Citrix Virtual Channel SDK for Citrix Workspace app for Linux

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 or XenDesktop servers. This version of the SDK provides support for writing new virtual channels for Workspace app for Linux. 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.
- 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.

System Requirements

You can build the Citrix Virtual Channel SDK on 4MB of disk space. The Virtual Channel SDK components for this release were built on Debian 5. You can compile the virtual channel on a similar system.

Development Environment Requirements

- GNU Compiler Collection (GCC) compiler on all platforms
- ARM Compiler (cross-compiler) requires a GCC cross-compiler
- GNU Make utility

Execution Environment Requirements

Server requirement

The Linux Virtual Channel SDK is supported on Citrix Presentation Server 4.5, Citrix XenApp, versions 5.0, 6.0, 6.5, 7.5, 7.6, 7.7, 7.8, 7.9, 7.11, 7.12, 7.13, 7.14, 7.15, and Citrix XenDesktop, versions 4.0, 5.0, 5.5, 5.6, 7, 7.1, 7.5, 7.6, 7.7, 7.8, 7.9, 7.11 and later.

Linux client requirement

Citrix Workspace app for Linux 1908 or later

> **Note**
>
> The Linux Virtual Channel SDK is supported for use with the client of the corresponding version number and any LCM fixes for that release.

Installing the Virtual Channel SDK

The Citrix Community Web site is the home of the Citrix Developer Network and all technical resources and discussions involving the use of Citrix SDKs. You can find access to SDKs, sample code and scripts, extensions and plug-ins, and SDK documentation. Also included are the Citrix Developer Network forums, where technical discussions take place around each of the Citrix SDKs.

1. Download the Virtual Channel SDK, vcsdk.tar.gz, from http://www.citrix.com/downloads/workspace-app/virtual-channel-sdks/virtual-channel-sdk.html to your user device.
2. Open a terminal window.
3. Run the installation file by typing

```
tar xvfz vcsdk.tar.gz
```

Uninstalling the Virtual Channel SDK

Remove the VCSDK directory to uninstall the Virtual Channel SDK by typing, for example,

```
rm -rf VCSDK
```

Architecture

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:

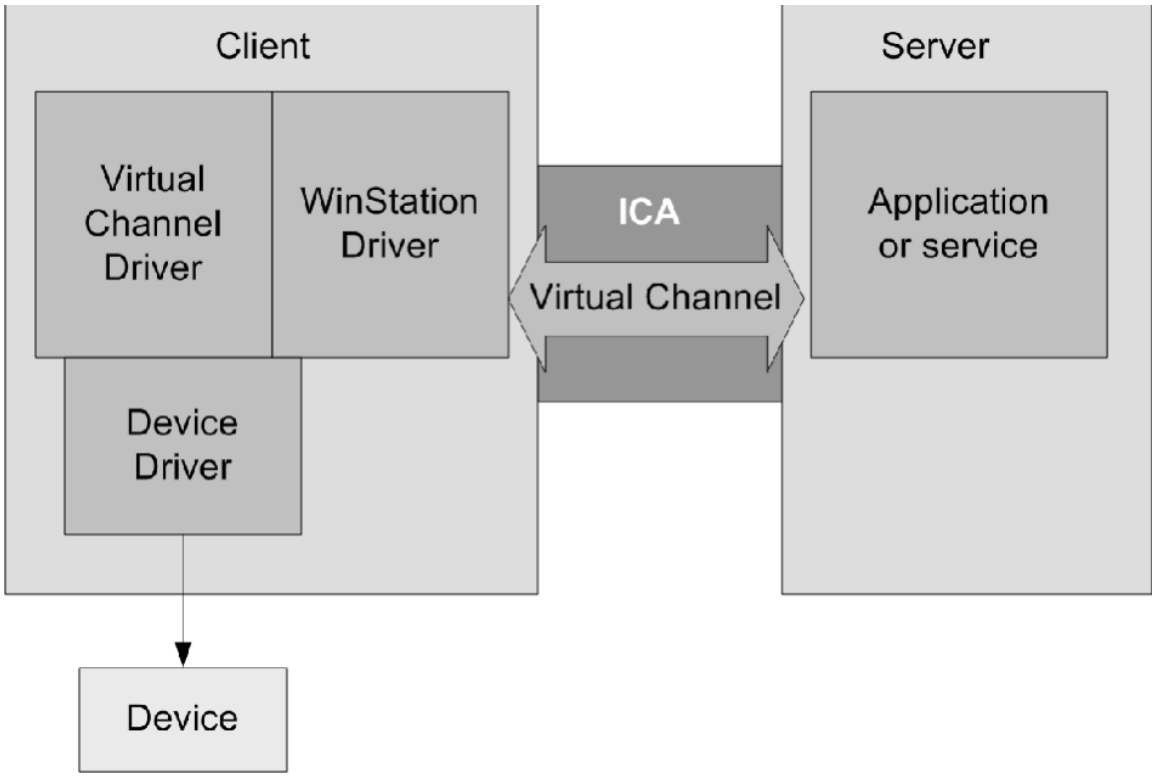
Server-side portion on the computer running XenApp or XenDesktop

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). The WinStation driver is also responsible for gathering and sending virtual channel data to the server over the ICA connection.

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 is sent by using the QueueVirtualWrite call for the newly written virtual drivers. The data can be sent at any point that the virtual driver is processing the main

process control flow. Do not send data from a thread within a virtual driver. Note that there is no way to alert the server virtual channel application that the 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 for more information.

Client WinStation Driver and Virtual Driver Interaction

The WinStation driver calls into the virtual driver on the event callbacks, timer callbacks, and on the periodic call to the DriverPoll function. The client runtime environment is single threaded and nonpreemptive; therefore, the virtual driver you write must never block. When control flow is passed to the virtual driver, the virtual driver must immediately perform the required operations and return control to the WinStation driver.

Because all transfers to the server require reserving an output buffer and buffers might be temporarily unavailable, the virtual driver must be prepared to delay sending all output until a later point.

The following process occurs when a user starts the client:

1. At client load time, the client engine reads the Configuration Storage in the configuration files 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 configuration files 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.a, 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.
3. For each virtual channel, the WinStation driver calls the DriverOpen function, which establishes and initializes the virtual channel. The WinStation driver passes the addresses of the output buffer management 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.

4. When virtual channel data arrives from the server, the WinStation driver calls the ICADataArrival function for that virtual driver.
5. To send data, the virtual channel driver has two options:
    - To use the QueueVirtualWrite function which is simple to use and offers the option for immediate data transfer. This is the method that should be used for all new virtual drivers.
    - To use the **deprecated** client-side helper functions (these addresses are obtained during initialization) to reserve an output buffer, fill it with data, and write the buffer.
6. Outgoing data must be placed in the WinStation driver's output buffers for transmission to the host. Checks for available space using OutBufReserve.
7. Fills in the buffer using AppendVdHeader and OutBufAppend.
8. Writes the data using OutBufWrite.

The WinStation driver does not preserve the output buffer data between calls to the virtual driver, so the virtual driver must complete the data output process before returning control.

Module.ini

The Workspace apps 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 configuration files 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 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.

Using Example Programs

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.

OXS: Demonstrates sub-window or overlay buffers, events, and timers.

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.

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

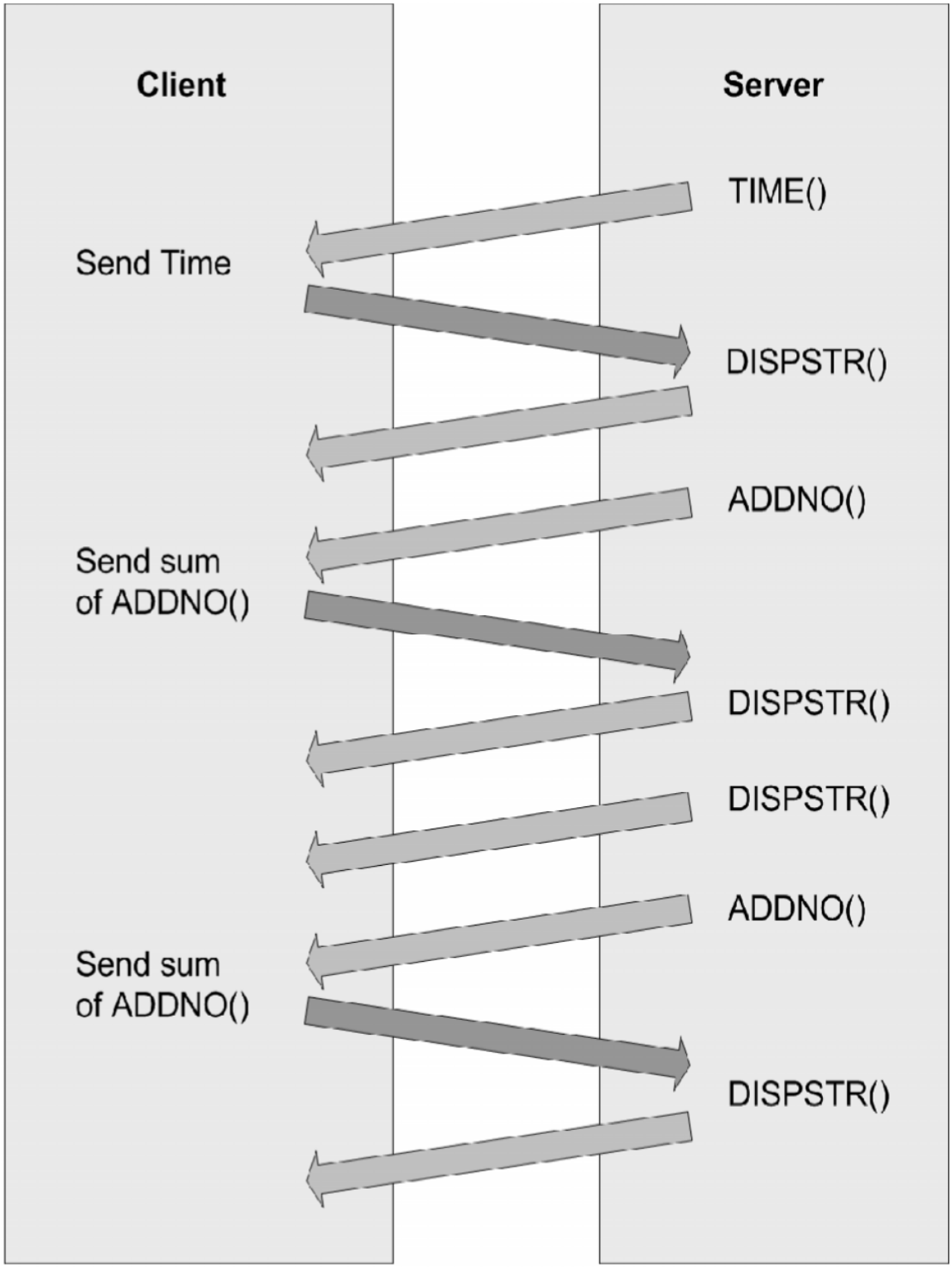
```
typedef struct MIXHEAD {
   USHORT    uType        // Packet type
   USHORT    uFunc;       // Index of Function
   ULONG     uLen;        // Length of data
   USHORT    fRetReq;     // True if return value required
   ULONG     dwRetVal;    // Return Value from client
   USHORT    dwLen1;      // 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

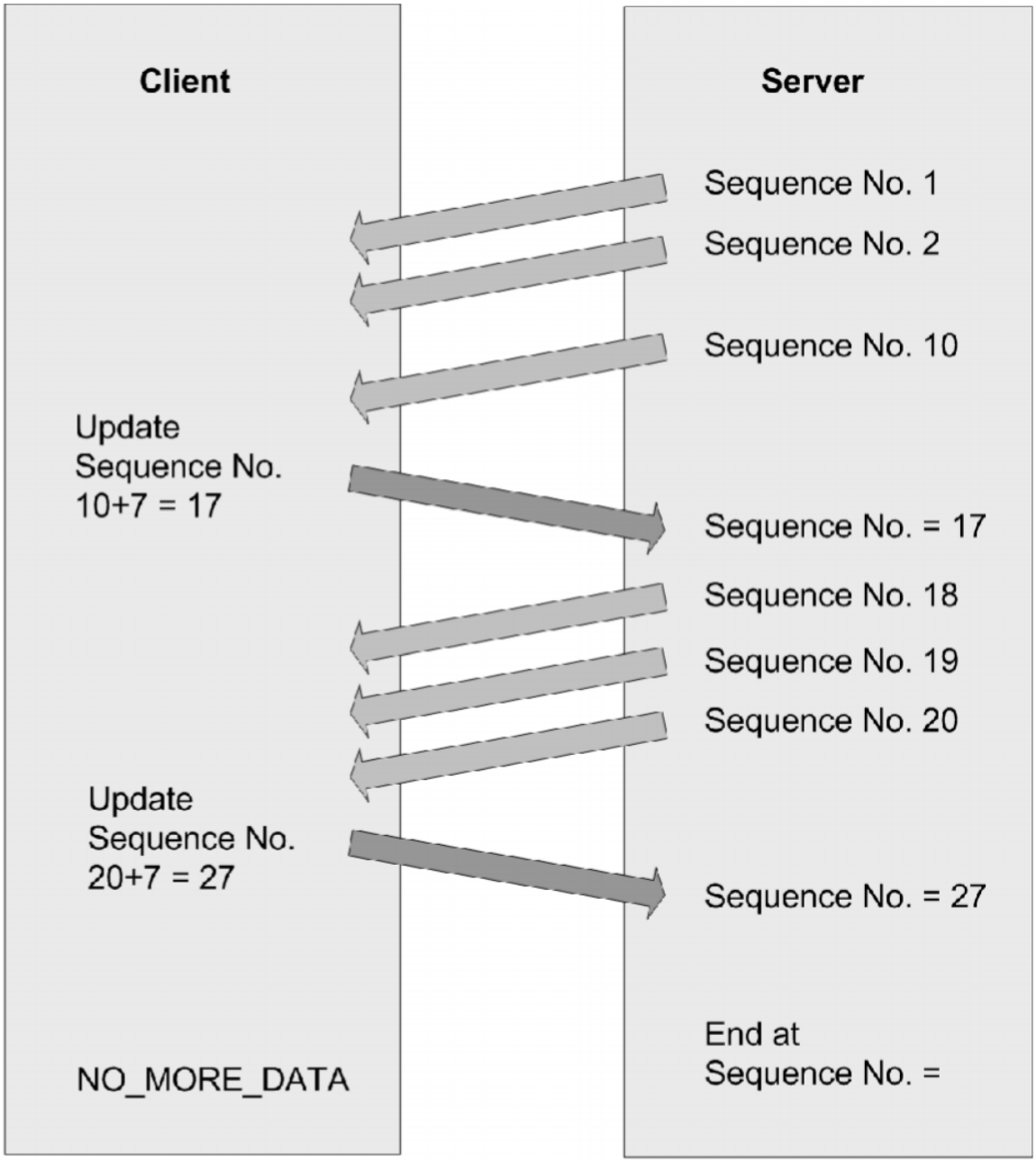
```
typedef struct OVER {
   USHORT     uSign;       // Signature
   USHORT     uType;       // Type, BEGIN or END, from server
   USHORT     uLen;        // Packet length from server
   USHORT     uCounter;    // Sequencer
   ULONG      ulServerMS;  // Server millisecond clock
} OVER, *POVER;
```

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.



OXS

OXS is the Noughts and Crosses game. The OXS virtual channel implements remote drawing of this game. The purpose of this is to demonstrate the following interfaces, which are specific to the Workspace app for Linux:

- Sub-windows or overlay buffers
- Events (the selection of File Descriptor)
- Timers

The client component of the OXS virtual driver implements the drawing of the game. All processing of the game play is performed on the server. To achieve client-side drawing, a sub-window (child) of the session window is obtained using the sub-window interface. The sub-window is placed over the corresponding area of the game window, and every movement to the server-side game window is mimicked by the sub-window. This creates the impression of a single application. Mouse clicks and interaction with the server-side OXS application are converted into Play Move, Winning Line, Move Window, and Close Window commands. These are received by the virtual driver and translated into corresponding actions and X drawing commands.

This sub-window technology is most useful for solutions such as local video decoding. The sub-window interface is not designed to take keyboard and mouse input. It is intended to render graphics.

In addition, the OXS virtual driver uses the events (Evt) interface. This monitors the X server file descriptor for Expose events, which allows a callback to redraw the game area for every event.

The OXS virtual driver also uses the timer (Tmr) interface. This works around a race condition between the OXS and Seamless virtual channels. Although the sub-window used for drawing mimics the movements of the server side window, if Seamless is used a race condition occurs between the movement of the Seamless server-side window and the sub-window. The timer is used to delay the sub-window position update until after the Seamless window move is complete.

Note: The sample application below is a basic one and does not use the MM_clip api. In this case, the overlaid session sub-window is never clipped and is always an on top square.

Clipping the session sub-window properly can give the appearance that it is in the remote sessions window stack, even though it is actually overlaid on top of the session. This is not something this example does.

Building Examples

Building a Server-side Example using Nmake

Examples of the latest server-side executables have been provided for testing. Please download the latest Windows Virtual Channel SDK in order to develop the server-side component.

Building a Client-side Example using Linux

1. To start a build, open a terminal.
2. Change src/examples/vc/client/unix/MakeCOMMONVD to match the target platform.

   Note: Linux does not require pingwire.c. This is detected by the presence of the environment variable, CROSS_COMPILER_PREFIX
3. Set the environment variable HOST_PREFIX to the target:
   - linux for Linux x64
   - linux for Linux x86
   - linuxarmhf for Linux armhf
4. For example, to build **vdping.dll**, type: `cd src/examples/vc/client/vdping/unix` and then type `make`. The production (retail) and debug builds of vdping.dll are built in the lib subdirectory. Pre-built production versions are provided in the separate directories, `src/examples/vc/client/vdping/linux/obj/retail` and `src/examples/vc/client/vdping/linuxarm/obj/retail`.
5. Perform the same process to build all other example virtual drivers.

Alternatively to build all example Virtual Drivers, type **cd src/examples/vc/client** and then run the **build** script.

Configuring the Virtual Driver

1. Copy the appropriate virtual driver for the platform to the client installation directory. For example, copy the Linux virtual driver, vdping.dll, in

src/examples/vc/client/vdping/unix/lib/linux/retail for a locally built version, or in src/examples/vc/client/vdping/linux/obj/retail for a pre-built version to the client installation directory as VDPING.DLL.

2. In the client installation directory, modify the file config/module.ini and make the following changes:
   - In the [ICA 3.0] section append “Ping” to the VirtualDriver list.
   - In the [ICA 3.0] section add “Ping=On”.
3. Add a new section [Ping] with the entry “DriverName=VDPING.DLL”.
4. Copy ctxping.exe.sample from src/examples/vc/server/ctxping to the Citrix server and rename it to ctxping.exe.
5. Run ctxping at the command prompt within a client session to implement the Ping virtual channel.

Running an Example Virtual Channel

Examples of the latest server-side executables have been provided for testing.

1. On a client configured with the client-side example, connect to a server running XenApp or XenDesktop with the associated server-side example (located at base/examples/vc/server in this SDK).
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 Linux 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 and create a debug.ini file in the current directory where the client is run.

When the debug module is installed on the client, the TRACE statements write the debug information to a file, ncs.log.<process id>. The following debug.ini contents create tracing for the example virtual channels:

```
\[ncs\]
traceFlags = +LOG_PRINTF
traceClasses = +TC_VD
traceFeatures = +TT_ALL
traceFile = ncs.log.\$\$
```

You can refine tracing by editing the traceFeatures line. For example, "traceFeatures = +TT_API1 | TT_API2" will only print trace statements of type TT_API1 and TT_API2.

The class flag for virtual channels is TC_VD. For the complete list of class and event flags, see logflags.h (located 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 debug version of the virtual driver to the client installation directory. For example, copy vdping.dll to the client installation directory as VDPING.DLL.
4. Ensure that config.ini and module.ini in the client installation directory are updated appropriately to load the new virtual driver, following the instructions for loading a production version of the virtual driver.
5. Create the debug.ini file in the current working directory.
6. Launch Workspace app to implement the new virtual channel.

Programming Guide

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 low-speed 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 structure. See dllapi.h (in base/inc/) for definitions of these structures.

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 six groups of functions: user-defined, virtual driver helper, memory INI, Workspace app for Linux sub-window interface, Workspace app for Linux event interface, and Workspace app for Linux timer interface.

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.a (provided with this SDK).

Function	Description
DriverClose	Frees private driver data. Called before unloading a virtual driver (generally upon client exit).
DriverGetLastError	Returns the last error set by the virtual driver. Not used; links with the common front end, VDAPI.
DriverInfo	Retrieves information about the virtual driver.
DriverOpen	Performs all initialization for the virtual driver. Called once when the client loads the virtual driver (at startup).
DriverPoll	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	Retrieves run-time information from the virtual driver.
DriverSetInformation	Sets run-time information in the virtual driver.
ICADataArrival	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 helper functions and the virtual driver passes pointers to the user-defined functions. Newer API functions QueueVirtualWrite, MM_*, Evt_*, and Tmr_* helper functions are callable directly by the virtual driver.

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.

Function	Description
QueueVirtualWrite	Queues a virtual write and stimulates packet output if required allowing the data to be sent without waiting for the poll. This must be used to send data to the server in all newly written virtual drivers. This replaces the deprecated functions below.
AppendVdHeader (**Deprecated**)	Appends a virtual driver header to a buffer.
OutBufAppend (**Deprecated**)	Appends data to a buffer.
OutBufReserve (**Deprecated**)	Checks for available output buffer space.
OutBufWrite (**Deprecated**)	Sends the buffer to the server.
VdCallWd	Used to query and set information from the WinStation driver (WD).

Memory INI Functions

Memory INI functions read data from the client engine configuration files stored in both the client installation directory for system wide settings and \$HOME/.ICAClient for user specific settings.

For each entry in appsrv.ini and wfclient.ini, there must be a corresponding entry in All_Regions.ini for the setting to take effect. For more information, refer to All_Regions.ini file in the \$ICAROOT/config directory.

Function	Description
miGetPrivateProfileBool	Returns a boolean value.
miGetPrivateProfileInt	Returns an integer value.
miGetPrivateProfileLong	Returns a long value.
miGetPrivateProfileString	Returns a string value.

Workspace app for Linux Sub-Window Interface

Workspace app for Linux sub-window interface allows a virtual channel to gain access to a sub-window of the client session in order to draw within a session. The sub-window interface is not designed to take keyboard and mouse input. It is simply for rendering graphics.

Function	Description
MM_clip	Sets the shape of the window.
MM_destroy_window	Destroys a window created by MM_get_window.
MM_get_window	Creates an operating system window that is a sub-window of an existing session window.
MM_set_geometry	Sets the size and position of a session sub-window.
MM_show_window	Makes a window visible.
MM_TWI_clear_new_window_function	Clears the callback for seamless window creation.
MM_TWI_set_new_window_function	Adds a callback for seamless window creation.

Workspace app for Linux Event (Evt) Interface

Workspace app for Linux event interface allows a virtual channel to select on a given file descriptor in the Workspace app for Linux event loop and receive a callback from the Workspace app for Linux event loop when the given conditions are met.

Function	Description
Evt_create	Allocates an event structure that can be used to fire a callback on an event.
Evt_destroy	Destroys previously created event structure.
Evt_remove_triggers	Removes any previously added file descriptor selections on a given file descriptor.
Evt_remove_triggers	Removes any previously added file descriptor selections on a given file descriptor.
Evt_signal	Calls the function stored in the event structure.
Evt_trigger_for_input	Connects the callback of an event structure to be triggered on the given file descriptor satisfying the input conditions.
Evt_trigger_for_output	Connects the callback of an event structure to be triggered on the given file descriptor satisfying the output conditions.

Workspace app for Linux Timer (Tmr) Interface

Workspace app for Linux timer interface allows a virtual channel to set up a recurrent timer that invokes a given callback. The timer is attached to the event loop of the Workspace app for Linux and is called from the event loop when the timer fires.

Function	Description
Tmr_create	Creates a timer object and returns its handle.
Tmr_destroy	Destroys a timer object given a printer to its handle and sets the handle to NULL.
Tmr_setEnabled	Enables or disables a timer object.
Tmr_setPeriod	Sets the timeout period for a timer.

Programming Reference

For function summaries, see:

- Client-Side Functions Overview [./programming-guide/#client-side-functions-overview]

AppendVdHeader (Deprecated)

> **Note**
>
> This function is deprecated. QueueVirtualWrite must be used in all

new virtual drivers.

Places an ICA virtual channel prefix on the output buffer prior to assembling and sending the buffer.

Calling Convention

```
INT WFCAPI AppendVdHeader (
  PWD pWd,
  USHORT Channel,
  USHORT ByteCount);
```

Parameters

pWD

Pointer to a WinStation driver control structure.

Channel

Virtual channel number.

ByteCount

Actual size in bytes of the virtual channel packet data to be sent. Do not include additional bytes reservered for the buffer overhead.

Return Values

If the function succeeds, the return value is CLIENT_STATUS_SUCCESS.

If the function fails, the return value is the error code associated with the failure; use GetLastError to get the extended error information.

Remarks

Call this function to prefix the virtual channel packet with the appropriate header information. Normally the virtual driver sees only the private packet data. However, when a virtual driver sends a virtual channel packet to a server application, it must use this function to prefix the data with the ICA header.

Use OutBufReserve to reserve a buffer prior to making this call. The virtual driver must use this function immediately after a successful OutBufReserve and before any other data is placed in the packet. This action uses the additional four bytes requested in OutBufReserve, so do not include this overhead in *ByteCount*.

If an ICA header or virtual channel data is appended to the buffer, the buffer must be sent to the server before the control leaves the virtual driver.

A pointer to this function is obtained from the VDWRITEHOOK structure after hook registration in DriverOpen. The VDWRITEHOOK structure also provides *pWd*.

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;
UINT16 uiSize;
wdqi.WdInformationClass = WdOpenVirtualChannel;
wdqi.pWdInformation = &OpenVirtualChannel;
wdqi.WdInformationLength = sizeof(OPENVIRTUALCHANNEL);
OpenVirtualChannel.pVCName = CTXPING_VIRTUAL_CHANNEL_NAME;
uiSize = sizeof(WDQUERYINFORMATION);
rc = VdCallWd(pVd, WDxQUERYINFORMATION, &wdqi, &uiSize);
/* 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 and set the channel mask to indicate which channel this driver will handle.

For example:

```
g_usVirtualChannelNum = OpenVirtualChannel.Channel;
pVdOpen->ChannelMask = (1L << g_usVirtualChannelNum);
```

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);
uiSize = sizeof(WDSETINFORMATION);
rc = VdCallWd( pVd, WDxSETINFORMATION, &wdsi, &uiSize);
/* do error processing here */
```

During the registration of the write hook, the WinStation driver passes entry points for the deprecated 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. 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);
}
```

5. 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 get periodic control to perform any action as required. With the Evt_ *and Tmr_* APIs, a more event driven implementation is possible so you may find that the DriverPoll is empty.

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).

puiSize

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

Return Values

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

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.

Return values that begin with CLIENT_ERROR_ are fatal errors; the ICA session is disconnected.

Remarks

Because the client engine is single threaded, a virtual driver is not allowed to block while waiting for a desired result (such as the availability of an output buffer) because this prevents the rest of the client from processing.

The Ping example includes examples of processing that can occur in DriverPoll.

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.

Evt_create

Allocates an event structure containing a callback that can be associated with the input or the output events of a particular file descriptor.

Calling Convention

```
VPSTATUS
Evt_create (
void \*hTC,
PFNDELIVER pDeliverFunc,
void \*pSubscriberId,
PEVT \*out);
```

Parameters

hTC

Pass NULL value as a dummy.

pDeliverFunc

The callback to call.

pSubscriberId

Data passed as an argument to the callback.

out

The event structure returned.

Return Value

The event structure created is returned with the out pointer argument. If the function succeeds, the return value is EVT_SUCCESS.

If the function fails because of insufficient memory, the return value is EVT_OBJ_CREATE_FAILED.

Remarks

The first argument of the callback pSubscriberId is the same as the pSubscriberId used to create the event structure.

The second argument nEvt is a pointer to the event structure responsible for the callback.

Evt_destroy

Destroys previously created event structure by freeing its memory and nulling the given pointer.

Calling Convention

```
VPSTATUS
Evt_destroy (
PEVT \*phEvt);
```

Parameters

phEvt

Pointer to the event object to destroy.

Return Value

If the function succeeds, the return value is EVT_SUCCESS.

Remarks

The event object to destroy must be removed from the event loop using Evt_remove_triggers, before Evt_destroy is called.

Evt_remove_triggers

Removes the previously setup file descriptor selections from the given file descriptor.

Calling Convention

```
VPSTATUS
Evt_remove_triggers (
Int fd);
```

Parameters

fd

The file descriptor to remove all selections from.

Return Value

If the function succeeds, the return value is EVT_SUCCESS.

Remarks

If both the input and output conditions are selected, both the conditions are removed.

Evt_signal

Calls the callback stored within the given event structure.

Calling Convention

```
VPSTATUS
Evt_signal (
PEVT hEvt);
```

Parameters

hEvt

The event structure containing the callback to call.

Return Value

If the function succeeds, the return value is EVT_SUCCESS.

Remarks

Calls the callback function directly. No conditions must be met prior to this call.

Evt_trigger_for_input

Connects the callback of an event structure to trigger on the given file descriptor when it satisfies the input conditions.

Calling Convention

```
VPSTATUS
Evt_trigger_for_input (
PEVT hEvt,
int fd);
```

Parameters

hEvt

The event structure to associate with the input conditions of the given file descriptor.

fd

The file descriptor.Workspace app

Return Value

If the function succeeds, the return value is EVT_SUCCESS.

If the function fails because of insufficient memory, the return value is EVT_OBJ_CREATE_FAILED.

Remarks

The Glib implementation of the event loop used by Citrix Workspace app for Linux watches for the input conditions G_IO_IN and G_IO_HUP.

Evt_trigger_for_output

Connects the callback of an event structure to trigger on the given file descriptor when it satisfies the output conditions.

Calling Convention

```
VPSTATUS
Evt_trigger_for_output (
PEVT hEvt,
int fd);
```

Parameters

hEvt

The event structure to associate with the ouput conditions of the given file descriptor.

fd

The file descriptor.Workspace app

Return Value

If the function succeeds, the return value is EVT_SUCCESS.

If the function fails because of insufficient memory, the return value is EVT_OBJ_CREATE_FAILED.

Remarks

The Glib implementation of the event loop used by Citrix Workspace app for Linux watches for the ouput conditions G_IO_OUT.

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

```
INT 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

The driver returns CLIENT_STATUS_SUCCESS.

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 files. 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 files.

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 files.

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.

MM_clip

Sets the shape of the operating system window “xwin” from the list of sorted rectangles.

Calling Convention

```
void
MM_clip (
UINT32 xwin,
int count,
struct tagTWI_RECT \*rects,
BOOLEAN extended)
```

Parameters

xwin

Operating system session sub-window.

count

Number of rectangles.

rects

Array of rectangles sorted by Y and X.

extended

TRUE for any extensions; otherwise, FALSE.

Return Values

There are no return values.

Remarks

The structure has four long integers for left, top, right, and bottom. Rectangles are YXsorted.

The last argument must be FALSE to start a fresh clipping update, and TRUE to add any clipping updates to the current clipping list.

MM_destroy_window

Destroys a window created by MM_get_window().

Calling Convention

```
void
MM_destroy_window (
UINT32 hwin,
UINT32 xwin,
```

Parameters

hwin

Host (seamless) window identifiers, ignored for non-seamless sessions.

xwin

x sub-window of the session window.

Return Values

There are no return values.

Remarks

MM_destroy_window also removes any window deletion callbacks added with the low level MM_TWI_set_deletion_call.

MM_get_window

Creates an operating system window "xwinp" that is a sub-window of an existing session window with a server handle "hwin".

Calling Convention

```
BOOLEAN
MM_get_window (
UINT32 hwin,
UINT32 \*xwinp,
```

Parameters

hwin

Host (seamless) window identifiers, ignored for non-seamless sessions.

xwinp

Local operating system window identifier. Returns the sub-window identifier of the session window. In this case, the X Window System is the operating system windowing system.

Return Values

If the parent (hwin) exists, the return value is TRUE. If the parent does not exist, the return value is FALSE.

If the return value is FALSE, the function, including window creation, still works. The root window, however, is used as a temporary parent.

A call to MM_get_window() or MM_set_geometry() can be used to reparent to any existing seamless window.

Remarks

When "0" is passed as the server handle in a non-seamless (single window) session, there can be an existing window, *xwinp that is reparented. The sub-window, however, is unmapped.

If the parent is seamless, *xwinp is protected by unmapping and reparenting it to the root before the parent is deleted.

MM_set_geometry

Sets the size and position for an existing sub-window, "xwin" of a session window with the server handle, "hwin".

Calling Convention

```
BOOLEAN
MM_set_geometry (
UINT32 hwin,
UINT32 xwin,
CTXMM_RECT \*rt);
```

Parameters

hwin

Host (seamless) window identifiers, ignored for non-seamless sessions.

xwin

Local operating system window identifier for the session sub-window. In this case, the X Window System is the operating system windowing system.

rt

CTXMM_RECT that describes the new window position and geometry.

Return Values

If the parent (hwin) exists, the return value is TRUE. If the parent does not exist, the return value is FALSE.

If the return value is TRUE, the sub-window is mapped on return.

Remarks

The CTXMM_RECT window rectangle is within the session coordinates which are not window relative and consist of four unsigned 32-bit integers for left, top, right, and bottom.

MM_show_window

Makes a sub-window visible.

Calling Convention

```
void
MM_show_window (
UINT32 xwin)
```

Parameters

xwin

Local operating system window identifier for the session sub-window. In this case, the X Window System is the operating system windowing system.

Return Values

There are no return values.

Remarks

This function is called when the parent seamless window arrives after the geometry is set.

There must, however, be a successful call to MM_get_window() initially.

The function can be called with exactly the same window identifiers as the previous one. It cannot be used if MM_set_geometry() previously returned TRUE.

MM_TWI_clear_new_window_function

Clears the callback function set up using MM_TWI_set_new_window_function.

Calling Convention

```
void
MM_TWI_clear_new_window_function (
void (\*) (UINT32))
```

Parameters

(*)(UINT32))

Callback function pointer to remove.

Return Values

There are no return values.

Remarks

Clears the callback for seamless window creation.

MM_TWI_set_new_window_function

Sets a callback function for seamless window creation.

Calling Convention

```
void
MM_TWI_set_new_window_function (
void (*) (UINT32));
```

Parameters

(*)(UINT32)

Callback function pointer to remove.

Return Values

There are no return values.

Remarks

When MM_get_window() fails because the seamless window is not yet created, MM_TWI_set_new_window_function can be used to watch the creation. The handle must be established only when required and should be removed immediately. The callback argument is the server window handle of a newly created seamless window.

OutBufAppend (Deprecated)

> **Note**
>
> This function is deprecated. QueueVirtualWrite must be used in all

new virtual drivers.

Adds virtual channel packet data to the current output buffer.

Calling Convention

```
INT WFCAPI OutBufAppend(
PWD pWd,
LPBYTE pData,
USHORT ByteCount);
```

Parameters

pWd

Pointer to a WinStation driver control structure.

pData

Pointer to the buffer containing the data to append.

ByteCount

Number of bytes to append to the buffer.

Return Values

If the function succeeds, it returns CLIENT_STATUS_SUCCESS.

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

Remarks

This function adds virtual channel packet data to the end of the current output buffer. A buffer of appropriate size must be reserved before calling this function.

The address for this function is obtained from the VDWRITEHOOK structure after hook registration. The VDWRITEHOOK structure also provides *pWd*.

This function can be called multiple times to build up the content of the buffer. It is not written until OutBufWrite is called. Attempts to write more data than was specified in OutBufReserve cause unpredictable results.

The packet header information must be filled in before this function is called.

If an ICA header or virtual channel data is appended to the buffer, the buffer must be sent to the server before the control leaves the virtual driver.

OutBufReserve (Deprecated)

> **Note**
>
> This function is deprecated. QueueVirtualWrite must be used in all

new virtual drivers.

Checks if a buffer of the requested size is available. This function does not allocate buffers because they are already allocated by the WinStation driver.

Calling Convention

```
INT WFCAPI OutBufReserve(
PWD pWd,
USHORT ByteCount);
```

Parameters

pWd

Pointer to a WinStation driver control structure.

ByteCount

Size in bytes of the buffer needed. This must be four bytes larger than the data to be sent.

Return Values

If a buffer of the specified size is available, the return value is CLIENT_STATUS_SUCCESS.

If a buffer of the specified size is not available, the return value is CLIENT_ERROR_NO_OUTBUF.

Remarks

After this function is called to reserve an output buffer, use the other OutBuf* helper functions to append data and then send the buffer to the server.

If a buffer of the specified size is not available, attempt the operation in a later DriverPoll call.

The developer determines the *ByteCount*, 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 address for this function is obtained from the VDWRITEHOOK structure after hook registration. The VDWRITEHOOK structure also provides the *pWd* address.

OutBufWrite (Deprecated)

> **Note**
>
> This function is deprecated. QueueVirtualWrite must be used in all

new virtual drivers.

Sends a virtual channel packet to XenApp or XenDesktop.

Calling Convention

```
INT WFCAPI OutBufWrite(
PWD pWd);
```

Parameters

pWd

Pointer to a WinStation driver control structure.

Return Values

If the function succeeds, it returns CLIENT_STATUS_SUCCESS.

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

Remarks

This function sends the current output buffer to the host. If a buffer was not reserved or no data was appended, this function does nothing.

If an ICA header or virtual channel data is appended to the buffer, the buffer must be sent to the server before DriverPoll returns.

The address for this function is obtained from the VDWRITEHOOK structure after hook registration. The VDWRITEHOOK structure also provides the *pWd* address.

QueueVirtualWrite

QueueVirtualWrite is an improved scatter gather interface. It queues a virtual write and stimulates packet output if required allowing data to be sent without having to wait for the poll.

Calling Convention

```
int WFCAPI
QueueVirtualWrite (
PWD pWd,
SHORT Channel,
LPMEMORY_SECTION pMemorySections,
USHORT NrOfMemorySections,
USHORT Flag);
```

Parameters

pWd

Pointer to a WinStation driver control structure.

Channel

The virtual channel number

pMemorySections

Pointer to an array memory sections.

NrOfMemorySections

The number of memory sections.

Flag

This can be FLUSH_IMMEDIATELY if the data is required to be sent immediately or ! FLUSH_IMMEDIATELY for lower priority data.

Return Values

If the function succeeds, that is queued successfully, the return value is CLIENT_STATUS_SUCCESS.

If the function fails because of unsuccessful queue, the return value is CLIENT_ERROR_NO_OUTBUF.

Remarks

The interface is simpler as it reduces the call sequence OutBufReserve, AppendVdHeader, OutBufAppend, and OutBufWrite down to a single QueveVirtualWrite call.

The data to be written across the chosen virtual channel is described by an array of MEMORY_SECTION structures, each of which contains a length and data pointer pair. This allows multiple non-contiguous data segments to be combined and written with a single QueueVirtualWrite.

Tmr_create

Creates a timer object and returns its handle.

Calling Convention

```
VPSTATUS
Tmr_create (
```

```
HND hTC,
UINT32 uiPeriod,
PVOID pvSubscriber,
PFNDELIVER pfnDeliver.
PTMR * phTimer);
```

Parameters

hTC

The value is NULL.

uiPeriod

The timeout for the timer in milliseconds.

pvSubscriber

Data passed as an argument to the callback.

pfnDeliver

The callback to call.

phTimer

The returned timer structure.

Return Values

If the function succeeds, the return value is TMR_SUCCESS.

If the function fails because of insufficient memory, the return value is TMR_OBJ_CREATE_FAILED.

Remarks

The default state of a newly created timer object is disabled. The "deliver" function is called when the timer fires.

Tmr_destroy

Destroys the timer object pointed to by the given handle and sets the handle to NULL.

Calling Convention

```
VPSTATUS
Tmr_destroy (
PTMR \* phTimer);
```

Parameters

phTimer

The timer to destroy.

Return Values

If the function succeeds, the return value is TMR_SUCCESS.

Remarks

Tmr_destroy is called for all timer objects when they are not required.

Tmr_setEnabled

Enables or disables a timer object.

Calling Convention

```
VPSTATUS
Tmr_setEnabled (
PTMR \* hTimer);
BOOL fEnabled);
```

Parameters

hTimer

The timer to enable or disable.

fEnabled

Enables or disables the timer.

Return Values

If the function succeeds, the return value is TMR_SUCCESS.

Remarks

Enabling a disabled timer restarts the timing period. Re-enabling an enabled timer, however, does not perform any action.

Tmr_setPeriod

Sets the timeout period for a timer.

Calling Convention

```
VPSTATUS
Tmr_setPeriod (
PTMR \* hTimer);
UNIT32 uiPeriod);
```

Parameters

hTimer

The timer to change the timeout period for.

uiPeriod

The new timeout period in milliseconds.

Return Values

If the function succeeds, the return value is TMR_SUCCESS.

Remarks

If the timer is already running, the timer is reset and fires after the new period. If the timer is disabled, the timeout period is updated but the timer remains disabled.

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,
PUINT16 puiSize);
```

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, used for both input and output.

puiSize

Size of parameter structure, used for both input and output.

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).