View Session Enhancement Programming Guide

VMware View Session Enhancement SDK 1.0

Contents

About This Book

The *View Session Enhancement Programming Guide* provides information about developing applications using the VMware® RPC and Overlay Application Programming Interface (API). VMware provides several different Software Development Kit (SDK) products, each of which targets different developer communities and target platforms. This guide is intended for developers who want to create applications that are used remotely over a VMware View connection. The supported platforms include VMware View 5.0 and greater.

Revision History

This book is revised with each release of the product or when necessary. A revised version can contain minor or major changes. Table 1 summarizes the significant changes in each version of this book.

Table 1: Revision History

Revision Date	Description
14OCT2011	Initial release of the VMware View Session Enhancement SDK providing support for VMware View 5.0

Intended Audience

This book is intended for anyone developing applications to run over a VMware View session running version 5.0 or greater.

View Session Enhancement SDK Programming Guide

The VMware RPC and Overlay APIs provide functions that you can use in a program that runs on the client or guest in a VMware View environment. This guide includes the following topics:

Installation Guide

Following are the steps required for installing the `View Session Enhancement` components on the varying systems that are supported by the SDK.

Remote VM - Agent

The vdpService.dll must be available on the Agent machine.

Local Machine - Client

Windows

The only installation step required is to move the vdpService.dll onto the machine, then run regsvr32.

Linux

The following are the steps to get the `View Session Enhancement` components installed on a Linux system:

1. Unzip and untar the `View Session Enhancement` file.
   `$ tar xzf vdpService-sdk-<buildnum>.tar.gz`
2. Copy the library to the plugin directory.
   `# cp vmware/lib/linux/release/libvdpService.so /usr/lib/pcoip/vchan_plugins/`
3. To enable overlay, create the file `/etc/vmware/vdp/host_overlay_plugins/config` and put the following line in the file: `/usr/lib/pcoip/vchan_plugins/libvdpService.so`.

After following the above steps, the `View Session Enhancement` components will be working. The user compiled plugin will need to be installed in `/usr/lib/vmware/view/vdpService`. The plugin may also be installed in a user defined location. If a non-standard location is used, the location will need to be set in the `VIEW_VDP_SERVICE_PLUGIN_DIR` environment variable.

Overview of the View Session Enhancement API

The View Session Enhancement APIs provide functions that developers can use to develop applications that run over a View connection using PCoIP. In particular, the APIs allow the application to be separated into two components: The remote UI and the local running code.

The View Session Enhancement APIs expose a message framework to the developer that allows for easy communication between the remote and local machines. All interaction with the APIs is done asynchronously.

Supported Operating Systems

The View Session Enhancement APIs run on any Linux or Windows operating system supported by View version 5.0 or later. Please see the VMware View documentation for supported operating systems.

How to Use the View Session Enhancement API

The View Session Enhancement APIs define functions and data types that are used to communicate between two machines, and to display remote media via an overlay. This sections covers the following topics:

- ”View Session Enhancement API Overview” Page 5
- ”View Session Enhancement Components” Page 6
- ”View Session Enhancement API Data Types” Page 7
- ”View Session Enhancement Program Flow” Page 9
- ”View Session Enhancement API Functions” Page 12
- ”View Session Enhancement Sink Functions” Page 52
- ”View Session Enhancement Error Codes” Page 66

View Session Enhancement API Overview

This section will give you a brief overview of the terms and definitions used throughout the View Session Enhancement APIs and this document.

This section is comprised of the following subsections:

- ”Connection” Page 5
- ”Channel” Page 6
- ”ChannelContext” Page 6
- ”Overlay” Page 6
- ”Remote Procedure Call (RPC)” Page 6
- ”Sink” Page 6
- ”Variant” Page 6

Connection

A connection, as used in the context of the View Session Enhancement APIs, refers to the View session over PCoIP. The connection cannot be altered through the View Session Enhancement APIs, but the current state of the connection can be determined. If the connection is not in the connected state (see page 66), then no other action may be taken with the APIs. Notifications will be received via the `VDPService_ChannelNotifySink` through the `OnConnectionStateChanged` callback (page 52). The current state of the connection can also be retrieved via the `GetConnectionState` method found in the `VDPService_ChannelInterface` (page 12).

Channel

The Channel in the View Session Enhancement APIs represents the connection between the remote application and the local plugin. The state of the channel will not necessarily match the status of the connection (see above).

Notifications of changes in the channel state will be received through the `VDPService_ChannelNotifySink` (page 52) that has been registered with that channel. The `OnChannelStateChanged` callback will deliver the state change. The current state of the channel can be queried by the `GetChannelState` method in the `VDPService_ChannelInterface` (page 12).

ChannelContext

A ChannelContext is a wrapper for the parameters and return values for a remote call. The ChannelContext holds all of the information for the receiver of a remote call via `Invoke` to determine what method was requested. Interaction with the ChannelContext is done using the `VDPRPC_ChannelContextInterface` (page 19).

Overlay

An overlay is a window or image set over another in order to have the "overlayed" image or window appear to be part of the underlying UI. This is typically done for video that is being played locally, but needs to appear as if it is being played on the remote machine.

Remote Procedure Call (RPC)

A remote procedure call (RPC) is an invocation of a method on a non-local machine. Typically, the remote machine will publish a set of methods that it responds to, and the client will invoke those methods via some channel. In the context of the View Session Enhancement APIs, a call to `Invoke` (page 18) will initiate an RPC.

Sink

The View Session Enhancement APIs use sinks, which are structs of function pointers, to communicate asynchronously with user code. Each API has one or more set of sinks that the user must register to receive necessary callbacks that will give the user important information. The various sinks defined in the View Session Enhancement APIs are covered beginning on page 52.

Variant

To ease cross-platform communication, all parameters used with the VDP RPC API are wrapped in a `VDP_RPC_VARIANT`. This struct holds an identifier that indicates the type of data held in the variant, and the data itself. Use of variants is done through the `VDPRPC_VariantInterface` (page 18).

View Session Enhancement Components

Any system that utilizes the View Session Enhancement APIs will be separated into two components: the code that will run on the remote desktop, which we will refer to as the Application; and the piece that will be installed on the local client, which we will refer to as the Plugin.

Each component has requirements that must be met to interact with the View Session Enhancement APIs. The following sections will discuss the shared requirements, and the requirements that are unique to each component.

Query Interface

Both the Application and Plugin will need to interact with the `VDP_SERVICE_QUERY_INTERFACE`, which is defined in `vdpService.h`. The difference is where the component receives a reference to this struct. The struct has two members; a version attribute, and a function pointer. The version attribute notifies the user's application which version of the APIs are available. The function pointer is how the user's code will access the other APIs in the system. The definition of the function pointer is listed in Example 1.

Example 1 VDP_SERVICE_QUERY_INTERFACE QueryInterface Function

```
Bool (*QueryInterface) (const GUID *iid, void *iface);
```

The QueryInterface() function is used to fetch the functions that the user needs to interact with the View Session Enhancement SDK. Table 2 lists the GUIDs that are defined by View Session Enhancement and which function lists the GUID will return.

Table 2: View Session Enhancement GUIDs

GUID	Returned Function List	Version	Header file
`GUID_VDPRPC_VariantInterface_V1`	`VDPRPC_VariantInterface`	v1	`vdprpc_interfaces.h`
`GUID_VDPRPC_ChannelObjectInterface_V1`	`VDPRPC_ChannelObjectInterface`	v1	
`GUID_VDPRPC_ChannelContextInterface_V1`	`VDPRPC_ChannelContextInterface`	v1	
`GUID_VDPOverlay_GuestInterface_V1`	`VDPOverlay_GuestInterface`	v1	`vdpOverlay.h`
`GUID_VDPOverlay_ClientInterface_V1`	`VDPOverlay_ClientInterface`	v1	
`GUID_VDPService_ChannelInterface_V1`	`VDPService_ChannelInterface`	v1	`vdpService_interfaces.h`

Example 2 shows how to request one of the interfaces discussed above.

Example 2 Fetching version one of the VDPService_ChannelInterface through QueryInterface

```
VDP_SERVICE_QUERY_INTERFACE qi;
VDPService_ChannelInterface ci;
qi.QueryInterface(GUID_VDPService_ChannelInterface_V1, &ci);
```

Application

The component that is used on the remote desktop will be launched by the user. Once the application has started and the `vdpService.dll` is loaded, the Application will need to call `VDPService_ServerInit()` in the DLL. Conversely, `VDPService_ServerExit()` must be called when the application is exiting. These functions are discussed in Table 3.

Table 3: View Session Enhancement Server Functions

Function	Description
`VDPService_ServerInit`	Invoked at the start of the Application. An identifying string (the token) must be passed to the function, and it will return a pointer to the `VDP_SERVICE_QUERY_INTERFACE` (page 7) and the channel handle for this Application (used to initialize user threads - page 13).
`VDPService_ServerExit`	Called when the Application is closing down.

Example 3 is a representation of how an application that wishes to use the View Session Enhancement APIs might initialize.

Plugin

The major difference between the Plugin and the Application, is that the code on the local client will be loaded by VMware View. Thus, the user compiled code will need to be built into a DLL or a shared object that will be loaded by the system. The Plugin must export a set of functions, which are all listed in Table 4.

View Session Enhancement API Data Types

The View Session Enhancement APIs use many different data types. This section will discuss the different types and is comprised of the following sections:

Example 3 View Session Enhancement Application Start Up

```
/* program startup (_tWinMain for example) */
VDP_SERVICE_QUERY_INTERFACE qi;
void *channelHandle;
VDPRPC_VariantInterface vi;
VDPOverlay_GuestInterface ogi;
/* other interfaces omitted */
VDPService_ServerInit("example" /* token */, &qi, &channelHandle);
qi.QueryInterface(GUID_VDPRPC_VariantInterface_V1, &vi);
qi.QueryInterface(GUID_VDPOverlay_GuestInterface_V1, &ogi);
/* ... */
```

Table 4: View Session Enhancement Exported Plugin Functions

Function	Description
`VDPService_PluginInit`	Invoked when the DLL or SO has been loaded. This is where the Plugin receives it's reference to the `VDP_SERVICE_QUERY_INTERFACE`.
`VDPService_PluginExit`	Invoked when the DLL or SO has been unloaded and the VMware View session has ended.
`VDPService_PluginGetTokenName`	This method is used by View Session Enhancement to match the Plugin to the Application. The token returned by this method must match the token passed to `VDPService_ServerInit` for the matching Application, or no communication will occur.
`VDPService_PluginCreateInstance`	Invoked when a new channel's identifier matches the one returned from `VDPService_PluginGetTokenName`. More than one instance of a plugin may exist. View Session Enhancement will match instances of the plugin to the correct channel.
`VDPService_PluginDestroyInstance`	Called when the channel that this plugin instance was running on has been closed.

- ”View Session Enhancement Data Types” Page 8
- ”VDP RPC Data Types” Page 8
- ”VDP Overlay Data Types” Page 9

View Session Enhancement Data Types

The `VDP Service API` uses the data types listed in the following tables.

Table 5: VDPService Data Types

Data Type	Description
`VDPService_ConnectionState`	This enum is used to indicate the current state of the remote connection.
`VDPService_ChannelState`	This enum is used to indicate the current state of a particular channel.

VDP RPC Data Types

The data types defined specifically for use with the VDP RPC API are listed in Table 6.

Table 6: VDPRPC Data Types

Data Type	Description
`VDP_RPC_VARENUM`	This enum is used to indicate the type of data stored in a `VDP_RPC_VARIANT`.
`VDP_RPC_BLOB`	Data type used when the data does not fit into any predefined `VDP_RPC_VARENUM`. Since the data is sent as is, `VDP Service` cannot protect against changes in byte endianess, so care must be taken as errors may occur.
`VDP_RPC_VARIANT`	Wraps the data for the RPC calls. Any data that will be sent using the `Invoke` call must be contained in a `VDP_RPC_VARIANT`.
`VDPRPC_ObjectState`	Represents the state of an object. Only objects in the `VDP_RPC_OBJ_CONNECTED` state can be used in the `Invoke` call.
`VDPRPC_ObjectConfigurationFlags`	These flags are used to configure new objects.

VDP Overlay Data Types

There are data types defined for use with the Overlay API. They can be found in the `vdpOverlay.h` header file, and are covered in Table 7 and Table 8.

Table 7: VDP Overlay Guest Data Types

Data Type	Description
`VDPOverlay_WindowId`	A uint32 representation of the native OS window.
`VDPOverlay_UserArgs`	Parameter that is passed through to the callback on the remote side.
`VDPOverlay_LayoutMode`	This enum represents all of the different layouts supported by the VDP Overlay API.
`VDPOverlay_Error`	Returned by many of the Overlay functions. Indicates the varying results that may occur.

Table 8: VDP Overlay Client Data Types

Data Type	Description
`VDPOverlayClient_ContextId`	Returned from the `VDPOverlayClient_Init`. This id is used in every call to the Client API.
`VDPOverlayClient_OverlayInfo`	Used in the call to `VDPOverlayClient.v1.GetInfo`. The user sets the first parameter (cbSize), and the rest of the struct is filled in after the GetInfo call.

View Session Enhancement Program Flow

Initialization

Application Start up of the remote side of the View Session Enhancement system is user controlled. Upon application launch, it is up to the user code to call the `VDPServer_Init` method and get the `VDP_SERVICE_QUERY_INTERFACE` structure. Upon receiving that structure, the user code should use the `QueryInterface` method to fetch all the interfaces required to do it's work.

Plugin Unlike the remote end, the local code will be initialized by the View Session Enhancement system. In the `VDPService_PluginInit` call, the user code should store the passed in reference to the `VDP_SERVICE_QUERY_INTERFACE` structure and use it to request all the interfaces needed to function. Note that at this point the user code has only been loaded. Once the matching application for the loaded plugin is started, then the `VDPService_PluginCreateInstance` is called. In this callback, the user may return a pointer that will be returned in each callback, so the user code can maintain state. To match a plugin and an application, the `VDPService_PluginGetTokenName` method is used, and the string returned from that function is compared to the string that was given by the application.

Before returning from the `VDPService_PluginCreateInstance` callback, the user code MUST call `Connect` from the `VDPService_ChannelInterface` (page 12).

Register Sinks In order to receive callbacks from the View Session Enhancement system, you must register sinks for different notifications. The first sink to register is the `VDPService_ChannelNotifySink` (page 52). This sink will notify you of changes to the connection state, the channel state, and will notify you when the application has created an object (see the ChannelObject section below). The sink is registered using the `RegisterChannelNotifySink` (page 14) method found in the `VDPService_ChannelInterface`. Once the sink is registered, you will receive a handle for that sink that may be used to unregister the sink. Registering the channel notify sink should be done prior to calling `Connect` to ensure that you will receive a notification when the channel is available.

NOTE: Upon registering the channel notify sink, you most likely will not receive a callback for a connection state change. This is due to the fact that by the time the application or plugin is started, the connection should be in the connected state. If you want to confirm that the connection is in the proper state prior to any actions, you may use the `GetConnectionState` (page 16) method.

Other sinks that exist are for individual channel objects (`VDPRPC_ObjectNotifySink` page 55), callbacks for each RPC call (`VDPRPC_RequestCallback` page 55), and important overlay notifications for the guest (`VDPOverlayGuest_Sink` page 58) and for the client (`VDPOverlayClient_Sink` page 58).

ThreadInit First, we will define the main thread. The main thread in the context of View Session Enhancement is the thread that the user called `VDPService_ServerInit` on, or the thread that the `VDPService_PluginCreateInstance` callback was received on. Any other thread created by the user that will use View Session Enhancement APIs must have the `ThreadInitialize` (page 13) method called prior to any other action.

If a thread that has been initialized will no longer need to interact with View Session Enhancement, you must uninitialize the thread using the `ThreadUninitialize` (page 13) method.

Channel

Connect/Disconnect In order for communication to occur, the channel between the Application and Plugin must be active. To initialize the channel connection, `Connect` (page 15) is used. It must be called on both sides of the connection for each channel. To shut down a channel, the `Disconnect` (page 16) method is used.

After calling `Disconnect` (or whenever the channel is in a disconnected state), you need to free all your channel objects using the `DestroyChannelObject` (page 27) method. After the channel is connected again, you must recreate any required objects.

RPC

The following information is specific to use of the RPC API. All VDPService initialization steps will still need to be done.

ChannelObject

Creation The last piece required for communication are the ChannelObjects. For an RPC to be sent, there must be an object with the same name on both sides of the connection. To create an object, you use the `CreateChannelObject` (page 27) method. The order of object creation (application first or plugin first) does not matter. The initial state of the object at this stage is disconnected.

Upon creating an object, a message is sent to the other side of the connection, and the `OnPeerObjectCreated` (page 54) callback will be received. A matching object must be created using the `CreateChannelObject` method before the objects can be used. Upon creating the matching object, the state of the object on both sides will be connected. Note that both sides should receive an object state change notification indicating this.

Invoke Once an object has been created, an RPC may be invoked. This is done using the `Invoke` (page 30) method. Be aware that the Invoke call must be made on the thread that the object was created, unless the object was configured to allow invoke on any thread (see `CreateChannelObject` on page 27).

The invoke call requires a ChannelContext. The context is a wrapper for all of the data for the RPC (command, parameters, etc.). You create a context using the `CreateContext` (page 29) function. After creating the context, you add all of the information for the call to the context using the `VDPRPC_ChannelContextInterface` (page 19) and then pass it to invoke.

NOTE: Though the context was user created, upon a successful Invoke call, the context will be freed by View Session Enhancement. This is due to the asynchronous nature of View Session Enhancement, since the context may not be sent by the time Invoke has returned.

NOTE: Each ChannelContext has a unique id, that may be recovered using the `GetId` (page 31) method. The id of the context passed to an Invoke call will be returned as a parameter in the OnDone and OnAbort handlers. This can be used to map the callbacks to the Invoke call that they refer to. The id of the context passed to the handlers will not match the originating context id, as this represents the return values given by the other end of the connection.

Variant All data added to the channel context must be in a `VDP_RPC_VARIANT` struct. Example 4 shows how to add data to a variant and append it to a context.

Example 4 Adding data to a Variant and appending it to a context

```
VDP_RPC_VARIANT var;
VDPRPC_VariantInterface varIface;
VDPRPC_ChannelContextInterface ctxtIface;
void *contextHandle;
varIface.v1.VariantFromInt32(&var, 32);
ctxtIface.v1.AppendParam(contextHandle, &var);
varIface.v1.VariantClear(&var);
varIface.v1.VariantFromString(&var, "sample string");
ctxtIface.v1.AppendNamedParam(contextHandle, "sample param", &var);
varIface.v1.VariantClear(&var);
```

Note that after each use of the variant, the `VariantClear` method is called. This is to ensure that all resources are properly freed before any subsequent uses.

OnInvoke On a successful Invoke call, the peer object receives an `OnInvoke` (page 56) callback. In this callback you receive a ChannelContext. This context contains all of the information that was given in the Invoke call. To respond, you add the appropriate return code and return values to the passed in channel context. This context will be returned to the caller once the `OnInvoke` call returns.

Shutdown

Application When it's time for the application to shut down, the user must call `VDPService_ServerExit`.

Plugin If the channel associated with a particular plugin instance is closed, the plugin's `VDPService_PluginDestroyInstance` method will be called. The plugin should free all it's resources and prepare to be shut down.

Overlay

The following instructions are for using the overlay portion of the SDK. All VDPService initialization steps will still need to be done.

Setup

Guest To use overlay, the first step is to initialize the guest interface. This is done using the `Init` (page 42) method. After a successful initialization, register the window that will be 'overlayed'. You do this by calling the `RegisterWindow` (page 43) method. The size and position of the registered window will be tracked and sent to the client automatically.

If the client does not reject the registered window, you will receive the `OnOverlayReady` callback. Once this callback is received, you must use the `EnableOverlay` (page 45) function to have the client side overlay be displayed.

When finished with the window, unregister it using `UnregisterWindow` (page 43).

Client On the client, the first step is also initialization. Do this by calling `Init` (page 49). You'll receive a context id from the init call. This is used to identify your plugin instance. When the guest has registered a window, the client will be notified via the `OnWindowRegistered` sink callback (page 61). You will receive a window id in this callback. This and the context id are required for updating the overlay.

Once you receive the `OnOverlayReady` (page 59) callback, you're ready to start displaying your media. To display an image, you use the `Update` (page 50) method. Note that unless the copyImage parameter is set to true, the View Session Enhancement system will not keep a copy of the image. Thus if you don't own the image resource or need to free it, this must be set.

When finished with the overlay interface on the client side, be sure to call the `Exit` (page 49) method.

View Session Enhancement API Functions

The View Session Enhancement APIs are broken into three header files containing the functions required to use all aspects of the APIs. Those categories are:

- "Channel Interaction Functions" Page 12
- "RPC Functions" Page 18
- "Overlay Functions" Page 41

Channel Interaction Functions

The View Session Enhancement SDK contains the header file `vdpService_interfaces.h`. This file declares a structure of function pointers that are used to interact with the remote connection, referred to as the Channel.

Table 9: VDPService ChannelInterface Function Members

v1	
Function	**Page**
`ThreadInitialize`	13
`ThreadUninitialize`	13
`Poll`	14
`RegisterChannelNotifySink`	14
`UnregisterChannelNotifySink`	15
`Connect`	15
`Disconnect`	16
`GetConnectionState`	16
`GetChannelState`	17

VDPService_ChannelInterface

Version 1 - ThreadInitialize

```
Bool (*ThreadInitialize)(void *channelHandle, uint32 unusedFlag);
```

Summary

Initialize thread for use with the View Session Enhancement APIs. This must be called on any thread that is not the main thread. This method should not be called on the thread that received the `VDPService_PluginCreateInstance` callback, or that the `VDPService_ServerInit` call was made from.

Parameters

channelHandle	-	This is the channel handle that is returned from the `VDPService_ServerInit` call or passed in from on the `VDPService_PluginCreateInstance` method. It represents the channel instance that this plugin instance is running on.
unusedFlag	-	Currently unused. Padding for future expansion.

Return Values

TRUE	-	If the thread was successfully initialized, TRUE is returned.
FALSE	-	Thread initialization failed.

Side Effects

None.

VDPService_ChannelInterface

Version 1 - ThreadUninitialize

```
Bool (*ThreadUninitialize)(void);
```

Summary

Uninitializes the calling thread. This frees all resources associated with View Session Enhancement. No API calls should be made from this thread after this call. This should only be called on threads that had `ThreadInitialize` invoked.

Parameters

None.

Return Values

TRUE	-	If the thread was successfully uninitialized, TRUE is returned.
FALSE	-	Thread uninitialization failed.

Side Effects

None.

VDPService_ChannelInterface

Version 1 - Poll

```
void (*Poll)(void);
```

Summary

The Poll method is used to allow the View Session Enhancement system to process any waiting events. This call is required on any thread that the `ThreadInitialize` (page 13) call was made to allow View Session Enhancement to function. If there are no waiting events, this call will just return. Note that **all** waiting events will be processed, so control may not be returned to you for some time. Most events will cause calls to registered sinks.

On Windows, if the thread uses it's own message loop, using the method is not required.

Parameters

None.

Return Values

None.

Side Effects

Callbacks may be fired.

VDPService_ChannelInterface

Version 1 - RegisterChannelNotifySink

```
Bool (*RegisterChannelNotifySink)(const VDPService_ChannelNotifySink *sink, void *userData, uint32 *sinkHandle);
```

Summary

Registers the given `VDPService_ChannelNotifySink` with the channel associated with the calling thread. You may register any number of sinks, and each will receive a callback when an event occurs. The sinkHandle parameter will be set to the handle assigned to the given sink. This is used to unregister the sink with the channel.

Parameters

sink	-	The sink to register with the channel.
userData	-	Data that will be passed into any callbacks to this sink. Can be NULL.
sinkHandle	-	Set to the handle assigned to this sink.

Return Values

TRUE	-	If the sink was successfully registered.
FALSE	-	If the sink failed to register.

Side Effects

None.

VDPService_ChannelInterface

Version 1 - UnregisterChannelNotifySink

```
Bool (*UnregisterChannelNotifySink)(uint32 sinkHandle);
```

Summary

Removes the sink associated with the given handle from the list of sinks the channel associated with the calling thread will notify of View Session Enhancement events.

Parameters

sinkHandle	-	The handle returned from `RegisterChannelNotifySink` of the sink to be unregistered.

Return Values

TRUE	-	If the sink that matches the given handle was unregistered.
FALSE	-	The sink is still registered with the channel.

Side Effects

None.

VDPService_ChannelInterface

Version 1 - Connect

```
Bool (*Connect)(void);
```

Summary

Starts the channel connection. This must be called on both the Application and Plugin side, though the order does not matter. This must be called prior to exiting the `VDPService_PluginCreateInstance` callback.

Parameters

None.

Return Values

TRUE	-	Call succeeded.
FALSE	-	Call failed.

Side Effects

None.

VDPService_ChannelInterface

Version 1 - Disconnect

```
Bool (*Disconnect)(void);
```

Summary

Closes the underlying channel connection. This may be called on either the Plugin or Application side.

Parameters

None.

Return Values

TRUE	-	Call succeeded.
FALSE	-	Call failed.

Side Effects

None.

VDPService_ChannelInterface

Version 1 - GetConnectionState

```
VDPService_ConnectionState (*GetConnectionState)(void);
```

Summary

Used to query the state of the underlying View session. Note that depending on when a sink was registered, you may not receive a callback noting the connection state has changed. This method can always be used to determine the state of the connection at any time.

Parameters

None.

Return Values

VDP_SERVICE_CONN_UNINITIALIZED	-	The View session cannot be found.
VDP_SERVICE_CONN_DISCONNECTED	-	The View session is currently inactive.
VDP_SERVICE_CONN_PENDING	-	The View session is not connected, but active on the calling end.
VDP_SERVICE_CONN_CONNECTED	-	The View session is active.

Side Effects

None.

VDPService_ChannelInterface

Version 1 - GetChannelState

```
VDPService_ChannelState (*GetChannelState)(void);
```

Summary

Used to query the current state of the channel connection between Application and Plugin instances. The channel to query is determined by the id of the calling thread.

Parameters

None.

Return Values

VDP_SERVICE_CHAN_UNINITIALIZED	-	The channel for this thread could not be found.
VDP_SERVICE_CHAN_DISCONNECTED	-	The channel is inactive.
VDP_SERVICE_CHAN_PENDING	-	The channel is open on the calling end, but not yet connected.
VDP_SERVICE_CHAN_CONNECTED	-	The channel is active.

Side Effects

None.

RPC Functions

The `vdprpc_interfaces.h` header file included in the View Session Enhancement SDK contains a set of structs of function pointers to send RPC messages.

Table 10: VDPRPC VariantInterface Function Members

v1	
Function	**Page**
`VariantInit`	20
`VariantCopy`	20
`VariantClear`	21
`VariantFromChar`	21
`VariantFromShort`	22
`VariantFromUShort`	22
`VariantFromInt32`	23
`VariantFromUInt32`	23
`VariantFromInt64`	24
`VariantFromUInt64`	24
`VariantFromFloat`	25
`VariantFromDouble`	25
`VariantFromStr`	26
`VariantFromBlob`	26

Table 11: VDPRPC ChannelObjectInterface Function Members

v1	
Function	**Page**
`CreateChannelObject`	27
`DestroyChannelObject`	27
`GetObjectState`	28
`GetObjectName`	28
`CreateContext`	29
`DestroyContext`	**??**
`Invoke`	30

Table 12: VDPRPC ChannelContextInterface Function Members

v1	
Function	**Page**
`GetId`	31
`GetCommand`	32
`SetCommand`	32
`GetNamedCommand`	33
`SetNamedCommand`	33
`GetParamCount`	34
`AppendParam`	34
`GetParam`	35
`AppendNamedParam`	36
`GetNamedParam`	36
`GetReturnCode`	37
`SetReturnCode`	37
`GetReturnValCount`	38
`AppendReturnVal`	39
`GetReturnVal`	39
`AppendNamedReturnVal`	40
`GetNamedReturnVal`	40

VDPRPC_VariantInterface

Version 1 - VariantInit

```
Bool (*VariantInit)(VDP_RPC_VARIANT *v);
```

Summary

Initializes the given `VDP_RPC_VARIANT`.

Parameters

v	-	The variant that will be initialized.

Return Values

TRUE	-	Variant was successfully initialized.
FALSE	-	Initialization failed.

Side Effects

None.

VDPRPC_VariantInterface

Version 1 - VariantCopy

```
Bool (*VariantCopy)(VDP_RPC_VARIANT *target, const VDP_RPC_VARIANT *src);
```

Summary

Copies the data held from the Variant src to the Variant target. Any data held by the target will be overwritten. Any data previously held in target will be freed before overwritting it with the data in src.

Parameters

target	-	The Variant to copy the data to.
src	-	The Variant to copy the data from.

Return Values

TRUE	-	Copy succeeded.
FALSE	-	Copy failed; target is unchanged.

Side Effects

None.

VDPRPC_VariantInterface

Version 1 - VariantClear

```
Bool (*VariantClear)(VDP_RPC_VARIANT *v);
```

Summary

Clears and frees any resources held by the given Variant. This should be called whenever you are finished with a Variant. This includes after successfully adding a Variant to a context.

Parameters

v	-	The Variant to clear.

Return Values

TRUE	-	The Variant is returned to initialized state.
FALSE	-	The Variant is unchanged.

Side Effects

None.

VDPRPC_VariantInterface

Version 1 - VariantFromChar

```
Bool (*VariantFromChar)(VDP_RPC_VARIANT *v, char c);
```

Summary

Stores the given char in the given Variant and sets the internal type properly.

Parameters

v	-	The Variant to set.
c	-	The char to store.

Return Values

TRUE	-	The char was successfully stored in the Variant.
FALSE	-	Setting the Variant failed.

Side Effects

None.

VDPRPC_VariantInterface

Version 1 - VariantFromShort

```
Bool (*VariantFromShort)(VDP_RPC_VARIANT *v, short s);
```

Summary

Stores the given short in the given Variant and sets the internal type properly.

Parameters

v	-	The Variant to set.
s	-	The short to store.

Return Values

TRUE	-	The short was successfully stored in the Variant.
FALSE	-	Setting the Variant failed.

Side Effects

None.

VDPRPC_VariantInterface

Version 1 - VariantFromUShort

```
Bool (*VariantFromUShort)(VDP_RPC_VARIANT *v, unsigned short us);
```

Summary

Stores the given unsigned short in the given Variant and sets the internal type properly.

Parameters

v	-	The Variant to set.
us	-	The unsigned short to store.

Return Values

TRUE	-	The unsigned short was successfully stored in the Variant.
FALSE	-	Setting the Variant failed.

Side Effects

None.

VDPRPC_VariantInterface

Version 1 - VariantFromInt32

```
Bool (*VariantFromInt32)(VDP_RPC_VARIANT *v, int32 i);
```

Summary

Stores the given int32 in the given Variant and sets the internal type properly.

Parameters

v	-	The Variant to set.
i	-	The int32 to store.

Return Values

TRUE	-	The int32 was successfully stored in the Variant.
FALSE	-	Setting the Variant failed.

Side Effects

None.

VDPRPC_VariantInterface

Version 1 - VariantFromUInt32

```
Bool (*VariantFromUInt32)(VDP_RPC_VARIANT *v, uint32 ui);
```

Summary

Stores the given uint32 in the given Variant and sets the internal type properly.

Parameters

v	-	The Variant to set.
ui	-	The uint32 to store.

Return Values

TRUE	-	The uint32 was successfully stored in the Variant.
FALSE	-	Setting the Variant failed.

Side Effects

None.

VDPRPC_VariantInterface

Version 1 - VariantFromInt64

```
Bool (*VariantFromInt64)(VDP_RPC_VARIANT *v, int64 i);
```

Summary

Stores the given int64 in the given Variant and sets the internal type properly.

Parameters

v	-	The Variant to set.
i	-	The int64 to store.

Return Values

TRUE	-	The int64 was successfully stored in the Variant.
FALSE	-	Setting the Variant failed.

Side Effects

None.

VDPRPC_VariantInterface

Version 1 - VariantFromUInt64

```
Bool (*VariantFromChar)(VDP_RPC_VARIANT *v, uint64 ui);
```

Summary

Stores the given uint64 in the given Variant and sets the internal type properly.

Parameters

v	-	The Variant to set.
ui	-	The uint64 to store.

Return Values

TRUE	-	The uint64 was successfully stored in the Variant.
FALSE	-	Setting the Variant failed.

Side Effects

None.

VDPRPC_VariantInterface

Version 1 - VariantFromFloat

```
Bool (*VariantFromFloat)(VDP_RPC_VARIANT *v, float f);
```

Summary

Stores the given float in the given Variant and sets the internal type properly.

Parameters

v	-	The Variant to set.
f	-	The float to store.

Return Values

TRUE	-	The float was successfully stored in the Variant.
FALSE	-	Setting the Variant failed.

Side Effects

None.

VDPRPC_VariantInterface

Version 1 - VariantFromDouble

```
Bool (*VariantFromDouble)(VDP_RPC_VARIANT *v, double d);
```

Summary

Stores the given double in the given Variant and sets the internal type properly.

Parameters

v	-	The Variant to set.
d	-	The double to store.

Return Values

TRUE	-	The double was successfully stored in the Variant.
FALSE	-	Setting the Variant failed.

Side Effects

None.

VDPRPC_VariantInterface

Version 1 - VariantFromStr

```
Bool (*VariantFromStr)(VDP_RPC_VARIANT *v, const char *str);
```

Summary

Stores a copy of the given const char * in the given Variant and sets the internal type properly.

Parameters

v	-	The Variant to set.
str	-	The const char * to copy.

Return Values

TRUE	-	The const char * was successfully copied into the Variant.
FALSE	-	Setting the Variant failed.

Side Effects

None.

VDPRPC_VariantInterface

Version 1 - VariantFromBlob

```
Bool (*VariantFromBlob)(VDP_RPC_VARIANT *v, VDP_RPC_BLOB *blob);
```

Summary

Stores a copy of the given `VDP_RPC_BLOB` in the given Variant. This should only be used for data that doesn't fit any of the other types. Data will be sent 'as is', so changes in architecture (sending from Linux client to Windows guest for example) can wreak havoc on your data. Please use caution.

Parameters

v	-	The Variant to set.
blob	-	The `VDP_RPC_BLOB` to copy.

Return Values

TRUE	-	The `VDP_RPC_BLOB` was successfully copied into the Variant.
FALSE	-	Setting the Variant failed.

Side Effects

None.

VDPRPC_ChannelObjectInterface

Version 1 - CreateChannelObject

```
Bool (*CreateChannelObject)(const char *name, const VDPRPC_ObjectNotifySink *sink, void *userData,
  VDPRPC_ObjectConfigurationFlags configFlags, void **objectHandle);
```

Summary

Creates a channel object with the given name. This call, with the same object name, must be made on both the Plugin and Application for communication to happen. Objects begin in the `VDP_RPC_OBJ_PENDING` state. Once the peer object is created (which may be prior to the call), the state will go to `VDP_RPC_OBJ_CONNECTED`. The sink registered with the object will receive notifications of events involving the new object. A handle for the created object will be returned in the objectHandle parameter.

Note that objects must be used on the thread on which the are created, unless configured with the `VDP_RPC_OBJ_CONFIG_INVOKE_ALLOW_ANY_THREAD` flag. If this option is used, the user is responsible for thread saftey.

Parameters

name	-	The name for the created object.
sink	-	Sink to be registered with the new object.
userData	-	Data that will be sent to all sink callbacks. Can be NULL.
configFlags	-	Set of configuration options for the new object.
objectHandle	-	Handle for the created object will be stored here.

Return Values

TRUE	-	The object was successfully created.
FALSE	-	Creation of the object failed.

Side Effects

None.

VDPRPC_ChannelObjectInterface

Version 1 - DestroyChannelObject

```
Bool (*DestroyChannelObject)(void *objectHandle);
```

Summary

Frees all resources associated with the given channel object.

Parameters

objectHandle	-	The handle, returned from `CreateChannelObject`, for the object to destroy.

Return Values

TRUE	-	The object was successfully destroyed.
FALSE	-	Destruction of the object failed.

Side Effects

None.

VDPRPC_ChannelObjectInterface

Version 1 - GetObjectState

```
VDPRPC_ObjectState (*GetObjectState)(void *objectHandle);
```

Summary

Used to query the current state of the given object.

Parameters

objectHandle	-	The handle, returned from `CreateChannelObject`, for the object to query.

Return Values

VDP_RPC_OBJ_UNINITIALIZED	-	Object with the given handle could not be found.
VDP_RPC_OBJ_DISCONNECTED	-	Matching peer object was destroyed.
VDP_RPC_OBJ_PENDING	-	Object created locally, waiting for other end to create peer object.
VDP_RPC_OBJ_CONNECTED	-	Given object is connected to it's peer on the other side of the channel.

Side Effects

None.

VDPRPC_ChannelObjectInterface

Version 1 - GetObjectName

```
Bool (*GetObjectName)(void *objectHandle, char *buf, uint32 bufSize);
```

Summary

Queries the given object for the name it was assigned at creation.

Parameters

objectHandle	-	The handle, returned from `CreateChannelObject`, for the object to query.
buf	-	The name of the object is stored in this parameter.
bufSize	-	The size of the passed in buf.

Return Values

TRUE	-	The name was successfully returned.
FALSE	-	An error occurred and the name was not returned.

Side Effects

None.

VDPRPC_ChannelObjectInterface

Version 1 - CreateContext

```
Bool (*CreateContext)(void *objectHandle, void **ppcontextHandle);
```

Summary

Allocates and returns a new channel context to be used for a RPC.

Parameters

objectHandle	-	A handle for a valid channel object.
ppcontextHandle	-	A handle for the new channel context will be returned here.

Return Values

TRUE	-	New context was successfully created and returned.
FALSE	-	Context creation failed.

Side Effects

None.

VDPRPC_ChannelObjectInterface

Version 1 - DestroyContext

```
Bool (*DestroyContext)(void *contextHandle);
```

Summary

Frees all resources associated with a given context. This should only be called on contexts that have not been sent using the `Invoke` call. Only contexts that will not be used should be destroyed by the user.

Parameters

contextHandle	-	The handle for the context to destroy.

Return Values

TRUE	-	Context was successfully destroyed.
FALSE	-	Destruction of the context failed.

Side Effects

None.

VDPRPC_ChannelObjectInterface

Version 1 - Invoke

```
Bool (*Invoke)(void *objectHandle, void *contextHandle, const VDPRPC_RequestCallback *callback, void *userData);
```

Summary

Initiates a RPC between the given object and it's peer on the other end of the channel.

Parameters

objectHandle	-	Handle for the object to send the RPC through.
contextHandle	-	A handle for the context containing the data for this RPC.
callback	-	User supplied callbacks to be used after the Invoke call.
userData	-	User supplied data that will be passed to the callback methods. Can be NULL.

Return Values

TRUE	-	Invoke call succeeded and RPC was sent.
FALSE	-	No RPC was sent due to an error.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - GetId

```
uint32 (*GetId)(void *contextHandle);
```

Summary

Returns the unique id for the given context. This id can be used to map callbacks to the `Invoke` call that they refer to.

Parameters

contextHandle	-	The handle for the context to be queried .

Return Values

uint32	-	The id for the given context.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - GetCommand

```
uint32 (*GetCommand)(void *contextHandle);
```

Summary

Queries the command code that was assigned to the given context. This should be used to determine the remote method that was being called. The command code is set using the `SetCommand` method. If 0 is returned, then the command code should be fetched using `GetNamedCommand`.

Parameters

contextHandle	-	Handle for the context to query.

Return Values

uint32	-	The uint32 command code set for this context. 0 indicates the command was not set as a uint32.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - SetCommand

```
Bool (*SetCommand)(void *contextHandle, uint32 command);
```

Summary

Sets the command code for the given context. The command code represents the remote method that the context is meant to represent. Note that you can also store the command as a string (using `SetNamedCommand` on page 33). Only one can be used though. If you call `SetNamedCommand` after this call, the uint32 command code will be overwritten. Note that 0 should not be used as the command code as View Session Enhancement uses 0 to indicate error.

Parameters

contextHandle	-	Handle for the context to set.
command	-	The command code for the context.

Return Values

TRUE	-	Context command code was successfully set.
FALSE	-	Unable to set the command code.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - GetNamedCommand

```
Bool (*GetNamedCommand)(void *contextHandle, char *buffer, int bufferSize);
```

Summary

Gets the command code assigned to the given context as a string. If the command wasn't stored as a string, this method will return NULL, and you should use the `GetCommand` method instead to get the command code.

Parameters

contextHandle	-	The context to query.
buffer	-	Out parameter that the name will be stored in.
bufferSize	-	Size of the buffer to store the name.

Return Values

TRUE	-	Named command successfully returned.
FALSE	-	The command was not stored as a string.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - SetNamedCommand

```
Bool (*SetNamedCommand)(void *contextHandle, const char *command);
```

Summary

Sets the command code for the given context with a name. Note that you can either set the command as a uint32 (using `SetCommand` on page 32) or as a string, using this method. Only one should be used. If both are used, the second command used will overwrite the previous command.

Parameters

contextHandle	-	Handle for the context to set.
command	-	Command string to use.

Return Values

TRUE	-	Command string successfully set.
FALSE	-	Unable to set the command string.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - GetParamCount

```
int (*GetParamCount)(void *contextHandle);
```

Summary

Returns the number of parameters appended to the given context.

Parameters

contextHandle	-	Handle to the context to query.

Return Values

int	-	Number of parameters stored in the given context.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - AppendParam

```
Bool (*AppendParam)(void *contextHandle, const VDP_RPC_VARIANT *v);
```

Summary

Add the given Variant to the context as a parameter for the method. Appends the parameter to the end of the list.

Parameters

contextHandle	-	Handle for the context to add the parameter to.
v	-	Variant to store in the context. A copy of the data will be made.

Return Values

TRUE	-	Data successfully stored.
FALSE	-	Failed to append the Variant to the context.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - GetParam

```
Bool (*GetParam)(void *contextHandle, int i, VDP_RPC_VARIANT *copy);
```

Summary

Fetches the parameter at the given index. The parameter list index begins at zero.

Parameters

contextHandle	-	The context to query.
i	-	Index of the parameter to fetch.
copy	-	Variant into which the parameter will be copied.

Return Values

TRUE	-	Parameter at the given index was successfully returned.
FALSE	-	Unable to find parameter at the given index.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - AppendNamedParam

```
Bool (*AppendNamedParam)(void *contextHandle, const char *name, const VDP_RPC_VARIANT *v);
```

Summary

Append the given Variant as a parameter to the given context and assign it a name. Note that the parameter is added to the end of the list with all parameters, even those without assigned names.

Parameters

contextHandle	-	The context to append the parameter to.
name	-	Name to assign the parameter.
v	-	The data for the new parameter.

Return Values

TRUE	-	Parameter successfully added.
FALSE	-	Unable to append the parameter to the given context.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - GetNamedParam

```
Bool (*GetNamedParam)(void *contextHandle, int index, char *name, int nameSize, VDP_RPC_VARIANT *copy);
```

Summary

Fetch the parameter at the given index and return the name, if any, that was assigned to the parameter. If no name was given, then the name parameter will remain untouched.

Parameters

contextHandle	-	The context to fetch the parameter from.
index	-	The index of the parameter to return.
name	-	The buffer to store the assigned name in. Can be NULL if not interested in the name.
nameSize	-	Size of the passed in buffer.
copy	-	Variant into which the parameter data will be copied.

Return Values

TRUE	-	Parameter at the given index returned and name (if any) found.
FALSE	-	Unable to fetch the parameter and name at the given index.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - GetReturnCode

```
uint32 (*GetReturnCode)(void *contextHandle);
```

Summary

Queries the value assigned to the given index as a return code. The return code is meant to indicate the success or failure of the remote method call, or as an error code.

Parameters

contextHandle	-	The handle of the context to query.

Return Values

uint32	-	Return code set for the given context.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - SetReturnCode

```
Bool (*SetReturnCode)(void *contextHandle, uint32 code);
```

Summary

Sets the return code for the given context. This should be done in the `OnInvoke` (page 56) callback. This value represents the success or failure of the remote call.

Parameters

contextHandle	-	Handle for the context to set.
code	-	Value for the return code.

Return Values

TRUE	-	Return code of the context set.
FALSE	-	Unable to set the return code.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - GetReturnValCount

```
int (*GetReturnValCount)(void *contextHandle);
```

Summary

Returns the number of Variants stored in the given context as return values.

Parameters

contextHandle	-	The context to query.

Return Values

int	-	Number of return values stored in the given context.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - AppendReturnVal

```
Bool (*AppendReturnVal)(void *contextHandle, const VDP_RPC_VARIANT *v);
```

Summary

Add the given Variant as a return value. The return values can be thought of as out parameters in a procedure call. The user can return any data desired here. The Variant will be added to the end of the return value list.

Parameters

contextHandle	-	Handle for the context to append to.
v	-	Data to append.

Return Values

TRUE	-	Return value was successfully added.
FALSE	-	Failed to add the given Variant as a return value.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - GetReturnVal

```
Bool (*GetReturnVal)(void *contextHandle, int i, const VDP_RPC_VARIANT *v);
```

Summary

Fetches the return value at the given index. Index of the return values begin at zero.

Parameters

contextHandle	-	Context to query.
i	-	Index of the return value to fetch.
v	-	Variant into which the return value data will be copied.

Return Values

TRUE	-	Return value successfully fetched.
FALSE	-	Failed to locate return value at the given index.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - AppendNamedReturnVal

```
Bool (*AppendNamedReturnVal)(void *contextHandle, const char *name, const VDP_RPC_VARIANT *v);
```

Summary

Similar to `AppendReturnVal` but also assigns a name to the return value. The return value will be added to the end of the list of all return values, even those without assigned names.

Parameters

contextHandle	-	Context to add the return value to.
name	-	Name for the given return value.
v	-	Data for the return value.

Return Values

TRUE	-	Name and return value successfully added.
FALSE	-	Failed to add return value.

Side Effects

None.

VDPRPC_ChannelContextInterface

Version 1 - GetNamedReturnVal

```
Bool (*GetNamedReturnVal)(void *contextHandle, int index, char *name, int nameSize, const VDP_RPC_VARIANT *v);
```

Summary

Fetches the return value at the given index. Also returns the name assigned to the return value. This may be NULL.

Parameters

contextHandle	-	Context to query.
index	-	Index of the return value to fetch.
name	-	Buffer to store the name into. Can be NULL.
nameSize	-	Size of the name buffer.
v	-	Variant to copy the return value data into.

Return Values

TRUE	-	Successfully fetched the return value and name at the given index.
FALSE	-	Failed to find the return value or the name.

Side Effects

None.

Overlay Functions

Finally, the `vdpOverlay.h` header file defines the set of functions to use in order to support overlay functionality in an application.

Table 13: VDPOverlay Guest Interface Function Members

v1	
Function	**Page**
`Init`	42
`Exit`	42
`RegisterWindow`	43
`UnregisterWindow`	43
`IsWindowRegistered`	44
`EnableOverlay`	45
`DisableOverlay`	45
`IsOverlayEnabled`	46
`SetLayoutMode`	47
`GetLayoutMode`	47
`SendMsg`	48

Table 14: VDPOverlay Client Interface Function Members

v1	
Function	**Page**
`Init`	49
`Exit`	49
`Update`	50
`GetInfo`	50
`SendMsg`	51

VDPOverlayGuest_Interface

Version 1 - Init

```
VDPOverlay_Error (*Init)(const VDPOverlayGuest_Sink *sink, void *userData);
```

Summary

Initializes the guest-side overlay library. This must be the first overlay API function called.

Parameters

sink	-	Function pointers that are called to notify user of overlay events.
userData	-	Parameter that is passed to sink function callbacks.

Return Values

VDP_OVERLAY_ERROR_SUCCESS	-	Initialization succeeded.
VDP_OVERLAY_ERROR_ALREADY_INITIALIZED	-	Init has already been called.
VDP_OVERLAY_ERROR_INVALID_PARAMETER	-	NULL sink parameter, or invalid sink version.
VDP_OVERLAY_ERROR_ALLOCATION_ERROR	-	Internal system error.

Side Effects

None.

VDPOverlayGuest_Interface

Version 1 - Exit

```
VDPOverlay_Error (*Exit)(void);
```

Summary

Frees all resources held by the View Session Enhancement Overlay APIs and unregisters all windows.

Parameters

None.

Return Values

VDP_OVERLAY_ERROR_SUCCESS	-	Overlay successfully shut down.
VDP_OVERLAY_ERROR_NOT_INITIALIZED	-	Overlay was never initialized.

Side Effects

None.

VDPOverlayGuest_Interface

Version 1 - RegisterWindow

```
VDPOverlay_Error (*RegisterWindow)(VDPOverlay_WindowId windowId, VDPOverlay_UserArgs userArgs);
```

Summary

Registers a window to be overlayed. The position, size, etc. of the window will be sent to the client so that a client-side plug-in can draw to an area of the desktop UI that will cover the window giving the illusion that the drawing is happening on the guest-side.

Parameters

windowId	-	The operating system window identifier.
userArgs	-	Data that will be passed to the client-side plugin-in.

Return Values

VDP_OVERLAY_ERROR_SUCCESS	-	Window was successfully registered.
VDP_OVERLAY_ERROR_NOT_INITIALIZED	-	Overlay not initialized.
VDP_OVERLAY_ERROR_INVALID_PARAMETER	-	Invalid window id.
VDP_OVERLAY_ERROR_ALLOCATION_ERROR	-	Internal system error.
VDP_OVERLAY_ERROR_WINDOW_ALREADY_REGISTERED	-	The given window id has already been registered with the Overlay system.

Side Effects

None.

VDPOverlayGuest_Interface

Version 1 - UnregisterWindow

```
VDPOverlay_Error (*UnregisterWindow)(VDPOverlay_WindowId windowId, VDPOverlay_UserArgs userArgs);
```

Summary

Unregisters a previously registered window. This will not only disable the client-side overlay, but also release any resources allocated to maintain the overlay.

Parameters

windowId	-	The operating system window identifier.
userArgs	-	Data that will be passed to the client-side plugin-in.

Return Values

VDP_OVERLAY_ERROR_SUCCESS	-	Window was unregistered.
VDP_OVERLAY_ERROR_NOT_INITIALIZED	-	Overlay has not been initialized.
VDP_OVERLAY_ERROR_WINDOW_NOT_REGISTERED	-	The given window id was never registered with Overlay.

Side Effects

None.

VDPOverlayGuest_Interface

Version 1 - IsWindowRegistered

```
Bool (*IsWindowRegistered)(VDPOverlay_WindowId windowId);
```

Summary

Determines if a window is currently registered with the guest-side Overlay API.

Parameters

windowId	-	Operating system window identifier.

Return Values

TRUE	-	Window is currently registered.
FALSE	-	The given window id is not registered.

Side Effects

None.

VDPOverlayGuest_Interface

Version 1 - EnableOverlay

```
VDPOverlay_Error (*EnableOverlay)(VDPOverlay_WindowId windowId, VDPOverlay_UserArgs userArgs);
```

Summary

Enables the client-side overlay. Once the window is registered, this function must be called to display the client-side overlay. The windowId must have been previously registered with the View Session Enhancement Overlay API.

Parameters

windowId	-	The operating system window identifier.
userArgs	-	Data that will be passed to the client-side plug-in.

Return Values

VDP_OVERLAY_ERROR_SUCCESS	-	Overlay enabled.
VDP_OVERLAY_ERROR_NOT_INITIALIZED	-	Overlay API has not been initialized.
VDP_OVERLAY_ERROR_OVERLAY_NOT_READY	-	The given windowId has not received the ready signal from the client.
VDP_OVERLAY_ERROR_WINDOW_NOT_REGISTERED	-	The given windowId has not been registered with the Overlay API.

Side Effects

None.

VDPOverlayGuest_Interface

Version 1 - DisableOverlay

```
VDPOverlay_Error (*DisableOverlay)(VDPOverlay_WindowId windowId, VDPOverlay_UserArgs userArgs);
```

Summary

Disables the client-side overlay. Disabling the overlay is a light-weight way to hide the client-side overlay. Unlike `UnregisterWindow` (page 43), resources used to maintain the overlay are not released.

Parameters

windowId	-	Operating system window identifier.
userArgs	-	Data that will be passed to the client-side plug-in.

Return Values

VDP_OVERLAY_ERROR_SUCCESS	-	Overlay disabled.
VDP_OVERLAY_ERROR_NOT_INITIALIZED	-	Overlay API has not been initialized.
VDP_OVERLAY_ERROR_OVERLAY_NOT_READY	-	The given windowId has not received the ready signal from the client.
VDP_OVERLAY_ERROR_WINDOW_NOT_REGISTERED	-	The given windowId has not been registered with the Overlay API.

Side Effects

None.

VDPOverlayGuest_Interface

Version 1 - IsOverlayEnabled

```
Bool (*IsOverlayEnabled)(VDPOverlay_WindowId windowId);
```

Summary

Queries whether the client-side overlay associated with the given windowId is currently enabled.

Parameters

windowId	-	Operating system window identifier.

Return Values

TRUE	-	The overlay is enabled.
FALSE	-	The overlay is disabled.

Side Effects

None.

VDPOverlayGuest_Interface

Version 1 - SetLayoutMode

```
VDPOverlay_Error (*SetLayoutMode)(VDPOverlay_WindowId windowId, VDPOverlay_LayoutMode layoutMode);
```

Summary

Sets the current layout mode for the overlay. The layout mode is used to determine how an image is drawn (e.g. scaled, cropped, etc.) when the size of the image doesn't match the size of the overlay.

Parameters

windowId	-	Operating system window identifier.
layoutMode	-	The desired layout mode.

Return Values

VDP_OVERLAY_ERROR_SUCCESS	-	Layout mode set.
VDP_OVERLAY_ERROR_NOT_INITIALIZED	-	Overlay API has not been initialized.
VDP_OVERLAY_ERROR_INVALID_PARAMETER	-	Invalid layout mode given.
VDP_OVERLAY_ERROR_WINDOW_NOT_REGISTERED	-	The given windowId has not been registered with the Overlay API.

Side Effects

None.

VDPOverlayGuest_Interface

Version 1 - GetLayoutMode

```
VDPOverlay_Error (*GetLayoutMode)(VDPOverlay_WindowId windowId, VDPOverlay_LayoutMode *pLayoutMode);
```

Summary

Gets the current layout mode for the overlay. The layout mode is used to determine how an image is drawn (e.g. scaled, cropped, etc.) when the size of the image doesn't match the size of the overlay.

Parameters

windowId	-	Operating system window identifier.
pLayoutMode	-	Current layout mode stored here.

Return Values

VDP_OVERLAY_ERROR_SUCCESS	-	Current layout successfully retrieved.
VDP_OVERLAY_ERROR_NOT_INITIALIZED	-	Overlay API not initialized.
VDP_OVERLAY_ERROR_INVALID_PARAMETER	-	pLayoutMode is NULL.
VDP_OVERLAY_ERROR_WINDOW_NOT_REGISTERED	-	windowId has not been registered with the Overlay API.

Side Effects

None.

VDPOverlayGuest_Interface

Version 1 - SendMsg

```
VDPOverlay_Error (*SendMsg)(VDPOverlay_WindowId windowId, void *msg, uint32 msgLen);
```

Summary

Sends a message to the client-side plug-in. The client's `OnUserMsg` (page 65) event handler will be called with the message.

Parameters

windowId	-	Operating system window identifier.
msg	-	Buffer that contains the message.
msgLen	-	Size of the msg buffer.

Return Values

VDP_OVERLAY_ERROR_SUCCESS	-	Message sent to the client.
VDP_OVERLAY_ERROR_NOT_INITIALIZED	-	Overlay API not initialized.
VDP_OVERLAY_ERROR_INVALID_PARAMETER	-	Error sending the supplied message.
VDP_OVERLAY_ERROR_WINDOW_NOT_REGISTERED	-	windowId has not been registered with Overlay API.

Side Effects

None.

VDPOverlayClient_Interface

Version 1 - Init

```
VDPOverlay_Error (*Init)(const VDPOverlayClient_Sink *sink, void *userData, VDPOverlayClient_ContextId *pContextI
```

Summary

This function initializes the client-side overlay library. This must be the first method called in the client-side Overlay API.

Parameters

sink	-	Function pointers that are called when events are generated by the Overlay API.
userData	-	Parameter that is passed to all sink callbacks. Can be NULL.
pContextId	-	Returns an id that is used to identify the instance of the API for this plugin instance.

Return Values

VDP_OVERLAY_ERROR_SUCCESS	-	Overlay client API initialized.
VDP_OVERLAY_ERROR_NOT_INITIALIZED	-	Internal View Session Enhancement initialization error.
VDP_OVERLAY_ERROR_INVALID_PARAMETER	-	sink or pContextId is NULL or sink reported an invalid version.
VDP_OVERLAY_ERROR_ALLOCATION_ERROR	-	Internal system error.

Side Effects

None.

VDPOverlayClient_Interface

Version 1 - Exit

```
VDPOverlay_Error (*Exit)(VDPOverlayClient_ContextId contextId);
```

Summary

Performs clean up operations and releases all allocated resources.

Parameters

contextId	-	The id returned from `Init`.

Return Values

VDP_OVERLAY_ERROR_SUCCESS	-	Overlay API properly shut down.
VDP_OVERLAY_ERROR_NOT_INITIALIZED	-	Overlay API was not initialized.
VDP_OVERLAY_ERROR_INVALID_PARAMETER	-	Invalid contextId.

Side Effects

None.

VDPOverlayClient_Interface

Version 1 - Update

```
VDPOverlay_Error (*Update)(VDPOverlayClient_ContextId contextId, VDPOverlay_WindowId windowId,
void *pImage, int32 width,int32 height, int32 pitch, Bool copyImage);
```

Summary

Updates the overlay with a new image. The updated image is displayed when the next frame is drawn.

Parameters

contextId	-	The id returned from `Init` (page 49).
windowId	-	Window id that was received from a previous `OnWindowRegistered` (page 61) event.
pImage	-	A pointer to the RGBX pixels to copy to the overlay.
width	-	The width, in pixels, of the image pointed to by pImage.
height	-	The height, in pixels, of the image pointed to by pImage.
pitch	-	The number of bytes that a single row of the image occupies (typically width * 4).
copyImage	-	If TRUE, a copy of the image data is made. If FALSE, the image data must remain valid until the next `Update` call is made.

Return Values

VDP_OVERLAY_ERROR_SUCCESS	-	Image updated.
VDP_OVERLAY_ERROR_NOT_INITIALIZED	-	Overlay API not initialized.
VDP_OVERLAY_ERROR_INVALID_PARAMETER	-	Invalid contextId, windowId, or image parameters.

Side Effects

None.

VDPOverlayClient_Interface

Version 1 - GetInfo

```
VDPOverlay_Error (*GetInfo)(VDPOverlayClient_ContextId contextId, VDPOverlay_WindowId windowId,
 VDPOverlayClient_OverlayInfo *pOverlayInfo);
```

Summary

Retrieves current information about the overlay.

Parameters

contextId	-	The id returned from `Init` (page 49).
windowId	-	Window id received from a previous `OnWindowRegistered` (page 61) event.
pOverlayInfo	-	Structure that will be filled in with information about the overlay.

Return Values

VDP_OVERLAY_ERROR_SUCCESS	-	Current information retrieved.
VDP_OVERLAY_ERROR_NOT_INITIALIZED	-	Overlay API not initialized.
VDP_OVERLAY_ERROR_INVALID_PARAMETER	-	pOverlayInfo NULL or invalid contextId/windowId.

Side Effects

None.

VDPOverlayClient_Interface

Version 1 - SendMsg

```
VDPOverlay_Error (*SendMsg)(VDPOverlayClient_ContextId contextId, VDPOverlay_WindowId windowId, void *msg, uint32
```

Summary

Sends a message to the guest. The guest's `OnUserMsg` (page **??**) event handler will be called with the message.

Parameters

contextId	-	The id returned from `Init` (page 49).
windowId	-	Window id returned from a previous `OnWindowRegistered` (page 61) event.
msg	-	The buffer that contains the message.
msgLen	-	Length, in bytes, of the msg buffer.

Return Values

VDP_OVERLAY_ERROR_SUCCESS	-	Message sent.
VDP_OVERLAY_ERROR_NOT_INITIALIZED	-	Overlay API not initialized.
VDP_OVERLAY_ERROR_INVALID_PARAMETER	-	Invalid contextId, windowId, or error with msg.

Side Effects

None.

View Session Enhancement Sink Functions

In order to interact and receive notifications of changes, you must register sinks with the View Session Enhancement APIs. There are common sinks, and sinks that are specific to RPC and Overlay. The sinks are covered in the following sections:

- "Channel Sinks" Page 52
- "RPC Sinks" Page 55
- "Overlay Sinks" Page 58

Channel Sinks

Table 15: VDPService ChannelNotifySink Function Members

v1	
Function	**Page**
`OnConnectionStateChanged`	53
`OnChannelStateChanged`	53
`OnPeerObjectCreated`	54

VDPService_ChannelNotifySink

Version 1 - OnConnectionStateChanged

```
void (*OnConnectionStateChanged)(void *userData, VDPService_ConnectionState currentState,
 VDPService_ConnectionState transientState, void *reserved);
```

Summary

This method will be invoked when a change in the underlying View session has changed it's state.

Parameters

userData	-	userData parameter passed in to the `RegisterChannelNotifySink` (page 14) method. May be NULL.
currentState	-	The current state of the connection.
transientState	-	The state change that caused the callback. This can be different than `currentState` if other state changes have already taken place and are waiting to be processed.
reserved	-	Unused parameter.

Return Values

None.

Side Effects

None.

VDPService_ChannelNotifySink

Version 1 - OnChannelStateChanged

```
void (*OnChannelStateChanged)(void *userData, VDPService_ChannelState currentState,
    VDPService_ChannelState transientState, void *reserved);
```

Summary

This method will be invoked when a change in the channel connection used by this plugin instance has occurred.

Parameters

userData	-	userData parameter passed in to the `RegisterChannelNotifySink` (page 14) method. May be NULL.
currentState	-	The current state of the channel.
transientState	-	The state change that caused the callback. This can be different than `currentState` if other state changes have already taken place and are waiting to be processed.
reserved	-	Unused parameter.

Return Values

None.

Side Effects

None.

VDPService_ChannelNotifySink

Version 1 - OnPeerObjectCreated

```
void (*OnPeerObjectCreated)(void *userData, const char *objName, void *reserved);
```

Summary

This method is invoked when an object was created on the other side of the channel connection, and no object with the same name exists locally.

Parameters

userData	-	userData parameter passed in to the `RegisterChannelNotifySink` (page 14) method. May be NULL.
objName	-	The name of the object created by the peer.
reserved	-	Unused parameter.

Return Values

None.

Side Effects

None.

RPC Sinks

Table 16: VDPRPC ObjectNotifySink Function Members

v1	
Function	**Page**
`OnInvoke`	56
`OnObjectStateChanged`	56

Table 17: VDPRPC RequestCallback Function Members

v1	
Function	**Page**
`OnDone`	57
`OnAbort`	57

VDPRPC_ObjectNotifySink

Version 1 - OnInvoke

```
void (*OnInvoke)(void *userData, void *contextHandle, void *reserved);
```

Summary

This method is invoked when the peer on the other end of the channel called `Invoke` (page 30). The contextHandle will be used to retrieve the data given by the peer, using the `VDPService_ChannelContextInterface` (page 19). This same context should be altered to hold the return values, and the context will be returned to the caller when this method returns.

Parameters

userData	-	The userData parameter passed to the `CreateChannelObject` (page 27) method. May be NULL.
contextHandle	-	Handle for the context that will contain the data for the call, and to hold the return values.
reserved	-	Unused parameter.

Return Values

None.

Side Effects

None.

VDPRPC_ObjectNotifySink

Version 1 - OnObjectStateChanged

```
void (*OnObjectStateChanged)(void *userData, void *reserved);
```

Summary

Called when the state of the object this sink was registered with has changed.

Parameters

userData	-	The userData parameter passed to the `CreateChannelObject` (page 27) method. May be NULL.
reserved	-	Unused parameter.

Return Values

None.

Side Effects

None.

VDPRPC_RequestCallback

Version 1 - OnDone

```
void (*OnDone)(void *userData, uint32 contextId, void *contextHandle);
```

Summary

This method is invoked when the `Invoke` (page 30) call this sink was registered with has returned from the peer. The contextId will map to the id of the context that was passed to the `Invoke` call. This will not match the id of the context pointed to by the contextHandle. The contextHandle will hold all of the return codes and values given by the peer.

Parameters

userData	-	The userData parameter passed to the `Invoke` method.
contextId	-	Id of the context that was passed to the `Invoke` method.
contextHandle	-	Handle for the context that holds all of the return data from the peer.

Return Values

None.

Side Effects

None.

VDPRPC_RequestCallback

Version 1 - OnAbort

```
void (*OnAbort)(void *userData, uint32 contextId, Bool userCancelled, uint32 reason);
```

Summary

This method is called when the `Invoke` call this sink was registered with failed due to a View Session Enhancement error.

Parameters

userData	-	The userData parameter passed to the `Invoke` method.
contextId	-	Id of the context that was passed to the `Invoke` method.
userCancelled	-	FALSE.
reason	-	A `VDP_RPC_E_*` error code (see page 66).

Return Values

None.

Side Effects

None.

Overlay Sinks

Table 18: VDPOverlay Guest Sink Function Members

v1	
Function	**Page**
`OnOverlayReady`	59
`OnOverlayRejected`	59
`OnOverlayCreateError`	60
`OnUserMsg`	60

Table 19: VDPOverlay Client Sink Function Members

v1	
Function	**Page**
`OnWindowRegistered`	61
`OnWindowUnregistered`	61
`OnOverlayEnabled`	62
`OnOverlayDisabled`	62
`OnWindowPositionChanged`	63
`OnWindowSizeChanged`	63
`OnWindowObscured`	64
`OnWindowVisible`	64
`OnLayoutModeChanged`	65
`OnUserMsg`	65

VDPOverlayGuest_Sink

Version 1 - OnOverlayReady

```
void (*OnOverlayReady)(void *userData, VDPOverlay_WindowId windowId, uint32 response);
```

Summary

This event handler is called when the client-side overlay is ready to be displayed. It does not mean that the overlay is enabled or even that the client-side has loaded an image into the overlay, just that the overlay was properly created and is ready to display an image.

Parameters

userData	-	The userData parameter that was passed to the `Init` call (page 42).
windowId	-	The windowId this callback corresponds to.
response	-	Client-side plugin response.

Return Values

None.

Side Effects

None.

VDPOverlayGuest_Sink

Version 1 - OnOverlayRejected

```
void (*OnOverlayRejected)(void *userData, VDPOverlay_WindowId windowId, uint32 reason);
```

Summary

This event handler is called when the client-side overlay was not created because the client side plug-in choose to reject it. Note that the window that is associated with the overlay is automatically unregistered.

Parameters

userData	-	The userData parameter that was passed to the `Init` call (page 42).
windowId	-	The windowId this callback corresponds to.
reason	-	The client-side plugin reason given for rejecting the overlay.

Return Values

None.

Side Effects

None.

VDPOverlayGuest_Sink

Version 1 - OnOverlayCreateError

```
void (*OnOverlayCreateError)(void *userData, VDPOverlay_WindowId windowId, VDPOverlay_Error error);
```

Summary

This event handler is called when the client-side overlay was not created due to an error. Note that the window that is associated with the overlay is automatically unregistered.

Parameters

userData	-	The userData parameter that was passed to the `Init` call (page 42).
windowId	-	The windowId this callback corresponds to.
error	-	The error that was encountered.

Return Values

None.

Side Effects

None.

VDPOverlayGuest_Sink

Version 1 - OnUserMsg

```
void (*OnUserMsg)(void *userData, VDPOverlay_WindowId windowId, void *msg, uint32 msgLen);
```

Summary

This event handler is called in response to a call to `SendMsg` (page 51) from the client.

Parameters

userData	-	The userData parameter that was passed to the `Init` call (page 42).
windowId	-	The windowId this message is sent to, or `VDP_OVERLAY_WINDOW_ID_NONE` if the message wasn't sent to a particular window.
msg	-	The message data. Not valid after the call returns.
msgLen	-	Length of msg, in bytes.

Return Values

None.

Side Effects

None.

VDPOverlayClient_Sink

Version 1 - OnWindowRegistered

```
void (*OnWindowRegistered)(void *userData, VDPOverlayClient_ContextId contextId, VDPOverlay_WindowId windowId,
  VDPOverlay_UserArgs userArgs, Bool *reject, uint32 *response);
```

Summary

This event handler is called when the guest-side application registers a window using the `RegisterWindow` (page 43) method.

The request can be rejected by setting `reject` to TRUE. `response` can be used to return a reason to the guest. `response` can also be used to send a message to the guest in the non-reject case.

Note that the windowId should be cached as it is required to identify the overlay to the Overlay API.

Parameters

userData	-	userData parameter passed to the `Init` (page 49) method.
contextId	-	The context id returned from the `Init` call.
windowId	-	The windowId representing the new overlay.
userArgs	-	Value sent by the guest-side in the `RegisterWindow` call.
reject	-	Set to TRUE to deny the request to create an overlay.
response	-	Response sent back to the guest.

Return Values

None.

Side Effects

None.

VDPOverlayClient_Sink

Version 1 - OnWindowUnregistered

```
void (*OnWindowUnregistered)(void *userData, VDPOverlayClient_ContextId contextId, VDPOverlay_WindowId windowId,
    VDPOverlay_UserArgs userArgs);
```

Summary

This event handler is called when the guest-side unregisters a window using the `UnregisterWindow` (page 43) method. The window id is no longer valid and the overlay associated with the window id is destroyed.

Parameters

userData	-	userData parameter passed to the `Init` (page 49) method.
contextId	-	The context id returned from the `Init` call.
windowId	-	Window id for the window that was unregistered.
userArgs	-	Value sent by the guest-side application in the `UnregisterWindow` call.

Return Values

None.

Side Effects

None.

VDPOverlayClient_Sink

Version 1 - OnOverlayEnabled

```
void (*OnOverlayEnabled)(void *userData, VDPOverlayClient_ContextId contextId, VDPOverlay_WindowId windowId,
 VDPOverlay_UserArgs userArgs);
```

Summary

This event handler is called when the guest-side enables the overlay using the `EnableOverlay` (page 45) method. This causes the current image in the overlay to be displayed.

Parameters

userData	-	userData parameter passed to the `Init` (page 49) method.
contextId	-	The context id returned from the `Init` call.
windowId	-	Window id that corresponds to the enabled overlay.
userArgs	-	Value passed by the guest-side to the `EnableOverlay` call.

Return Values

None.

Side Effects

None.

VDPOverlayClient_Sink

Version 1 - OnOverlayDisabled

```
void (*OnOverlayDisabled)(void *userData, VDPOverlayClient_ContextId contextId, VDPOverlay_WindowId windowId,
  VDPOverlay_UserArgs userArgs);
```

Summary

This event handler is called when the guest-side disables the overlay using the `DisableOverlay` (page 45) method. This causes the current image in the overlay to be hidden. The overlay image data is maintained and will be re-displayed when the overlay is re-enabled.

Parameters

userData	-	userData parameter passed to the `Init` (page 49) method.
contextId	-	The context id returned from the `Init` call.
windowId	-	Window id that corresponds to the disabled overlay.
userArgs	-	Value passed by the guest-side to the `DisableOverlay` call.

Return Values

None.

Side Effects

None.

VDPOverlayClient_Sink

Version 1 - OnWindowPositionChanged

```
void (*OnWindowPositionChanged)(void *userData, VDPOverlayClient_ContextId contextId, VDPOverlay_WindowId windowI
   int32 x, int32 y);
```

Summary

This event handler is called when the guest-side window which the overlay is tracking changes position. The overlay will be drawn at the new location. This is for information only, no action is required by the plugin.

Parameters

userData	-	userData parameter passed to the `Init` (page 49) method.
contextId	-	The context id returned from the `Init` call.
windowId	-	Window id that corresponds to the repositioned overlay.
x	-	New X position with the display.
y	-	New Y position with the display.

Return Values

None.

Side Effects

None.

VDPOverlayClient_Sink

Version 1 - OnWindowSizeChanged

```
void (*OnWindowSizeChanged)(void *userData, VDPOverlayClient_ContextId contextId, VDPOverlay_WindowId windowId,
     int32 width, int32 height);
```

Summary

This event handler is called when the guest-side window which the overlay is tracking changes size. The old overlay image will be redrawn according to the layout mode of the overlay. This is for information only, no action is required by the plugin.

Parameters

userData	-	userData parameter passed to the `Init` (page 49) method.
contextId	-	The context id returned from the `Init` call.
windowId	-	Window id that corresponds to the resized overlay.
width	-	New width of the window.
height	-	New height of the window.

Return Values

None.

Side Effects

None.

VDPOverlayClient_Sink

Version 1 - OnWindowObscured

```
void (*OnWindowRegistered)(void *userData, VDPOverlayClient_ContextId contextId, VDPOverlay_WindowId windowId);
```

Summary

This event handler is called when the guest-side window which the overlay is tracking is completely obscured. The client-side can use this as a hint to scale back drawing to the overlay.

Parameters

userData	-	userData parameter passed to the `Init` (page 49) method.
contextId	-	The context id returned from the `Init` call.
windowId	-	Window id that corresponds to the obscured overlay.

Return Values

None.

Side Effects

None.

VDPOverlayClient_Sink

Version 1 - OnWindowVisible

```
void (*OnWindowVisible)(void *userData, VDPOverlayClient_ContextId contextId, VDPOverlay_WindowId windowId);
```

Summary

This event handler is called when the guest-side window which the overlay is tracking was obscured but now is at least partially visible.

Parameters

userData	-	userData parameter passed to the `Init` (page 49) method.
contextId	-	The context id returned from the `Init` call.
windowId	-	Window id that corresponds to the no longer obscured overlay.

Return Values

None.

Side Effects

None.

VDPOverlayClient_Sink

Version 1 - OnLayoutModeChanged

```
void (*OnLayoutModeChanged)(void *userData, VDPOverlayClient_ContextId contextId, VDPOverlay_WindowId windowId,
     VDPOverlay_LayoutMode layoutMode);
```

Summary

This event handler is called when the layout mode for the overlay is changed. This is for information only, no action is required by the plugin.

Parameters

userData	-	userData parameter passed to the `Init` (page 49) method.
contextId	-	The context id returned from the `Init` call.
windowId	-	Window id that corresponds to the referenced overlay.
layoutMode	-	The new layout mode.

Return Values

None.

Side Effects

None.

VDPOverlayClient_Sink

Version 1 - OnUserMsg

```
void (*OnUserMsg)(void *userData, VDPOverlayClient_ContextId contextId, VDPOverlay_WindowId windowId,
  void *msg, uint32 msgLen);
```

Summary

This event handler is called when the guest-side application has called the `SendMsg` (page 48) method.

Parameters

userData	-	userData parameter passed to the `Init` (page 49) method.
contextId	-	The context id returned from the `Init` call.
windowId	-	Window id that the message was sent to, or `VDP_OVERLAY_WINDOW_ID_NONE` if the message wasn't sent to a particular window.
msg	-	Message data. Not valid once the handler returns.
msgLen	-	Length, in bytes, of msg.

Return Values

None.

Side Effects

None.

View Session Enhancement Error Codes

While interacting with the View Session Enhancement APIs, you may encounter error codes. This section covers those codes, where you may encounter them, and what the error indicates.

OnAbort Reason Codes

When using the `VDPRPC_ChannelObjectInterface.v1.OnInvoke()` method, if the call failed due to a View Session Enhancement error, then the supplied OnAbort method will be called. The last parameter to this method will be one of the following error codes:

Table 20: OnAbort Reason Error Codes

Code	Description
`VDP_RPC_E_APARTMENT_UNINITIALIZED`	The `OnInvoke` call was made on a thread that has not been initialized to be used with the View Session Enhancement APIs.
`VDP_RPC_E_APARTMENT_THREAD`	The `OnInvoke` call involved an object that was not created on the calling thread, and the object was not configured to allow `OnInvoke` calls on different threads.
`VDP_RPC_E_OBJECT_NOT_CONNECTED`	The object handle used for the `OnInvoke` points to an object that is not connected. This indicates that the peer object on the remote side has not yet been created.
`VDP_RPC_E_PARAMETER`	One of the required parameters passed to the `OnInvoke` call was invalid.
`VDP_RPC_E_MEMORY`	The system failed to allocate the required memory to send the request.

VDP Overlay Error Codes

Many of the methods defined in the `vdpOverlay.h` header file return a VDPOverlay_Error value. The potential values for this type are listed in Table 21.

Table 21: VDPOverlay_Error Codes

Code	**Description**
`VDP_OVERLAY_ERROR_SUCCESS`	No error occurred. The call succeeded.
`VDP_OVERLAY_ERROR_NOT_INITIALIZED`	The call failed because the VDP Overlay components were not properly loaded in the VMware View environment.
`VDP_OVERLAY_ERROR_ALREADY_INITIALIZED`	This error can only be returned from the `VDPOverlayGuest_Interface.v1.Init()` call, and indicates that the guest Overlay system has already been initialized.
`VDP_OVERLAY_ERROR_INVALID_PARAMETER`	One of the required parameters passed to the method call was invalid.
`VDP_OVERLAY_ERROR_ALLOCATION_ERROR`	The system failed to allocate the required memory or system resource to handle the method call.
`VDP_OVERLAY_ERROR_NO_MORE_OVERLAYS`	This error results from a failed attempt to register a window. This may be due to a client-side error, or the window that the register attempt was made on has already been registered with a different Plugin. It is an error that maybe received in the `VDPOverlayGuest_Sink.v1.OnOverlayCreateError` callback.
`VDP_OVERLAY_ERROR_OVERLAY_REJECTED`	This error results from a failed attempt to register a window. It indicates that the client-side did not accept the plugin. This error will be in the reason field of the `VDPOverlayGuest_Sink.v1.OnOverlayRejected()` callback.
`VDP_OVERLAY_ERROR_OVERLAY_NOT_READY`	Error returned when either the `VDPOverlayGuest_Interface.v1.EnableOverlay` and `VDPOverlayGuest_Interface.v1.DisableOverlay` and indicates that the registered window that was indicated is not ready (i.e., the `VDPOverlayGuest_Sink.v1.OnOverlayReady()` callback has not yet been received).
`VDP_OVERLAY_ERROR_WINDOW_NOT_REGISTERED`	The passed in window id to the overlay method has not yet been registered. This can be returned from most Overlay methods.
`VDP_OVERLAY_ERROR_WINDOW_ALREADY_REGISTERED`	The window that has been attempted to register has already been registered. This error can be returned from the `VDPOverlayGuest_Interface.v1.RegisterWindow()` method.

Sample Code

In the View Session Enhancement SDK deliverable, you will find a directory `samples`. This directory contains sample code that shows examples of everything listed above.

The simulator sample directory contains code that exercises all of the RPC APIs to send basic messages over the VMware View connection. Instructions for compiling and installing the example code can be found in the readme files under that directory.