are included in the PDK PTRENTRY module:

```
BOOL EXPORT CALLBACK InitializeRoute(HWND hRoute,
   LPSTR lpszHost, LPSTR lpszQueue, UINT uTimeout,
   LPSTR lpszQueueType, LPSTR lpszRoute);
```

This function allocates and initializes a route.

```
void EXPORT CALLBACK HostOpen(HWND hRoute,
   LPASESSION aSession);
```

This function indicates that HostRequest has completed successfully.

```
void EXPORT CALLBACK QueueOpen(HWND hRoute,
   LPASESSION aSession);
```

This function indicates that QueueRequest has completed successfully.

```
void EXPORT CALLBACK QueueClosed(HWND hRoute,
   LPASESSION aSession);
```

This function indicates that QueueRequest has failed or QueueRelease has completed successfully.

```
void EXPORT CALLBACK HostClosed(HWND hRoute,
  LPASESSION aSession);
```

This function indicates that HostRequest has failed or HostRelease has completed successfully.

```
void EXPORT CALLBACK DataFromHost(HWND hRoute,
  LPASESSION aSession);
```

This function indicates that data has been received from the host session.

```
void EXPORT CALLBACK DataFromQueue(HWND hRoute,
  LPASESSION aSession);
```

This function indicates that data has been received from the queue session.

```
void EXPORT CALLBACK SendToHostDone(HWND hRoute,
  LPASESSION aSession);
```

This function indicates that SendToHost has completed successfully.

```
void EXPORT CALLBACK SendToQueueDone(HWND hRoute,
  LPASESSION aSession);
```

This function indicates that SendToQueue has completed successfully.

```
BOOL EXPORT CALLBACK RcvErrorFromHost(HWND hRoute,
  LPASESSION aSession, RESULT Status);
```

This function indicates that RcvFromHost has failed. A TRUE return value indicates that the error was handled. A FALSE return value causes PTR to handle the error in a default manner.

```
BOOL EXPORT CALLBACK RcvErrorFromQueue(HWND hRoute,
  LPASESSION aSession, RESULT Status);
```

This function indicates that RcvFromQueue has failed. A TRUE return value indicates that the error was handled. A FALSE return value causes PTR to handle the error in a default manner.

```
BOOL EXPORT CALLBACK SendErrorFromHost(HWND hRoute,
   LPASESSION aSession, RESULT Status);
```

This function indicates that SendToHost has failed. A TRUE return value indicates that the error was handled. A FALSE return value causes PTR to handle the error in a default manner.

```
BOOL EXPORT CALLBACK SendErrorFromQueue(HWND
   hRoute, LPASESSION aSession, RESULT Status);
```

This function indicates that SendToQueue has failed. A TRUE return value indicates that the error was handled. A FALSE return value causes PTR to handle the error in a default manner.

```
void EXPORT CALLBACK StatusFromHost(HWND hRoute,
   LPASESSION aSession, STATUS Status);
```

This function indicates a status change on the host session.

```
void EXPORT CALLBACK StatusFromQueue(HWND hRoute,
   LPASESSION aSession, STATUS Status);
```

This function indicates a status change on the queue session.

```
void EXPORT CALLBACK TimeOut(HWND hRoute, int
   iKey);
```

This function indicates that the SetTimeOut time has expired.

```
void EXPORT CALLBACK TerminateRoute(HWND hRoute);
```

This function closes and deallocates a route.

```
BOOL EXPORT ConfigProc(HWND hWnd, BOOL bSupported,
   LPSTR lpszRoute);
```

This function supports the "Advanced" configuration button in the PTR Quick Configuration dialog box. It calls UserConfigProc, which should be defined in the project source file.

```
CPtrRoute *HandleToObject(HWND hRoute);
```

This function converts a route handle to a pointer to a CPtrRoute object. Searches the RouteList for an entry with the same route handle. When found, it returns the item's pointer. It returns NULL if the list is empty or if *hRoute* was not found in any of the entries on the list.

PTRSESS Module

Description

This module handles the session class or structure. It defines the session data elements and contains code to allocate, free, and initialize each host or queue session.

Notes

The same session class or structure is used for both host and queue sessions. All data items are publicly accessed from the object or structure pointer.

Function

The following function is included in the PDK PTRSESS module:

```
CSession::CSession(LPSTR lpszSessionName) :
   Cobject();
```

This function allocates and initializes a CSession object or SESSION structure. The passed name is stored in *m_SessionName*.

PTRTIMER Module

Description

This module handles the multiplexed timer class or structure. It defines the data elements, and contains code to allocate, free, and initialize each multiplexed timer.

Functions

The following functions are included in the PDK PTRTIMER module:

```
CPtrTimer::CPtrTimer(UINT iTicks, UINT uType,
  HSESSION hSession);
```

This function allocates and initializes a CPtrTimer object or PTRTIMER structure. The passed parameters are stored in the data elements of the new timer.

```
BOOL CPtrTimer::Tick();
```

This function is called for each timer by the TimeOut multiplexed timer entry point. Decrements *m_TicksRemaining*. It returns TRUE if the timer count has reached zero or returns FALSE if the timer has not yet expired.

```
void CPtrTimer::GetData(UINT &uType, HSESSION
  &hSession);
```

This function is called when a timer expires to recover the timer's type and session handle.

TASK Module

Description

This module handles the task class or structure. It defines the data elements and contains code to allocate, free, and initialize each pending task.

Functions

The following functions are included in the PDK TASK module:

```
CTask::CTask(UINT uTask, LPSTR pBuffer, UINT
   uBufSize, STATUS Status);
```

This function is called by AddPending to allocate and initialize a Ctask object or TASK structure. The passed parameters are stored in the data elements of the new task.

```
void CTask::GetData(UINT &uTask, LPSTR &pBuffer,
   UINT &uBufSize, STATUS &Status);
```

This function is called by ProcessHostPending and ProcessQueuePending to extract data elements from the pending task.

THRU and NULL Modules

Description

Each of these modules contains sample source code for a host filter. THRU is the primary module intended for developer modification. Add your processing here to customize THRU.HFF into your host filter. You can copy this file and create several different host filters, each with only a single unique module.

Notes

In Visual C++, if the CPtrApp member functions perform all the required processing, there is no need to override these functions in the derived CPtrRoute class. If you override a new function, add the new version to the CPtrRoute class definition in THRU.H as well as write the new function in THRU.CPP.

If an overridden function is not declared *virtual* in CPtrApp, the caller context determines the destination of each call. Calls from the CPtrApp class (in the host filter library) call the CPtrApp (library) version of the function, and the CPtrRoute calls call the CPtrRoute version of the function. Though CPtrRoute functions can always call the CPtrApp version of a function by prefixing the CPtrApp:: *context specifier*, CPtrApp library functions are not able to access the CPtrRoute version unless the function is declared virtual in the CPtrApp class.

Initialization and Termination Functions

The following initialization and termination functions are included in the PDK THRU and NULL modules:

```
CPtrRoute::CPtrRoute() : CPtrApp();
```

This function allocates and initializes a CPtrRoute object. It returns a pointer to the new route or returns NULL if allocation and initialization failed.

```
CPtrRoute::~CPtrRoute();
```

This function deallocates any user structures appended to the ROUTE structure.

```
BOOL CPtrRoute::InitializeRoute(HWND hRoute, LPSTR
   lpszRoute, LPSTR lpszHost, LPSTR lpszQueue, UINT
   uTimeout, LPSTR lpszQueueType);
```

This function performs runtime startup initialization of a route. It generally opens one or both default sessions.

Event Handler Functions

The following event handler functions are included in the PDK THRU and NULL modules:

```
void CPtrRoute::HostClosed(HSESSION hHost);
```

This function is called when HostRequest fails or HostRelease completes successfully. It calls the default CPtrApp::HostClosed function to clean up the session structure. It then determines the post-cleanup action to be taken when a host session is closed. This varies according to the requirements of each host filter. The sample THRU host filter starts a TO_HOST_RECONN timer.

> **Note:** Since HostClosed can be called from within HostRequest, reopening a host session immediately from this function can cause recursion.

```
void CPtrRoute::QueueClosed(HSESSION hQueue);
```

This function is called when QueueRequest fails or QueueRelease completes successfully. It calls the default CPtrApp::QueueClosed function to clean up the session structure. It then determines the post-cleanup action to be taken when a queue session is closed. This varies according to the requirements of each host filter. The sample THRU host filter starts a TO_QUEUE_RECONN timer.

> **Note:** Since QueueClosed can be called from within QueueRequest, reopening a queue session immediately from this function can cause recursion.

```
void CPtrRoute::DataFromHost(HSESSION hHost, LPSTR
  pBuffer, UINT uBufSize);
```

This function is called when data arrives on the host session. The host filter may not access this data buffer after returning from this call or calling RcvFromHost again. The data is copied to a locally allocated buffer in the PrepDataFromHost routine. The processing of received host data varies according to the requirements of each host filter.

> **Note:** GetDataFromHost may result in an immediate call to DataFromHost. To maintain correct buffer ordering, GetDataFromHost should be avoided until after the received data is processed and queued for output.

```
void CPtrRoute::DataFromQueue(HSESSION hQueue,
   LPSTR pBuffer, UINT uBufSize);
```

This function is called when data arrives on the queue session. The host filter may not access this data buffer after returning from this call or after calling RcvFromQueue again. The data is copied to a locally allocated buffer in the PrepDataFromHost routine. Processing of received queue data varies according to the requirements of each host filter.

> **Note:** GetDataFromQueue may result in an immediate call to DataFromQueue. To maintain correct buffer ordering, GetDataFromQueue should be avoided until after the received data is processed and queued for output.

```
BOOL CPtrRoute::DataErrorFromHost(HSESSION hHost,
   RESULT Error);
```

This function is called when RcvFromHost fails. It determines the action to be taken when a receive error occurs on a host session. This varies according to the requirements of each host filter. The sample THRU host filter calls CloseHost if the error is severe or terminal or calls GetDataFromHost if the error is only a warning or informational. It returns TRUE to indicate that the error was handled or returns FALSE to have PTR handle the error in a default manner.

```
BOOL CPtrRoute::DataErrorFromQueue(HSESSION hQueue,
   RESULT Error);
```

This function is called when RcvFromQueue fails. It determines the action to be taken when a receive error occurs on a queue session. This varies according to the requirements of each host filter. The sample THRU host filter calls CloseQueue if the error is severe or terminal or calls GetDataFromQueue if the error is only a warning or informational. It returns TRUE to indicate that the error was handled or returns FALSE to have PTR handle the error in a default manner.

```
BOOL CPtrRoute::SendErrorFromHost(HSESSION hHost,
  RESULT Error);
```

This function is called when SendToHost fails. It determines the action to take when a transmit error occurs on a host session, which varies according to each host filter's requirements. The sample THRU host filter calls CloseHost if the error is severe or terminal or calls ProcessHostPending if the error is informational or a warning. It returns TRUE to indicate that the error was handled or returns FALSE to have PTR handle the error in a default manner.

```
BOOL CPtrRoute::SendErrorFromQueue(HSESSION hQueue,
  RESULT Error);
```

This function is called when SendToQueue fails. It determines the action to be taken when a transmit error occurs on a queue session. This varies according to the requirements of each host filter. The sample THRU host filter calls CloseQueue if the error is severe or terminal or calls ProcessQueuePending if the error is only a warning or informational. It returns TRUE to indicate that the error was handled or returns FALSE to have PTR handle the error in a default manner.

Configuration Function

The following configuration function is included in the PDK THRU and NULL modules:

```
UINT .i).UserConfigProc;(HWND hWnd, BOOL
  bSupported, LPSTR lpszRoute);
```

This function is an optional advanced configuration support. If advanced host filter configuration is not implemented, returning FALSE all the time is recommended.

If bSupported is TRUE, return one of these flag mask values:

0—Indicates that advanced configuration is not supported.

1—Indicates that advanced configuration is supported and the route with the supplied name has already been configured.

3—Indicates that advanced configuration is supported, but the route with the supplied name is not configured in the host filter. PTR will require you to configure this route within the host filter before exiting the Route Configuration dialog box.

If *bSupported* is FALSE, the user has pressed the Config button in the PTR Configuration edit dialog box. The value of *hWnd* is the PTR Configuration's parent window handle for use with application-modal dialog boxes. The route name currently entered in the PTR Configuration edit dialog box is provided in *lpszRoute* for use by the host filter's configuration. The return value is ignored.

The host filter temporarily takes over the configuration process at this point, displaying dialog boxes to the user and manipulating a separate configuration database keyed by the indicated route name. The following standards are recommended for host filters that support multiple routes:

- If *lpszRoute* is NULL or points to an empty string, ask the user to enter a route name in PTR before performing host filter configuration and exit without further action.
- If *lpszRoute* is already in the host filter configuration, load and display the configuration for this route and allow the user to edit it.
- If *lpszRoute* is not already in the host filter configuration, display a list of route names currently in the host filter configuration and allow the user to either add a new entry or update the route name of an existing entry.
- You should provide a method of deleting old or unused route entries. This could be done when the user adds a new route. One way of forcing this issue is to establish a maximum number of routes configured and disable the New option when the configuration table is full.

Status and Error Messages

A

In This Appendix

This appendix contains the following topics:

Statuses Sent by Host Filter

These definitions are also included in the file ICPTRSTS.H.

The host filter can send one of the following statuses:

Status	Sub-Type	Sub-Value
Start of Document (SOD)	IC_PTR_SOD (0x06)	0x00
End of Document (EOD)	IC_PTR_EOD (0x07)	0x00
Abort	IC_PTR_ABORT(0x09)	0x00

Start of Document (SOD)

SubType: C_PTR_SOD (0x06)

This status is sent to the printer queue to signify that the host filter wants to gain exclusive access to the queue. If the Queue is already locked by use of the QueueLock function, the Start of Document status is used to signify the beginning of a document. If the printer queue has not been manually locked, the host filter must wait for the status message Printer Ready before issuing any other statuses like GetDataFromQueue or SendDataToQueue.

End of Document (EOD)

SubType: C_PTR_EOD (0x07)

This status is sent to the printer queue to signify that the host filter wants to release its access to the queue. If the Queue was locked by the QueueLock function, the End of Document status is used to signify the end of a document. If the printer queue has not been manually locked, the queue is no longer available for use by the host filter. After End of Document has been called, Start of Document status must be called in order to regain access to the queue.

Abort

SubType: IC_PTR_ABORT(0x09)

This status is sent to a queue to abort printing without flushing pending data. It suggests that a serious communication or processing error has occurred that prevents proper route operation.

Statuses Received by Host Filter

The host filter can receive any of the following statuses.

Status	Sub-Type	Sub-Value
Printer Ready	IC_PTR_PRINTERREADY (0x00)	0x00
Printer Not Ready: Offline	IC_PTR_NOTREADY (0x01)	IC_PTR_OFFLINE (0x00)
Printer Not Ready: Out of Paper	IC_PTR_NOTREADY (0x01)	IC_PTR_OUTOFPAPER (0x01)
Printer Not Ready: Paper Jam	IC_PTR_NOTREADY (0x01)	IC_PTR_PAPERJAM (0x02)
Printer Not Ready: Printer Busy	IC_PTR_NOTREADY (0x01)	IC_PTR_BUSY (0x03)
No Device Connected	IC_PTR_NODEVICE (0x02)	0x00
Error: Communications Error	IC_PTR_ERROR (0x03)	IC_PTR_COMMERROR (0x00)
Error: Block Checksum	IC_PTR_ERROR (0x03)	IC_PTR_BLKCHK (0x01)
Acknowledge Abort	IC_PTR_ACKABORT (0x04)	0x00
Acknowledge End of Document	IC_PTR_ACKEOD (0x05)	0x00
Acknowledge Lock	IC_PTR_ACKLOCK (0x0A)	0x00
Lock Timeout	IC_PTR_LOCKTIMEOUT (0x0B)	0x00

Printer Ready

SubType: IC_PTR_PRINTERREADY (0x00)

This status is received in response to a Start of Document. When received, the printer Queue is locked for exclusive access by the host filter. The host filter can now send additional statuses and/or issue GetDataFromQueue or SendDataToQueue.

Send an End of Document status to release the lock of the queue unless it was manually locked by a QueueLock call. The EOD finishes the last SendDataToQueue call and cancels any pending GetDataFromQueue call.

Related calls include: GetDataFromQueue, SendDataToQueue, QueueLock, QueueUnlock.

Printer Not Ready

SubType: IC_PTR_NOTREADY (0x01)

This status is received in response to a Start of Document. This call means that the SOD call was rejected. The SOD call must be reissued to gain control of the printer. This Status has the following SubValues:

Offline

SubValue: IC_PTR_OFFLINE (0x00)

The printer is offline.

Out of Paper

SubValue:C_PTR_OUTOFPAPER (0x01)

The printer is out of paper.

Paper Jam

SubValue: IC_PTR_PAPERJAM (0x02)

There is a paper jam in the printer.

Printer Busy

SubValue: IC_PTR_BUSY (0x03)

The printer queue is currently locked by another host filter.

No Device Connected

SubType: IC_PTR_NODEVICE (0x02)

This status is received in response to an attempt to send data to a session that has no device connected to it. Its receipt is contingent upon the ability of the communications path to identify this condition.

Error

SubType: IC_PTR_ERROR (0x03)

These statuses indicate that the intelligent destination device has reported a protocol or format error and has invoked contingency handling. Their receipt is contingent upon the ability of the communications path to identify these conditions.

Communications Error

SubValue: IC_PTR_COMMERROR(0x00)

This status indicates that an interface protocol rule has been violated. For example, the destination device was expecting data in a different format.

Block Checksum

SubValue: IC_PTR_BLKCHK(0x01)

This status indicates that the data received by the device did not validate correctly.

Acknowledge Abort

SubType: IC_PTR_ACKABORT(0x04)

This status is sent to indicate that the previous IC_PTR_ABORT status has been honored. Pending session data has been discarded.

Acknowledge End of Document

SubType: IC_PTR_ACKEOD (0x05)

This status is received in response to an End of Document status. When received, the printer queue has finished sending the last of its data and is ready for the next document. This status can be ignored.

Acknowledge Lock

SubType: IC_PTR_ACKLOCK(0x0A)

This status is received in response to QueueAllocate when the printer queue is available for use (locked) by this route. It is functionally equivalent to IC_PTR_PRINTERREADY.

LockTimeout

SubType: IC_PTR_LOCKTIMEOUT(0x0B)

This status is received in response to QueueAllocate when the nonzero timeout period has expired and the requested queue remains unavailable. It is functionally equivalent to the RESULT IC_PTR_QUEUEBUSY.

Error Message Table

A host filter can receive the following PTR error messages:

Error	Value	Description
Bad Buffer Length	IC_ERROR_BUFFER_LENGTH (0x0050)	The buffer passed to either SendToHost or SendToQueue is too long.
Bad Route Handle	IC_ERROR_BADROUTE (0x0051)	The Route Handle is invalid.
Queue Locked	IC_ERROR_QUEUELOCKED (0x0053)	The queue is currently locked by another host filter.
Queue not Locked	IC_ERROR_QUEUENOTLOCKED (0x0055)	An attempt was made to perform an operation that requires the queue to be locked, such as SendToQueue.
Wrong session type	IC_ERROR_BADSESSION_ NOTPRINTER (0x0056)	An attempt was made to either call a host specific API call with a queue session handle or call a queue specific API call with a host session handle.
Queue not Requested	IC_ERROR_BADSESSION_ NOTREQUESTED (0x0057)	An attempt was made to lock a queue that has not been requested by this route via the QueueRequest function.
Session Not Open	IC_ERROR_BADSESSION_ NOTOPEN (0x0058)	An attempt was made to lock a session that has not been requested by any routes via the QueueRequest function.
Duplicate Request	IC_ERROR_DUPLICATEREQUEST (0x0059)	An attempt was made to allocate a queue with a QueueAllocate already pending on the same route.
Queue Busy	IC_ERROR_QUEUEBUSY (0x005A)	The queue is not available to be locked by QueueLock or QueueAllocate. This is only returned by QueueAllocate if the timeout period is zero.
Queue Waiting	IC_ERROR_QUEUEWAITING (0x005B)	The QueueAllocate request is pending. The queue is not immediately available, but may become available before the nonzero timeout period elapses.

Troubleshooting

B

In This Appendix

This appendix contains the following topic:

General Troubleshooting Procedures

If you have problems using the PTR Host Filter Development Kit, complete the following steps:

1 **Check to see that your system meets the minimum requirements necessary to use the product.** Refer to "Prerequisites" on page 2 for this information.

2 **Check the configuration options.** Most problems are caused by incorrect software configuration. Check to make sure you have selected the correct configuration options.

3 **Check your connections.** 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 **Resolve errors.** If you are receiving error messages, refer to Appendix A, "Status and Error Messages," beginning on page 135 for information about these errors.

6 **Consult your distributor**. If you cannot identify and solve the problem without assistance, contact your product distributor. Call from a location where you have access to the problem PC.

Index

T

V

W