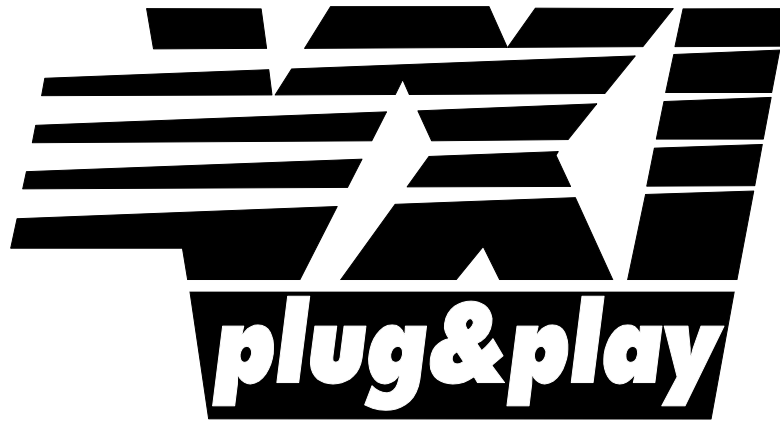


Systems Alliance

VPP-4.3.5: VISA Shared Components

June 7, 2016

Revision 5.6



Systems Alliance

VPP-4.3.5 Revision History

This section is an overview of the VPP-4.3.5 specification revision history.

Revision 1.0, October 16, 2008

First draft of the VPP-4.3.5 specification, based on Section 6 of the VPP-4.3.4 specification. This specification covers all aspects of the VISA Shared Components binaries and installation.

Revision 5.0, June 9, 2010

Added USBTMC components to shared component specification.

Revision 5.1, March 6, 2013

Added Windows 7 and Windows 8 to the list of supported operating systems.

Revision 5.4, May 14, 2014

Made changes to specify the VISA .NET shared components. Added observations to clarify the mechanism used by the VISA Shared Components installers to register the Primary Interop Assembly. Changed the version to 5.4 to ensure that all VISA specifications being voted on at the same time have the same version.

Revision 5.5, February 11, 2015

Made changes to clarify the behavior of the viParseRsrc/viParseRsrcEx functions. Modified the uninstallation behavior of the VISA Shared Components and VISA.NET Shared Installer. Clarified the necessary prerequisite software for installing VISA.NET Shared Components.

Revision 5.5, August 6, 2015

Removed Windows 2000 and added Windows 10 to the list of supported operating systems.

Revision 5.6, June 7, 2016

Removed Windows XP and Windows Vista from the list of supported operating systems.

NOTICE

VPP-4.3.4: *VISA Shared Components* is authored by the IVI Foundation member companies. For a vendor membership roster list, please visit the IVI Foundation web site at www.ivifoundation.org, or contact the IVI Foundation at 2515 Camino del Rio South, Suite 340, San Diego, CA 92108.

The IVI Foundation wants to receive your comments on this specification. You can contact the Foundation through email at ivilistserver@ivifoundation.org, through the web site at www.ivifoundation.org, or you can write to the IVI Foundation, 2515 Camino del Rio South, Suite 340, San Diego, CA 92108.

Warranty

The IVI Foundation and its member companies make no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The IVI Foundation and its member companies shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Trademarks

Product and company names listed are trademarks or trade names of their respective companies.

No investigation has been made of common-law trademark rights in any work.

Table of Contents

Section 1:	Introduction to the VXIplug&play Systems Alliance and the IVI Foundation	1
Section 2:	Overview of VISA Shared Components Specification	1
2.1.	Objectives of This Specification	2
2.2.	Audience for This Specification	3
2.3.	Scope and Organization of This Specification	4
2.4.	Application of This Specification	5
2.5.	References	6
2.6.	Definition of Terms and Acronyms	7
2.7.	Conventions	8
Section 3:	VISA Shared Components	8
3.1.	VXIplug&play Infrastructure	2
3.2.	VISA Plug-In Architecture Components	3
3.2.1.	VISA Header Files	3
3.2.2.	The VISA Router	3
3.2.3.	The Conflict Resolution Manager	6
3.2.4.	VISA Utilities	37
3.3.	VISA COM Components	38
3.4.	VISA .NET Components	39
3.4.1.	Conflict Resolution Manager .NET API	39
3.5.	VISA Shared USBTMC Device Driver	46
Section 4:	VISA Shared Components Installers	47
4.1.	Installing VISA Shared Components On 32-Bit Operating Systems	48
4.2.	Installing VISA Shared Components On 64-Bit OS's	54
4.3.	Installing VISA .NET Shared Components	61
Appendix A:	Implementation Files	63
A.1	Contents of the visaRouter.h File	63
A.2	Contents of the ConflictMgr.h File	63
A.3	Contents of the ConflictMgr.def File	68
A.4	Contents of the visaUtilities.h File	69
A.5	Contents of the visaUtilities.def File	69

Section 1: Introduction to the VXIplug&play Systems Alliance and the IVI Foundation

The VXIplug&play Systems Alliance was founded by members who shared a common commitment to end-user success with open, multivendor VXI systems. The alliance accomplished major improvements in ease of use by endorsing and implementing common standards and practices in both hardware and software, beyond the scope of the VXIbus specifications. The alliance used both formal and de facto standards to define complete system frameworks. These standard frameworks gave end-users "plug & play" interoperability at both the hardware and system software level.

The IVI Foundation is an organization whose members share a common commitment to test system developer success through open, powerful, instrument control technology. The IVI Foundation's primary purpose is to develop and promote specifications for programming test instruments that simplify interchangeability, provide better performance, and reduce the cost of program development and maintenance.

In 2002, the VXIplug&play Systems Alliance voted to become part of the IVI Foundation. In 2003, the VXIplug&play Systems Alliance formally merged into the IVI Foundation. The IVI Foundation has assumed control of the VXIplug&play specifications, and all ongoing work will be accomplished as part of the IVI Foundation.

All references to VXIplug&play Systems Alliance within this document, except contact information, are there to preserve consistency with the original set of specifications.

Section 2: Overview of VISA Shared Components Specification

This VISA specification is a document authored by the *VXIplug&play* Systems Alliance. The technical work embodied in this document and the writing of this document was performed by the VISA Technical Working Group.

This section provides a complete overview of the *VISA Shared Components* specification, and gives readers general information that may be required to understand how to read, interpret, and implement individual aspects of this specification. This section is organized as follows:

- Objectives of this specification
- Audience for this specification
- Scope and organization of this specification
- Application of this specification
- References
- Definitions of terms and acronyms
- Conventions

2.1. Objectives of This Specification

The *VISA Shared Components* specification describes the deployment of a common set of VISA components for developing multi-vendor software programs, including instrument drivers. There are two sets of components which are installed separately.

- The VISA Shared Components installer includes the VISA COM components and the VISA Plug-In Architecture components provided by the IVI Foundation.
- The VISA .NET Shared Components installer includes the VISA .NET components provided by the IVI Foundation. The VISA .NET Shared Components installer has a dependency on the VISA Shared Components installer.

These installers are available from the IVI Foundation to guarantee a consistent, reliable installation.

2.2. Audience for This Specification

The primary audience is I/O vendors who wish to implement and install VISA-compliant I/O software.

Secondary audiences include instrument driver developers—whether an instrument vendor, system integrator, or end user—who wishes to implement instrument driver software that is compliant with the *VXIplug&play* standards, and instrumentation end users and application programmers who wish to implement applications that utilize instrument drivers compliant with this specification.

2.3. Scope and Organization of This Specification

This specification is organized in sections, with each section discussing a particular aspect of the VISA model.

Section 1, *Introduction to the VXIplug&play Systems Alliance and the IVI Foundation*, explains the VXIplug&play Systems Alliance and its relation to the IVI Foundation.

Section 2, *Overview of VISA I/O Components and Installation Specification*, provides an overview of this specification, including the objectives, scope and organization, application, references, definition of terms and acronyms, and conventions.

Section 3, *VISA Shared Components*, provides an overview of the files and required infrastructure that make up the VISA Shared Components.

Section 4, *VISA Shared Components Installers*, discusses the implementation of the VISA Shared Components installers.

2.4. Application of This Specification

This specification describes standards to be used by developers of VISA and VISA COM I/O library software. After June 1, 2007 every release of a VISA I/O library must use installers provided by the IVI Foundation to install shared VISA components.

It is also useful as a reference for developers and end users of VISA I/O library software, as well as *VXIplug&play* and IVI instrument drivers.

2.5. References

The following documents contain information that you may find helpful as you read this document:

- VPP-1, *VXIplug&play Charter Document*
- VPP-2, *System Frameworks Specification*
- VPP-3.1, *Instrument Drivers Architecture and Design Specification*
- VPP-3.2, *Instrument Functional Body Specification*
- VPP-3.3, *Instrument Driver Interactive Developer Interface Specification*
- VPP-3.4, *Instrument Driver Programmatic Developer Interface Specification*
- VPP-4.1, *VISA-I Main Specification*
- VPP-4.2, *The VISA Transition Library*
- VPP-4.3, *The VISA Library*
- VPP-4.3.2, *VISA Implementation Specification for Textual Languages*
- VPP-4.3.3, *VISA Implementation Specification for the G Language*
- VPP-4.3.4, *VISA Implementation Specification for COM*
- VPP-4.3.6, *VISA Implementation Specification for .NET*
- VPP-6, *Installation and Packaging Specification*
- VPP-7, *Soft Front Panel Specification*
- VPP-9, *Instrument Vendor Abbreviations*
- VXI-1, *VXIbus System Specification*, Revision 1.4, VXIbus Consortium
- VXI-11, *TCP/IP Instrument Protocol*, VXIbus Consortium

2.6. Definition of Terms and Acronyms

The following are some commonly used terms within this document

API	Application Programmers Interface. The direct interface that an end user sees when creating an application. The VISA API consists of the sum of all of the operations, attributes, and events of each of the VISA Resource Classes. The VISA COM I/O API consists of a collection of COM interfaces. The VISA COM .NET API consists of a collection of .NET interfaces and classes.
COM	Component Object Model, a Microsoft technology for reusable software components.
Component	A DLL or EXE that implements executable ANSI C, COM, or .NET code. Executable components may be accompanied by other supporting files, such as help files.
Instrument	A device that accepts some form of stimulus to perform a designated task, test, or measurement function. Two common forms of stimuli are message passing and register reads and writes. Other forms include triggering or varying forms of asynchronous control.
Instrument Driver	Library of functions for controlling a specific instrument.
VISA	Virtual Instrument Software Architecture. This is the general name given to this document and its associated architecture. The architecture consists of two main VISA components: the VISA Resource Manager and the VISA Instrument Control Resources.
VISA COM	VISA for COM. VISA COM is an architecture that provides VISA functionality via a COM API.
VISA Shared Components	A common set of VISA components for which there must be only one implementations. That implementation is provided by the IVI Foundation, and is described in this document.
WOW64	Windows On Windows 64, a Microsoft technology for allowing the execution of 32-bit native code programs on 64-bit operating systems.

2.7. Conventions

Throughout this specification you will see the following headings on certain paragraphs. These headings instill special meaning on these paragraphs.

Rules must be followed to ensure compatibility with the System Framework. A rule is characterized by the use of the words **SHALL** and **SHALL NOT** in bold upper case characters. These words are not used in this manner for any purpose other than stating rules.

Recommendations consist of advice to implementers that will affect the usability of the final device. They are included in this standard to draw attention to particular characteristics that the authors believe to be important to end user success.

Permissions are included to *authorize* specific implementations or uses of system components. A permission is characterized by the use of the word **MAY** in bold upper case characters. These permissions are granted to ensure specific System Framework components are well defined and can be tested for compatibility and interoperability.

Observations spell out implications of rules and bring attention to things that might otherwise be overlooked. They also give the rationale behind certain rules, so that the reader understands why the rule must be followed.

A Note on the text of the specification: Any text that appears without heading should be considered as description of the standard and how the architecture was intended to operate. The purpose of this text is to give the reader a deeper understanding of the intentions of the specification including the underlying model and specific required features. As such, the implementer of this standard should take great care to ensure that a particular implementation does not conflict with the text of the standard.

Section 3: VISA Shared Components

The VISA shared components are a common set of VISA components for developing multivendor software programs, including VISA I/O libraries and a variety of instrument drivers.

The components are “shared” because multiple VISA, VISA COM and VISA .NET vendor-specific implementations must share a single copy of each component. Because there may only be a single copy of the component per PC, and the behavior of each component is precisely described, the IVI Foundation supplies a standard implementation of each of them; in fact, the IVI Foundation implementation of each shared component *must* be used wherever the component is called for.

The VISA Shared Components installer includes the *VXIplug&play* infrastructure, VISA COM components, and VISA Plug-In Architecture components. The IVI Shared Components are available from the IVI Foundation in the form of standard installers that guarantee a consistent, reliable installation.

The *VXIplug&play* infrastructure includes the framework directory structure, registry entries, and environment variables. They are described in *VPP-2, System Frameworks Specification*.

The VISA Plug-In Architecture components include shared files that allow multiple vendor-specific VISA libraries to be installed on a single PC. In the past, the standard did not directly support this, as each vendor-specific VISA library had to install a file named *visa32.dll* in the system directory. To avoid potential backwards compatibility issues with older versions of VISA that follow this naming standard, the VISA Plug-In Architecture is available only on 64-bit PCs.

The VISA COM components include shared VISA COM functionality, including the VISA COM type library, the associated PIA, the Global Resource Manager, and Basic Formatted IO. These components are available for both 32-bit and 64-bit PCs. They are described in *VPP-4.3.4, VISA Implementation Specification for COM*.

The VISA .NET Shared Components installer includes the standard VISA .NET API and shared VISA .NET functionality such as the Global Resource Manager, the Formatted IO and the Conflict Manager. This set of shared functionality is provided in the VISA .NET assembly which is available for both 32-bit and 64-bit PCs. The functionality provided by the assembly is described in *VPP-4.3.6, VISA Implementation Specification for .NET*.

3.1. *VXIplug&play* Infrastructure

The VXIplug&play infrastructure includes the directories, HKLM\SOFTWARE registry keys and values, and environment variables for each installed *VXIplug&play* framework.

The WinNT framework is installed on 32-bit Windows operating systems.

Both the WinNT and Win64 frameworks are installed on 64-bit Windows operation systems. On 64-bit operations systems, the WinNT framework is installed to the appropriate Windows On Windows 64 (WOW64) directories and registry keys.

3.2. VISA Plug-In Architecture Components

3.2.1. VISA Header Files

VISA header files are now included in the VISA shared component installers, because they are needed for VISA development.

The VISA header files are:

- `visa.h`
- `visatype.h`

See *VPP-4.3.2: VISA Implementation Specification for Textual Languages* for a description of these files. The content of each file is listed as an appendix to VPP-4.3.2.

Vendor-specific WIN64 framework VISA installers may not overwrite `visa.h` or `visatype.h`. Vendor-specific WINNT framework VISA installers may not overwrite `visatype.h`, but may overwrite `visa.h` to maintain backwards compatibility with previous vendor-specific versions of `visa.h`.

3.2.2. The VISA Router

The VISA Router is supported only on Windows 7 (64-bit), Windows 8 (64-bit), and Windows 10 (64-bit).

The VISA Router component includes the following files:

- `visa64.dll`
- `visa64.lib`
- `visa64.def` (documentation only, not installed)
- `visaRouter.h`

The VISA Router implements entry points defined by the VISA API, but only so that it can call the corresponding entry points in vendor-specific implementations of VISA. VISA users call the VISA API through the VISA Router. The VISA Router routes calls to the appropriate vendor-specific VISA, and also handles callbacks from the vendor-specific VISA to the calling program. The VISA object handles used by the Visa Router are unique and different from the VISA object handles returned from the underlying vendor-specific VISA libraries. The VISA Router takes care of mapping its object handles to the handles used by each of the underlying VISA libraries.

For improved performance, when only a single vendor-specific VISA is installed, the VISA Router acts as a simple pass-through. The object handles returned to the user are those returned from the installed vendor-specific VISA.

Once a session is opened in the VISA Router, most of the VISA entry points simply map the VISA session handle passed into the call to the handle of the underlying vendor-specific VISA, call the underlying VISA, and return the results. There are a few entry points, however, where the process is more complicated because the router needs either to call more than one underlying VISA or determine which underlying VISA to call. The behavior of these entry points is documented in the following sections. In addition, `visaRouter.h` includes additional items needed by the Visa Router, also documented below.

In the entry points described below where calls are made to more than one of underlying VISA libraries, the order in which these calls are made is as follows:

1. The “preferred” VISA, if one is defined.
2. Each installed VISA in lexical order of the VISA GUID.

3.2.2.1. viOpenDefaultRM

The first viOpenDefaultRM call in a process gets a list of the vendor-specific VISA libraries that are installed and enabled. It loads each of these VISA libraries and returns a session handle to the user. Subsequent calls to viOpenDefaultRM use the list of open VISA libraries generated in the first call, but they return a new unique session handle to the user. When the last defaultRM session is closed, all of the vendor-specific VISA libraries are unloaded if the value of the VI_ATTR_UNLOAD_PLUGINS_IF_LAST_RM attribute is set to VI_TRUE.

3.2.2.2. viOpen

The viOpen call finds a vendor-specific VISA that can successfully parse and open the resource string passed in. Because it is possible that more than one vendor-specific VISA can open a resource, this function checks the enabled vendor-specific VISA libraries in the following order and returns with the first one that succeeds:

1. The VISA that the user chose to handle devices on the interface the resource string specifies.
2. The VISA that was last used to successfully open this resource.
3. The “preferred” VISA, if one was specified.
4. Each VISA in the list of installed VISA libraries that has not already been tried.

If none of the VISA libraries can parse the resource string, a VI_ERROR_RSRC_NFOUND error is returned. If at least one VISA can parse it, but none can open it, the error code from the first VISA to parse it is returned.

3.2.2.3. viFindRsrc/viFindNext

The following algorithm generates the list of resources returned:

1. Create an empty master list of resources to be returned.
2. Call viFindRsrc/viFindNext on each underlying VISA in turn, starting with the preferred VISA if there is one.
3. For each resource found:
 - a. Remember the viFindRsrc/viFindNext name.
 - b. Call viParseRsrcEx and remember the canonical name returned.
 - c. Call the conflict manager and remember whether the resource is on a “chosen” interface. (There is no distinction between user chosen and resource manager chosen.)
 - d. Compare the canonical name with the canonical names already in the master list.
 - e. If the canonical name does not exist in the master list, add the resource to the master list.
 - f. If the current resource is on a chosen interface, replace the matching element in the master list with this one.
4. Return the names returned by the underlying viFindRsrc/viFindNext calls to the user.

3.2.2.4. viParseRsrc/viParseRsrcEx

The following algorithm is used to parse a resource:

1. Call viParseRsrc/viParseRsrcEx on each underlying VISA starting with the preferred VISA if there is one. Continue until a VISA implementation returns success or there are no more VISA implementations left.

2. If there was a successful parse in step 1, remember the parse result. If none of the VISA libraries succeed, return the status code returned by the first VISA called.
3. Use the parse result from step 2 to query the conflict manager to find if there is a chosen VISA for this resource. (There is no distinction between user chosen and resource manager chosen.)
4. If there is a chosen VISA for this resource and it hasn't been already called in step 1, call `viParseRsrc/viParseRsrcEx` on that VISA.
5. If the call succeeds, return the parse result obtained from the chosen VISA. Otherwise, return the parse result remembered in step 2.

3.2.2.5. `viGetAttribute`

The behavior of `viGetAttribute` depends on the type of object it is being called on.

For objects returned from `viOpen` or event objects, if the attribute being requested is a multivendor VISA attribute defined in `visaRouter.h` (`VI_ATTR_UNDERLYING_VISA_SESSION`, `VI_ATTR_MULTI_MANF_NAME`, `VI_ATTR_MULTI_SPEC_VERSION`, `VI_ATTR_MULTI_MANF_ID`, or `VI_ATTR_MULTI_IMPL_VERSION`), the appropriate result and/or status code is returned. For any other attribute, the `viGetAttribute` of the underlying VISA is called and the result is returned.

For objects returned from `viOpenDefaultRM` or `viFindRsrc`, if the attribute being requested is defined in `visaRouter.h` as above, the appropriate result and/or status code is returned. This algorithm also applies for the attribute `VI_ATTR_UNLOAD_PLUGINS_IF_LAST_RM` for sessions returned by `viOpenDefaultRM`. Otherwise, `viGetAttribute` is called on each underlying VISA, and the results are returned from the first VISA to succeed. If none of the VISA calls succeed, `VI_ERROR_NSUP_ATTR` is returned.

3.2.2.6. `viSetAttribute`

The behavior of `viSetAttribute` depends on the type of object it is being called on.

For objects returned from `viOpen` or event objects, if the attribute being set is a multivendor VISA attribute defined in `visaRouter.h` (`VI_ATTR_UNDERLYING_VISA_SESSION`, `VI_ATTR_MULTI_MANF_NAME`, `VI_ATTR_MULTI_SPEC_VERSION`, `VI_ATTR_MULTI_MANF_ID`, or `VI_ATTR_MULTI_IMPL_VERSION`), the appropriate result and/or status code is returned. For any other attribute, the `viSetAttribute` of the underlying VISA is called and the result is returned.

For objects returned from `viOpenDefaultRM` or `viFindRsrc`, if the attribute being set is defined in `visaRouter.h` as above, the appropriate result and/or status code is returned. Otherwise, the attribute is set on each underlying VISA. If one or more of the underlying VISA libraries returns a status value greater or equal to `VI_SUCCESS`, the status returned to the user is the status returned by the first underlying VISA that returned a successful status code. If none of the underlying VISA libraries return a successful status code, the unsuccessful status code of the first underlying VISA called is returned.

For `viSetAttribute`, the `VI_ATTR_UNLOAD_PLUGINS_IF_LAST_RM` attribute (which applies only to `viOpenDefaultRM`) is handled differently. The state of the attribute is saved in the VISA Router, and each of the underlying VISA libraries is called with this attribute as well, and the status returned from the underlying VISA libraries is ignored. The status returned to the user will be `VI_SUCCESS`.

3.2.2.7. visaRouter.h Additions

visaRouter.h defines the following router-specific attributes:

Symbolic Name	Access Privilege		Data Type	Default Value
VI_ATTR_UNDERLYING_VISA_SESSION	RO	Local	ViSession	N/A
VI_ATTR_MULTI_SPEC_VERSION	RO	Global	ViVersion	N/A
VI_ATTR_MULTI_MANF_NAME	RO	Global	ViString	IVI Foundation
VI_ATTR_MULTI_MANF_ID	RO	Global	ViUInt16	0x3FFF
VI_ATTR_MULTI_IMPL_VERSION	RO	Global	ViVersion	N/A
VI_ATTR_UNLOAD_PLUGINS_IF_LAST_RM	RW	Global	ViBoolean	VI_FALSE

3.2.3. The Conflict Resolution Manager

The Conflict Resolution Manager is supported on both the WINNT and WIN64 frameworks.

In cases where more than one vendor-specific VISA library can connect to an interface, the conflict resolution manager provides information regarding available vendor-specific VISA libraries and user preferences.

RULE 3.2.1

VISA vendors **SHALL** use the IVI Foundation implementation of the conflict resolution manager and **SHALL NOT** create or use a vendor-specific version.

OBSERVATION 3.2.1

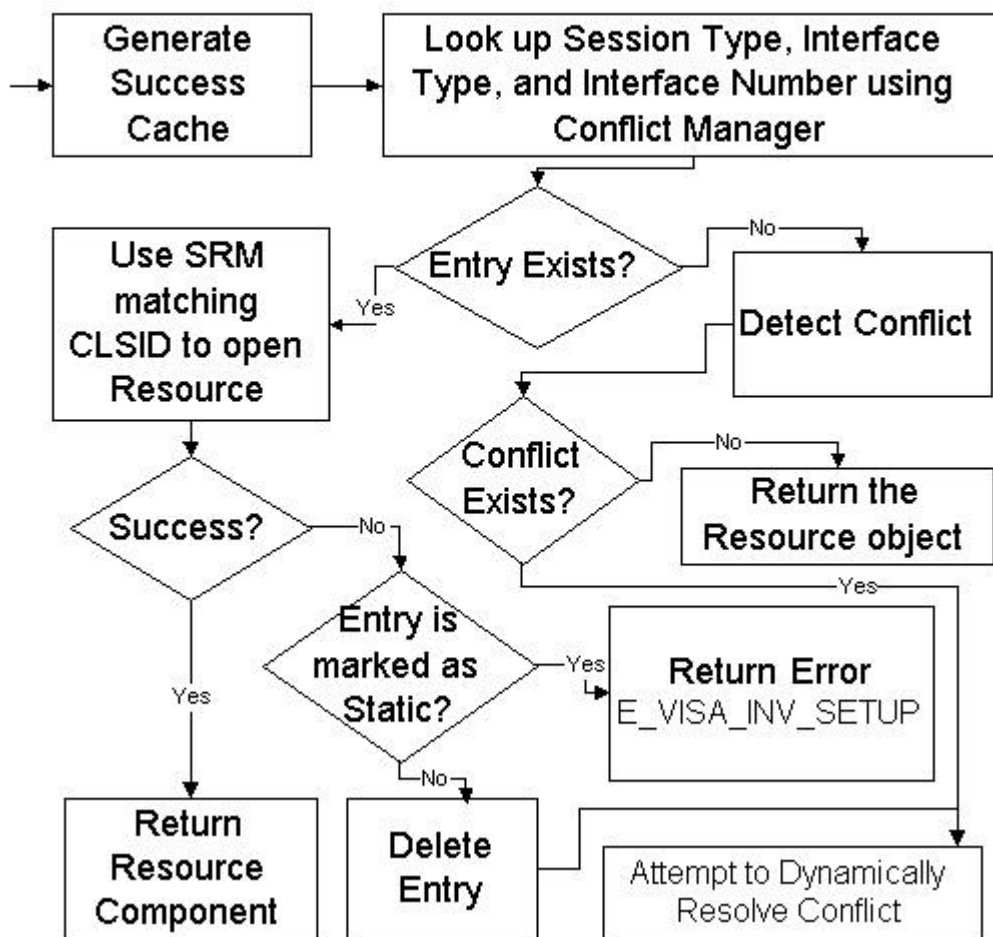
VISA vendors may want to include various capabilities for manipulating the VISA conflict resolution process using the Conflict Resolution Manager. The purpose of describing the API details of the Conflict Resolution Manager in this document is to document it so that vendors can use it correctly.

3.2.3.2. How Conflict Resolution Works

Conflict Resolution

When multiple VISA libraries are present on a system, some method of determining which VISA library shall be used is required. The Conflict Resolution Manager follows the algorithm outlined below to provide information regarding available vendor-specific VISA libraries and user preferences. The algorithm takes into account previous results in combination with user-defined preferences. The algorithms in this section use the information the conflict resolution manager provides, but the implementation of these algorithms is in the VISA Router, VISA .NET Global Resource Manager and the VISA-COM Global Resource Manager. (The diagrams reflect COM details at points, but the general logic applies to all three.)

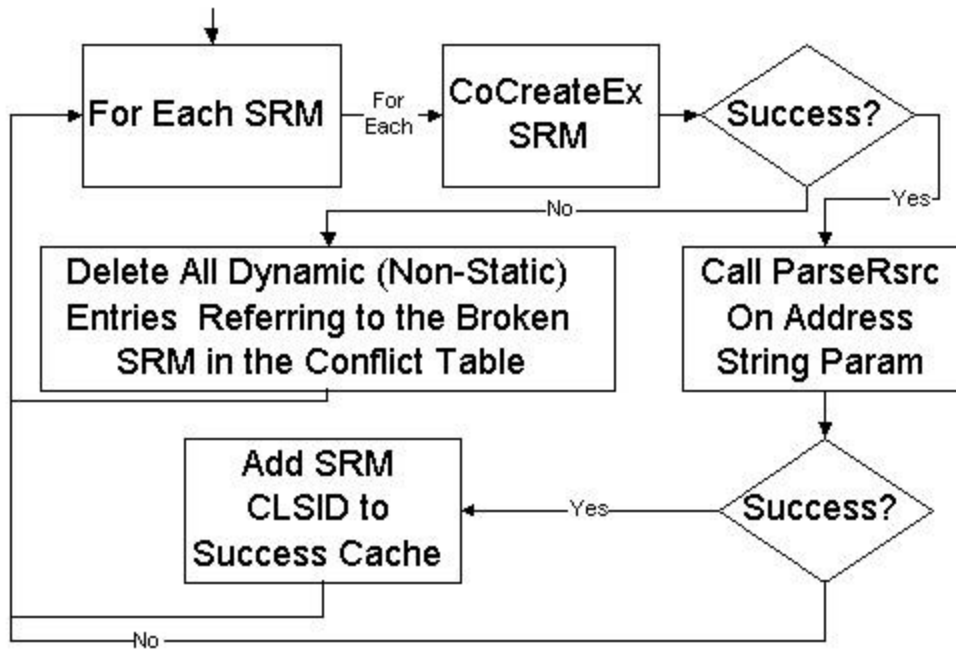
Conflict Resolution Sub



Success Cache Generation

A portion of the conflict resolution algorithm requires a listing of which VISA libraries previously opened the desired resource. This listing is known to the VISA Conflict Resolution Manager as the success cache, and it is generated using the algorithm below.

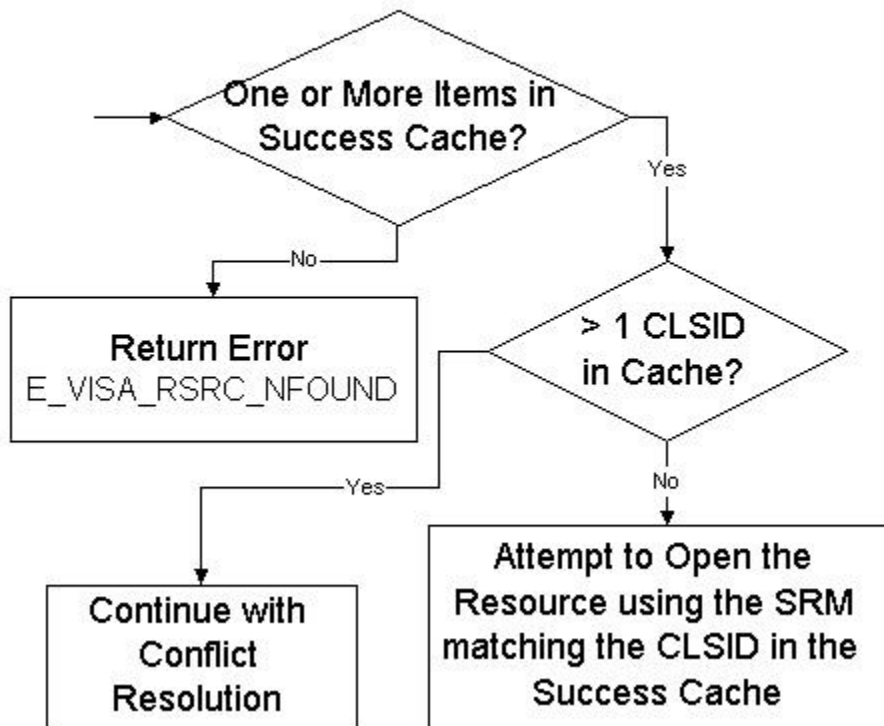
Success Cache Generation



Conflict Detection

The conflict detection subroutine outlined below is used when determining which VISA library to use for a specified resource. The goal of this subroutine is to determine whether any libraries can open the given resource and ensuring that the VISA library used previously to open that resource is used.

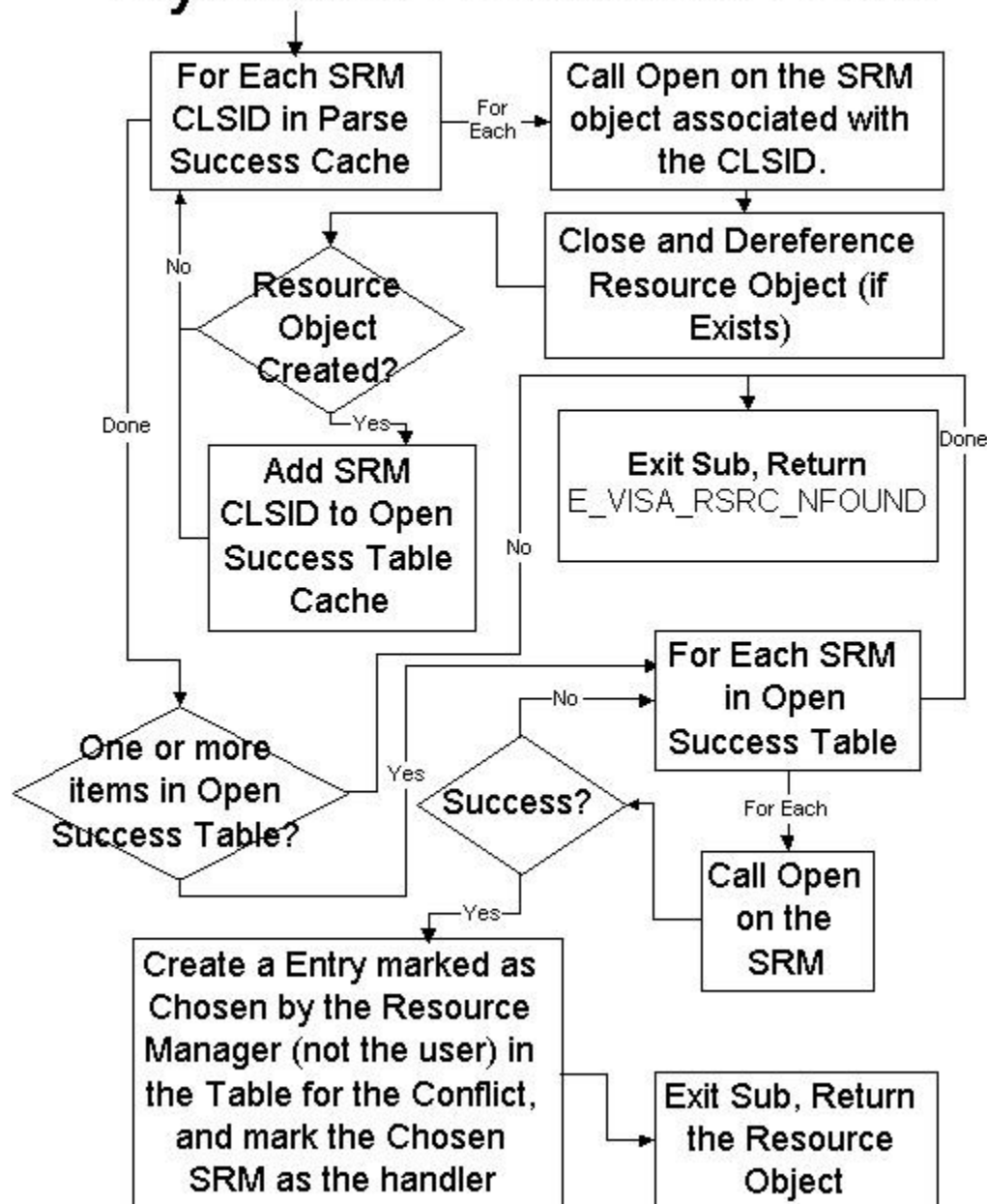
Conflict Detection Sub



Dynamic Resolution

The Dynamic Resolution Subroutine is used when the resource has not been previously opened or if the VISA library previously used has failed. This subroutine finds a VISA library that can open the resource if one exists, and note that entry in the settings cache.

Dynamic Resolution Sub



3.2.3.3. Supported Enumerations for the Conflict Manager functions

RULE 3.2.2

Each function described in section 3.2.3 that accepts a `conflictHandlerType` parameter **SHALL** accept one of the following values:

```
VISACM_HANDLER_NOT_CHOSEN (0)
VISACM_HANDLER_CHOSEN_BY_RSRC_MGR (1)
VISACM_HANDLER_CHOSEN_BY_USER (2)
```

RULE 3.2.3

Each function described in section 3.2.3 that accepts a `flushBehavior` parameter **SHALL** accept one of the following values:

```
VISACM_FLUSH_OVERWRITE_ALWAYS (0)
VISACM_FLUSH_WRITE_IF_UNCHANGED (1)
VISACM_FLUSH_WRITE_OR_RELOAD (2)
```

RULE 3.2.4

Each function described in section 3.2.3 that accepts an `apiType` parameter **SHALL** accept one of the following values:

```
VISACM_API_C_AND_COM (0)
VISACM_API_DOTNET (1)
```

RULE 3.2.5

For each function in this section that takes an `apiType` parameter, there **SHALL** be a corresponding legacy function without that parameter and without '2' in the function name. Each legacy function **SHALL** call the corresponding new function with the `apiType` parameter set to `VISACM_API_C_AND_COM`.

3.2.3.4. VISACM_ClearEntireTable()

Purpose

Clear all cached settings stored by the Conflict Manager.

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The Conflict Manager's settings have been successfully cleared.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that <code>VISACM_Initialize()</code> has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_ALLOC	Insufficient system resources to make the change to the Conflict Manager settings.

Description

This function clears all settings currently cached by the Conflict Manager. After this function is called, there will be no preferred VISA, no disabled VISA libraries, and no record of any instruments having been accessed by a VISA library.

3.2.3.5. VISACM_ClearResourceHandlersFromTable2(apiType)**Purpose**

Deletes all records corresponding to the given API type of resources that were previously attempted to be opened by a VISA library.

Parameters

Name	In/Out	Type	Description
apiType	IN	ViInt16	An enum specifying the type of the API corresponding to the records to be deleted.

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The records of all previous resource opening attempts for the given API type were deleted by the Conflict Manager.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_INV_PARAMETER	An invalid apiType parameter value was passed in by the user.

Description

This function clears all records of any attempted opens for the given API type by any VISA library on resources. Following this call, all resources will be treated as never opened for the given API type upon the next open attempt.

3.2.3.6. VISACM_Close()

Purpose

Signals to the Conflict Manager that the caller is finished with Conflict Manager actions.

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The Conflict Manager handled the close operation successfully.

Error Codes	Description
VI_ERROR_CLOSING_FAILED	The Conflict Manager failed to successfully close. This can occur if the Conflict Manager has not been correctly initialized, or if writing settings to file failed.

Description

This function should be called when the client process is finished using the Conflict Manager. Calling this function ensures that new settings are stored and that cleanup occurs properly.

3.2.3.7. VISACM_CreateHandler2(apiType, interfaceType, interfaceNumber, sessionType, guid_SRM, conflictHandlerType, comments)

Purpose

Creates a record of an open attempt by a VISA library on a resource for the given API type.

Parameters

Name	In/Out	Type	Description
apiType	IN	ViInt16	An enum specifying the type of the API supported by the calling VISA library.
interfaceType	IN	ViUInt16	The target resource's interface type. This is identical to the VI_ATTR_INTF_TYPE specified by the VISA specification.
interfaceNumber	IN	ViUInt16	The target resource's interface number. This is identical to the VI_ATTR_INTF_NUM specified by the VISA specification.
sessionType	IN	ViConstString	The type of resource being opened. For example, GPIB has both INSTR and BACKPLANE resources. This parameter would hold that type.
guid_SRM	IN	ViConstString	The GUID of the calling VISA library. The GUID should be passed in the format XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX.
conflictHandlerType	IN	ViInt16	An enumeration signaling whether the setting is a user-specified setting, a Conflict Manager defined success, or a failure.
comments	IN	ViConstString	Any extra comments regarding this setting. Passing NULL signals that no comments should be kept.

Return Values**Type ViStatus**

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The desired setting was added to the Conflict Manager's stored settings.

Error Codes	Description
VI_ERROR_ALLOC	Insufficient system resources to make the change to the Conflict Manager settings.
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_INV_RSRC_NAME	The GUID string passed by the user was not a valid GUID string.
VI_ERROR_INV_SETUP	The current Conflict Manager settings are incompatible with the desired setting. For example, the new record's VISA is a disabled VISA which cannot have any stored records.
VI_ERROR_USER_BUF	A buffer passed by the user was not a valid buffer.
VI_ERROR_INV_PARAMETER	An invalid apiType or conflictHandlerType parameter value was passed in by the user.

Description

This call is used to create new records in the Conflict Manager's settings cache. For end users, the main use would be to specify a specific VISA for a given resource for the given API type. When a setting is user-specified (as opposed to Conflict Manager specified), it is never overridden, even if the open attempt fails.

Any new setting created by this function must conform to any settings already made. For example, a record may not be created for a disabled VISA.

3.2.3.8. VISACM_DeleteHandler2(apiType, interfaceType, interfaceNumber, sessionType, guid_SRM)

Purpose

Deletes a setting corresponding to both a specific VISA library and a specific resource for the given API type.

Parameters

Name	In/Out	Type	Description
apiType	IN	ViInt16	An enum specifying the type of the API corresponding to the record to be deleted.
interfaceType	IN	ViUInt16	The target resource's interface type. This is identical to the VI_ATTR_INTF_TYPE specified by the VISA specification.
interfaceNumber	IN	ViUInt16	The target resource's interface number. This is identical to the VI_ATTR_INTF_NUM specified by the VISA specification.
sessionType	IN	ViConstString	The type of resource being opened. For example, GPIB has both INSTR and BACKPLANE resources. This parameter would hold that type.
guid_SRM	IN	ViConstString	The GUID of the calling VISA library. The GUID should be passed in the format XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX.

Return Values**Type ViStatus**

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The record either does not exist or was successfully cleared.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_INV_RSRC_NAME	The GUID string passed by the user was not a valid GUID string.
VI_ERROR_INV_SETUP	The current Conflict Manager settings are incompatible with the desired setting. For example, the new record's VISA is a disabled VISA which cannot have any stored records.
VI_ERROR_USER_BUF	A buffer passed by the user was not a valid buffer.
VI_ERROR_INV_PARAMETER	An invalid apiType parameter value was passed in by the user.

Description

This function is called when a setting needs to be deleted. Much like the above VISACM_CreateHandler2() function, the function should be used by external applications to manipulate user-specified resource settings.

3.2.3.9. VISACM_DeleteHandlerByGUID2(apiType, guid_SRM)

Purpose

Delete all records corresponding to the given API type for a VISA library identified by GUID.

Parameters

Name	In/Out	Type	Description
apiType	IN	ViInt16	An enum specifying the type of the API corresponding to the records to be deleted.
guid_SRM	IN	ViConstString	The GUID of the VISA library to remove. The GUID should be passed in the format XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	All records for the specified VISA were successfully removed.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_INV_RSRC_NAME	The GUID string passed by the user was not a valid GUID string.
VI_ERROR_USER_BUF	A buffer passed by the user was not a valid buffer.
VI_ERROR_INV_PARAMETER	An invalid apiType parameter value was passed in by the user.

Description

This function deletes all records corresponding to the given API type for a specified VISA library. This function is designed for use when a VISA library is removed from the system.

3.2.3.10. VISACM_DeleteResourceByIndex2(apiType, resourceIndex)**Purpose**

Delete all records corresponding to the given API type for a resource specified by index.

Parameters

Name	In/Out	Type	Description
apiType	IN	ViInt16	An enum specifying the type of the API corresponding to the records to be deleted.
resourceIndex	IN	ViInt32	The index of the resource to remove

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	All records for the specified resource were successfully removed.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_RSRC_NFOUND	The index provided was outside the bounds of the Conflict Manager's records.
VI_ERROR_INV_PARAMETER	An invalid apiType parameter value was passed in by the user.

Description

This function deletes a resource from the Conflict Manager's cache. The index used by this function corresponds to the index used by VISACM_QueryResource2().

3.2.3.11. VISACM_FindChosenHandler2(apiType, interfaceType, interfaceNumber, sessionType, guid_SRM[], conflictHandlerType)

Purpose

Find the VISA library selected by the Conflict Manager for the given API type to open the specified resource.

Parameters

Name	In/Out	Type	Description
apiType	IN	ViInt16	An enum specifying the type of the API to be supported by the desired handler library.
interfaceType	IN	ViUInt16	The target resource's interface type. This is identical to the VI_ATTR_INTF_TYPE specified by the VISA specification.
interfaceNumber	IN	ViUInt16	The target resource's interface number. This is identical to the VI_ATTR_INTF_NUM specified by the VISA specification.
sessionType	IN	ViConstString	The type of resource being opened. For example, GPIB has both INSTR and BACKPLANE resources. This parameter would hold that type.
guid_SRM	OUT	ViChar[]	The GUID of the selected VISA library.
conflictHandlerType	OUT	ViInt16	An enum representing whether the resource was chosen by the Conflict Manager or specified by the user

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The Conflict Manager successfully supplied the VISA library for the given API type to be used for opening the resource.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_RSRC_NFOUND	No library was found for the given API type that was previously selected to open this resource.
VI_ERROR_USER_BUF	A buffer passed by the user was not valid.
VI_ERROR_INV_PARAMETER	An invalid apiType or conflictHandlerType parameter value was passed in by the user.

Description

This function is used to determine which installed VISA library for the given API type will be used to open the supplied resource. If no VISA library has previously opened the device for the given API type, and no user setting has been supplied, an error is returned.

3.2.3.12. VISACM_FlushConflictFile(flushBehavior, fileOnDiskWasNewer)**Purpose**

Flush any new settings made to the Conflict Manager settings file on disk.

Parameters

Name	In/Out	Type	Description
flushBehavior	IN	ViInt16	An enum specifying how the Conflict Manager should react if the settings file has been edited by another process.
fileOnDiskWasNewer	OUT	ViPBoolean	A Boolean signaling whether the file on the disk is newer than the one loaded by the client process.

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The settings were successfully flushed to the settings file on disk.

Error Codes	Description
VI_ERROR_FILE_ACCESS	The Conflict Manager failed to write to the settings file. This is likely due to invalid permissions.
VI_ERROR_INV_MODE	The value passed for the behavior was invalid.
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_WARN_NULL_OBJECT	Either no settings had been changed in memory, or the settings file was not overwritten based on the specified behavior.

Description

This function flushes any changes made to Conflict Manager settings to the settings file on disk. The flushBehavior parameter determines what actions the function takes if the file on disk has been updated by another process. If the settings are dirty but could not be flushed because the VISACM_FLUSH_WRITE_IF_UNCHANGED behavior was specified and the file on disk was newer, the settings remain dirty and the function returns VI_WARN_NULL_OBJECT.

3.2.3.13. VISACM_GetConflictTableFilename(filename)**Purpose**

Get the name of the Conflict Manager settings file.

Parameters

Name	In/Out	Type	Description
filename	OUT	ViChar[]	The path and name of the Conflict Manager settings file.

Return Values**Type** ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The settings file information was successfully returned.

Error Codes	Description
VI_ERROR_FILE_ACCESS	The Conflict Manager could not access the required path.
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_INV_SETUP	The path required by the Conflict Manager does not exist.
VI_ERROR_USER_BUF	A buffer passed by the user was not valid.

Description

Get the name of the file where the Conflict Manager settings will be kept. This file does not necessarily need to exist.

3.2.3.14. VISACM_GetIsDirty(isDirty)**Purpose**

Get whether any Conflict Manager settings have been changed since the last flush of the settings to disk.

Parameters

Name	In/Out	Type	Description
isDirty	OUT	ViPBoolean	The variable specifying whether the settings in memory have been written to disk.

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The status of the settings in memory has been successfully returned.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_USER_BUF	The output variable pointer passed to the function was invalid.

Description

This function returns whether the settings in memory have changed since the last write to disk.

3.2.3.15. VISACM_GetInstalledVisa2(apiType, index, vendorID, guid_SRM[], visaPathLocation[], visaFriendlyName[], comments[])**Purpose**

Get information about an installed VISA library for the given API type based on a supplied index.

Parameters

Name	In/Out	Type	Description
apiType	IN	ViInt16	An enum specifying the type of the API corresponding to the VISA library.
index	IN	ViInt32	The index determining which VISA library's information gets returned.
vendorID	OUT	ViPUInt16	The vendor ID number of the VISA library's vendor.

guid_SRM	OUT	ViChar[]	The GUID associated with the VISA library.
visaPathLocation	OUT	ViChar[]	The location of the vendor-specific VISA implementation.
visaFriendlyName	OUT	ViChar[]	The friendly name of the VISA library, determined by the vendor.
comments	OUT	ViChar[]	Any comments regarding the specified VISA library.

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The information for the VISA library at the given index was successfully returned.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_RSRC_NFOUND	A VISA library corresponding to the given index could not be found. This could be due to an out-of-range index or corruption of the VISA information in the registry.
VI_ERROR_USER_BUF	A buffer passed by the user was not valid.
VI_ERROR_INV_PARAMETER	An invalid apiType parameter value was passed in by the user.

Description

This function returns information for a vendor-specific VISA implementation based on a given index. While most of the data returned by this function is mainly for informational use, the GUID can be used by other functions to determine what resources a vendor-specific VISA library is currently set to access. The GUID is also used by the VISA COM Global Resource Manager to create instances of the Vendor Specific VISA COM resource manager class. For VISA C, the visaPathLocation value is the absolute path on disk to the vendor specific VISA C Library that can be loaded by the VISA C Router. For VISA .NET, the visaPathLocation value is the Assembly Qualified Name of the resource manager class in the vendor specific VISA .NET assembly that can be instantiated by the VISA .NET Global Resource Manager described in *VPP-4.3.6, VISA Implementation Specification for .NET*.

3.2.3.16. VISACM_GetInstalledVisaCount2(apiType, numberOfVisas)**Purpose**

Get the total number of vendor-specific VISA libraries installed on the system that support the given API type.

Parameters

Name	In/Out	Type	Description
apiType	IN	ViInt16	An enum specifying the type of the API.
numberOfVisas	OUT	ViPInt32	The number of VISA libraries installed on the system that support the given API type.

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The number of VISA libraries on the system that support the given API type was successfully returned.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_RSRC_NFOUND	No installed VISA libraries that support the given API type were found.
VI_ERROR_USER_BUF	The output variable passed to the function was invalid.
VI_ERROR_INV_PARAMETER	An invalid apiType parameter value was passed in by the user.

Description

This functions returns the total number of vendor-specific VISA libraries that support the specified API type installed on the system. The value returned by this can be used to iterate through the installed VISA libraries using the VISACM_GetInstalledVisa2() function.

3.2.3.17. VISACM_GetResourceCount2(apiType, numberRsrcs)**Purpose**

Gets the number of resources with settings stored by the Conflict Manager.

Parameters

Name	In/Out	Type	Description
apiType	IN	ViInt16	An enum specifying the type of the API for which to query the number of resources.
numberRsrcs	OUT	ViPInt32	The number of resources with settings stored in the Conflict Manager.

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The number of resources stored by the Conflict Manager corresponding to the specified API type was successfully returned.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize() has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_USER_BUF	The output variable passed to the function was invalid.
VI_ERROR_INV_PARAMETER	An invalid apiType parameter value was passed in by the user.

Description

This function gets the number of resources with settings corresponding to the specified API type stored by the Conflict Manager. The value returned by this function can be used to iterate through resource records using VISACM_QueryResource2().

3.2.3.18. VISACM_GetStoreConflictsOnly(storeConflicts)**Purpose**

Returns whether the Conflict Manager is storing settings only for resources with multiple VISA libraries able to parse the resource string.

Parameters

Name	In/Out	Type	Description
storeConflicts	OUT	ViBoolean	A variable representing whether the Conflict Manager is storing only resources with multiple VISA libraries finding the resource.

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The setting was successfully returned by the function.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_USER_BUF	The output variable passed to the function was invalid.

Description

This function returns whether the Conflict Manager is storing settings only for resources with multiple VISA libraries able to parse the resource string. If this setting is false, then the Conflict Manager stores settings for all resources, even when no conflicts exist.

3.2.3.19. VISACM_GetVisaEnabled2(apiType, guid_SRM, enabled)**Purpose**

Returns whether a specific VISA library is enabled or disabled.

Parameters

Name	In/Out	Type	Description
apiType	IN	ViInt16	An enum specifying the type of the API supported by the given VISA library.
guid_SRM	IN	ViConstString	The GUID identifying the VISA for which to retrieve its enabled status.
Enabled	OUT	ViPBoolean	The variable specifying whether the VISA is enabled or disabled for the specified API type

Return Values**Type** ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The enabled status for the VISA was returned successfully.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_INV_RSRC_NAME	The GUID supplied to the function was invalid. GUID values should be in XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX format.
VI_ERROR_USER_BUF	The output variable passed to the function was invalid.
VI_ERROR_INV_PARAMETER	An invalid apiType parameter value was passed in by the user.

Description

The function returns whether a given VISA is enabled for the given API type. A VISA that is disabled will not be used by the Conflict Manager for records corresponding to the given API type and may not have any other settings associated with it.

3.2.3.20. VISACM_GetVisaPreferred2(apiType, guid_SRM[])**Purpose**

Get the GUID for the preferred VISA library for the specified API type.

Parameters

Name	In/Out	Type	Description
apiType	IN	ViInt16	An enum specifying the type of the API.
guid_SRM	OUT	ViChar[]	The GUID of the preferred VISA library for the given API type on the system. The ViChar array should be at least 36 characters long.

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The GUID for the preferred VISA library for the given API type was returned successfully.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_RSRC_NFOUND	No preferred VISA is specified to handle the given API type on the system.
VI_ERROR_USER_BUF	The output variable passed to the function was invalid.
VI_ERROR_INV_PARAMETER	An invalid apiType parameter value was passed in by the user.

Description

This function returns the GUID of the preferred VISA library for the specified API type. If no VISA library has been specified as preferred for that API type, an error is returned.

3.2.3.21. VISACM_Initialize()**Purpose**

Initialize the Conflict Manager library for use by the client application.

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The Conflict Manager library was initialized successfully.

Error Codes	Description
VI_ERROR_ALLOC	The Conflict Manager failed to initialize.

Description

This function initializes the Conflict Manager library for use by a client application. This function must be called before any other function for the Conflict Manager to be used.

3.2.3.22. VISACM_QueryResource2(apiType, resourceIndex, interfaceType, interfaceNumber, sessionType[], numHandlers)**Purpose**

Queries the properties of a resource stored in the Conflict Manager settings cache based on a provided index and the given API type.

Parameters

Name	In/Out	Type	Description
apiType	IN	ViInt16	An enum specifying the type of the API to query.
resourceIndex	IN	ViInt32	The index of the resource for which to retrieve information.
interfaceType	OUT	ViPUInt16	The target resource's interface type. This is identical to the VI_ATTR_INTF_TYPE specified by the VISA specification.
interfaceNumber	OUT	ViPUInt16	The target resource's interface number. This is identical to the VI_ATTR_INTF_NUM specified by the VISA specification.
sessionType	OUT	ViChar[]	The type of resource being opened. For example, GPIB has both INSTR and BACKPLANE resources. This parameter would hold that type.
numHandlers	OUT	ViPInt16	The number of records stored for the specified resource.

Return Values**Type ViStatus**

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The information for the specified resource was returned successfully.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_RSRC_NFOUND	A resource corresponding to the given index and API type could not be found. This is most likely due to an out-of-range index.
VI_ERROR_SYSTEM_ERROR	The function failed to access the Conflict Manager settings successfully.
VI_ERROR_USER_BUF	A buffer passed by the user was not valid.
VI_ERROR_INV_PARAMETER	An invalid apiType parameter value was passed in by the user.

Description

This function queries the properties of the resources corresponding to the given index and API type. This function is mainly meant to be used to iterate through resource stored by the Conflict Manager.

VISACM_QueryResourceHandler2() can be used to iterate through the handler records for each resource for the given API type .

3.2.3.23. VISACM_QueryResourceHandler2(apiType, resourceIndex, handlerIndex, guid_SRM[], conflictHandlerType, comments[])

Purpose

Query a record for how a given VISA library is set to handle a specific resource for the given API type.

Parameters

Name	In/Out	Type	Description
apiType	IN	ViInt16	An enum specifying the type of the API to query.
resourceIndex	IN	ViInt32	The index corresponding to the resource.
handlerIndex	IN	ViInt32	A provided index corresponding to which VISA library record should be returned.
guid_SRM	OUT	ViChar[]	The GUID of the VISA library specified by this record
conflictHandlerType	OUT	ViInt16	An enum representing whether this VISA library set by the user to handle this resource, chosen by the Conflict Manager to handle this resource, or set to not handle the resource.
comments	OUT	ViChar[]	A character string containing any comments stored with the record.

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The information from the record was returned successfully.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_RSRC_NFOUND	Either the resource index or the handler index was invalid.
VI_ERROR_SYSTEM_ERROR	The function failed to access the Conflict Manager settings successfully.
VI_ERROR_USER_BUF	A buffer passed by the user was not valid.
VI_ERROR_INV_PARAMETER	An invalid apiType or conflictHandlerType parameter value was passed in by the user.

Description

This function queries how a specific VISA library is set to handle a given resource for the given API type. A VISA library can either be chosen by the user, chosen by the Conflict Manager, or set to not handle a given resource. Using VISACM_QueryResource2() in tandem with this function can provide a complete picture of the Conflict Manager's configuration.

3.2.3.24. VISACM_ReloadFile()**Purpose**

Reload settings from the Conflict Manager's settings file.

Return Values

Type ViStatus This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The settings were successfully loaded from the Conflict Manager's settings file on disk.

Error Codes	Description
VI_ERROR_ALLOC	The settings failed to load due to a failed allocation.
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.

Description

This function reloads the settings from the Conflict Manager's settings file. If the file cannot be read, the settings revert to the default settings.

3.2.3.25. VISACM_SetStoreConflictsOnly(storeConflicts)**Purpose**

Sets whether the Conflict Manager should store only conflicts in the table.

Parameters

Name	In/Out	Type	Description
storeConflicts	IN	ViBoolean	Determines whether the Conflict Manager will store only conflicted entries or all entries.

Return Values

Type ViStatus This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The setting was set successfully.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.

Description

This function sets whether the Conflict Manager stores only conflicts in the table or the results of all resource openings. Saving all of the results from opening results in a larger table, but prevents the VISA Router from repeating the conflict arbitration algorithm on every open.

3.2.3.26. VISACM_SetVisaEnabled2(apiType, guid_SRM, enabled)**Purpose**

Enables or disables a specified VISA library.

Parameters

Name	In/Out	Type	Description
apiType	IN	ViInt16	An enum specifying the type of the API for which to enable or disable the specified VISA library.
guid_SRM	IN	ViConstString	The GUID of the VISA library to enable or disable. The GUID should be provided in XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX format.
enabled	IN	ViBoolean	Specifies whether the VISA library should be enabled or disabled.

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The VISA library was successfully disabled.

Error Codes	Description
VI_ERROR_ALLOC	The Conflict Manager could not allocate needed memory for the new setting.
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_INV_RSRC_NAME	The specified GUID was not valid.
VI_ERROR_USER_BUF	A buffer passed by the user was not valid.
VI_ERROR_INV_PARAMETER	An invalid apiType parameter value was passed in by the user.

Description

Enables or disables a specified VISA library for the specified API type. A disabled VISA library will have all of its records corresponding to the specified API type deleted from the conflict table and will not be used when opening any resources for that API type. When reenabling a VISA library, the user may want to clear the entire table to repopulate the table with settings using the reenabled VISA library.

3.2.3.27. VISACM_SetVisaPreferred2(apiType, guid_SRM)

Purpose

Sets a VISA library as the preferred VISA library for the given API type.

Parameters

Name	In/Out	Type	Description
apiType	IN	ViInt16	An enum specifying the type of the API for which to set the given preferred library.
guid_SRM	IN	ViConstString	The GUID corresponding to the VISA library to set as preferred.

Return Values

Type ViStatus

This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The VISA library was successfully set as the preferred VISA library.

Error Codes	Description
VI_ERROR_INV_OBJECT	The Conflict Manager Library has not been properly initialized. Ensure that VISACM_Initialize () has been called before attempting to configure Conflict Manager Settings.
VI_ERROR_INV_RSRC_NAME	The specified GUID was not valid.
VI_ERROR_INV_SETUP	The VISA library specified to be preferred is disabled for the given API type. This is not a valid setup and returns an error instead of making this setting.
VI_ERROR_USER_BUF	A buffer passed by the user was not valid.
VI_ERROR_INV_PARAMETER	An invalid apiType parameter value was passed in by the user.

Description

This function sets a VISA library as the preferred VISA library for the specified API type. As the preferred VISA library, this library is used to open all resources on the system for the given API type, unless that resource has a user-specified preference for that API type.

3.2.4. VISA Utilities

The VISA Utilities component is supported only on the WIN64 framework.

The VISA Utilities component provides support to the VISA Router for mapping the VISA object handles of the underlying vendor-specific VISA object handles to the VISA object handles that are presented to the user. `visaUtilities.dll` exports a number of entry points, most of which are used internally by the VISA router and are not documented in this specification.

Only the `getUserVi()` entry point is documented here because, given a vendor-specific VISA object handle, it returns the user-visible VISA object handle. This is useful for vendor-provided utilities that deal with vendor-specific object handles, but need to display the user-visible object handles returned from the VISA Router.

3.2.4.1. `getUserVi (underlyingVi, underlyingManfId)`

Purpose

Map a vendor-specific VISA object handle to the user-visible VISA object handle used by the VISA Router.

Parameters

Name	In/Out	Type	Description
<code>underlyingVi</code>	IN	<code>const ViSession</code>	A vendor-specific VISA object handle.
<code>underlyingManfId</code>	IN	<code>const ViUInt16</code>	The <code>VI_ATTR_RSRC_MANF_ID</code> value for this vendor-specific VISA.

Return Values

Type `ViSession`

This is the user-visible VISA object handle that corresponds to the underlying vendor-specific VISA object handle that was passed in.

If 0 is passed in as the `underlyingVi`, the value returned is 0. If there is no corresponding user-visible VISA object handle, the value returned is that of the underlying VI.

Description

This function is useful for vendor-provided utilities that deal with vendor-specific object handles, but which need to display the objecthandle that the VISA Router has presented to the user. The VISA Router hides the vendor-specific handle from the user, so a vendor utility that presents objectinformation to the user needs a way to convert the hidden vendor-specific objecthandle to the user-visible objecthandle that has meaning to the user.

Note that both the vendor-specific VISA objecthandle and the vendor Manufacturer ID are required to determine the user-visible VISA objecthandle. This is because the vendor-specific VISA objecthandle may not be unique.

3.3. VISA COM Components

Table 3.3.1 shows a list of files included in the VISA Shared Components. See *VPP-4.3.4, VISA Implementation Specification for COM*, for details of these components.

Component Name	32-Bit	64-Bit	Description
GlobMgr.dll	X	X	The Global Resource Manager (GRM) COM Component and the VISA COM I/O type library.
BasFrmIO.dll	X	X	A standard implementation of the VISA COM IFormattedIO488 interface.
Ivi.Visa.Interop.dll Ivi.Visa.Interop.xml	X	X	The primary interop assembly (PIA) for the VISA COM I/O type library, along with the PIA intellisense help file.
Ivi.Visa.Interop.config Policy.X.X.Ivi.Visa. Interop.xml	X		The policy file that forwards calls from previous versions of the VISA COM PIA to the current version. X.X may stand for any previous version that is forwarded to the current PIA. There may be multiple policy files, one for each major.minor version that is forwarded.
IviPiaRegistration.bat	X	X	A batch program that can be run by users to make sure that all of the IVI PIAs are properly registered in the Global Assembly Cache (GAC).

Table 3.3.1

3.4. VISA .NET Components

Table 3.3.2 shows a list of files included in the VISA .NET Shared Components installer. See *VPP-4.3.6, VISA Implementation Specification for .NET*, for additional details of the functionality provided by this component.

Component Name	Any CPU	Description
Ivi.Visa.dll	X	The VISA .NET Shared Assembly that provides the implementation of the GlobalResourceManager class and an implementation of the IMessageBasedFormattedIO interface defined in the VPP 4.3.6 specification. In addition, this assembly also provides the implementation of the ConflictManager class defined in this specification.

Table 3.4.1

3.4.1. Conflict Resolution Manager .NET API

The VISA .NET Shared Components provide a .NET API for the Conflict Resolution Manager. This API consists of several enumerations and classes that .NET clients can use to access the Conflict Resolution Manager. The methods available in these classes correspond to the C API functions that are described in section 3.2.3.

RULE 3.4.1

VISA .NET Shared Components **SHALL** provide the only implementation of the Conflict Resolution Manager .NET API.

3.4.1.2. Enumerations

Conflict Manager .NET API defines the following enumerations. All enumerations are defined in the Ivi.Visa.ConflictManager namespace.

- ApiType
- FlushBehavior
- HandlerType

3.4.1.2.1 ApiType

Definition

```
public enum ApiType
{
    CAndCom = 0,
    DotNet = 1
}
```

OBSERVATION 3.4.1

The `ApiType` enumeration consists of all the VISA API Types that are supported by this version of the Conflict Manager API functions. This enumeration corresponds to the defined values for the `apiType` parameter used in the Conflict Manager C API functions described in section 3.2.3.

3.4.1.2.2 FlushBehavior

Definition

```
public enum FlushBehavior
{
    OverwriteAlways = 0,
    WriteIfFileOnDiskUnchanged = 1,
    WriteOrReload = 2
}
```

OBSERVATION 3.4.2

The `FlushBehavior` enumeration specifies the behavior of the `ConflictManager FlushConflictFile()` method when it saves the Conflict Manager settings to the file on disk. This enumeration corresponds to the `flushBehavior` parameter of the `VISACM_FlushConflictFile()` function described in section 3.2.3. The values are described in the following table.

Value	Description
<code>OverwriteAlways</code>	Save the conflict file even if there are changes to the file on disk since it was read.
<code>WriteIfFileOnDiskUnchanged</code>	Save only if there are no changes to the file on disk since it was read.
<code>WriteOrReload</code>	Save only if there are no changes to the file on disk was read. If the file on disk is newer, <code>FlushConflictFile</code> will discard the local changes and reload the file from disk.

3.4.1.2.3. HandlerType

Definition

```
public enum HandlerType
{
    NotChosen = 0,
    ChosenByResourceManager = 1,
    ChosenByUser = 2
}
```

OBSERVATION 3.4.3

The `HandlerType` enumeration represents the possible status settings of the `VisaImplementation` in a Conflict Manager record with respect to the `HardwareInterface` with which the handler is associated. This enumeration corresponds to the `conflictHandlerType` parameter used in the Conflict Manager C API functions described in section 3.2.3. The values are described in the following table.

Value	Description
<code>NotChosen</code>	The <code>VisaImplementation</code> is not the chosen handler for the <code>HardwareInterface</code> .
<code>ChosenByResourceManager</code>	The <code>VisaImplementation</code> is the chosen handler for the <code>HardwareInterface</code> and it was chosen by a Resource Manager.
<code>ChosenByUser</code>	The <code>VisaImplementation</code> is the chosen handler for the <code>HardwareInterface</code> and it was chosen by a user.

3.4.1.3. Exceptions

Accessing the ConflictManager .NET API methods and properties throw various standard .NET exceptions to indicate error conditions.

3.4.1.4. Classes

Conflict Manager .NET API defines the following classes. All classes are defined in the Ivi.Visa.ConflictManager namespace.

- HardwareInterface
- VisaImplementation
- ConflictManager

3.4.1.4.1. HardwareInterface

Description

The HardwareInterface class consolidates information related to hardware interfaces. `Type` specifies the hardware interface type (GPIB, ASRL, etc.). `Number` specifies the board number within the interface type. `ResourceClass` specifies the resource class for this session (INSTR, INTFC, etc.).

Definition

```
public sealed class HardwareInterface : IEquatable<HardwareInterface>
{
    public Int16 Number { get; private set; }
    public String ResourceClass { get; private set; }
    public Int32 Type { get; private set; }
    public HardwareInterface(Int32 type, Int16 number, String resourceClass)
    {...}

    public static Boolean operator ==(HardwareInterface intf1, HardwareInterface
intf2) {...}
    public static Boolean operator !=(HardwareInterface intf1, HardwareInterface
intf2) {...}
    public override Boolean Equals(object o) {...}
    public Boolean Equals(HardwareInterface other) {...}
}
```

Corresponding C API Features

The HardwareInterface class has several .NET properties that correspond to parameters to functions in the C API. The following table shows property-parameter equivalence for this class.

Property Name	C API Parameter Name
Number	interfaceNumber
ResourceClass	sessionType
Type	interfaceType

There is no corresponding C API function for the constructor or the equality methods. The only job of the constructor is to create an instance of the object and set the values of the related properties.

3.4.1.4.2. VisaImplementation

Description

The `VisaImplementation` class consolidates information related to VISA implementations that are installed on the system. `HandlerId` specifies the GUID that identifies the VISA implementation for the given API type. `ResourceManufacturerId` specifies the ID number of the vendor of this implementation. (These numbers are defined in the VPP-9 specification.). `Location` specifies the location of the DLL or class that contains the implementation. If the `ApiType` is `CAndCom`, this is the full path for the resource manager DLL. If the `ApiType` is `DotNet`, this is the assembly qualified name for the VISA implementation's resource manager class. `FriendlyName` specifies a descriptive name for the implementation. `Comments` specifies additional information about the implementation. `ApiType` represents the type of API supported by the implementation. `Enabled` specifies whether a resource manager will ever load or instantiate this VISA implementation.

Definition

```
public sealed class VisaImplementation : IEquatable<VisaImplementation>
{
    public ApiType ApiType { get; private set; }
    public String Comments { get; private set; }
    public Boolean Enabled { get; set; }
    public String FriendlyName { get; private set; }
    public Guid HandlerId { get; private set; }
    public String Location { get; private set; }
    public Int32 ResourceManufacturerId { get; private set; }

    public VisaImplementation(
        Guid handlerId,
        Int32 resourceManufacturerId,
        String location,
        String friendlyName,
        String comments,
        ApiType apiType) {...}
    public static Boolean operator ==(VisaImplementation visal, VisaImplementation
visa2) {...}
    public static Boolean operator !=(VisaImplementation visal, VisaImplementation
visa2) {...}
    public override Boolean Equals(object o) {...}
    public Boolean Equals(VisaImplementation other) {...}
}
```

Corresponding C API Features

The `VisaImplementation` class has several .NET properties that correspond to parameters to the functions in the C API. The following table shows property-parameter equivalence for this class.

Property Name	C API Parameter Name
<code>ApiType</code>	<code>apiType</code>
<code>Comments</code>	<code>comments (VISACM_GetInstalledVisa2())</code>
<code>Enabled</code>	<code>enabled</code>
<code>FriendlyName</code>	<code>visaFriendlyName</code>
<code>HandlerId</code>	<code>guid_SRM</code>
<code>Location</code>	<code>visaPathLocation</code>
<code>ResourceManufacturerId</code>	<code>vendorID</code>

There is no corresponding C API function for the constructor or equality methods. The only job of the constructor is to create an instance of the object and set the values of the related properties.

3.4.1.4.4. ConflictManager

Description

The ConflictManager class provides the primary .NET API for the Conflict Resolution Manager. The methods in this class correspond to one or more of the C API functions described in section 3.2.3. The ConflictFilePath property is the path to the file on disk that is used by the Conflict Manager to store the settings. The IsDirty property indicates whether there have been any changes to the in-memory copy of the settings since the settings were last saved or initially loaded. The StoreConflictsOnly property specifies whether the Conflict Manager will only store records when there is a conflict.

Definition

```
public sealed class ConflictManager : IDisposable
{
    public Boolean StoreConflictsOnly { get; set; }
    public String ConflictFilePath { get; }
    public Boolean IsDirty { get; }

    public ConflictManager() {...}
    ~ConflictManager() {...}
    public void Dispose() {...}

    public void ClearTable() {...}
    public void CreateHandler(HardwareInterface intf,
                             VisaImplementation visa,
                             HandlerType type) {...}
    public void CreateHandler(HardwareInterface intf,
                             VisaImplementation visa,
                             HandlerType type,
                             String comments) {...}
    public void FlushConflictFile(FlushBehavior behavior) {...}
    public void FlushConflictFile(FlushBehavior behavior,
                                  out Boolean fileOnDiskWasNewer) {...}
    public VisaImplementation GetChosenHandler(ApiType apiType,
                                                HardwareInterface intf) {...}
    public VisaImplementation GetChosenHandler(ApiType apiType,
                                                HardwareInterface intf,
                                                out HandlerType handlerType) {...}
    public List<VisaImplementation> GetHandlers(ApiType apiType,
                                                HardwareInterface intf) {...}
    public List<VisaImplementation> GetInstalledVisas(ApiType apiType) {...}
    public List<HardwareInterface> GetInterfaces(ApiType apiType) {...}
    public VisaImplementation GetPreferredVisa(ApiType apiType) {...}
    public void ReloadFile() {...}
    public void RemoveHandler(HardwareInterface intf, VisaImplementation visa) {...}
    public void RemoveHandlers(ApiType apiType) {...}
    public void RemoveHandlers(ApiType apiType, HardwareInterface intf) {...}
    public void RemoveHandlers(VisaImplementation visa) {...}
    public void SetPreferredVisa(VisaImplementation visa) {...}
}
```

Corresponding C API Features

The `ConflictManager` class has several .NET properties that correspond to parameters to functions in the C API. The following table shows property-parameter equivalence for this class.

Property Name	C API Parameter Name
<code>StoreConflictsOnly</code>	<code>storeConflicts</code>
<code>ConflictFilePath</code>	<code>filename</code>
<code>IsDirty</code>	<code>isDirty</code>

The `ConflictManager` class has several methods that correspond to one or more functions in the C API. following table shows method-function equivalence for hardware interfaces.

Method Name	C API Function Name
<code>Constructor</code>	<code>VISACM_Initialize()</code>
<code>Dispose</code>	<code>VISACM_Close()</code>
<code>ClearTable()</code>	<code>VISACM_ClearEntireTable()</code>
<code>CreateHandler(HardwareInterface, VisaImplementation, HandlerType)</code>	<code>VISACM_CreateHandler2()</code>
<code>CreateHandler(HardwareInterface, VisaImplementation, HandlerType, String)</code>	<code>VISACM_CreateHandler2()</code>
<code>FlushConflictFile(FlushBehavior)</code>	<code>VISACM_FlushConflictFile()</code>
<code>FlushConflictFile(FlushBehavior, out Boolean)</code>	<code>VISACM_FlushConflictFile()</code>
<code>GetChosenHandler(ApiType, HardwareInterface)</code>	<code>VISACM_FindChosenHandler2()</code>
<code>GetChosenHandler(ApiType, HardwareInterface, out HandlerType)</code>	<code>VISACM_FindChosenHandler2()</code>
<code>GetHandlers(ApiType, HardwareInterface)</code>	<code>VISACM_GetResourceCount2()</code> Followed by (per resource) <code>VISACM_QueryResource2()</code> Followed by (per handler) <code>VISACM_QueryResourceHandler2()</code>
<code>GetInstalledVisas(ApiType)</code>	<code>VISACM_GetInstalledVisaCount2()</code> Followed by (per VISA) <code>VISACM_GetInstalledVisa2()</code> <code>VISACM_GetVisaEnabled2()</code>
<code>GetInterfaces(ApiType)</code>	<code>VISACM_GetResourceCount2()</code> Followed by (per resource) <code>VISACM_QueryResource2()</code>
<code>GetPreferredVisa(ApiType)</code>	<code>VISACM_GetVisaPreferred2()</code>
<code>ReloadFile()</code>	<code>VISACM_ReloadFile()</code>
<code>RemoveHandler(HardwareInterface, VisaImplementation)</code>	<code>VISACM_DeleteHandler2()</code>
<code>RemoveHandlers(ApiType)</code>	<code>VISACM_ClearResourceHandlersFromTable2()</code>
<code>RemoveHandlers(ApiType, HardwareInterface)</code>	<code>VISACM_DeleteResourceByIndex2()</code>
<code>RemoveHandlers(VisaImplementation)</code>	<code>VISACM_DeleteHandlerByGUID2()</code>
<code>SetPreferredVisa(VisaImplementation)</code>	<code>VISACM_SetVisaPreferred2()</code>

3.5 VISA Shared USBTMC Device Driver

Table 3.5.1 shows a list of files included in the VISA Shared Components. See *IVI 6.2, VISA Interoperability Requirement for USBTMC Specification*, for details of this component.

Component Name	32-Bit	64-Bit	Description
ausbtmc.sys	X	X	The signed kernel binary containing the USBTMC driver logic.
ausbtmc.cat	X	X	The catalog file containing the digital signature for the kernel driver.
ausbtmc.inf	X	X	The setup information file detailing installation behavior for the kernel driver.

Table 3.5.1

Section 4: VISA Shared Components Installers

Section 3 described the components required for a complete VISA .NET and VISA COM I/O implementation. This section covers the details of the installation and gives detailed requirements of the components' implementation.

There are two VISA Shared Components installers, one for 32-bit Windows operating systems and one for 64-bit operating systems. The 32-bit installer installs 32-bit components only. The details about the components installed by this installer are in section 4.1. The 64-bit installer installs both 32-bit and 64-bit components, with 32-bit components installed so that they can be run using Microsoft's Windows on Windows 64 (WOW64) technology. The details about the components installed by this installer are in section 4.2.

Similarly, there are two VISA .NET Shared Components installers, one for 32-bit Windows operating systems and one for 64-bit operating systems. The details of the components installed by the VISA .NET Shared Components installers are in section 4.3.

4.1. Installing VISA Shared Components On 32-Bit Operating Systems

TERMS

The following terms are used in this section.

- *<SYSTEM32DIR>* is the Windows system directory for executables. The default is C:\Windows\System32, but this may be changed when Windows is installed.
- *<PROGRAMFILES>* is the Windows Program Files directory. The default is C:\Program Files, but this may be changed when Windows is installed.
- *<ALLUSERSAPPDATA>* is the Windows directory where data that is accessible to all users is stored (as opposed to data that is accessible to the current user).
- *<VXIPNPPATH>* is the target directory for the VISA components on 32-bit systems.
- *<VISADATAPATH>* is the directory used for VISA data files, and the conflict resolution table in particular.

RULE 4.1.1

The VISA Shared Components installer for Windows 7 (32-bit), Windows 8 (32-bit), and Windows 10 (32-bit) **SHALL** be named *VISA Shared Components* and **SHALL** have its own entry in the Windows Add/Remove Programs list.

RULE 4.1.2

Every 32-bit vendor-specific VISA installer released after June 1, 2009 **SHALL** use the VISA Shared Components installer to install shared VISA COM or VISA Plug-In Architecture files, or to create standard VISA directories, registry keys and values, or environment variables, and **SHALL NOT** install the components or create standard VISA directories, registry keys and values, or environment variables as part of the vendor-specific installer.

RULE 4.1.3

The value of *<VXIPNPPATH>*:

- **IF** the registry key HKLM\SOFTWARE\VXIPNP_Alliance\VXIPNP\CurrentVersion exists and contains the value *VXIPNPPATH*, and this value designates a directory that exists, and the directory or any of the subdirectories contain any files, **THEN** the value of *<VXIPNPPATH>* **SHALL** be the value of the this key's *VXIPNPPATH* string value.
- **OTHERWISE**, the default value of *<VXIPNPPATH>* **SHALL** be *<PROGRAMFILES>\IVI Foundation\VISA* **AND** the VISA Shared Components installer **SHALL** allow the user to change the value of *<VXIPNPPATH>*.

OBSERVATION 4.1.1

The above rules regarding the VISA Shared Components installer dictate that it detect existing VISA components and other *VXIplug&play* components such as instrument drivers. If the *VXIplug&play* registry keys point to a *VXIplug&play* root directory, and that directory exists, but no files exist in the hierarchy to which they point, it is acceptable for the VISA Shared Components installer to allow the user to change the 32-bit VISA base directory. In this case, the VISA Shared Components installer changes the *VXIplug&play* registry keys and the environment variables if necessary, but it leaves the old, empty directory hierarchy alone (that is, it does not remove it).

RULE 4.1.4

The value of *<VISADATAPATH>* **SHALL** be *<ALLUSERSAPPDATA>\IVI Foundation\VISA*.

RULE 4.1.5

The VISA Shared Components installer **SHALL** create any of the following directories that do not already exist:

- <VXIPNPPATH>
- <VXIPNPPATH>\VisaCom
- <VXIPNPPATH>\VisaCom\Primary Interop Assemblies
- <VXIPNPPATH>\WinNT
- <VXIPNPPATH>\WinNT\Bin
- <VXIPNPPATH>\WinNT\include
- <VXIPNPPATH>\WinNT\lib
- <VXIPNPPATH>\WinNT\lib\bc
- <VXIPNPPATH>\WinNT\lib\msc
- <VXIPNPPATH>\WinNT\lib_x64
- <VXIPNPPATH>\WinNT\lib_x64\msc
- <VISADATAPATH>

RULE 4.1.6

The VISA Shared Components installer **SHALL** install the following files, unless newer versions of the files are already installed:

- <SYSTEM32DIR>\visaConfMgr.dll (32-bit executable)
- <VXIPNPPATH>\VisaCom\GlobMgr.dll (32-bit executable)
- <VXIPNPPATH>\VisaCom\BasFrmIO.dll (32-bit executable)
- <VXIPNPPATH>\VisaCom\Primary Interop Assemblies\Ivi.Visa.Interop.dll (32-bit executable)
- <VXIPNPPATH>\VisaCom\Primary Interop Assemblies\Ivi.Visa.Interop.xml
- <VXIPNPPATH>\VisaCom\Primary Interop Assemblies\Ivi.Visa.Interop.config
- <VXIPNPPATH>\VisaCom\Primary Interop Assemblies\Policy.1.0.Ivi.Visa.Interop.dll (32-bit executable, one for each major/minor previous version that must be redirected to the current version of Ivi.Visa.Interop.dll. *XX* stand for the previous major/minor versions being redirected.)
- <VXIPNPPATH>\VisaCom\Primary Interop Assemblies\IviPiaRegistration.bat
- <VXIPNPPATH>\WinNT\include\visa.h
- <VXIPNPPATH>\WinNT\include\visatype.h
- <VXIPNPPATH>\WinNT\lib_x64\msc\visa64.lib
- ausbtmc.cat (location determined by Windows during driver installation process)
- ausbtmc.inf (location determined by Windows during driver installation process)
- ausbtmc.sys (32-bit kernel driver file, location determined by Windows during driver installation process)

RULE 4.1.7

The VISA Shared Components installer **SHALL NOT** replace an existing conflict resolution table file.

RULE 4.1.8

The VISA Shared Components installer **SHALL** create registry keys and values for GlobMgr.dll and BasFrmIO.dll as if regsvr32 was run on them. It **SHALL NOT** self register these components.

OBSERVATION 4.1.2

Note that the registry keys and values that are created by self-registering 32-bit COM type libraries on a 32-bit operating system are different than the registry keys and values created by self-registering the same 32-bit type library on a 64-bit system. These changes are more extensive than just relocating keys to the WOW64 registry hive—the number of subkeys actually is different. If the 32-bit installer creates registry keys and values that match the self-registration keys and values on a 32-bit operating system, they will not match the self-registration keys and values on a 64-bit operating system, and vice versa. This is one reason why the 32-bit installer installs only on 32-bit operating systems.

RULE 4.1.9

The VISA Shared Components installer **SHALL** create registry keys and values for `Ivi.Visa.Interop.dll`, the Primary Interop Assembly (PIA), as if the 32-bit version of `regasm` was run on it. It **SHALL NOT** run `regasm` from within the installer. The resulting keys and values created **SHALL** be equivalent to running the following command:

```
regasm <PIA>
```

OBSERVATION 4.1.3

Prior to version 5.4, this specification did not specify the exact `regasm` command that would be equivalent to the registration done by the VISA Shared Components installer. However, the installers compliant to those versions of this specification register the PIA in a manner that is equivalent to running the following command:

```
regasm <PIA> /codebase
```

While this doesn't cause any problem on 32-bit Operating Systems supported by the 32-bit versions of this installer, there are problems on 64-bit Operating Systems supported by the 64-bit versions of this installer. See OBSERVATION 4.2.5 for more details. To be consistent, the 32-bit installer will also use the command without the `/codebase` option.

RULE 4.1.10

The VISA Shared Components installer **SHALL** create the following additional registry keys and values under `HKLM\SOFTWARE`:

- `VXIPNP_Alliance`
- `VXIPNP_Alliance\IVIVISACOM`
- `VXIPNP_Alliance\IVIVISACOM\CurrentVersion`
 - Value: `InstallerVersion`—The version of the VISA Shared Components installer.
 - Value: `Version`—The product version of `GlobMgr.dll`
- `VXIPNP_Alliance\VXIPNP`
- `VXIPNP_Alliance\VXIPNP\CurrentVersion`
 - Value: `FRAMEWORK_PATH`—`<VXIPNPPATH>\WinNT`
 - Value: `VXIPNPPATH`—`<VXIPNPPATH>`

Note that there no default values for any of the above keys.

RULE 4.1.11

The VISA Shared Components installer **SHALL** install on Windows 7 32-bit, Windows 8 32-bit, and Windows 10 32-bit. Service Pack requirements may vary by version of the installer and are documented on the IVI web site for each version of the installer. The VISA Shared Components installer **SHALL NOT** install on any 64-bit Windows operating system.

RULE 4.1.12

The VISA Shared Components installer **SHALL** require that Internet Explorer 5.01 or higher already be installed.

PERMISSION 4.1.1

Vendors that include the VISA Shared Components installer in their distributions **MAY** place additional restrictions on the number of operating systems, service packs, or versions of Internet Explorer with which their distributions are compatible.

RULE 4.1.13

The VISA Shared Components installer **SHALL** install 64-bit VISA shared components, directories, registry keys, and environment variables, as necessary to enable development of 64-bit VISA and VISA COM applications.

RULE 4.1.14

The VISA Shared Components installer **SHALL NOT** include the .NET framework installer and **SHALL NOT** require that the .NET framework already be installed.

RULE 4.1.15

The VISA Shared Components installer **SHALL** create the VXIPNPPATH environment variable with a value of <VXIPNPPATH>.

RULE 4.1.16

The VISA Shared Components installer **SHALL** require that the user has administrative privileges.

RULE 4.1.17

The VISA Shared Components installer **SHALL** require the user to accept the IVI Foundation license.

RULE 4.1.18

The VISA Shared Components installer **SHALL** provide command line options to:

- Run silently (/q)
msiexec.exe /i <PathToMSI> /q
- Set VXIPNPPATH
msiexec.exe /i <PathToMSI> VXIPNPPATHDIR=<CustomVXIPNPPath>
- Repair the installation (/f)
msiexec.exe /f <PathToMSI>

RULE 4.1.19

The VISA Shared Components uninstaller **SHALL** search for the following dependent software:

- Vendor Specific Resource Managers
- IVI Shared Components
- VISA.NET Shared Components

IF dependent software is found, the uninstaller **SHALL** present a warning dialog to the user **AND** the dialog's default response **SHALL** be Cancel.

OBSERVATION 4.1.4

Even if dependent software is detected, a user still will be allowed to override the default and uninstall the VISA Shared Components. This is important for corner cases such as downgrading and cleaning up systems where some other component's uninstallation failed.

RULE 4.1.20

The VISA Shared Components uninstaller **SHALL** detect, after removing the files it installed, whether any remaining files or nonempty folders remain. **IF** no files remain, **THEN** the VISA Shared Components uninstaller **SHALL** remove the entire VXIplug&play directory structure, registry keys, and environment variable.

OBSERVATION 4.1.5

If the VISA Shared Components uninstaller detects remaining files, it leaves the VXIplug&play directory structure, registry keys, and environment variable intact. This is a "leak," but it is unavoidable.

RULE 4.1.21

On Windows 7 32-bit and higher, the VISA Shared Components installer, if invoked in dialog mode without admin privileges, **SHALL** prompt for elevation. If the installer is invoked in silent mode without admin privileges, a failure condition exists and the installer **SHALL** abort.

RULE 4.1.22

On Windows 7 32-bit and higher, the VISA Shared Components installer **SHALL** set the attributes of the VXIplug&play directory to allow modification without admin privileges. (This is an interim solution that will be reverted when the IVI Foundation specifies a revised directory structure that avoids placing writable files in the Program Files directory. Therefore, drivers and applications should not rely on users having write access to the VXIplug&play directory.)

RULE 4.1.23

The USBTMC kernel binary (ausbtmc.sys) and its associated files (ausbtmc.cat, ausbtmc.inf) **SHALL** be installed using the DIFx package provided by Microsoft. Please visit MSDN for more details regarding DIFx and device driver installation in general.

4.2. Installing VISA Shared Components On 64-Bit OS's

TERMS

The following terms are used in this section.

- `<SYSTEM32DIR>` is the Windows system directory for 64-bit executables. The default is `C:\Windows\System32`, but this may be changed when Windows is installed.
- `<SYSWOW64DIR>` is the Windows system directory for 32-bit executables. The default is `C:\Windows\SysWOW64`, but this may be changed when Windows is installed.
- `<PROGRAMFILES>` is the Windows Program Files directory. The default is `C:\Program Files`, but this may be changed when Windows is installed.
- `<PROGRAMFILESx86>` is the Windows Program Files (x86) directory. The default is `C:\Program Files (x86)`, but this may be changed when Windows is installed.
- `<ALLUSERSAPPDATA>` is the Windows directory where application data that is accessible to all users is stored (as opposed to data that is accessible to the current user).
- `<VXIPNPPATH>` is the target directory for the 32-bit VISA components.
- `<VXIPNPPATH64>` is the target directory for the 64-bit VISA components.
- `<VISADATAPATH>` is the directory used for VISA data files, and the conflict resolution table in particular.

RULE 4.2.1

The VISA Shared Components installer for 64-bit versions of the Windows operating system **SHALL** be named *VISA Shared Components 64-Bit* and **SHALL** have its own entry in the Windows Add/Remove Programs list.

RULE 4.2.2

Every 64-bit vendor-specific VISA installer **SHALL** use the VISA Shared Components 64-bit installer to install shared VISA COM or VISA Plug-In Architecture files, and to create standard VISA directories, registry keys and values, or environment variables, and **SHALL NOT** install the components or create standard VISA directories, registry keys and values, or environment variables as part of the vendor-specific installer.

RULE 4.2.3

The VISA Shared Components 64-Bit installer **SHALL** install both the 32-bit VISA shared components, directories, registry keys, and environment variables, and the 64-bit VISA shared components, directories, registry keys, and environment variables.

OBSERVATION 4.2.1

The above rule serves several purposes. First, it eliminates problems that the 32-bit installer would have registering 32-bit type libraries on both 32-bit and 64-bit operating systems. Second, it minimizes potential issues with interactions between the 32-bit and 64-bit installers on 64-bit operating systems. Third, it supports cross-developing 32-bit and 64-bit executables on 64-bit operating systems.

RULE 4.2.4

The value of <VXIPNPPATH64>:.

- **IF** the registry key HKLM\SOFTWARE\VXIPNP_Alliance\VXIPNP\CurrentVersion exists and contains the value VXIPNPPATH, and this value designates a directory that exists, and the directory and all subdirectories do not contain any files, **THEN** the value of <VXIPNPPATH64> **SHALL** be the value of the this key's VXIPNPPATH string value.
- **OTHERWISE**, the default value of <VXIPNPPATH64> **SHALL** be <PROGRAMFILES>\IVI Foundation\VISA, **AND** the VISA Shared Components installer **SHALL** allow the user to change the value of <VXIPNPPATH64>.

RULE 4.2.5

The value of <VXIPNPPATH>:.

- **IF** the registry key HKLM\SOFTWARE\Wow6432Node\VXIPNP_Alliance\VXIPNP\CurrentVersion exists and contains the value VXIPNPPATH, and this value designates a directory that exists, and the directory and all subdirectories do not contain any files, **THEN** the value of <VXIPNPPATH> **SHALL** be the value of the this key's VXIPNPPATH string value.
- **OTHERWISE**, the default value of <VXIPNPPATH> **SHALL** be <PROGRAMFILESx86>\IVI Foundation\VISA, **AND** the VISA Shared Components installer **SHALL** allow the user to change the value of <VXIPNPPATH>.

OBSERVATION 4.2.2

The above rules dictate that the 64-bit installer detect existing VISA components and other *VXIplug&play* components such as instrument drivers.

- If the 32-bit *VXIplug&play* registry keys point to a 32-bit *VXIplug&play* root directory, and that directory exists, but no files exist in the hierarchy to which they point, it is acceptable for the VISA Shared Components installer to allow the user to change the 32-bit VISA base directory. In this case, the VISA Shared Components installer changes the 32-bit *VXIplug&play* registry keys and the environment variable if necessary, but it leaves the old, empty 32-bit directory hierarchy alone (that is, it does not remove it).
- If the 64-bit *VXIplug&play* registry keys point to a 64-bit *VXIplug&play* root directory, and that directory exists, but no files exist in the hierarchy to which they point, it is acceptable for the VISA Shared Components installer to allow the user to change the 64-bit VISA base directory. In this case, the VISA Shared Components installer changes the 64-bit *VXIplug&play* registry keys and the environment variable if necessary, but it leaves the old, empty 64-bit directory hierarchy alone (that is, it does not remove it).

RULE 4.2.6

The VISA Shared Components 64-Bit installer **SHALL** install 64-bit executables to 64-bit directories and 32-bit executables to the appropriate WOW64 directory. If the user specifies a directory that would result in 32-bit shared components being installed in a 64-bit only directory, the installer **SHALL** redirect the path to the corresponding WOW64 32-bit directory.

RULE 4.2.7

The value of <VISADATAPATH> **SHALL** be <ALLUSERSAPPDATA>\IVI Foundation\VISA.

OBSERVATION 4.2.3

<VISADATAPATH> is the same for both 32-bit and 64-bit components.

RULE 4.2.8

The VISA Shared Components 64-Bit installer **SHALL** create any of the following directories that do not already exist:

- <VXIPNPPATH>
- <VXIPNPPATH>\VisaCom
- <VXIPNPPATH>\VisaCom\Primary Interop Assemblies
- <VXIPNPPATH>\WinNT
- <VXIPNPPATH>\WinNT\Bin
- <VXIPNPPATH>\WinNT\include
- <VXIPNPPATH>\WinNT\lib
- <VXIPNPPATH>\WinNT\lib\bc
- <VXIPNPPATH>\WinNT\lib\msc
- <VXIPNPPATH>\WinNT\lib_x64
- <VXIPNPPATH>\WinNT\lib_x64\msc
- <VXIPNPPATH64>
- <VXIPNPPATH64>\VisaCom64
- <VXIPNPPATH64>\VisaCom64\Primary Interop Assemblies
- <VXIPNPPATH64>\Win64
- <VXIPNPPATH64>\Win64\Bin
- <VXIPNPPATH64>\Win64\include
- <VXIPNPPATH64>\Win64\lib_x64
- <VXIPNPPATH64>\Win64\lib_x64\msc
- <VISADATAPATH>

RULE 4.2.9

The VISA Shared Components 64-Bit installer **SHALL** install the following files, unless newer versions of the files are already installed:

- <SYSWOW64DIR>\visaConfMgr.dll (32-bit executable)
- <VXIPNPPATH>\VisaCom\GlobMgr.dll (32-bit executable)
- <VXIPNPPATH>\VisaCom\BasFrmIO.dll (32-bit executable)
- <VXIPNPPATH>\VisaCom\Primary Interop Assemblies\Ivi.Visa.Interop.dll (32-bit executable)
- <VXIPNPPATH>\VisaCom\Primary Interop Assemblies\Ivi.Visa.Interop.xml
- <VXIPNPPATH>\VisaCom\Primary Interop Assemblies\Ivi.Visa.Interop.config
- <VXIPNPPATH>\VisaCom\Primary Interop Assemblies\Policy.X.X.Ivi.Visa.Interop.dll (32-bit executable, one for each major/minor previous version that needs to be redirected to the current version of Ivi.Visa.Interop.dll. X.X stand for the previous major/minor versions being redirected.)
- <VXIPNPPATH>\VisaCom\Primary Interop Assemblies\IviPiaRegistration.bat
- <VXIPNPPATH>\WinNT\include\visa.h

- <VXIPNPPATH>\WinNT\include\visatype.h
- <VXIPNPPATH>\WinNT\lib_x64\msc\visa64.lib
- <SYSTEM32DIR>\visa64.dll (64-bit executable)
- <SYSTEM32DIR>\visaConfMgr.dll (64-bit executable)
- <SYSTEM32DIR>\visaUtilities.dll (64-bit executable)
- <VXIPNPPATH64>\VisaCom64\GlobMgr.dll (64-bit executable)
- <VXIPNPPATH64>\VisaCom64\BasFrmIO.dll (64-bit executable)
- <VXIPNPPATH64>\VisaCom64\Primary Interop Assemblies\Ivi.Visa.Interop.dll (64-bit executable)
- <VXIPNPPATH64>\VisaCom64\Primary Interop Assemblies\Ivi.Visa.Interop.xml
- <VXIPNPPATH64>\VisaCom64\Primary Interop Assemblies\Ivi.Visa.Interop.config (Needed only if a policy file also is being installed.)
- <VXIPNPPATH64>\VisaCom64\Primary Interop Assemblies\IviPiaRegistration64.bat
- <VXIPNPPATH>\VisaCom64\Primary Interop Assemblies\Policy.X.X.Ivi.Visa.Interop.dll (64-bit executable, one for each major/minor previous version that needs to be redirected to the current version of Ivi.Visa.Interop.dll. X.X stand for the previous major/minor versions being redirected.)
- <VXIPNPPATH>\Win64\include\visa.h
- <VXIPNPPATH>\Win64\include\visatype.h
- <VXIPNPPATH>\Win64\lib_x64\msc\visa64.lib
- ausbtmc.cat (location determined by Windows during driver installation process)
- ausbtmc.inf (location determined by Windows during driver installation process)
- ausbtmc.sys (64-bit kernel driver file, location determined by Windows during driver installation process)

RULE 4.2.10

The VISA Shared Components installer **SHALL NOT** replace an existing conflict resolution table file.

RULE 4.2.11

The VISA Shared Components 64-Bit installer **SHALL** first create registry entries for the 32-bit and 64-bit versions of `GlobMgr.dll` and `BasFrmIO.dll` as if the 32-bit version of `regsvr32` was run on the 32-bit versions and then the 64-bit version of `regsvr32` was run on the 64-bit versions. It **SHALL NOT** self-register these components.

OBSERVATION 4.2.4

Note that the registry entries created by self-registering 32-bit COM type libraries on a 32-bit operating system are different than the registry entries created by self-registering the same 32-bit type library on a 64-bit system. These changes are more extensive than just relocating keys to the WOW64 registry hive—the number of subkeys is actually different. If the 32-bit installer creates registry entries that match the self-registration entries on a 32-bit operating system, they will not match the self-registration entries on a 64-bit operating system, and vice versa. This is one reason why the 64-bit installer installs only on 64-bit operating systems.

RULE 4.2.12

The VISA Shared Components 64-Bit installer **SHALL** first create registry entries for the 32-bit and 64-bit versions of `Ivi.Visa.Interop.dll`, the Primary Interop Assembly (PIA), as if the 32-bit version of `regasm` was run on the 32-bit version, and then the 64-bit version of `regasm` was run on the 64-bit version. It **SHALL NOT** run `regasm` from within the installer. The resulting keys and values created **SHALL** be equivalent to running the following command, once for 32-bit and then for 64-bit:

```
regasm <PIA>
```

OBSERVATION 4.2.5

Prior to version 5.4, this specification did not specify the exact `regasm` command that would be equivalent to the registration done by the VISA Shared Components installer. However, the installers compliant to those versions of this specification register the PIA in a manner that is equivalent to running the following command, once for 32-bit and then for 64-bit:

```
regasm <PIA> /codebase
```

`Regasm` with the `/codebase` option adds the `PrimaryInteropAssemblyCodeBase` registry value to the COM type library registry key and the `CodeBase` registry value to subkeys of the `CLSID\<GUID>` key for classes defined in the type library. The issue is with the type library PIA codebase value, however. This registry value is the physical PIA file path of a PIA for the COM type library. In the case of the VISA shared components, the value prior to 2014 was the location of the 32-bit version of the PIA. This worked for both 32-bit and 64-bit development in Visual Studio 2005 and 2008. However, this prevents 64-bit development in Visual Studio 2010, 2012, and 2013 because these versions of Visual Studio require references to a PIA of the same bitness as the current project type.

As a result, the type library `PrimaryInteropAssemblyCodeBase` registry value will no longer be added to VISA shared component type library keys with the release of version 5.4 of the VISA Shared Components. Without this key, all versions of Visual Studio resolve references to the PIA by using the GAC. This means that the reference is automatically resolved with the correct bitness of PIA from the GAC. However, the PIA IntelliSense help files are not installed to the GAC, with the resulting downside that this help is not available to developers in Visual Studio IntelliSense. In addition, the `CodeBase` registry value to subkeys of the `CLSID\<GUID>` key for classes defined in the type library will also be omitted. The net effect is that the VISA Shared Components now register the PIA with the corresponding COM type library and COM classes in a manner that is equivalent to running the following command:

```
regasm <PIA>
```

RULE 4.2.13

The VISA Shared Components 64-Bit installer **SHALL** create the following additional registry keys and values under `HKLM\SOFTWARE`:

- `VXIPNP_Alliance`
- `VXIPNP_Alliance\IVIVISACOM`
- `VXIPNP_Alliance\IVIVISACOM\CurrentVersion`
 - Value: `InstallerVersion`—The version of the VISA Shared Components installer.
 - Value: `Version`—The product version of `GlobMgr.dll`
- `VXIPNP_Alliance\VXIPNP`
- `VXIPNP_Alliance\VXIPNP\CurrentVersion`
 - Value: `FRAMEWORK_PATH`—`<VXIPNPPATH64>\Win64`
 - Value: `VXIPNPPATH`—`<VXIPNPPATH64>`

The VISA Shared Components 64-Bit installer **SHALL** create the following additional registry keys and values under HKLM\SOFTWARE\Wow6432Node:

- VXIPNP_Alliance
- VXIPNP_Alliance\IVIVISACOM
- VXIPNP_Alliance\IVIVISACOM\CurrentVersion
 - Value: InstallerVersion—The version of the VISA Shared Components installer.
 - Value: Version—The product version of GlobMgr.dll
- VXIPNP_Alliance\VXIPNP
- VXIPNP_Alliance\VXIPNP\CurrentVersion
 - Value: FRAMEWORK_PATH—<VXIPNPPATH>\WinNT
 - Value: VXIPNPPATH—<VXIPNPPATH>

RULE 4.2.14

The VISA Shared Components 64-Bit installer **SHALL** install on Windows 7 64-bit, Windows 8 64-bit, and Windows 10 64-bit. Service Pack requirements may vary by version of the installer, and are documented on the IVI web site for each version of the installer. The VISA Shared Components 64-Bit installer **SHALL NOT** install on any 32-bit Windows operating system.

PERMISSION 4.2.1

Vendors that include the VISA Shared Components 64-Bit installer in their distributions **MAY** place additional restrictions on the number of operating systems, service packs, or versions of Internet Explorer with which their distributions are compatible.

RULE 4.2.15

The VISA Shared Components 64-Bit installer **SHALL NOT** include the .NET framework installer and **SHALL NOT** require that the .NET framework already be installed.

RULE 4.2.16

The VISA Shared Components 64-Bit installer **SHALL** create the VXIPNPPATH environment variable with a value of <VXIPNPPATH> and the VXIPNPPATH64 environment variable with a value of <VXIPNPPATH64>.

RULE 4.2.17

The VISA Shared Components 64-Bit installer **SHALL** require that the user has administrative privileges.

RULE 4.2.18

The VISA Shared Components 64-Bit installer **SHALL** require the user to accept the IVI Foundation license.

RULE 4.2.19

The VISA Shared Components 64-Bit installer **SHALL** provide command line options to:

- Run silently (/q)
msiexec.exe /i <PathTo64-BitMSI> /q
- Set VXIPNPPATH64 and VXIPNPPATH
msiexec.exe /i <PathTo64-BitMSI> VXIPNPPATH64DIR=<Custom64-BitVXIPNPPath> VXIPNPPATHDIR=<CustomVXIPNPPath>
- Repair the installation (/f)
msiexec.exe /f <PathTo64-BitMSI>

RULE 4.2.20

The VISA Shared Components 64-Bit uninstaller **SHALL** search for the following dependent software:

- Vendor Specific Resource Managers
- IVI Shared Components
- VISA.NET Shared Components

IF dependent software is found, the uninstaller **SHALL** present a warning dialog to the user, **AND** the dialog's default response **SHALL** be Cancel.

OBSERVATION 4.2.6

Even if dependent software is detected, a user still will be allowed to override the default and uninstall the VISA Shared Components. This is important for corner cases such as downgrading and cleaning up systems where some other component's uninstallation failed.

RULE 4.2.21

The VISA Shared Components 64-Bit uninstaller **SHALL** detect, after removing the files it installed, whether any remaining files or non-empty folders remain. **IF** no files remain, **THEN** the VISA COM uninstaller **SHALL** remove the entire VXIplug&play directory structure and registry keys.

OBSERVATION 4.2.7

If the VISA Shared Components 64-Bit uninstaller detects remaining files, it leaves the VXIplug&play directory structure and registry keys intact. This is a "leak," but it is unavoidable.

RULE 4.2.22

If the VISA Shared Components 64-Bit installer is invoked in dialog mode without admin privileges, it **SHALL** prompt for elevation. If the installer is invoked in silent mode without admin privileges, a failure condition exists, and the installer **SHALL** abort.

RULE 4.2.23

On 64-bit versions of the Windows operating system, the VISA Shared Components installer **SHALL** set the attributes of the VXIplug&play directory to allow modification without admin privileges. (This is an interim solution that will be reverted when the IVI Foundation specifies a revised directory structure that avoids placing writable files in the Program Files directory. Therefore, drivers and applications should not rely on users having write access to the VXIplug&play directory.)

RULE 4.2.24

The USBTMC 64-bit kernel binary (ausbtmc.sys) and its associated files (ausbtmc.cat, ausbtmc.inf) **SHALL** be installed using the DIFx package provided by Microsoft. Please visit MSDN for more details regarding DIFx and device driver installation in general.

4.3. Installing VISA .NET Shared Components

TERMS

The following terms are used in this section.

- `<GAC>` is the Windows system directory for the Global Assembly Cache.
- `<VXIPNPPATH>` is the target directory for the 32-bit VISA components. See sections 4.1 and 4.2 for the default value and relevant constraints.
- `<VXIPNPPATH64>` is the target directory for the 64-bit VISA components. See section 4.2 for the default value and relevant constraints.

PREREQUISITES

The following software must be installed before the VISA.NET Shared Components are installed.

- .NET Framework 2.0 or higher. VISA.NET is based on this version of the .NET Framework. The .NET Framework Client Profile is not sufficient to meet this condition.
- VISA Shared Components. VISA.NET is installed under base directories created by the VISA Shared Components. The VISA Shared Components installer also includes the Conflict Resolution Manager which is required by the VISA.NET Global Resource Manager.

RULE 4.3.1

The VISA.NET Shared Components installer **SHALL** require that the prerequisites are installed before making any modifications to the install PC.

OBSERVATION 4.3.1

The VISA.NET Shared Components installer does not install the .NET Framework 2.0 (or higher) or the VISA Shared Components.

VISA.NET Assembly Name and Location

The VISA.NET Shared Components install a single .NET assembly that includes all of the common API, the standard implementation of `IMessageBasedFormattedIO`, and the VISA.NET Global Resource Manager (GRM). The installer may also include policy files that redirect references to previous versions of the VISA.NET assembly to the current version.

RULE 4.3.2

The 32-bit version of the VISA.NET Shared Components installer **SHALL** support the exact set of Operating Systems that are supported by the 32-bit version of the VISA Shared Components installer, as described in RULE 4.1.11.

RULE 4.3.3

The 64-bit version of the VISA.NET Shared Components installer **SHALL** support the exact set of Operating Systems that are supported by the 64-bit version of the VISA Shared Components installer, as described in RULE 4.2.14.

RULE 4.3.4

On 32-bit and 64-bit Windows operating systems, the VISA.NET Shared Components installer **SHALL** create the directory “`<VXIPNPPATH>\Microsoft.NET\Framework32\<FrameworkVersion>\VISA.NET Shared Components <version>`”, where `<version>` is the version of the VISA.NET Shared Components installer. This directory is known as the *version-specific 32-bit VISA.NET install directory*.

RULE 4.3.5

On 64-bit Windows operating systems, the VISA.NET Shared Components installer **SHALL** create the directory “<VXIPNPPATH64>\Microsoft.NET\Framework64\<FrameworkVersion>\VISA.NET Shared Components <version>\”, where <version> is the version of the VISA.NET Shared Components installer. This directory is known as the *version-specific 64-bit VISA.NET install directory*.

RULE 4.3.6

The VISA.NET assembly **SHALL** be named Ivi.Visa.dll. The format of the product, file, and assembly versions of this assembly **SHALL** be <MajorVersion>.<MinorVersion>.<Build>.0 (e.g. <Revision> is always 0).

RULE 4.3.7

On 32-bit and 64-bit Windows operating systems, the VISA.NET Shared Components installer **SHALL** install the Ivi.Visa.dll assembly into the version-specific 32-bit VISA.NET install directory.

RULE 4.3.8

On 64-bit Windows operating systems, the VISA.NET Shared Components installer **SHALL** install the Ivi.Visa.dll assembly into the version-specific 64-bit VISA.NET install directory.

RULE 4.3.9

Policy file names **SHALL** conform to Microsoft conventions. The VISA.NET Shared Components installer **SHALL** install any related Policy files into the version-specific VISA.NET install directory.

RULE 4.3.10

The VISA.NET Shared Components installer **SHALL** install the VISA.NET assembly and any related policy files into the Global Assembly Cache (GAC).

VISA.NET Shared Components Installer Version**RULE 4.3.11**

The major and minor fields of the VISA.NET Shared Components installer version **SHALL** be identical to the major and minor fields of the version of the VISA.NET assembly.

Uninstalling the VISA.NET Shared Components**RULE 4.3.12**

The VISA.NET Shared Components uninstaller **SHALL** search for the following dependent software:

- Vendor Specific Resource Managers

IF dependent software is found, the uninstaller **SHALL** present a warning dialog to the user **AND** the dialog’s default response **SHALL** be Cancel.

OBSERVATION 4.3.2

Even if dependent software is detected, a user still will be allowed to override the default and uninstall the VISA.NET Shared Components. This is important for corner cases such as downgrading and cleaning up systems where some other component’s uninstallation failed.

Appendix A: Implementation Files

A.1 Contents of the visaRouter.h File

```

/*****
Distributed by IVI Foundation Inc.

Do not modify the contents of this file.
-----

Title   : visaRouter.h
Date    : 08-12-2008
Purpose : Define VISA attributes that apply to the VISA router.

*****/

#ifndef __VISAROUTER_HEADER__
#define __VISAROUTER_HEADER__

#define VI_ATTR_UNLOAD_PLUGINS_IF_LAST_RM (0x3FFF018CUL) // ViBoolean
#define VI_ATTR_UNDERLYING_VISA_SESSION   (0x3FFFA000UL) // ViSession
#define VI_ATTR_MULTI_SPEC_VERSION        (0x3FFFA001UL) // ViVersion
#define VI_ATTR_MULTI_MANF_NAME           (0x3FFFA002UL) // ViString
#define VI_ATTR_MULTI_MANF_ID             (0x3FFFA003UL) // ViUInt16
#define VI_ATTR_MULTI_IMPL_VERSION        (0x3FFFA004UL) // ViVersion

#endif // __VISAROUTER_HEADER__

```

A.2 Contents of the ConflictMgr.h File

```

/*-----*/
/* Distributed by IVI Foundation Inc. */
/* Do not modify the contents of this file. */
/*-----*/
/*
/* Title   : visaConflictMgr.h
/* Date    : 06-19-2014
/* Purpose : Include file for the VISA Conflict Resolution Manager
/*
/*-----*/

#ifndef VISACM_CON_MGR
#define VISACM_CON_MGR

#include "visatype.h"

/* these enumerations are for conflict handler types */
#define VISACM_HANDLER_NOT_CHOSEN 0
#define VISACM_HANDLER_CHOSEN_BY_RSRC_MGR 1
#define VISACM_HANDLER_CHOSEN_BY_USER 2

/* these enumerations are for api types */
#define VISACM_API_C_AND_COM 0
#define VISACM_API_DOTNET 1

```

```

/* these enumerations are for FlushConflictFile */
#define VISACM_FLUSH_OVERWRITE_ALWAYS 0
#define VISACM_FLUSH_WRITE_IF_UNCHANGED 1
#define VISACM_FLUSH_WRITE_OR_RELOAD 2

#define VISACM_STRING_SIZE 256
#define VISACM_GUID_STRING_SIZE 39 /* 32 hex digits + 4
dashes + null term char + surrounding braces */

#ifdef __cplusplus || defined(__cplusplus__)
extern "C" {
#endif

/* Initialize the DLL for use */
ViStatus _VI_FUNC VISACM_Initialize();

/* Close the DLL after use */
ViStatus _VI_FUNC VISACM_Close();

/* toggle conflict storage */
ViStatus _VI_FUNC VISACM_SetStoreConflictsOnly(ViBoolean storeConflicts); /* in */

/* get storage setting */
ViStatus _VI_FUNC VISACM_GetStoreConflictsOnly(ViPBoolean storeConflicts); /* out */

/* get conflict table filename */
ViStatus _VI_FUNC VISACM_GetConflictTableFilename(ViChar filename[]); /* out */

/* get number of conflict table entries */
/* OBSOLETE, apiType = 0 */
ViStatus _VI_FUNC VISACM_GetResourceCount(ViPInt32 numberRsrcs); /* out */

/* get number of conflict table entries */
ViStatus _VI_FUNC VISACM_GetResourceCount2(
    ViInt16 apiType, /* in */
    ViPInt32 numberRsrcs); /* out */

/* clear all of the resource handlers from the table */
/* OBSOLETE, apiType = 0 */
ViStatus _VI_FUNC VISACM_ClearResourceHandlersFromTable();

/* clear all of the resource handlers from the table */
ViStatus _VI_FUNC VISACM_ClearResourceHandlersFromTable2(ViInt16 apiType); /* in */

/* create a new handler in conflict table */
/* OBSOLETE, apiType = 0 */
ViStatus _VI_FUNC VISACM_CreateHandler(
    ViUInt16 interfaceType, /* in */
    ViUInt16 interfaceNumber, /* in */
    ViConstString sessionType, /* in */
    ViConstString guid_SRM, /* in */
    ViInt16 conflictHandlerType, /* in */
    ViConstString comments); /* in */

/* create a new handler in conflict table */
ViStatus _VI_FUNC VISACM_CreateHandler2(
    ViInt16 apiType, /* in */
    ViUInt16 interfaceType, /* in */
    ViUInt16 interfaceNumber, /* in */
    ViConstString sessionType, /* in */
    ViConstString guid_SRM, /* in */
    ViInt16 conflictHandlerType, /* in */
    ViConstString comments); /* in */

```



```

/* delete a handler by specifying ALL identifying attributes */
/* OBSOLETE, apiType = 0 */
ViStatus _VI_FUNC VISACM_DeleteHandler(
    ViUInt16 interfaceType,          /* in */
    ViUInt16 interfaceNumber,       /* in */
    ViConstString sessionType,      /* in */
    ViConstString guid_SRM);        /* in */

/* delete a handler by specifying ALL identifying attributes */
ViStatus _VI_FUNC VISACM_DeleteHandler2(
    ViInt16 apiType,                /* in */
    ViUInt16 interfaceType,         /* in */
    ViUInt16 interfaceNumber,       /* in */
    ViConstString sessionType,      /* in */
    ViConstString guid_SRM);        /* in */

/* delete all handlers for a specific GUID (for use when VISA fails to load) */
/* OBSOLETE, apiType = 0 */
ViStatus _VI_FUNC VISACM_DeleteHandlerByGUID(ViConstString guid_SRM); /* in */

/* delete all handlers for a specific GUID (for use when VISA fails to load) */
ViStatus _VI_FUNC VISACM_DeleteHandlerByGUID2(
    ViInt16 apiType,                /* in */
    ViConstString guid_SRM);        /* in */

/* delete handler by index */
/* OBSOLETE, apiType = 0 */
ViStatus _VI_FUNC VISACM_DeleteResourceByIndex(ViInt32 resourceIndex); /* in */

/* delete handler by index */
ViStatus _VI_FUNC VISACM_DeleteResourceByIndex2(
    ViInt16 apiType,                /* in */
    ViInt32 resourceIndex);         /* in */

/* find a handler for a interface number, type, and session type */
/* OBSOLETE, apiType = 0 */
ViStatus _VI_FUNC VISACM_FindChosenHandler(
    ViUInt16 interfaceType,          /* in */
    ViUInt16 interfaceNumber,       /* in */
    ViConstString sessionType,      /* in */
    ViChar guid_SRM[],              /* out */
    ViPInt16 conflictHandlerType);  /* out */

/* find a handler for a interface number, type, and session type */
ViStatus _VI_FUNC VISACM_FindChosenHandler2(
    ViInt16 apiType,                /* in */
    ViUInt16 interfaceType,         /* in */
    ViUInt16 interfaceNumber,       /* in */
    ViConstString sessionType,      /* in */
    ViChar guid_SRM[],              /* out */
    ViPInt16 conflictHandlerType);  /* out */

/* get conflict table entry, mainly used to iterate through all entries */
/* OBSOLETE, apiType = 0 */
ViStatus _VI_FUNC VISACM_QueryResource(
    ViInt32 resourceIndex,           /* in */
    ViPUInt16 interfaceType,         /* out */
    ViPUInt16 interfaceNumber,       /* out */
    ViChar sessionType[],            /* out */
    ViPInt16 numHandlers);           /* out */

/* get conflict table entry, mainly used to iterate through all entries */

```

```

ViStatus _VI_FUNC VISACM_QueryResource2(
    ViInt16 apiType,                /* in */
    ViInt32 resourceIndex,          /* in */
    ViPUInt16 interfaceType,        /* out */
    ViPUInt16 interfaceNumber,      /* out */
    ViChar sessionType[],           /* out */
    ViPInt16 numHandlers);          /* out */

/* get a handler for a specific interface and session type */
/* OBSOLETE, apiType = 0 */
ViStatus _VI_FUNC VISACM_QueryResourceHandler(
    ViInt32 resourceIndex,          /* in */
    ViInt32 handlerIndex,           /* in */
    ViChar guid_SRM[],              /* out */
    ViPInt16 conflictHandlerType,   /* out */
    ViChar comments[]);             /* out */

/* get a handler for a specific interface and session type */
ViStatus _VI_FUNC VISACM_QueryResourceHandler2(
    ViInt16 apiType,                /* in */
    ViInt32 resourceIndex,          /* in */
    ViInt32 handlerIndex,           /* in */
    ViChar guid_SRM[],              /* out */
    ViPInt16 conflictHandlerType,   /* out */
    ViChar comments[]);             /* out */

/* get preferred VISA GUID and name */
/* OBSOLETE, apiType = 0 */
ViStatus _VI_FUNC VISACM_GetVisaPreferred(ViChar guid_SRM[]); /* out */

/* get preferred VISA GUID and name */
ViStatus _VI_FUNC VISACM_GetVisaPreferred2(
    ViInt16 apiType,                /* in */
    ViChar guid_SRM[]);             /* out */

/* set preferred VISA */
/* OBSOLETE, apiType = 0 */
ViStatus _VI_FUNC VISACM_SetVisaPreferred(ViConstString guid_SRM); /* in */

/* set preferred VISA */
ViStatus _VI_FUNC VISACM_SetVisaPreferred2(
    ViInt16 apiType,                /* in */
    ViConstString guid_SRM);        /* in */

/* get the number of VISA libraries installed and registered on the system */
/* OBSOLETE, apiType = 0 */
ViStatus _VI_FUNC VISACM_GetInstalledVisaCount(ViPInt32 numberOfVisas); /* out */

/* get the number of VISA libraries installed and registered on the system */
ViStatus _VI_FUNC VISACM_GetInstalledVisaCount2(
    ViInt16 apiType,                /* in */
    ViPInt32 numberOfVisas);        /* out */

/* get installed VISA by index */
/* OBSOLETE, apiType = 0 */
ViStatus _VI_FUNC VISACM_GetInstalledVisa(
    ViInt32 index,                  /* in */
    ViPUInt16 vendorID,             /* out */
    ViChar guid_SRM[],              /* out */
    ViChar visaPathLocation[],      /* out */
    ViChar visaFriendlyName[],      /* out */
    ViChar comments[]);             /* out */

```

```

/* get installed VISA by index */
ViStatus _VI_FUNC VISACM_GetInstalledVisa2(
    ViInt16 apiType,
    ViInt32 index,
    ViPUInt16 vendorID,
    ViChar guid_SRM[],
    ViChar visaPathLocation[],
    ViChar visaFriendlyName[],
    ViChar comments[]);

/* get if a VISA is enabled */
/* OBSOLETE, apiType = 0 */
ViStatus _VI_FUNC VISACM_GetVisaEnabled(
    ViConstString guid_SRM,
    ViPBoolean enabled);

/* get if a VISA is enabled */
ViStatus _VI_FUNC VISACM_GetVisaEnabled2(
    ViInt16 apiType,
    ViConstString guid_SRM,
    ViPBoolean enabled);

/* set whether a VISA is enabled or disabled */
/* OBSOLETE, apiType = 0 */
ViStatus _VI_FUNC VISACM_SetVisaEnabled(
    ViConstString guid_SRM,
    ViBoolean enabled);

/* set whether a VISA is enabled or disabled */
ViStatus _VI_FUNC VISACM_SetVisaEnabled2(
    ViInt16 apiType,
    ViConstString guid_SRM,
    ViBoolean enabled);

/* flush the table in memory to file */
ViStatus _VI_FUNC VISACM_FlushConflictFile(
    ViInt16 behavior,
    ViPBoolean fileOnDiskWasNewer);

/* get whether any changes have been made to the table since load */
ViStatus _VI_FUNC VISACM_GetIsDirty(ViPBoolean isDirty);

/* reload the file from disk regardless of any changes in memory */
ViStatus _VI_FUNC VISACM_ReloadFile();

/* reset the table to no handlers, no preferred VISA, and no disabled VISAs */
ViStatus _VI_FUNC VISACM_ClearEntireTable();

#ifdef __cplusplus || defined(__cplusplus__)
} /* extern "C" brace */
#endif

#endif /* endif for VISACM_CON_MGR macro */

```

A.3 Contents of the ConflictMgr.def File

```

EXPORTS
    VISACM_GetConflictTableFilename           @1
    VISACM_SetStoreConflictsOnly             @14
    VISACM_GetStoreConflictsOnly             @15
    VISACM_FlushConflictFile                 @17
    VISACM_Initialize                        @18
    VISACM_Close                             @19
    VISACM_GetIsDirty                        @20
    VISACM_ReloadFile                        @23
    VISACM_ClearEntireTable                   @24

; New functions added to support API Type parameter
    VISACM_CreateHandler2                    @25
    VISACM_DeleteHandler2                    @26
    VISACM_DeleteHandlerByGUID2              @27
    VISACM_DeleteResourceByIndex2            @28
    VISACM_FindChosenHandler2                @29
    VISACM_QueryResource2                    @30
    VISACM_QueryResourceHandler2             @31
    VISACM_ClearResourceHandlersFromTable2   @32
    VISACM_GetVisaPreferred2                 @33
    VISACM_SetVisaPreferred2                 @34
    VISACM_GetInstalledVisa2                 @35
    VISACM_GetResourceCount2                 @36
    VISACM_GetInstalledVisaCount2            @37
    VISACM_SetVisaEnabled2                   @38
    VISACM_GetVisaEnabled2                   @39

; Obsolete functions, exported for backwards compatibility
    VISACM_CreateHandler                     @2
    VISACM_DeleteHandler                     @3
    VISACM_DeleteHandlerByGUID               @4
    VISACM_DeleteResourceByIndex             @5
    VISACM_FindChosenHandler                 @6
    VISACM_QueryResource                     @7
    VISACM_QueryResourceHandler              @8
    VISACM_ClearResourceHandlersFromTable    @9
    VISACM_GetVisaPreferred                   @10
    VISACM_SetVisaPreferred                   @11
    VISACM_GetInstalledVisa                   @12
    VISACM_GetResourceCount                   @13
    VISACM_GetInstalledVisaCount              @16
    VISACM_SetVisaEnabled                     @21
    VISACM_GetVisaEnabled                     @22

```

A.4 Contents of the visaUtilities.h File

```

/*****
Distributed by IVI Foundation Inc.

Do not modify the contents of this file.
-----

Title   : visaUtilities.h
Date    : 08-12-2008
Purpose : Define a function to obtain the user 'vi' from the internal 'vi'.
          This can be used to convert the 'vi' is used internally by a
          vendor VISA to the 'vi' that user sees which was returned by the
          VISA router.

*****/

#ifdef VISAUTILITIES_H
#define VISAUTILITIES_H

#ifdef __cplusplus || defined(__cplusplus)
    extern "C" {
#endif

ViSession _VI_FUNC getUserVi(const ViSession underlyingVi, const ViUInt16
underlyingManfId);

#ifdef __cplusplus || defined(__cplusplus)
    }
#endif

#endif // VISAUTILITIES_H

```

A.5 Contents of the visaUtilities.def File

```

LIBRARY      "visaUtilities"

EXPORTS
; Functions used internally by the VISA router
    viTableAdd             @128
    viTableRemove          @129
    viTableLookup          @130
    viTableGetSessionCount @131
    viTableAddToUserViMap  @132
    viTableRemoveFromUserViMap @133

; Functions that can be called externally
    getUserVi              @144

```