Virtual Channel SDK Programmer Guide

Version 18.0

Contents

Chapter 1: Using the Virtual Channel SDK 6

Requirements 7
System Requirements 7
Development Environment 7
Execution Environment 7
Build Process 8

Chapter 2: Architecture 9

Virtual Channel Overview 10
ICA and Virtual Channel Data Packets 11
Client WinStation Driver and Virtual Driver 12
Interaction 12
Module.ini 13
Virtual Channel Packets 13
Flow Control 14
Windows Monitoring API 15
Overview 15
Key points 15
Using the APIs 15
Programming guide 20
Programming reference 20
Citrix Dynamic Virtual Channel Protocol 23
Architecture 23
How to write DVC component over ICA 23
Citrix Dynamic Virtual Channel Setup 24
Naming Static Virtual Channel 26
Steps to write DVC component over ICA 27

Chapter 3: Using Example Programs 28

Ping 29
Packet Format 29
Mix 29
Packet Format 30
Sequence of Events 30
Over 32
Packet Format - From Server to Client 33
Packet Format - From Client to Server 33
Sequence of Events 34

Building Examples *35*
Building a Server-side Example using Visual Studio or .NET 35
Building a Client-side Example for Win32 using ..... **Error! Bookmark not defined.**
Visual Studio ..... **Error! Bookmark not defined.**

Preparing and Deploying a Virtual Driver *36*
To deploy the MSI 36
To add a virtual channel after installation 36

Running an Example Virtual Channel *37*

Debugging a Win32 virtual driver *37*

Deploying Client Virtual Channels *38*

Remotely *38*
Administrative Template Changes for Ping Example 40
Best Practices 41

Chapter 4: Programming Guide 42

Design Suggestions *43*

Server-Side Functions Overview *44*

Client-Side Functions Overview *45*
User-Defined Functions 45
Virtual Driver Helper Functions 46
Memory INI Functions 47

Chapter 5: Programming Reference 48

DriverClose *49*

DriverGetLastError *49*

DriverInfo *50*

DriverOpen *52*

DriverPoll *56*

DriverQueryInformation *57*

DriverSetInformation *58*

SendData *59*

ICADataArrival *60*

miGetPrivateProfileBool *61*

miGetPrivateProfileInt *62*

miGetPrivateProfileLong *62*

miGetPrivateProfileString *63*

QueueVirtualWrite *64*

VdCallWd *65*

WFVirtualChannelClose *66*

WFVirtualChannelOpen *67*

WFVirtualChannelPurgeInput *68*

WFVirtualChannelPurgeOutput *68*

WFVirtualChannelQuery *69*

WFVirtualChannelRead *70*

WFVirtualChannelWrite *71*

Disclaimer

This document is furnished "AS IS." Citrix Systems, Inc. disclaims all warranties regarding the contents of this document, including, but not limited to, implied warranties of merchantability and fitness for any particular purpose. This document may contain technical or other inaccuracies or typographical errors. Citrix Systems, Inc. reserves the right to revise the information in this document at any time without notice. This document and the software described in this document constitute confidential information of Citrix Systems, Inc. and its licensors, and are furnished under a license from Citrix Systems, Inc.

Chapter 1

Using the Virtual Channel SDK

Topics:

- ***Requirements***
- ***Installing the Virtual Channel SDK***

The Citrix Virtual Channel Software Development Kit (SDK) provides support for writing server-side applications and client-side drivers for additional virtual channels using the ICA protocol. The server-side virtual channel applications are on XenApp and XenDesktop servers. This version of the SDK provides support for writing new virtual channels for the Citrix plug-ins for Win32. If you want to write virtual drivers for other client platforms, contact Citrix.

The Virtual Channel SDK provides:

- The Citrix Virtual Driver Application Programming Interface (VDAPI) used with the virtual channel functions in the Citrix Server API SDK (WFAPI SDK) to create new virtual channels. The virtual channel support provided by VDAPI is designed to make writing your own virtual channels easier.
- The Windows Monitoring API, which enhances the visual experience and support for third-party applications integrated with ICA.
- Working source code for several virtual channel sample programs that demonstrate programming techniques.

The Virtual Channel SDK requires the WFAPI SDK to write the server side of the virtual channel.

Note: For Presentation Server Client Versions 6.0 to 9.0, use the Virtual Server SDK Version 2.3. To write virtual channel drivers for DOS 32 clients or clients earlier than Version 6.0, use the Virtual Channel SDK Version 2.1.

Requirements

System Requirements

You need to have Citrix Receiver for Windows 4.11 and install WinFrame API SDK.
You can build the virtual drivers and applications on any platform. To run, they require a server running XenApp or XenDesktop.

Development Environment

Use Microsoft Visual Studio 2017 and .NET 4.0. Server-side development also requires the WFAPI SDK.

Although the compiler software packages include C++, only C code is used in this SDK. This SDK has not been tested with any other compilers or any other combinations.

Execution Environment

Server requirement: XenApp 6.5 or higher (earlier versions of XenApp and XenDesktop are also supported but the Windows Monitoring API is currently supported only on XenApp 5 Feature Pack 2 for Windows Server 2003, XenApp 6.0, and XenDesktop 4).

Windows 32 client requirement: Receiver for Windows 4.11

Build Process

The source supplied in this SDK includes Solution Files for use with the Microsoft Visual Studio.

The client-side virtual driver is designed to be built using Visual Studio 2017, the solution file for client-side is located at \src\examples\vc\client\client_examples_win32.sln

You can build this solution from visual interface of Visual Studio 2017, with "Configuration" set to "Release" and "Platform" set to "Win32".

Compiled object files and the binaries generated are placed in the Relese folder located inside the project subfolder.
Eg. for vdmix the output folder would be src\examples\vc\client\vdmix\Release

The server-side examples can be built with Visual Studio 2017, solution files are located at \src\examples\vc\server\server_2017.sln

Compiled object files and the binaries generated are placed in the Release folder located inside the project subfolder.

Eg. for ctxmix built using VS 2017 the output folder would be src\examples\vc\server\ctxmix\Release.

The components can also be built with debugging information turned on. The output directory changes to Debug for debug objects.

Chapter 2

Architecture

Topics:

- ***Virtual Channel Overview***
- ***ICA and Virtual Channel Data Packets***
- ***Client WinStation Driver and Virtual Driver Interaction***
- ***Virtual Channel Packets***
- ***Flow Control***
- ***Windows Monitoring API***

A Citrix Independent Computing Architecture (ICA) virtual channel is a bidirectional error-free connection for the exchange of generalized packet data between a server running Citrix XenApp and a client device. Developers can use virtual channels to add functionality to clients. Uses for virtual channels include:

- Support for administrative functions
- New data streams (audio and video)
- New devices, such as scanners, card readers, and joysticks)

Virtual Channel Overview

An ICA virtual channel is a bidirectional error-free connection for the exchange of generalized packet data between a client and a server running Citrix XenApp or XenDesktop. Each implementation of an ICA virtual channel consists of two components:

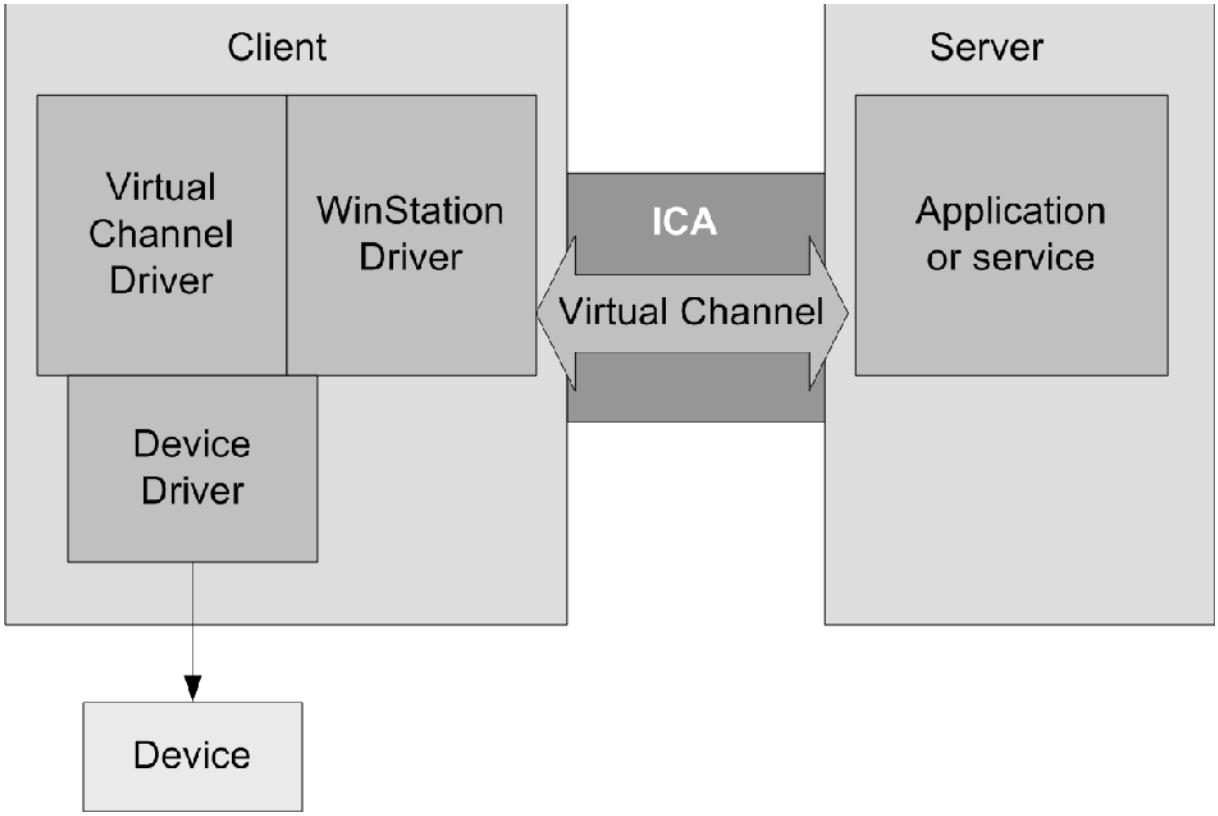
Server-side portion on the computer running XenApp or XenDesktop

The virtual channel on the server side is a normal Win32 process; it can be either an application or a Windows NT service.

Client-side portion on the client device

The client-side virtual channel driver is a dynamically loadable module (.DLL) that executes in the context of the client. You must write your virtual driver.

This figure illustrates the virtual channel client-server connection:



*

The WinStation driver is responsible for demultiplexing the virtual channel data from the ICA data stream and routing it to the correct processing module (in this case, the virtual driver DLL). The WinStation driver is also responsible for gathering and sending virtual channel data to the server over the ICA connection. On the client side, the WinStation driver is also called the client engine, or simply the engine.

The following is an overview of client-server data exchange using a virtual channel:

1. The client connects to the server running XenApp or XenDesktop. The client passes information about the virtual channels it supports to the server.
2. The server-side application starts, obtains a handle to the virtual channel, and optionally queries for additional information about the channel.
3. The client-side virtual driver and server-side application pass data using the following two methods:

    - If the server application has data to send to the client, the data is sent to the client immediately. When the client receives the data, the WinStation driver demultiplexes the virtual channel data from the ICA stream and passes it immediately to the client virtual driver.

    - If the client virtual driver has data to send to the server, the data may be sent immediately, or it may be sent the next time the WinStation driver polls the virtual driver. When the data is received by the server, it is queued until the virtual channel application reads it. There is no way to alert the server virtual channel application that data was received.

4. When the server virtual channel application is finished, it closes the virtual channel and frees any allocated resources.

ICA and Virtual Channel Data Packets

Virtual channel data packets are encapsulated in the ICA stream between the client and the servers. Because ICA is a presentation-level protocol and runs over several different transports, the virtual channel application programming interface (API) enables developers to write their protocols without worrying about the underlying transport. The data packet is preserved.

For example, if 100 bytes are sent to the server, the same 100 bytes are received by the server when the virtual channel is demultiplexed from the ICA data stream. The compiled code runs independently of the currently configured transport protocol.

The ICA engine provides the following services to the virtual channel:

Packet encapsulation
ICA virtual channels are packet-based, meaning that if one side performs a write with a certain amount of data, the other side receives the entire block of data when it performs a read. This contrasts with TCP, for example, which is stream-based and requires a higher-level protocol to parse out packet boundaries. Stated another way, virtual channel packets are contained within the ICA stream, which is managed separately by system software.

Error correction
ICA provides its own reliability mechanisms even when the underlying transport is unreliable. This guarantees that connections are error free and that data is received in the order in which it is sent.

Flow control
The virtual channel API provides several types of flow control. This allows designers to structure their channels to handle only a specific amount of data at any one time. See Flow Control on page 14 for more information.

Client WinStation Driver and Virtual Driver Interaction

Client virtual drivers may elect to send data in either of two modes:

- Polling mode
- Immediate mode

If operating in the polling mode, the WinStation driver polls each virtual driver regularly by calling its DriverPoll function. When DriverPoll is called, the virtual driver should immediately check any necessary state information, send any queued data, and return control to the WinStation driver.

If operating in the immediate mode, the virtual driver may send data at any time. For example, suppose the driver receives a packet from the server in the ICADataArrival function. In the immediate send-data mode, the driver may send data immediately in response to the packet received, and then control will return to the WinStation driver from the ICADataArrival function.

Whenever the virtual driver attempts to send data to the server, it should be prepared for a data send operation to sometimes be declined. This may occur because the WinStation Driver supports a reasonable but not excessive amount of queued backlog waiting to be sent to the server. If a send operation is declined, the virtual driver must arrange to retry the send later.

In any case, whether operating in the polling or immediate mode, the virtual driver must never block. When any of the driver functions are called by the WinStation driver, the virtual driver must immediately check any necessary state information, send any queued data, and return control to the WinStation driver.

The following process occurs when a user starts the client:

1. At client load time, the client engine reads the Configuration Storage in the registry to determine the modules to configure, including how to configure the virtual channel drivers.
2. The client engine loads the virtual channel drivers defined in the Configuration Storage in the registry by calling the Load function, which must be exported explicitly by the virtual channel driver .DLL. The Load function is defined in the static library file Vdapi.lib, which is provided in this SDK. Every driver must link with this library file. The Load function forwards the driver entry points defined in the .DLL to the client engine.

1. For each virtual channel, the WinStation driver calls the DriverOpen function, which establishes and initializes the virtual channel. The WinStation driver passes the address of one of the send-data output functions in the WinStation driver to the virtual channel driver. The virtual channel driver passes the address of the ICADataArrival function to the WinStation driver. The WinStation driver calls the DriverOpen function for each virtual driver when the client loads, not when the virtual channel is opened by the server-side application.
2. When virtual channel data arrives from the server, the WinStation driver calls the ICADataArrival function for that virtual driver.
3. If using the send-data polling mode, the virtual driver cannot initiate data transfers. Instead, the WinStation driver calls DriverPoll to poll for data to send to the server. To send data, the virtual channel driver can use the QueueVirtualWrite function (this address is obtained during

initialization) to send a block of data to the server-side version of the channel. During DriverPoll, the virtual driver may try to send one or more packets (VirtualWrites) until there is no more data to send, or the QueueVirtualWrite function returns the error code CLIENT_ERROR_NO_OUTBUF, to indicate that there is no more buffer space, and that the data in question has not been accepted, and must be retried later (normally on a later DriverPoll).

4. If using the immediate send-data mode, the virtual driver may send data at any time. To send data, the virtual channel driver will use the SendData function (this address is obtained during initialization) to send a block of data to the server-side version of the channel. The virtual driver may try to send one or more packets (VirtualWrites) until there is no more data to send, or the SendData function returns the error code CLIENT_ERROR_NO_OUTBUF, to indicate that there is no more buffer space, and that the data in question has not been accepted, and must be retried later. The retry will normally happen when the WinStation driver presents a special "notification" call to the virtual driver's DriverPoll function. The notification DriverPoll call is made when the WinStation driver detects that buffers have been freed and the send data operation may be retried.

Module.ini

The XenApp plug-ins use settings stored in Module.ini to determine which virtual channels to load. Driver developers can also use Module.ini to store parameters for virtual channels. Module.ini changes are effective only before the installation. After the installation, you must modify the Configuration Storage in the registry to add or remove virtual channels.

Use the memory INI functions to read data from Configuration Storage.

Virtual Channel Packets

ICA does not define the contents of a virtual channel packet. The contents are specific to the particular virtual channel and are not interpreted or managed by the ICA data stream manager. You must develop your own protocol for the virtual channel data.

A virtual channel packet can be any length up to the maximum size supported by the ICA connection. This size is independent of size restrictions on the lower-layer transport. These restrictions affect the server-side WFVirtualChannelRead and WFVirtualChannelWrite functions and the QueueVirtualWrite and SendData functions on the client side. The maximum packet size is 5000 bytes (4996 data bytes plus 4 bytes of packet overhead generated by the ICA datastream manager).

Both the virtual driver and the server-side application can query the maximum packet size. See DriverOpen for an example of querying the maximum packet size on the client side.

Flow Control

ICA virtual channels provide support for downstream (server to client) flow control, but there is currently no support for upstream flow control. Data received by the server is queued until used.

Some transport protocols such as TCP/IP provide flow control, while others such as IPX do not. If data flow control is needed, you might need to design it into your virtual channel.

Choose one of three types of flow control for an ICA virtual channel: **None**, **Delay**, or **ACK**. Each virtual channel can have its own flow control method. The flow control method is specified by the virtual driver during initialization.

None

ICA does not control the flow of data. It is assumed the client can process all data sent. You must implement any required flow control as part of the virtual channel protocol. This method is the most difficult to implement but provides the greatest flexibility. The Ping example does not use flow control and does not require it.

Delay

Delay flow control is a simple method of pacing the data sent from the server. When the client virtual driver specifies delay flow control, it also provides a delay time in milliseconds. The server waits for the specified delay time between each packet of data it sends.

ACK

ACK flow control provides what is referred to as a sliding window. With ACK flow control, the client specifies its maximum buffer size (the maximum amount of data it can handle at any one time). The server sends up to that amount of data. The client virtual driver sends an ACK ICA packet when it completes processing all or part of its buffer, indicating how much data was processed. The server can then send more data bytes up to the number of bytes acknowledged by the client.

This ACK is not transparent—the virtual driver must explicitly construct the ACK packet and send it to the server. The server sends entire packets; if the next packet to be sent is larger than the window, the server blocks the send until the window is large enough to accommodate the entire packet.

Windows Monitoring API

Overview

These APIs allow creating solutions that synchronize the visual aspects of an application that runs on a host (XenApp or XenDesktop) with corresponding visual elements that are running on the Citrix Plug-in. The APIs consist of two different parts: client-side and host-side. The client-side component exposes previously unavailable functionality to third parties. This includes getting information about the ICA window on the client desktop (such as handle, dimensions, panning and scaling), the corresponding client window to a given host window, and setting up a callback function to be called when the ICA window changes. The host component is part of the WinFrame API, and allows for tracking window positions on the host through kernel mode calls.

Key points

The APIs provide the following features:

- Allow efficient tracking of windows on a host through the WinFrame API.
- Provide methods for synchronizing with the client desktop display with the Virtual Channel SDK.
- Provide an improved visual experience and support to third-party applications for better ICA integration.

Using the APIs

Getting started

Headers: wdapi.h
Libraries: wdica30.lib

Architecture

The APIs provide two distinct components with their own architectures: host-side and client-side. The host component is part of the WinFrame API, and provides updates on tracked windows. You can then communicate this data to Citrix Plug-in so as to synchronize window positions. The client-side component in the Virtual Channel SDK then allows third parties to synchronize with the ICA window. It provides them with information about the ICA window's dimensions and handle, as well as whether it is panning or scaling. As a whole, the APIs allow third-party applications to better integrate with ICA and provide a better visual experience.

The client side extends the current WdQueryInformation system in the Virtual Channel SDK to expose functionality that was previously unavailable to third parties. Users call the pre-existing VdCallWd function to call the WinStation driver's QueryInformation function which performs the requested task.

Samples

Get ICA Window Information

This sample shows how information about the ICA window is gathered. It populates a structure with information about the current state of the ICA window. This includes its dimensions, handle, view area dimensions and offset (for example, panning), as well as its current mode (for example, scaling, panning, seamless).

```
WDQUERYINFORMATION wdQueryInfo;
UINT16 uiSize;
int rc;
WDICAWINDOWINFO infoParam;

wdQueryInfo.WdInformationClass = WdGetICAWindowInfo;
wdQueryInfo.pWdInformation = &infoParam;
wdQueryInfo.WdInformationLength = sizeof(infoParam);
uiSize = sizeof(wdQueryInfo);

rc = VdCallWd(g_pVd, WDxQUERYINFORMATION, &wdQueryInfo,
              &uiSize);
if(CLIENT_STATUS_SUCCESS == rc)
{
   // Successfully populated infoParam with ICA window
   // information
}
```

Get Corresponding Client Window

This sample shows how to get the corresponding client window for a given server window.

```
WDQUERYINFORMATION wdQueryInfo;
UINT16 uiSize;
int rc;
HWND window  = 0x42; // example server window handle

wdQueryInfo.WdInformationClass =
              WdGetClientWindowFromServerWindow;
wdQueryInfo.pWdInformation = &window;
wdQueryInfo.WdInformationLength = sizeof(window);
uiSize = sizeof(wdQueryInfo);

rc = VdCallWd(g_pVd, WDxQUERYINFORMATION, &wdQueryInfo,
              &uiSize);

if(CLIENT_STATUS_SUCCESS  == rc)
{
   // Success, pWdInformation now points to the
   //corresponding client window hwnd.
}
```

Register ICA Window Callback

This sample shows how to register a callback function that is called when the ICA window changes. It registers a user defined callback function named Foo. Afterward, whenever the ICA window changes, Foo is called with the current ICA window mode passed in. More information about the ICA window is then gathered using WdGetICAWindowInfo information class, as demonstrated in the first sample.

```
WDQUERYINFORMATION wdQueryInfo;
UINT16 uiSize;
int rc;
WDREGISTERWINDOWCALLBACKPARAMS callbackParams;

callbackParams.pfnCallback = &Foo; // Your callback function
wdQueryInfo.WdInformationClass =
                          WdRegisterWindowChangeCallback;
wdQueryInfo.pWdInformation = &callbackParams;
wdQueryInfo.WdInformationLength = sizeof(callbackParams);
uiSize = sizeof(wdQueryInfo);

rc = VdCallWd(g_pVd, WDxQUERYINFORMATION, &wdQueryInfo,
              &uiSize);
if(CLIENT_STATUS_SUCCESS  == rc)
{
    // Callback successfully registered.
    // Function Foo will be called whenever the ICA window
    // mode, position, or size changes.
}
```

Unregister ICA Window Callback

This sample shows how to unregister a previous ICA window change callback function. It unregisters the callback function Foo from the previous example. The callback function is no longer called when the ICA window changes.

```
WDQUERYINFORMATION wdQueryInfo;
UINT16 uiSize;
int rc;

wdQueryInfo.WdInformationClass =
                          WdUnregisterWindowChangeCallback
wdQueryInfo.pWdInformation = &callbackParams.Handle;
                             // Previously returned handle
wdQueryInfo.WdInformationLength = sizeof(callbackParams.Handle);
uiSize = sizeof(wdQueryInfo);

rc = VdCallWd(g_pVd, WDxQUERYINFORMATION, &wdQueryInfo, &uiSize);
if(CLIENT_STATUS_SUCCESS == rc)
{
    // Callback successfully unregistered
}
```

Programming guide

The APIs as a whole provide better window control for applications that coordinate windows between the host and client desktop. In general, the host uses the WinFrame API component of the API to track windows of interest. The host listens on an assigned mail slot for tracking updates about its windows. These updates are then communicated to the Citrix Plug-in, where they are used to properly position corresponding windows. The Citrix Plug-in uses the client-side portion of the APIs in the Virtual Channel SDK to synchronize its windows with the ICA window. The Citrix Plug-in can be notified when the ICA window changes, and thus make any necessary changes to other third-party applications.

Programming reference

Structures:

WDQUERYINFORMATION

Pre-existing structure passed to the WinStation driver's QueryInformation method. Stores input as well as resulting output.

```
typedef struct _WDQUERYINFORMATION
{
    WDINFOCLASS WdInformationClass;
    LPVOID pWdInformation;
    USHORT WdInformationLength;
    USHORT WdReturnLength;
} WDQUERYINFORMATION, * PWDQUERYINFORMATION;
```

- WdInformationonClass: Set to the enum value corresponding to the API function you want to call.
- pWdInformation: Necessary input parameters, if any, for this function call. If the call returns anything, it is stored here as well.
- WdInformationLength: Set to the size of the input to which pWdInformation point.
- WdReturnLength: Filled in upon return; the size of the return value to which pWdInformation points.

WDICAWINDOWINFO

Struct passed as input when using the WdGetICAWindowInfo information class. Upon successful return, this is populated with information about the ICA window.

```
typedef struct _WDICAWINDOWINFO
{
    HWND hwnd;
    WDICAWINDOWMODE mode;
    UINT32 xWinWidth, yWinHeight, xViewWidth, yViewHeight;
    INT xViewOffset, yViewOffset;
} WDICAWINDOWINFO, * PWDICAWINDOWINFO;
```

- hwnd: ICA window handle.
- mode: Current mode of the ICA window (for example, scaling, panning, seamless).
- xWinWidth: Width of the ICA window.
- yWinHeight: Height of the ICA window.
- xViewWidth: Width of the ICA window's view area.
- yViewHeight: Height of the ICA window's view area.
- xViewOffset: How much the view area is offset in the x dimension (horizontal panning).
- yViewOffset: How much the view area is offset in the y dimension (vertical panning).

WDREGISTERWINDOWCALLBACKPARAMS

Struct passed as input when using the WdRegisterWindowChangeCallback information class.

```
typedef struct _WDREGISTERWINDOWCALLBACKPARAMS
{
    PFNWD_WINDOWCHANGED pfnCallback;
    UINT32 Handle;
} WDREGISTERWINDOWCALLBACKPARAMS, *PWDREGISTERWINDOWCALLBACKPARAMS;
```

- pfnCallback: The user defined function to be called when the ICA window changes (for example, its mode, dimensions, view). This function should have the following header, with the UINT parameter being the current mode of the ICA window (see WDICAWINDOWMODE): typedef VOID (_cdecl * PFNWD_WINDOWCHANGED) (UINT32);
- Handle: Upon successful return this handle is populated. It can later be used to identify the handle when unregistering the callback.

Unions

WDICAWINDOWMODE

Union used to store the ICA window's current mode.

```
typedef union _WDICAWINDOWMODE
{
    struct
    {
       UINT Reserved : 1;
       UINT Seamless : 1;
       UINT Panning : 1;
       UINT Scaling  : 1;
    } Flags;
    UINT Value;
} WDICAWINDOWMODE;
```

- Reserved: Reserved portion of the mode, not currently used.
- Seamless: ICA window is currently in seamless mode.
- Panning: ICA window is currently panning (that is, scrolled vertically/horizontally).
- Scaling: ICA window is currently scaling.
- Value: Raw value of the mode.

Enumerations

The WDINFOCLASS enumeration has four values used by the Windows Monitoring API:

- WdGetICAWindowInfo
- WdGetClientWindowFromServerWindow
- WdRegisterWindowChangeCallback
- WdUnregisterWindowChangeCallback

Citrix Dynamic Virtual Channel Protocol

Architecture

The primary purpose of the DVC protocol is to provide a generic connection-based communication infrastructure over traditional Static Virtual Channels (SVCs).

Dynamic Virtual Channels (or DVCs) are multiplexed over SVCs. In general, one SVC is used per technology remoted over ICA. The DVC protocol provides the ability to create and communicate between logically connected dynamic virtual channel endpoints.

A Dynamic Virtual Channel is an end-to-end connection created between an application running on the ICA host (first endpoint) and an application running on the ICA client (second endpoint, referred to as DVC listener). The end-to-end DVC connection is established and maintained over an ICA connection.

Individual DVC instances are created and maintained by DVC managers. There is a DVC manager running on the host (implemented as a device driver and service) and another on the client (implemented as a virtual driver DLL). The host is responsible for creating dynamic virtual channels and the client is responsible for creating and maintaining connections to client-side DVC applications.

Once the DVC connection is established, both the host and the client-side DVC applications can send data messages to each other. These messages can be initiated by either side, and sending and receiving a message is the same on either side.

The protocol allows for multiple static channels to be used for DVC. By default, each DVC plug-in, representing a specific technology, runs on a separate SVC. This allows the administrator to prioritize individual DVC-remoted technologies by managing the priority of their respective SVC. However, the DVC client can also be configured such that a SVC can be shared by two or more DVC Plug-ins. This may be desirable in the rare case when the number of DVC plug-ins is more than the available SVCs. Currently a maximum of 64 SVCs are supported over ICA.

How to write DVC component over ICA

Microsoft's DVC is implemented over the Remote Desktop Protocol and the Citrix DVC protocol is implemented over the ICA protocol. To write the DVC component over ICA, Microsoft's DVC API can be used.

The Microsoft DVC client-side APIs are found in:
http://msdn.microsoft.com/en-us/library/bb540853(VS.85).aspx

The server-side APIs are found in:
http://msdn.microsoft.com/en-us/library/bb540857(VS.85).aspx.

Citrix Dynamic Virtual Channel Setup

The following steps occur during the lifetime of a dynamic virtual channel:

1. The client DVC Manager enumerates all the registered DVC plug-ins and sends the list to the host, which consists of pairs of DVC Plug-in Friendly Name and corresponding SVC Name. The DVC plug-in Friendly Name could be a DLL name or any other friendly name available. The SVC name is either administrator-assigned or generated from the friendly name. The SVC name is always truncated to 7 characters plus a NULL terminator (8 total), which is the SVC name size used by the ICA protocol. The list is sent as part of the CAPABILITY_DYNAMIC_VIRTUAL_CHANNEL ICA capability exchanged during the initial ICA handshake at the WinStation Driver (WD) level. See the ICA3.0 protocol specification (ica30.doc) for details on the format of CAPABILITY_DYNAMIC_VIRTUAL_CHANNEL.

   Remarks:

   - Normally, DVC plug-ins are registered according to the requirements defined by Microsoft. Citrix provides additional DVC plug-in registration for two purposes:

     a. Enumeration of known 3rd party plug-ins that are not properly registered and cannot be enumerated otherwise.
     b. Optional assignment by administrator of explicit SVC name per plug-in.

   - If the SVC name is generated from a friendly name, as opposed to administrator-assigned, then accidental collision with other SVCs is avoided as follows:

     a. First the original name is truncated to 6 characters. Then a decimal digit is appended starting from 0 and up to 9 such that the new name is unique. If the name still is not unique, then step b is performed.
     b. First the original name is truncated to 5 characters. Then two decimal digits are appended starting from 0 and up to 99 such that the new name is unique.

   - Name collision may occur with both SVCs used for DVC and other standard Citrix or 3rd party SVCs. The range from 0 to 99 used to create unique SVC names is sufficient, since currently ICA support only up to 64 SVCs.

   - The Client DVC Manager registers N number of SVCs with the WD in DriverOpen. In general, this will be 1 SVC per DVC Plug-in enumerated. However, plug-ins may share the same SVC name if:

     a. The administrator has explicitly assigned the same SVC name for more than one DVC Plug in via the Citrix ICA Client DVC Registration.
     b. There are no more SVCs available.

   - In the future the Client DVC Manager might assign and load a separate SVC per DVC listener (as opposed to plug-in). Currently, this is not possible because new listeners may become available at any point after loading of a plug-in and the current ICA architecture does not allow dynamic loading of SVCs at the client. All SVCs are loaded during the ICA handshake.

2. The host DVC Manager opens a SVC for each channel name received from the client via the CAPABILITY_DYNAMIC_VIRTUAL_CHANNEL ICA capability.

3. The host sends a list of supported DVC capabilities to the client and requests the list of client-

supported capabilities. Currently, for optimization purposes, DVC capabilities are negotiated over one of the opened SVCs and they are assumed to apply to all SVCs.

4. The client responds with a list of supported capabilities. The list is sent over the same SVC.

5. The host then commits the DVC capabilities to be used for the lifetime of the ICA connection. The list is sent over the same SVC.

6. For each DVC Plug-in the client sends a list of all currently available listeners hosted by the DVC Plug-in. Each list is sent over the respective SVC:

   a. Immediately after the DVC capabilities are committed by the host.
   b. And at any point a new listener starts or an existing listener shuts down.

   Remarks: Although it is theoretically possible for a listener to shut down at any point, in practice once a listener starts, it does not shut down until the whole DVC plug-in shuts down.

7. The host reads the list of listeners, caches any future updates from the client and keeps a mapping in a table.

8. When subsequently the Virtual Channel Open API (WTSVirtualChannelOpenEx) is called by a host application, the host looks up the listener name in its table, assigns a new Channel Number in the range from 0 to 64K and links it to the respective listener. The Channel Number is unique within a listener. A DVC Create request is then sent over the SVC associated with the listener. The client responds over the same SVC with success or failure to create the channel (DVC instance) on the specified listener. The client's response is communicated to the host application.

9. If a DVC channel is successfully created, data messages can be exchanged between the application running on the ICA host and the DVC Listener running on the ICA client. Sending and receiving messages is symmetrical between the host and client, and either side can initiate sending a message.

10. Eventually a DVC channel is closed by either the host or the client. A close is triggered when the host application closes the DVC instance handle but can also be triggered by a client listener or by the client or host DVC Managers, for example, upon error.

11. All DVC packets are sent over SVC packets.

12. When the ICA client exists, the client DVC Manager shuts down and unloads all the DVC Plug-ins, which in turn shut down their listeners.

Naming Static Virtual Channel

The Channel Name field provides a name for the static virtual channel to use for a specific DVC plug-in. By default the static channel name to use will be automatically generated using the module file name of the DVC plug-in. To ensure that a unique name is generated, upon collision one or two digits may be used at the end of the name to make it unique while keeping the name length at a maximum of seven characters. The channel name field is explained as follows:

Section: ChannelName
Feature: DVC
Attribute Name: INI_DVC_PLUGIN_<DVC plugin name>
Definition location: inc\icaini.h
Data Type: String
Access Type: Read
Unix Specific: No
Present in ADM: No

Values:

	Static virtual channel name	

INI Location:

INI File	Section	Value
Module.ini	[DVC_Plugin_<DVC plugin name>]	

Registry Location:

Registry Key	Value
HKEY_LOCAL_MACHINE\SOFTWARE\Citrix\ICA Client\Engine\Configuration\Advanced\Modules\DVC_Plugin_<DVC plugin name>	*

The static virtual channel name can be modified using the above locations if the administrator wants to give the explicit name.

Steps to write DVC component over ICA

This section explains how to write a DVC component over ICA using an example. Citrix DVC protocol uses the existing interfaces provided by Microsoft to develop DVC components over ICA. For more details on how to write DVC Server components and DVC Client components refer to:

http://msdn.microsoft.com/en- us/library/bb540858(VS.85).aspx
and
http://msdn.microsoft.com/en-us/library/bb540854(VS.85).aspx
respectively.

Chapter 3

Using Example Programs

Topics:

- ***Ping***
- ***Mix***
- ***Over***
- ***Building Examples***
- ***Preparing and Deploying a Virtual Driver***
- ***Running an Example Virtual Channel***
- ***Debugging a Win32 virtual driver***
- ***Deploying Client Virtual Channels Remotely***

The example programs included with the Virtual Channel SDK are buildable, working virtual channels. Use these examples to:

- Verify your Virtual Channel SDK installation is correct by building a known working example program.
- Provide working examples of code that can be modified to suit your requirements.
- Explore the features and functionality provided in the Virtual Channel SDK.

Each of these example programs comprises a client virtual driver and a server application. The server-side application is run from the command line within an ICA session. A single virtual channel comprises an application pair.

The example programs included with the Virtual Channel SDK are:

- Ping: Records the round-trip delay time for a test packet sent over a virtual channel.
- Mix: Demonstrates a mechanism to call functions (for example, to get the time of day) on a remote client.
- Over: Simple asynchronous application that demonstrates how to code an application where the server must receive a response from the client asynchronously, and where the type of packet being sent to the client is different from the type received.

Each example includes a description of the program, packet format, and other necessary information.

Ping

Ping is a simple program that records the round-trip delay time for a test packet sent over a virtual channel. The server sends a packet to the client and the client responds with a packet containing the time it received the original packet from the server. This sequence is repeated a specified number of times, and then the program displays the round-trip time for each ping and the average round-trip delay time.

For this example, there is no significant difference between a BEGIN packet and an END packet. The two types of packets are provided as an example for writing your own virtual channel protocols.

This program demonstrates:

- How to transfer data synchronously. The sequence of events is: {SrvWrite, ClntRead, ClntWrite, SrvRead} {SrvWrite, ClntRead} {...}. The server waits for the client to reply before sending the next packet.
- How to read parameter data (in this case, the number of times to send packets to the client) from the Module.ini files.

Ping uses the SendData function to transmit data immediately, rather than waiting to be polled.

Packet Format

The following packet is exchanged between the client and the server.

```
typedef struct PING
{
     USHORT uSign;         // Signature
     USHORT uType;         // Type, BEGIN or END, from server
     USHORT uLen;          // Packet length from server
     USHORT uCounter;      // Sequencer
     ULONG  ulServerMS;    // Server millisecond clock
     ULONG  ulClientMS;    // Client millisecond clock
} PING, *PPING;
```

Mix

Mix demonstrates a mechanism that can be used to call functions on a remote client (for example to get the time of day). This program demonstrates an extensible scheme for making function calls from the server to the client that allows the server to specify when it expects a response from the client and when it does not. This method can increase performance, because the server does not have to constantly wait for a reply from the client.

The server calls a series of simple functions:

- AddNo: Add two numbers and return the sum as the return value.
- DispStr: Write a string to the log file. There is no return value (write-only function).

- Gettime: Read the client time and return it as the return value. The actual implementation of these functions is on the client side. The server conditionally waits for the response from the client, depending on the function being executed. For example, the server waits for the result of the AddNo or Gettime function, but not the write-only function DispStr, which returns no result.

Packet Format

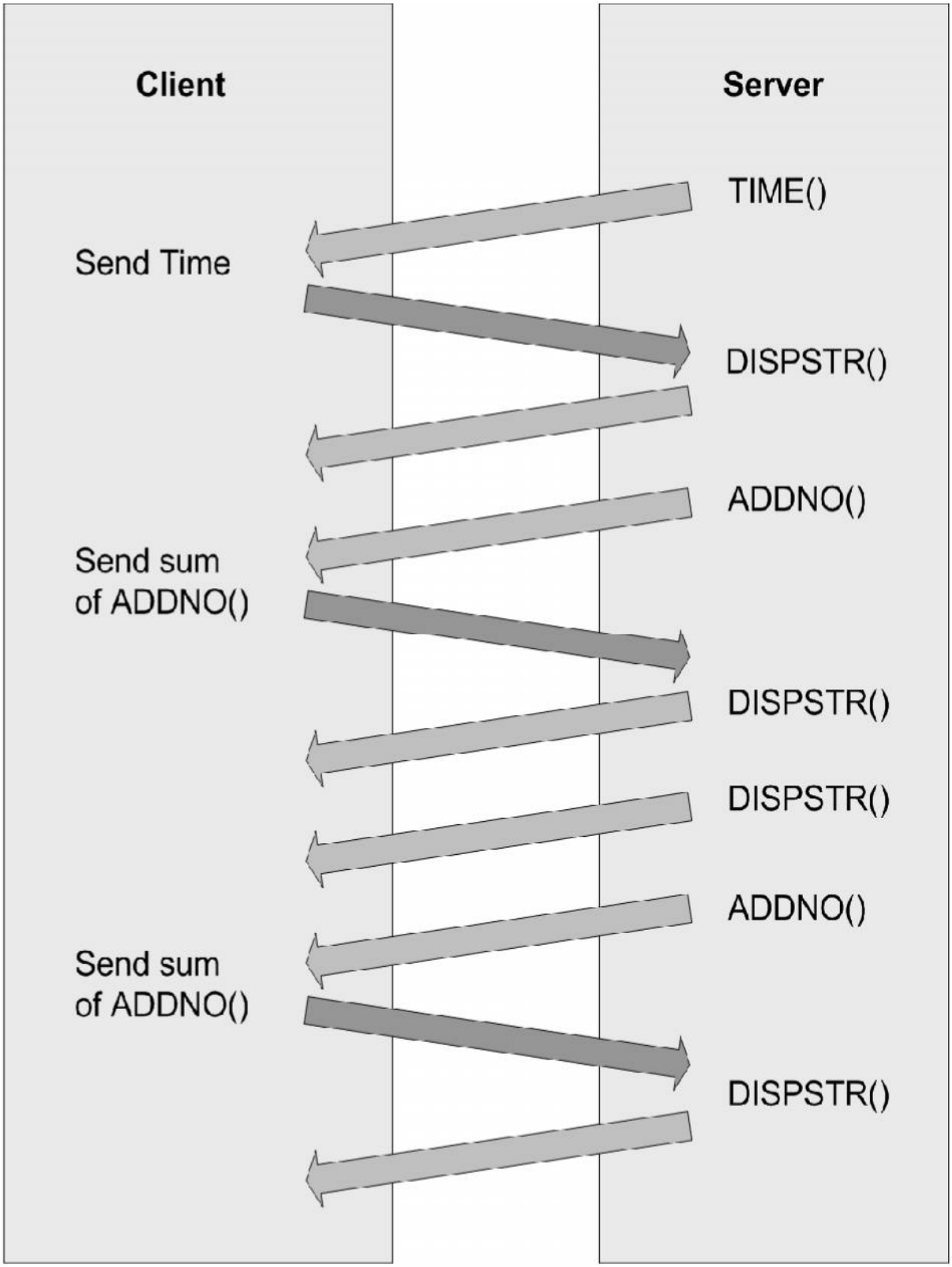
```
typedef struct MIXHEAD
{
     USHORT   uType;         // Packet type
     USHORT   uFunc;         // Index of Function
     ULONG    uLen;          // Length of data
     USHORT   fRetReq;       // True if return value required
     ULONG    dwRetVal;      // Return Value from client
     USHORT   dwLen1;        // length of data for #1 LpVoid
     USHORT   dwLen2;        // length of data for #2 LpVoid
} MIXHEAD, *PMIXHEAD;
```

The data consists of the above structure followed by the arguments to the function being called. uLen is the total length of the data being sent, including the arguments. DwLen1 is the length of the data pointed to by a pointer argument.

Sequence of Events

The Mix program demonstrates the following sequence of events. See the graphic on the next page.

This figure illustrates the sequence of events that occurs when you use the Mix program, starting at the top.



Over

Over is a simple asynchronous application. It demonstrates how to code an application in which the server must receive a response from the client asynchronously, and the type of packet being sent to the client is different from the type received.

When the Over program begins, it:

1. Spawns a thread that waits for a response from the client.
2. Begins sending data packets with sequence numbers to the client.
3. (After sending the last packet of data) sends a packet with a sequence number of NO_MORE_DATA, and then closes the connection.

The client receives packets and inspects the sequence number. For every sequence number divisible by 10, the client increases the sequence number by 7 and sends a response to the server. These numbers are chosen arbitrarily to demonstrate that the client can asynchronously send data to the server at any time.

The packet type used to send data from the server to the client is different from the packet type used to receive data from the client.

Packet Format - From Server to Client

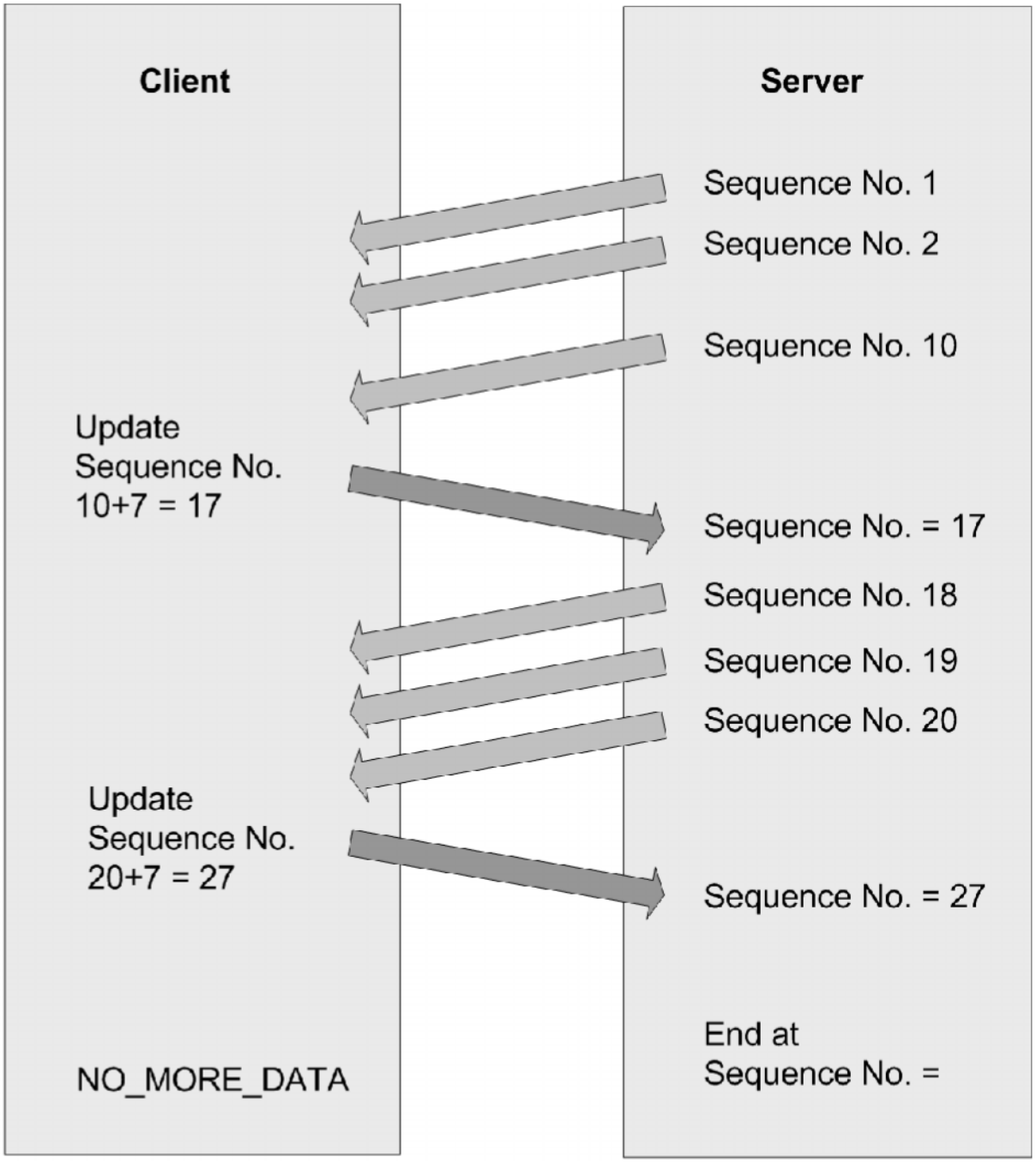
```
typedef struct OVER
{
     USHORT uSign;        // Signature
     USHORT uType;        // Type, BEGIN or END, from server
     USHORT uLen;         // Packet length from server
     USHORT uCounter;     // Sequencer
     ULONG  ulServerMS;   // Server millisecond clock
} OVER, *POVER;
```

Packet Format - From Client to Server

```
typedef struct DRVRESP
{
     USHORT  uType;       // Type OVERFLOW_JUMP from client
     USHORT  uLen;        // Packet length from client
     USHORT  uCounter;    // seqUencer
} DRVRESP, * PDRVRESP;
```

Sequence of Events

This figure illustrates the sequence of events that occurs when you use the Over program, starting at the top.



Building Examples

Building a Server-side Example using Visual Studio or .NET

1. Create a new Win32 console project. Citrix recommends that the project name be associated with the example (for example, ctxping). You can set the location of the project to the src\examples\vc\server directory so that the .c source files are readily available.

2. Add the following directories to include the search path of the C++ preprocessor in the project settings (where vcsdk is the directory in which you installed the Virtual Channel SDK):

   - vcsdk\src\examples\vc\shared\inc
   - vcsdk\src\shared\inc\Citrix

3. Point to the wfapi include and library paths. Open file wfapi.mak from vcsdk installation path\src\examples\build.

   - Set WFAPILIB to the full path of WFAPI lib directory.
   - Set WFAPIINC to the full path of WFAPI include directory.

   The WFAPI SDK installs Wfapi.lib into the designated library directory.

Building a Client-side Example for Win32 using Visual Studio

1. Change to directory <vcsdk_unzipped_location>\src\examples\vc\
2. At the command prompt, set an environment variable:

   WFAPILIBPATH = C:\Program Files (x86)\Citrix\WfApiSDK

   or

   WFAPILIBPATH = C:\Program Files\Citrix\WfApiSDK

   depending on architecture.
3. For each example you want to build, type:

   cd client

   msbuild client_examples_win32.sln /p:Configuration=Release /p:Platform=win32 /verbosity:detailed

Preparing and Deploying a Virtual Driver

Before installing a virtual driver on a client, copy the virtual driver for the platform to the client device and configure the client MSI.

1. Copy the appropriate virtual driver for the platform to the directory on the device where the client is installed. The virtual driver is the .Dll file in \src\examples\vc\client\driver\platform\obj\retail, where driver is vdmix, vdover, or vdping. The default installation directory for the Win32 client is %SystemDrive%\Program Files\Citrix\ICA Client.
2. Open the standard client MSI package with the Microsoft packaging tools (for example, Orca in the Windows Installer SDK).
3. Add the virtual channel .DLL to the MSI package.
4. Modify the Configuration Storage file /configuration/module.ini.

   Caution: Editing the Registry incorrectly can cause serious problems that may require you to reinstall your operating system. Citrix cannot guarantee that problems resulting from the incorrect use of Registry Editor can be solved. Use the Registry Editor at your own risk. Be sure to back up the registry before you edit it.

   a. Locate the VirtualDriverEx string REG_SZ value in the HKEY_LOCAL_MACHINE\SOFTWARE\Citrix\ICA Client\Engine\Configuration\Advanced\Modules\ICA3.0 key. Append the name of the virtual driver to the end of this line, for example: VirtualDriverEx = Ping.

   b. Under the HKEY_LOCAL_MACHINE\SOFTWARE\Citrix\ICA Client\Engine\Configuration\Advanced\Modules key, create a new <driver> key, where <driver> is Mix, Over, or Ping. For Ping, the section would be: HKEY_LOCAL_MACHINE\SOFTWARE\Citrix\ICA Client\Engine\Configuration\Advanced\Modules\Ping. Add the following string REG_SZ values under the above key:

```
DriverName              = VDPING.DLL
DriverNameWin16         = VDPINGW.DLL

DriverNameWin32         = VDPINGN.DLL
PingCount               = 3
```

   The client engine uses DriverName, DriverNameWin16, and DriverNameWin32 to determine the module filename to load for each platform. PingCount is a tunable parameter used by the Ping virtual channel.

5. Repackage the MSI for deployment.

To deploy the MSI

Deploy your MSI package with Windows Active Directory Services or Microsoft Systems Management Server. See your Windows or Systems Management Server documentation for more information. No further configuration is necessary.

To add a virtual channel after installation

Because the Module.ini file is installed in the registry, modifying the file after installation has no effect. To add a virtual channel after installation, use the Group Policy template or

change the registry keys corresponding to those in the Module.ini file at the following registry location:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Citrix\ICAClient\Engine\Configuration\Advanced
\Modules
```

Running an Example Virtual Channel

1. On a client configured with the client-side example, connect to a server running XenApp with the associated server-side example.
2. Within the ICA session, run the server-side executable.

   The server-side example queries the client-side virtual driver, and then displays the driver information. Use the -d parameter to display detailed information.

   For Ping only: CTXPING sends PingCount separate pings. PingCount has a default value of three, but can be set in the [Ping] section of the Module.ini file. Each ping consists of a BEGIN packet and an END packet.

Debugging a Win32 virtual driver

Use the TRACE feature to log events on the client. To enable the TRACE statements, you must build the debug version of the virtual driver. When the debug module is installed on the client, the TRACE statements write the debug information to a file.

At run time, you can specify which class and event flags to trace. This allows you to trace only the sections you need, minimizing performance degradation.

The class flag for virtual channels is 00000080. For the complete list of class and event flags, see Logflags.h (in src\inc\).

1. Compile the debug version of the virtual driver for the client platform.
2. If it is running, close the client on the client device.
3. Copy the compiled debug version of the library into the directory on the client device where the client is installed. For example, for the Ping example, copy VdpingN.dll to C:\Program Files\Citrix\ICA Client.
4. Change to the directory containing the client and type:
   wfcrun32 connection /c:xxxxxxxx /e:yyyyyyyy /logfile:filename where:
   connection is the name of the connection in Remote Application Manager.
   xxxxxxxx are the event flags you want to log.
   yyyyyyyy are the class flags you want to log.
   filename is the relative path of the file to which you want to save the log.
   The client stores the Appsrv.ini file in each user's profile directory. When starting the ICA session with event logging, add /iniappsrv:%userprofile%\"application data"\icaclient\appsrv.ini to the end of the command line above.

Deploying Client Virtual Channels Remotely

To deploy virtual channels remotely, make changes based on the following administrative template (.adm) file.

CustomVC is a placeholder for the channel name of the virtual channel.

```
;Group Policy template for Citrix Online Plug-in.
;Citrix Online Plug-in Client Extensions template
;Description:
;This file is provided as a base for third-party extensions
;to the Citrix Online Plug-in client.
;Copyright (C) Citrix Systems, Inc. All Rights Reserved.
;
CLASS MACHINE
CATEGORY !!Citrix
  CATEGORY !!ICAClient
    CATEGORY !!Third Party
      #if version >= 4
      EXPLAIN !!Explain_Third Party
      #endif
      ; Continued below...
```

```
        ; Continued from above
        ; Remotely define virtual channel
        POLICY !!Policy_CustomVirtualChannel
          EXPLAIN !!Explain_CustomVirtualChannel
          KEYNAME "Software\Policies\Citrix\ICA Client\Engine\Lockdown
          Profiles\All Regions\Lockdown\Virtual Channels\Third
          Party\CustomVC"
          VALUENAME "VCEnable"
          VALUEON"true,false" VALUEOFF"false" ACTIONLISTON
          KEYNAME"Software\Citrix\ICA Client\Engine\Lockdown
          Profiles\All Regions\Lockdown\Virtual Channels\Third
          Party\CustomVC"
                  VALUENAME "VCEnable"
                  VALUE""
                  KEYNAME "Software\Citrix\ICA
          Client\Engine\Configuration\Advanced\Modules\ICA 3.0"
                  VALUENAME "VirtualDriverEx"
                  VALUE"CustomVC"
                  KEYNAME "Software\Citrix\ICA
          Client\Engine\Configuration\Advanced\Modules\CustomVC"
                  VALUENAME "DriverName"
                  VALUE"Unsupported"
                  KEYNAME "Software\Citrix\ICA
          Client\Engine\Configuration\Advanced\Modules\CustomVC"
                  VALUENAME "DriverNameWin16"
                  VALUE"Unsupported"
                  KEYNAME "Software\Citrix\ICA
          Client\Engine\Configuration\Advanced\Modules\CustomVC"
                  VALUENAME
                  "DriverNameWin32" VALUE
                   "VDCustomVC.DLL"
               END ACTIONLISTON
               ACTIONLISTOFF
                  KEYNAME "Software\Citrix\ICA
          Client\Engine\Configuration\Advanced\Modules\ICA 3.0"
                  VALUENAME
                  "VirtualDriverEx" VALUE""
                  END ACTIONLISTOFF
         END POLICY
      END CATEGORY
    END CATEGORY
END CATEGORY
[strings]
Citrix="Citrix Components"
ICAClient="Presentation Server Client"
Third Party="Extensions"
Explain_Third Party="These policies control extensions to the
        standard Citrix Presentation Server Client."
Policy_CustomVirtualChannel="Additional Virtual Channel"
Explain_CustomVirtualChannel=" This policy controls a virtual
       channel.\n\nSupplier:\nMy company.\n\nReference:"
```

Administrative Template Changes for Ping Example

For the ping virtual channel example, edit the .adm template file as follows (changes in the text are in boldface).

Note: The Memory INI functions require the lines in the example referencing VCEnable. Every parameter used by the virtual channel must appear in this file. The client uses these to place security restrictions on the virtual channels.

```
;
;  Group Policy template for Citrix Online Plug-in Client
;  Ping virtual channel example template
;  Description:
;  A Group Policy template to remotely configure the Ping
;  Virtual Channel
;
;  Copyright (C) Citrix Systems. Inc. All Rights Reserved.
;
CLASS MACHINE
CATEGORY !!Citrix
  CATEGORY !!ICAClient
    CATEGORY !!Third Party
      #if version >= 4
       EXPLAIN !!Explain_Third Party
      #endif
      ;
      ; Remotely configure the Ping Virtual Channel
      ;
      POLICY !!Policy_PingVirtualChannel
       EXPLAIN !!Explain_PingVirtualChannel
       KEYNAME "Software\Policies\Citrix\ICA Client\Engine\Lockdown
       Profiles\All Regions\Lockdown\Virtual Channels\Third
       Party\ping"
          VALUENAME "VCEnable"
          VALUEON"true,false"
          VALUEOFF"false"
       ACTIONLISTON
          KEYNAME"Software\Citrix\ICA Client\Engine\Lockdown\
                 Profiles\All Regions\Lockdown\Virtual
                 Channels\Third Party\ping"
            VALUENAME "VCEnable"
            VALUE ""
          KEYNAME"Software\Citrix\ICA Client\Engine\Configuration\
                 Advanced\Modules\ICA 3.0"
            VALUENAME "VirtualDriverEx"
            VALUE "ping"
          KEYNAME"Software\Citrix\ICA Client\Engine\Configuration\
                 Advanced\Modules\pin g "
            VALUENAME "DriverName"
            VALUE "Unsupported"
          ; Continued below...
```

```
            ; Continued from above.
            KEYNAME "Software\Citrix\ICA Client\Engine\Configuration\
                   Advanced\Modules\ping"
              VALUENAME "DriverNameWin16"
              VALUE "Unsupported"
            KEYNAME "Software\Citrix\ICA Client\
                   Engine\Configuration\Advanced\Modules\ping"
              VALUENAME "DriverNameWin32"
              VALUE "vdpingn.dll"
          END ACTIONLISTON
          ACTIONLISTOFF
            KEYNAME "Software\Citrix\ICA Client\Engine\
                   Configuration\Advanced\Modules\ICA3.0"
              VALUENAME "VirtualDriverEx"
              VALUE ""
          END ACTIONLISTOFF
        END POLICY
      END CATEGORY
    END CATEGORY
END CATEGORY
[strings]
  Citrix="Citrix Components"
  ICAClient="Presentation Server Client"
  Third Party="Extensions"
  Explain_Third Party="These policies control extensions to the standard
                       Citrix Presentation Server Client."
  Policy_PingVirtualChannel="Example Ping Virtual Channel"
  Explain_PingVirtualChannel=" This policy controls the example
                       Ping virtual channel.\n\nSupplier:\n
                       My company.\n\nReference:Example001"
```

Best Practices

Citrix recommends using the .adm file to customize the following parts of the Group Policy GUI:

- Specify a name (Additional Virtual Channel in the template file) that describes the functionality provided by the virtual channel.
- Description text
- Supplier
- Reference (for example, add a URL or email address to access further information).
- GUI that appears when the policy is double-clicked (optionally and as required).

  Do not change the name of the Citrix Components \Presentation Server Client

  \Extensions folder.

Deploy the virtual channel DLL remotely using existing management tools and enable and configure using the above GUI. You can do this to entire groups of computers within an organization.

Chapter 4

Programming Guide

Topics:

- ***Design Suggestions***
- ***Server-Side Functions Overview***
- ***Client-Side Functions Overview***

Virtual channels are referred to by a seven-character (or shorter) ASCII name. In several previous versions of the ICA protocol, virtual channels were numbered; the numbers are now assigned dynamically based on the ASCII name, making implementation easier.

When developing virtual channel code for internal use only, you can use any seven-character name that does not conflict with existing virtual channels. Use only upper and lowercase ASCII letters and numbers. Follow the existing naming convention when adding your own virtual channels.

The predefined channels, which begin with the OEM identifier CTX, are for use only by Citrix.

Design Suggestions

Follow these suggestions to make your virtual channels easier to design and enhance:

- When you design your own virtual channel protocol, allow for the flexibility to add features. Virtual channels have version numbers that are exchanged during initialization so that both the client and the server detect the maximum level of functionality that can be used. For example, if the client is at Version 3 and the server is at Version 5, the server does not send any packets with functionality beyond Version 3 because the client does not know how to interpret the newer packets.
- Because the server side of a virtual channel protocol can be implemented as a separate process, it is easier to write code that interfaces with the Citrix-provided virtual channel support on the server than on the client (where the code must fit into an existing code structure). The server side of a virtual channel simply opens the channel, reads from and writes to it, and closes it when done.

  Writing code for the server-side is similar to writing an application, which uses services exported by the system. It is easier to write an application to handle the virtual channel communication because it can then be run once for each ICA connection supporting the virtual channel.

  Writing for the client-side is similar to writing a driver, which must provide services to the system in addition to using system services. If a service is written, it must manage multiple connections.
- If you are designing new hardware for use with new virtual channels (for example, an improved compressed video format), make sure the hardware can be detected so that the client can determine whether or not it is installed. Then the client can communicate to the server if the hardware is available before the server uses the new data format. Optionally, you could have the virtual driver translate the new data format for use with older hardware.
- There might be limitations preventing your new virtual channel from performing at an optimum level. If the client is connecting to the server running XenApp through a modem or serial connection, the bandwidth might not be great enough to properly support audio or video data. You can make your protocol adaptive, so that as bandwidth decreases, performance degrades gracefully, possibly by sending sound normally but reducing the frame rate of the video to fit the available bandwidth.
- To identify where problems are occurring (connection, implementation, or protocol), first get the connection and communication working. Then, after the virtual channel is complete and debugged, do some time trials and record the results. These results establish a baseline for measuring further optimizations such as compression and other enhancements so that the channel requires less bandwidth.
- The time stamp in the pVdPoll variable can be helpful for resolving timing issues in your virtual driver. It is a ULONG containing the current time in milliseconds. The pVdPoll variable is a pointer to a DLLPOLL or DLL_HPC_POLL structure. See Dllapi.h (in src\inc\) for definitions of these structures.

Server-Side Functions Overview

Server-side functions are entry points to virtual channel services provided by the ICAsubsystem on the XenApp or XenDesktop server. Wfapi.h contains constants and function prototypes.

Use these functions to open and close virtual channels and to read, write, query, and purge incoming or outgoing data.

The words IN and OUT in the function calling conventions are for clarification only. They are defined as blank in Windef.h. If you do not have access to Windef.h, add the following to a header file for your project:

```
#ifndef IN
#define IN
#endif
#ifndef OUT
#endif
```

Function	Description
WFVirtualChannelClose on page 66	Closes an open virtual channel handle.
WFVirtualChannelOpen on page 67	Opens a handle to a specific virtual channel.
WFVirtualChannelPurgeInput on page 70	Purges all queued input data sent from the client to the server on a specific virtual channel.
WFVirtualChannelPurgeOutput on page 68	Purges all queued output data sent from the server to the client on a specific virtual channel.
WFVirtualChannelQuery on page 69	Returns data related to a virtual channel.
WFVirtualChannelRead on page 70	Reads data from a virtual channel.
WFVirtualChannelWrite on page 71	Writes data to a virtual channel.

Client-Side Functions Overview

The client software is built on a modular configurable architecture that allows replaceable, configurable modules (such as virtual channel drivers) to handle various aspects of an ICA connection. These modules are specially formatted and dynamically loadable. To accomplish this modular capability, each module (including virtual channel drivers) implements a fixed set of function entry points.

There are three groups of functions: user-defined, virtual driver helper, and memory INI.

User-Defined Functions

To make writing virtual channels easier, dynamic loading is handled by the WinStation driver, which in turn calls user-defined functions. This simplifies creating the virtual channel because all you have to do is fill in the functions and link your virtual channel driver with Vdapi.lib (provided with this SDK).

Function	Description
DriverClose on page 48	Frees private driver data. Called before unloading a virtual driver (generally upon client exit).
DriverGetLastError on page 49	Returns the last error set by the virtual driver. Not used; links with the common front end, VDAPI.
DriverInfo on page 50	Retrieves information about the virtual driver.
DriverOpen on page 52	Performs all initialization for the virtual driver. Called once when the client loads the virtual driver (at startup).
DriverPoll on page 56	Allows driver to check timers and other state information, sends queued data to the server, and performs any other required processing. Called periodically to see if the virtual driver has any data to write.
DriverQueryInformation on page 57	Retrieves run-time information from the virtual driver.
DriverSetInformation on page 58	Sets run-time information in the virtual driver.
ICADataArrival on page 60	Indicates that data was delivered. Called when data arrives on the virtual channel.

Virtual Driver Helper Functions

The virtual driver uses helper functions to send data and manage the virtual channel. When the WinStation driver initializes the virtual driver, the WinStation driver passes pointers to the helper functions and the virtual driver passes pointers to the user-defined functions.

VdCallWd is linked in as part of VDAPI and is available in all user-implemented functions. The others are obtained during DriverOpen when VdCallWd is called with the WDxSETINFORMATION parameter.

Virtual channel drivers can send data from private buffers via the SendData or QueueVirtualWrite functions obtained during DriverOpen. Either of these functions may decline to accept the data if the WinSation Driver itself cannot buffer it. The channel will then need to retry the send operation on the next DriverPoll.

Function	Description
SendData on page 59	To send a packet of channel protocol to the server, with a notification option.
QueueVirtualWrite on page 64	To send a packet of channel protocol to the server. This is a legacy function. Use SendData above for new virtual drivers.
VdCallWd on page 65	Used to query and set information from the WinStation driver (WD).

Memory INI Functions

Memory INI functions read data that the client engine reads from the Configuration Storage in the registry and stored in a memory INI structure. These functions must be used because some client devices store this information in ROM, and only the engine has access to this INI data. The Memory INI functions read values from this Memory INI structure as if the values are being read from the Configuration Storage. The Configuration Storage for specifying virtual channels is in Software\Citrix\ICA Client\Configuration\Advanced\Modules\.

Important: Access to configuration data might be limited depending on security restrictions. In particular, the virtual channel might not have access to all contents of the ICA file. This is controlled by registry keys in HKEY_LOCAL_MACHINE\SOFTWARE\Citrix\ICA Client\Engine\Lock down Profiles\All Regions\Lock down. You can use the Group Policy file to modify the registry keys.

Function	Description
miGetPrivateProfileBool on page 61	Returns a Boolean value.
miGetPrivateProfileInt on page 62	Returns an integer value.
miGetPrivateProfileLong on page 60	Returns a long value.
miGetPrivateProfileString on page 63	Returns a string value.

Chapter 5

Programming Reference

Topics:

- ***DriverClose***
- ***DriverGetLastError***
- ***DriverInfo***
- ***DriverOpen***
- ***DriverPoll***
- ***DriverQueryInformation***
- ***DriverSetInformation***
- ***SendData***
- ***ICADataArrival***
- ***miGetPrivateProfileBool***
- ***miGetPrivateProfileInt***
- ***miGetPrivateProfileLong***
- ***miGetPrivateProfileString***
- ***QueueVirtualWrite***
- ***VdCallWd***
- ***WFVirtualChannelClose***
- ***WFVirtualChannelOpen***
- ***WFVirtualChannelPurgeInput***
- ***WFVirtualChannelPurgeOutput***
- ***WFVirtualChannelQuery***
- ***WFVirtualChannelRead***
- ***WFVirtualChannelWrite***

For function summaries , see:

- Server-Side Functions Overview on page 44
- Client-Side Functions Overview on page 45

DriverClose

The WinStation driver calls this function prior to unloading the virtual driver, when the ICA connection is being terminated.

Calling Convention

```
INT Driverclose( PVD pVD, PDLLCLOSE pVdClose, PUINT16 puiSize);
```

Parameters

pVD
Pointer to a virtual driver control structure.

pVdClose
Pointer to a standard driver close information structure.

puiSize
Pointer to the size of the driver close information structure. This is an input parameter.

Return Values

If the function succeeds the return value is CLIENT_STATUS_SUCCESS.

If the function fails, the return value is the CLIENT_ERROR_* value corresponding to the error condition; see Clterr.h (in src\inc\) for a list of error values beginning with CLIENT_ERROR.

Remarks

When DriverClose is called, all private driver data is freed. The virtual driver does not need to deallocate the virtual channel or write hooks.

The pVdClose structure currently contains one element – NotUsed. This structure can be ignored.

DriverGetLastError

This function is not used but is available for linking with the common front end, VDAPI.

Calling Convention

```
INT DriverGetLastError(PVD pVD,PVDLASSTERROR pVdLastError);
```

Parameters

pVD
Pointer to a virtual driver control structure.

pVdLastError
Pointer to a structure that receives the last error information.

Return Value

The driver returns CLIENT_STATUS_SUCCESS.

Remarks

This function currently has no practical significance for virtual drivers; it is provided for compatibility with the loadable module interface.

DriverInfo

Gets information about the virtual driver, such as the version level of the driver.

Calling Convention

```
INT DriverInfo(PVD pVD, PDLLINFO pVdInfo, PUINT16 puiSize);
```

Parameters

pVD
Pointer to a virtual driver control structure.

pVdInfo
Pointer to a standard driver information structure.

puiSize
Pointer to the size of the driver information structure. This is an output parameter.

Return Value

If the function succeeds, it returns CLIENT_STATUS_SUCCESS.

If the function fails because the buffer pointed to by pVdInfo is too small, it returns CLIENT_ERROR_BUFFER_TOO_SMALL. Normally, when a CLIENT_ERROR_* result code is returned, the ICA session is disconnected. CLIENT_ERROR_BUFFER_ TOO_SMALL is an exception and does not result in the ICA session being disconnected. Instead, the WinStation driver attempts to call DriverInfo again with the ByteCount of pVdInfo returned by the failed call.

Remarks

When the client starts, it calls this function to retrieve module-specific information for transmission to the host. This information is returned to the server side of the virtual channel by WFVirtualChannelQuery.

The virtual driver must support this call by returning a structure in the pVdInfo buffer. This structure can be a developer-defined virtual channel-specific structure, but it must begin with a VD_C2H structure, which in turn begins with a MODULE_C2H structure. All fields of the VD_C2H structure must be filled in except for the ChannelMask field. See ica-c2h.h (in src\inc\) for definitions of these structures.

The virtual driver must first check the size of the information buffer given against the size that the virtual driver requires (the VD_C2H structure). The size of the input buffer is given in pVdInfo->ByteCount.

If the buffer is too small to store the information that the driver needs to send, the correct size is filled into the ByteCount field and the driver returns CLIENT_ERROR_BUFFER_TOO_SMALL.

If the buffer is large enough, the driver must fill it with a module-defined structure. At a minimum, this structure must contain a VD_C2H structure. The VD_C2H structure must be the first data in the buffer; additional channel-specific data can follow. All relevant fields of this structure are filled in by this function. The flow control method is specified in the VDFLOW structure (an element of the VD_C2H structure). The Ping example contains a flow control selection.

The WinStation driver calls this function twice at initialization, after calling DriverOpen. The first call contains a NULL information buffer and a buffer size of zero. The driver is expected to fill in pVdInfo->ByteCount with the required buffer size and return CLIENT_ERROR_BUFFER_TOO_SMALL. The WinStation driver allocates a buffer of that size and retries the operation.

The data buffer pointed to by pVdinfo->pBuffer must not be changed by the virtual driver. The WinStation driver stores byte swap information in this buffer.

The parameter puiSize must be initialized to the size of the driver information structure.

DriverOpen

Initializes the virtual driver. The client engine calls this user-written function once when the client is loaded.

Calling Convention

```
INT DriverOpen(PVD pVD, PVDOPEN pVdOpen. PUINT16 puiSize);
```

Parameters

pVD
Pointer to the virtual driver control structure. This pointer is passed on every call to the virtual driver.

pVdOpen
Pointer to the virtual driver Open structure.

puiSize
Pointer to the size of the virtual driver Open structure. This is an output parameter.

Return Values

If the function succeeds, it returns CLIENT_STATUS_SUCCESS.

If the function fails, it returns the CLIENT_ERROR_* value corresponding to the error condition; see Clterr.h (in src\inc\) for a list of error values beginning with CLIENT_ERROR

Remarks

The code fragments in this section are taken from the vdping example.

The DriverOpen function must:

1. Allocate a virtual channel.

   Fill in a WDQUERYINFORMATION structure and call VdCallWd. The WinStation driver fills in the OpenVirtualChannel structure (including the channel number) and the data in pVd.

```
WDQUERYINFORMATION wdqi;
OPENVIRTUALCHANNEL OpenVirtualChannel;

wdqi.WdInformationClass = WdOpenVirtualChannel;
wdqi.pWdInformation = &OpenVirtualChannel;

wdqi.WdInformationLength = sizeof(OPENVIRTUALCHANNEL);
OpenVirtualChannel.pVCName = CTXPING_VIRTUAL_CHANNEL_NAME;
rc = VdCallWd(pVd, WDxQUERYINFORMATION, &wdqi);

/* do error processing here */
```

After the call to VdCallWd, the channel number is assigned in the OpenVirtualChannel structure's Channel element. Save the channel number.

For example:

```
g_usVirtualChannelNum = OpenVirtualChannel.Channel;
```

2. Optionally specify a pointer to a private data structure.

   If you want the virtual driver to allocate memory for state data, it can have a pointer to this data returned on each call by placing the pointer in the virtual driver structure, as follows:

   ```
   pVd->pPrivate = pMyStructure;
   ```

3. Exchange entry point data with the WinStation driver.

   The virtual driver must register a write hook with the client WinStation driver. The write hook is the entry point of the virtual driver to be called when data is received for this virtual channel. The WinStation driver returns pointers to functions that the driver must use to fill in output buffers and sends data to the WinStation driver for transmission to the server.

   ```
   WDSETINFORMATION wdsi;
   VDWRITEHOOK vdwh;

   // Fill in a write hook structure
   vdwh.Type = g_usVirtualChannelNum;
   vdwh.pVdData = pVd;
   vdwh.pProc = (PVDWRITEPROCEDURE) ICADataArrival;

   // Fill in a set information structure
   wdsi.WdInformationClass = WdVirtualWriteHook;
   wdsi.pWdInformation = &vdwh;
   wdsi.WdInformationLength = sizeof(VDWRITEHOOK);
   rc = VdCallWd( pVd, WDxSETINFORMATION, &wdsi);
   /* do error processing here */
   ```

   During the registration of the write hook, the WinStation driver passes entry points for the output buffer virtual driver helper functions to the virtual driver in the VDWRITEHOOK structure. The DriverOpen function saves these in global variables so helper functions in the virtual driver can use them. The WinStation driver also passes a pointer to the WinStation driver data area, which the DriverOpen function also saves (because it is the first argument to the virtual driver helper functions).

   ```
   // Record pointers to functions used
   // for sending data to the host.
   pWd = vdwh.pWdData;
   pOutBufReserve = vdwh.pOutBufReserveProc;
   pOutBufAppend = vdwh.pOutBufAppenProc;
   pOutBufWrite = vdwh.pOutBufWriteProc;
   pAppendVdHeader = vdwh.pAppendVdHeaderProc;
   ```

4. Determine the version of the WinStation driver.

   New virtual drivers should determine whether the WinStation driver supports the new SendData API and “no polling” mode. Use the WdVirtualWriteHookEx information class to retrieve this information:

```
// Do extra initialization to determine if
// we are talking to an HPC client.
wdsi.WdInformationClass = WdVirtualWriteHookEx;
wdsi.pWdInformation = &vdwhex;
wdsi.WdInformationLength = sizeof(VDWRITEHOOKEX);
vdwhex.usVersion = HPC_VD_API_VERSION_LEGACY;  // Set version
                     // to 0; older clients will do nothing
rc = VdCallWd(pVd, WDxQUERYINFORMATION, &wdsi, &uiSize);
if(CLIENT_STATUS_SUCCESS != rc)
{
    return( rc );
}
g_fIsHpc = (HPC_VD_API_VERSION_LEGACY != vdwhex.usVersion);
                   // If version returned, this is HPC or later
g_pSendData = vdwhex.pSendDataProc; // save HPC SendData
                                    // API address
```

   The usVersion that is returned may be one of the following values:

```
typedef enum _HPC_VD_API_VERSION
{
    HPC_VD_API_VERSION_LEGACY = 0,    // legacy VDs
    HPC_VD_API_VERSION_V1 =     1,    // VcSdk API Version 1
} HPC_VD_API_VERSION;
```

   If the usVersion returned is HPC_VD_API_VERSION_LEGACY, the engine is an earlier engine. Any other value indicates the newer engine. The actual version returned indicates the version of the API supported. Currently the only other value that will be returned is HPC_VD_API_VERSION_V1. The g_fIsHpc flag should be set to indicate that the newer API is available.

   The WdVirtualWriteHookEx call also returns a pointer (g_pSendData). This is a pointer to the SendData function. Save this value for later use.

5. Set the API options in the WinStation driver.

   If this virtual driver is loaded by the HPC WinStation driver, set the API options this driver will use:

```
if(g_fIsHpc)
{
   WDSET_HPC_PROPERITES hpcProperties;

   hpcProperties.usVersion = HPC_VD_API_VERSION_V1;
   hpcProperties.pWdData = g_pWd;
   hpcProperties.ulVdOptions = HPC_VD_OPTIONS_NO_POLLING;
   wdsi.WdInformationClass = WdHpcProperties;
   wdsi.pWdInformation = &hpcProperties;
   wdsi.WdInformationLength = sizeof(WDSET_HPC_PROPERITES);
   rc = VdCallWd(pVd, WDxSETINFORMATION, &wdsi, &uiSize);
   if(CLIENT_STATUS_SUCCESS != rc)
   {
      return(rc);
   }
}
```

   The usVersion field is set to inform the engine of the version of the VD API that this driver will use. This allows the engine to maintain the compatibility of the VD API for this driver at this level, even if the engine API changes in the future.

   The pWdData pointer must point to the same data that was pointed to by the pWdData field returned by the WdVirtualWriteHook VdCallWd call earlier in DriverOpen.

   The ulVdOptions field is a bitwise OR of any of the following bit definitions:

```
typedef enum _HPC_VD_OPTIONS
{
   HPC_VD_OPTIONS_NO_POLLING =0x0001,  // Flag indicating that
                              // channels on this VD do not
                              // require send data polling
   HPC_VD_OPTIONS_NO_COMPRESSION =0x0002  // Flag indicating
                              // that channels on this VD
                              // send data that does not
                              // need reducer compression
} HPC_VD_OPTIONS;
```

6. Allocate all memory needed by the driver and do any initialization. You can obtain the maximum ICA buffer size from the MaximumWriteSize element in the VDWRITEHOOK structure that is returned.

   Note: vdwh.MaximumWriteSize is one byte greater than the actual maximum that you can use because it also includes the channel number.

```
g_usMaxDataSize = vdwh.MaxiumWriteSize - 1;
if(NULL == (pMyData = malloc( g_usMaxDataSize )))
{
   return(CLIENT_ERROR_NO_MEMORY);
}
```

7. Return the size of the VDOPEN structure in puiSize. This is used by the client engine to determine the version of the virtual channel driver.

DriverPoll

Allows the virtual driver to check timers and other state information, send queued data to the server, and perform any other required processing. This function may be called on a regular basis by the main client poll loop.

Calling Convention

```
INT DriverPoll(PVD pVD, PVOID pVdPoll, PUINT16 puiSize);
```

Parameters

pVD
Pointer to a virtual driver control structure.

pVdPoll
Pointer to one of the driver poll information structures (DLLPOLL or DLL_HPC_POLL).

puiSize
Pointer to the size of the driver poll information structure. This is an output parameter.

Return Values

If the function succeeds, it returns CLIENT_STATUS_SUCCESS.

If the driver has no data on this polling pass, it returns CLIENT_STATUS_NO_DATA.

If all virtual channels return CLIENT_STATUS_NO_DATA, the WinStation driver may slow down the polling process

If the sending of data via either the QueueVirtualWrite or the SendData function is blocked (CLIENT_ERROR_NO_OUTBUF), DriverPoll should return CLIENT_STATUS_ERROR_RETRY so the WinStation driver does not slow polling. The virtual driver should then try again the next time it is polled.

If the virtual driver cannot allocate an output buffer, it returns CLIENT_STATUS_ERROR_RETRY so the WinStation driver does not slow polling. The virtual driver then attempts to get an output buffer the next time it is polled.

If polling has been disabled via the HPC_VD_OPTIONS_NO_POLLING option, DriverPoll will be called at least once, and then only when the virtual driver has asked to be polled, or when it has asked to be notified when a send operation can be retried. The return values have the same setting as in the polling case above. Return CLIENT_STATUS_SUCCESS if all data has been sent successfully. Return CLIENT_STATUS_NO_DATA if there is no data available to send. Return CLIENT_STATUS_ERROR_RETRY if the send operation was blocked and the virtual driver has more data to send.

Return values that begin with CLIENT_ERROR_* are fatal errors; the ICA session will be disconnected.

Remarks

A virtual driver is not allowed to block while waiting for a desired result (such as the availability of an output buffer).

The Ping example includes examples of processing types that can occur in Driver Poll.

DriverQueryInformation

Gets run-time information from the virtual driver.

Calling Convention

```
INT DriverQueryInformation(PVD pVD,
                           PVDQUERYINFORMATION pVdQueryInformation,
                           PUINT16 puiSize);
```

Parameters

pVD
Pointer to a virtual driver control structure.

pVdQueryInformation
Pointer to a structure that specifies the information to query and the results buffer.

puiSize
Pointer to the size of the query information and resolves structure. This is an output parameter.

Return Value

The function returns CLIENT_STATUS_SUCCESS.

Remarks

This function currently has no practical significance for virtual drivers; it is provided for compatibility with the loadable module interface. There are no general purpose query functions at this time other than LastError. The LastError query is accomplished through the DriverGetLastError function.

DriverSetInformation

Sets run-time information in the virtual driver.

Calling Convention

```
INT DriverSetInformation(PVD pVD,
                         PVDSETINFORMATION pVdSetInformation,
                         PUINT16 puiSize);
```

Parameters

pVD
Pointer to a virtual driver control structure.

pVdSetInformation
Pointer to a structure that specifies the information class, a pointer to any additional data, and the size in bytes of the additional data (if any).

puiSize
Pointer to the size of the information structure. This is an input parameter.

Return Value

The function returns CLIENT_STATUS_SUCCESS.

Remarks

This function can receive two information classes:

- VdDisableModule: When the connection is being closed.
- VdFlush: When WFPurgeInput or WFPurgeOutput is called by the server-side virtual channel application. The VdSetInformation structure contains a pointer to a VDFLUSH structure that specifies which purge function was called.

SendData

Sends a virtual channel packet to the server, with a notification option.

Calling Convention

```
INT WFCAPI SendData(DWORD pWd, USHORT usChannel,
                    LPBYTE pData,USHORT usLen,
                    LPVOID pUserData, UINT32 uiFlags);
```

Parameters

pWd
Pointer to a WinStation driver control structure.

usChannel
The virtual channel number.

pData
Pointer to the data buffer containing the virtual channel data to send.

usLen
Length in bytes of the data in the data buffer.

pUserData
This is a pointer to arbitrary VD user data that will be passed back to the callback function (DriverPoll) on notification that the send should be retried. See DriverPoll and the SENDDATA_NOTIFY flag described below.

uiFlags
Flags to control the operation of the SendData function. This value consists of a number of flags bitwise OR'ed together. Each of the flags controls some aspect of the SendData interface. Currently there is only one flag defined. See the SENDDATA_* enum:

- SENDDATA_NOTIFY: If this flag is set, and when the SendData return code is CLIENT_ERROR_NO_OUTBUF indicating that the engine had no buffers to accommodate the outbound packet, the engine will notify the virtual driver later when it can retry the send operation. The notification occurs via the DriverPoll method.

Return Value

The SendData function will return one of the following values:

- CLIENT_STATUS_SUCCESS:
  - The data was copied into virtual write buffers.
  - The user's buffer is free.
  - No callback will occur, even if the SENDDATA_NOTIFY flag is set.
  - The next SendData call can be issued immediately.

- CLIENT_ERROR_NO_OUTBUF:
  - The virtual write could not be scheduled (out of VirtualWrite buffers). If a notification was requested, DriverPoll will be driven with the notification at some later time when the virtual driver should retry sending.
  - If no notification was requested, the virtual driver should return from DriverPoll and wait for the next poll before retrying the send. This assumes that the virtual driver had selected the polled mode of operation.

- CLIENT_ERROR_BUFFER_STILL_BUSY:
  - If the user has called SendData requesting a notification, and the return code was CLIENT_ERROR_NO_OUTBUF, the user must not issue another SendData call until the notification has occurred. If another call is issued before the notification occurs, the CLIENT_ERROR_BUFFER_STILL_BUSY return code will result.

- CLIENT_ERROR_*: Any other error should cause the virtual driver and the session to close.

Note: If the user has specified HPC_VD_OPTIONS_NO_POLLING in the HPC channel options, then the virtual driver must assume that its DriverPoll function will not be called again after receiving one of these errors.

Remarks

This function is used to send channel protocol to the server. The engine either accepts all the data, or refuses it all, in which case the channel will need to retry later (normally inside DriverPoll).

The address for this function is obtained from the VDWRITEHOOKEX structure after hook registration in pSendDataProc. The VDWRITEHOOK structure provides pWd.

ICADataArrival

The WinStation driver calls this function when data is received on a virtual channel being monitored by the driver. The address of this function is passed to the WinStation driver during DriverOpen.

Calling Convention

```
VOID wfcapi ICADataArrival(PVD pVD, USHORT uchan, LPBYTE pBuf,
                           USHORT Length);
```

Parameters

pVD
Pointer to a virtual driver control structure.

uChan
Virtual channel number.

pBuf
Pointer to the data buffer containing the virtual channel data as sent by the server-side application.

Length
Length in bytes of the data in the buffer.

Return Value

No value is returned from this function.

Remarks

This function name is a placeholder for a user-defined function; the actual function does not have to be called ICADataArrival, although it does have to match the function signature (parameters and return type). The address of this function is given to the WinStation driver during DriverOpen. Although ICA prefixes packet control data to the virtual channel data, this prefix is removed before this function is called.

After the virtual driver returns from this function, the WinStation driver considers the data delivered. The virtual driver must save whatever information it needs from this packet if later processing is required.

Do not allow this function to block. Use your own thread or the DriverPoll function (with polling enabled) for any required deferred processing.

The virtual driver can send data to the server on receipt of this data from within the ICADataArrival function, but be aware that the send operation may return an immediate error when buffers are not available to accommodate the send operation. The virtual driver may not block in this function waiting for the sending operation to complete.

If the virtual driver is handling multiple virtual channels, use the uChan parameter to determine the channel over which this data is to be sent. See DriverOpen for more information.

miGetPrivateProfileBool

Gets a Boolean value from a section of the Configuration Storage.

Calling Convention

```
INT miGetPrivateProfileBool(PCHAR lpszSection, PCHAR lpszEntry,
                            BOOL bDefault);
```

Parameters

lpszSection
Name of section to query.

lpszEntry
Name of entry to query.

bDefault

Default value to use.

Return Values

If the requested entry is found, the entry value is returned; otherwise, bDefault is returned.

Remarks

A Boolean value of TRUE can be represented by on, yes, or true in the Configuration Storage. All other strings are interpreted as FALSE.

miGetPrivateProfileInt

Gets an integer from a section of the Configuration Storage.

Calling Convention

```
INT miGetPrivateProfileInt(PCHAR lpszSection, PCHAR lpszEntry,
                           INT iDefault);
```

Parameters

lpszSection
Name of section to query.

lpszEntry
Name of entry to query.

iDefault
Default value to use.

Return Values

If the requested entry is found, the entry value is returned; otherwise, iDefault is returned.

miGetPrivateProfileLong

Gets a long value from a section of the Configuration Storage.

Calling Convention

```
INT miGetPrivateProfileLong(PCHAR lpszSection, PCHAR lpszEntry,
                            LONG lDefault);
```

Parameters

lpszSection
Name of section to query.

lpszEntry
Name of entry to query.

lDefault
Default value to use.

Return Values

If the requested entry is found, the entry value is returned; otherwise, lDefault is returned.

miGetPrivateProfileString

Gets a string from a section of the Configuration Storage.

Calling Convention

```
INT miGetPrivateProfileString(PCHAR lpszSection, PCHAR lpszEntry,
                              PCHAR lpszDefault,
                              PCHAR lpszReturnBuffer, INT cbSize);
```

Parameters

lpszSection
Name of section to query.

lpszEntry
Name of entry to query.

lpszDefault
Default value to use.

lpszReturnBuffer
Pointer to a buffer to hold results.

cbSize
Size of lpszReturnBuffer in bytes.

Return Values

This function returns the string length of the value returned in lpszReturnBuffer (not including the trailing NULL).

If the requested entry is found and the size of the entry string is less than or equal to cbSize, the entry value is copied to lpszReturnBuffer; otherwise, iDefault is copied to lpszReturnBuffer.

Remarks

lpszDefault must fit in lpszReturnBuffer. The caller is responsible for allocating and deallocating lpszReturnBuffer.

lpszReturnBuffer must be large enough to hold the maximum length entry string, plus a NULL termination character. If an entry string does not fit in lpszReturnBuffer, the lpszDefault value is used.

QueueVirtualWrite

Sends a virtual channel packet to the server.

Calling Convention

```
INT WFCAPI QueueVirtualWrite(PWD pWd, USHORT ChannelNumber,
                             LPMEMORY_SECTION pMemorySections,
                             USHORT NumberOfSections, USHORT FlushControl);
```

Parameters

pWd
Pointer to a WinStation driver control structure.
ChannelNumber
The virtual channel number
pMemorySections
Pointer to an array of one or more elements of the structure MEMORY_SECTION (see below) containing the pure body of the protocol, i.e. excluding the virtual write header defining the channel and the length. This will get constructed by the QueueVirtualWrite function itself. Normally the protocol body will already be in a single contiguous block of memory. But if not, then multiple sections can be defined, and the destination function will copy all the different pieces into the appropriate internal WD queue.
NumberOfSections
The number of memory sections, normally 1.
FlushControl
Indicates whether a 'flush to wire' operation should be triggered after the data has been successfully queued. If this value is FLUSH_IMMEDIATELY, then if the line conditions permit, the WD will attempt to send this new virtual write immediately over the wire, after also flushing any earlier queued data for the same or higher priority. If the data is not time critical (within the span of about 50ms), it may be better not to force a flush at this point, so that the data (if small) may go over the wire together with other data, so making better use of the wire bandwidth. The value to use if an immediate flush is not required is !FLUSH_IMMEDIATELY.

Memory section

This structure has the definition:

```
typedef struct _MEMORY_SECTION
{
    UINT length;              // Length of data
    LPBYTE pSection;          // Address of data
} MEMORY_SECTION, far * LPMEMORY_SECTION;
```

Return Values

If the function succeeds, it returns CLIENT_STATUS_SUCCESS. If it cannot currently accept the data, it returns CLIENT_ERROR_NO_OUTBUF. If being called from DriverPoll, then the return value for the DriverPoll should normally be set to CLIENT_STATUS_ERROR_RETRY in this case. If the function fails, it returns an error code associated with the failure; use GetLastError to get the extended error information.

Remarks

This function is used to send channel protocol to the server. The engine either accepts all the data, or refuses it all, in which case the channel will need to retry later (normally inside DriverPoll).

The address for this function is obtained from the VDWRITEHOOK structure after hook Registration in pQueueVirtualWriteProc. The VDWRITEHOOK structure also provides pWd. Each successful call will ultimately result in a single block of protocol, with length = total length of all memory sections, being delivered to the server-side channel.

VdCallWd

Calls the client WinStation driver to query and set information about the virtual channel. This is the main method for the virtual driver to access the WinStation driver. For general-purpose virtual channel drivers, this sets the virtual write hook.

Calling Convention

```
INT VdCallWd( PVD pVd, USHORT ProcIndex, PVOID pParam);
```

Parameters

pVd
Pointer to a virtual driver control structure.

ProcIndex
Index of the WinStation driver routine to call. For virtual drivers, this can be either WDxQUERYINFORMATION or WDxSETINFORMATION.

pParam
Pointer to a parameter structure.

Return Values

If the function succeeds, it returns CLIENT_STATUS_SUCCESS.

If the function fails, it returns an error code associated with the failure; use DriverGetLastError to get the extended error information.

Remarks

This function is a general purpose mechanism to call routines in the WinStation driver. The only valid uses of this function for a virtual driver are:

- To allocate the virtual channel using WDxQUERYINFORMATION.
- To exchange function pointers with the WinStation driver during DriverOpen using WDxSETINFORMATION.

For more information, see DriverOpen or the Ping example.

On successful return, the VDWRITEHOOK structure contains pointers to the output buffer virtual driver helper functions, and a pointer to the WinStation driver control block (which is needed for buffer calls).

WFVirtualChannelClose

Closes an open virtual channel handle.

Calling Convention

```
BOOL WINAPI WFVirtualChannelClose(IN HANDLE hChannelHandle);
```

Parameter

hChannelHandle
Handle to a previously opened virtual channel. Use WFVirtualChannelOpen to obtain a handle for a specific channel.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE; call GetLastError to get extended error information.

Remarks

When this function is called, any data waiting to be sent to the client is discarded. Call this function when the server-side virtual channel application is finished.

The client- side virtual driver is not closed until the ICA session is closed. If the virtual driver sends

data to the server after the server-side application closes, the data is queued on the server and eventually discarded.

To prevent the virtual driver from sending data after the server-side application closes, you might need to incorporate a packet type into your virtual channel protocol to notify the virtual driver that the server-side application is closing.

WFVirtualChannelOpen

Opens a handle to a specific virtual channel.

Calling Convention

```
HANDLE WINAPI WFVirtualChannelOpen(IN HANDLE hServer,
                                   IN DWORD SessionId,
                                   IN LPSTR pVirtualName  // ANSI name
                                  );
```

Parameters

hServer
Handle to a server running XenApp. To specify the current server, use the constant WF_CURRENT_SERVER_HANDLE. Use WFOpenServer to obtain a handle for a specific server. For more information about the WFOpenServer function, see the WFAPI SDK documentation.

SessionId
Server session ID. Use the constant WF_CURRENT_SESSION to specify the current session. To obtain the session ID of a specific session, use WFEnumerateSessions. For more information about session IDs and WFEnumerateSessions, see the WFAPI SDK documentation.

pVirtualName
Pointer to the virtual channel name. This is an ASCII string (even when Unicode is defined) of no more than seven characters.

Return Values

If the function succeeds, it returns a handle to the specified virtual channel.

If the function fails, it returns NULL; call GetLastError for extended error information.

Remarks

The WinStation driver opens the channel by name, assigns a channel number, and returns a handle. The server-side virtual channel application uses this handle to read and write data to the virtual channel.

WFVirtualChannelPurgeInput

Purges all queued input data sent from the client to the server on a specific virtual channel.

Calling Convention

```
BOOL WINAPI WFVirtualChannelPurgeInput(IN HANDLE hChannelHandle);
```

Parameter

hChannelHandle
Handle to a previously opened virtual channel. To obtain a handle for a specific channel, use WFVirtualChannelOpen.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE; call GetLastError to get extended error information.

Remarks

Output buffers and queued data received from the client are discarded.

This function sends a message to the client WinStation driver, which then calls the client virtual driver's DriverSetInformation function with the VdFlush information class. For most virtual channels, this function is not necessary and you can use the Ping example function without modification.

WFVirtualChannelPurgeOutput

Purges all queued output data sent from the server to the client on a specific virtual channel.

Example of use: in an audio application in which the user starts playing a different audio file, use this function to discard the audio that was queued to be sent to the client from the first file played.

Calling Convention

```
BOOL WINAPI WFVirtualChannelPurgeOutput(IN HANDLE hChannelHandle);
```

Parameter

hChannelHandle
Handle to a previously opened virtual channel. To obtain a handle for a specific channel, use WFVirtualChannelOpen.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. Call GetLastError to get extended error information.

Remarks

Output buffers and data queued to be sent to the client are discarded.

This function sends a message to the client WinStation driver, which then calls the client virtual driver's DriverSetInformation function with the VdFlush information class. For most virtual channels, this function is not necessary and you can use the Ping example function without modification.

WFVirtualChannelQuery

Returns data related to a virtual channel. This information is obtained when the ICA connection is initiated and the WinStation driver calls the DriverInfo function.

Calling Convention

```
BOOL WINAPI WFVirtualChannelQuery(IN HANDLE hChannelHandle,
                                  IN WF_VIRTUAL_CLASS VirtualClass,
                                  OUT PVOID *ppBuffer,
                                  OUT DWORD *pBytesReturned
                                 );
```

Parameters

hChannelHandle
Handle to a previously opened virtual channel. To obtain a handle for a specific channel, use WFVirtualChannelOpen.

VirtualClass
Type of information to request. Currently, the only defined value is WFVirtualClientData , which returns virtual channel client module data.

ppBuffer
Pointer to the address of a variable to receive the data. The buffer is allocated within this function and is deallocated by using WFFreeMemory.

pBytesReturned

Pointer to a DWORD that is updated with the length of the data returned in the allocated buffer (upon successful return).

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE; Call GetLastError to get extended error information.

Remarks

ppBuffer begins with the structure VD_C2H, which begins with the structure MODULE_C2H. These two structures are defined in Ica-c2h.h (in src\inc\). See the Ping example for more information.

WFVirtualChannelRead

Reads data from a virtual channel.

Calling Convention

```
BOOL WINAPI WFVirtualChannelRead(IN HANDLE hChannelHandle,
                                 IN ULONG TimeOut,
                                 OUT PCHAR pBuffer,
                                 IN ULONG BufferSize,
                                 OUT PULONG pBytesRead
                                 );
```

Parameters

hChannelHandle
: Handle to a previously opened virtual channel. To obtain a handle for a specific channel, use WFVirtualChannelOpen.

TimeOut
: Length of time to wait for data (in milliseconds). If this value is zero, the function returns immediately whether or not data is available. If this value is -1, the function continues waiting until there is data to read.

pBuffer
: Buffer to receive the data read from the virtual channel.

BufferSize
: Size in bytes of the buffer needed.

pBytesRead
: Pointer to a variable that receives the number of bytes read.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE; call GetLastError to get extended error information.

Remarks

The developer determines the BufferSize, which can be any length up to the maximum size supported by the ICA connection. This size is independent of size restrictions on the lower-layer transport.

- If the server is running XenApp or a version of Presentation Server 3.0 Feature Release 2 or later, the maximum packet size is 5000 bytes (4996 bytes of data plus the 4-byte packet overhead generated by the ICA datastream manager).
- If the server is running a version of Presentation Server earlier than 3.0 Feature Release 2, the maximum packet size is 2048 bytes (2044 bytes of data plus the 4-byte packet overhead generated by the ICA datastream manager).

If more data is received than the buffer can hold, the entire packet is discarded.

If no data is received, pBuffer and pBytesRead are unmodified. The function fails if the read times out.

The server-side virtual channel application is not notified when data is received. Instead, the data is queued until the application uses WFVirtualChannelRead to retrieve it.

WFVirtualChannelWrite

Writes data to a virtual channel.

Calling Convention

```
BOOL WINAPI WFVirtualChannelWrite(IN HANDLE hChannelHandle,
                                  IN PCHAR pBuffer,
                                  IN ULONG Length,
                                  OUT PULONG pBytesWritten
                                  );
```

Parameters

hChannelHandle
: Handle to a previously opened virtual channel. To obtain a handle for a specific channel, use WFVirtualChannelOpen.

pBuffer
: Buffer containing data to write to the virtual channel. This must be four bytes larger than the largest buffer written by the client.

Length
: Size in bytes of the buffer needed. This must be four bytes larger than the data

written by the application.

pBytesWritten
Pointer to a ULONG variable that receives the number of bytes written.

Return Values

If the data is sent, the return value is TRUE.

If the data is not sent, the return value is FALSE; call GetLastError to get extended error information.

Remarks

The developer determines the Length, which can be any length up to the maximum size supported by the ICA connection. This size is independent of size restrictions on the lower-layer transport.

- If the server is running XenApp or a version of Presentation Server 3.0 Feature Release 2 or later, the maximum packet size is 5000 bytes (4996 bytes of data plus the 4-byte packet overhead generated by the ICA datastream manager).
- If the server is running a version of Presentation Server earlier than 3.0 Feature Release 2, the maximum packet size is 2048 bytes (2044 bytes of data plus the 4-byte packet overhead generated by the ICA datastream manager).

The WinStation driver calls the client virtual driver's ICADataArrival function.