IVI-6.3: IVI VISA PXI Plug-in

December 19, 2022
Revision 2.1

Important Information

The IVI-6.3: IVI VISA PXI Plug-in Specification 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.

The IVI Foundation wants to receive your comments on this specification. You can contact the Foundation through the web site at www.ivifoundation.org.

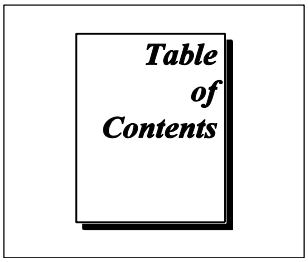
Warranty

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.



IVI VISA PXI Plug-in Revision History

This section is an overview of the revision history of the IVI VISA PXI Plug-in specification.

Table 1. IVI VISA PXI Plug-in Class Specification Revisions

Status	Description
Revision 1.0	First version of specification.
Revision 2.0, October 19, 2018	Major change to add support for Linux
Revision 2.1, December 19, 2022	Change the PXI plug in registry key to reflect the current registry location.

1 Overview of the IVI VISA PXI Plug-in Specification

The IVI VISA plug-in provides a mechanism for various VISA libraries to access custom I/O software for devices that use PXI I/O. The plug-in provides a user-level call that that presumably in turn calls a corresponding kernel driver that has been configured by the operating system based on the hardware installed to interface with the hardware.

Figure 1, *VISA PXI Plug-in mechanism*, shows conceptually how the VISA PXI plug-in fits into the I/O architecture.

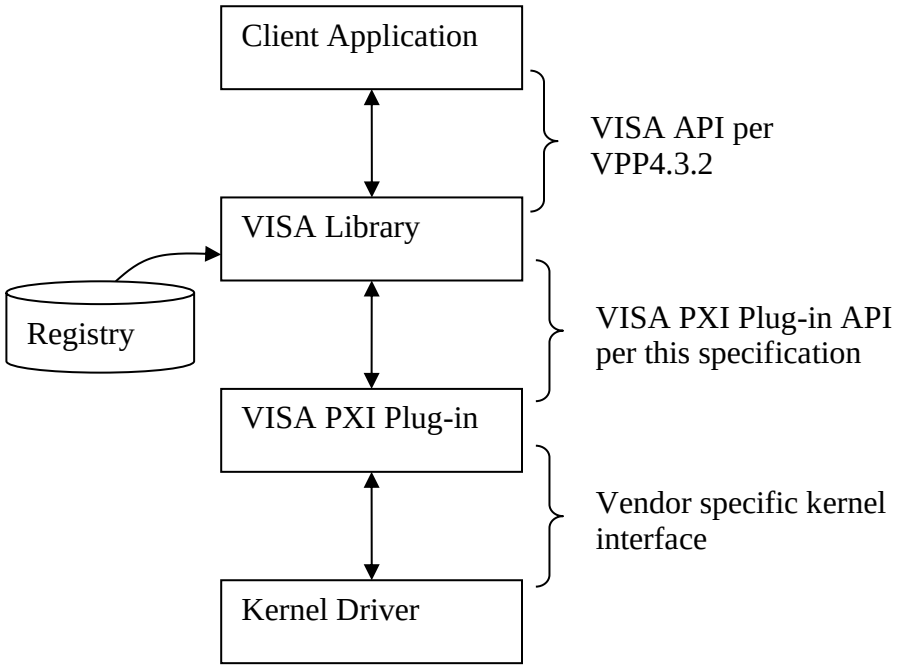


Figure 1 VISA PXI Plug-in mechanism provides standard access to vendor specific kernel drivers

1.1 IVI VISA PXI Plug-in Overview

When a client application calls VISA to find PXI resources or open a PXI session, VISA consults the registry to locate PXI plug-ins. VISA dynamically loads the plug-ins and initializes them, finding the correct plug-in for a device by getting the list of devices found by each plug-in. VISA completes all subsequent I/O to the device using the functions defined in this document. If no plug-in is found, the VISA library may use vendor specific techniques to complete the I/O operations or return an error from the call.

1.2 References

Several other documents and specifications are related to this specification. These other related documents are the following:

Microsoft

- Microsoft Platform SDKs for Windows operating systems
- Microsoft DDKs for Windows operating systems

IVI Foundation (www.ivifoundation.org)

- VPP-4.3 & 4.3.x, The VISA Library and detailed VISA and VISA-COM specifications

PXISA (www.pxisa.org)

- PXI-4, PXI Module Description File Specification

1.3 Definitions of Terms and Acronyms

This section defines terms and acronyms that are specific to the VISA PXI Plug-in definition:

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.

2 PXI Plug-in Requirements

The following sections have general PXI Plug-in requirements.

2.1 Identification of a PXI Plug-in

The following sections describe the registration and identification of PXI plugs-ins on Windows and Linux.

2.1.1 Identification of a PXI Plug-in on Windows

This section describes how a Windows client determines the appropriate plug-in to call for a given PXI device.

Devices provide a plug-in by specifying the plug-in DLL in the registry. To aid in the organization of the registry, the vendor name is included, however that does not necessarily affect the behavior of the client.

The registry default OS bitness key is located at:

```
HKLM\
    Software\
        IVI\
            VisaPxiPlugin\
                <PlugInName>\
```

Modules that support 64-bit systems **SHALL** also provide 32-bit DLL's. On 64-bit systems, the 32-bit registry entry **SHALL** be:

```
HKLM\
    Software \
        Wow6432Node \
            IVI\
                VisaPxiPlugin\
                    <PlugInName>\
```

The *PlugInName* key **SHALL** include the vendor name, concatenated with a plug-in-specific string to make the *PlugInName* unique.

The required registry values to write to this key are listed in Table 2, *VISA PXI Plug-in registry entries*.

Table 2 VISA PXI Plug-in registry entries

Registry Entry	Type	Content
Library	String	Full path of the DLL
SpecVersion	String	Version of this specification to which the DLL complies, in the format "major.minor". Currently this **SHALL** be "1.0".

2.1.2 Identification of a PXI Plug-in on Linux

This section describes how a Linux client determines the appropriate plug-in to call for a given Linux device. Note that this specification only supports 64-bit Linux.

Devices provide a plug-in by specifying the absolute path to the plug-in SO (shared object) in an .INI file. The vendor name is embedded in the filename, but that does not affect the behavior of the client.

The filename **SHALL** be a unique name, chosen by the vendor, that incorporates the vendor name along with any other elements, at the vendor's discretion, needed to ensure the name is unique. The filename **shall** have a .ini suffix. A separate file is provided for each PXI plug-in registered.

INI File location: the file is placed in the <*PXIPLUGINREGPATH*> directory as defined in Section 4.3 *Installing VISA Shared Components on Linux Operating Systems*, of *VPP-4.3.5 VISA Shared Components*. The *owner:group* of the file **SHALL** be *root:root*. The permissions **SHALL** be set to 644 (that is, read-write by owner and read-only by group and others).

The file is a text file. The syntax is a basic INI file syntax and **SHALL** contain the following name value pairs with the syntax shown here:

```
[DEFAULT]
Library="<path to the .so file>"
SpecVersion=2.0
```

Where:

<*path to the .so file*>	This is the absolute path to the shared object file that provides the API. Relative paths are not supported

Note that the spec version refers to the version of this document with the lowest numeric specification version that is code compatible with the API implementation.

2.2 VISA Behavior When Multiple Plug-ins Support the Same Module

Multiple plug-ins may return information about the same module, this is normal behavior. The following define the appropriate behavior for the plug-in client (which is nominally VISA):

- If a single plug-in returns that it is the primary plug-in, the client should use it.
- If no plug-in returns that it is the primary plug-in, but several plug-ins support a module, the client is permitted to use a vendor specific algorithm to choose one.
- If multiple plug-ins return that they are primary, the client is permitted to use a vendor specific algorithm to choose one or flag an error.

2.3 VISA PXI Plug-in API

The PXI Plug-in API **SHALL** implement all of the functions as defined below.

The DLL function calls are all defined as *stdcall* for Windows 32-bit.

3 API Definition

The following sections define the required API.

3.1 PpiInitializePlugin

A client should call this plug-in function first, for example, immediately after loading the plug-in. A plug-in **SHALL** implement this by doing whatever one-time (per-process) initialization is needed. Since a plug-in may be loaded by multiple clients in the same process, it **SHALL** reference count how many times this function has been called, so it does not prematurely clean up when PpiFinalizePlugin() is called. In particular, the plug-in **SHALL NOT** do any work on any calls to PpiInitializePlugin() except the first one.

If a plug-in returns an error from this function, a client should not make any further calls to that plug-in.

If a client does not call this function, the behavior is undefined. The plug-in is permitted to either return an error from other functions or perform its one-time (per-process) initialization lazily.

```
ViStatus PpiInitializePlugin();
```

Parameters

None.

Return Value

If the operation completes successfully, the return value is VI_SUCCESS.

If the operation fails the return value is less than zero (VI_SUCCESS). If the operation fails an appropriate VISA error code is returned.

3.2 PpiGetDeviceIDs

Get the PCI device information for the devices supported by this DLL that have been enumerated by the OS.

A plug-in **SHALL** do whatever is necessary to ensure any cache is up-to-date, so the results of this call are always valid at the time the call is made. For example, if a configuration file was updated in another process, it **SHALL** re-initialize its state information to prevent it from returning invalid data. Calling this function must not invalidate any existing session handles, even if the device to which that handle refers is no longer returned by this function.

If called with *includeNonPrimary* as VI_TRUE, it is possible for the same device to be returned from multiple plug-ins. Not more than 1 plug-in should ever return isPrimaryArray[] of VI_TRUE for a given device; otherwise, system behavior and reproducibility are not defined.

```
ViStatus PpiGetDeviceIDs(
  __in        ViBoolean     includeNonPrimary,
  __in        ViInt32       arrayElementCount,
  __in, out   ViAUInt64     deviceIdArray,
  __in, out   ViABoolean    isPrimaryArray,
  __out       ViPInt32      deviceCount
);
```

Parameters

includeNonPrimary[in]	Whether the caller is interested in getting back information about devices for which this plug-in is not the actual driver.
arrayElementCount [in]	The maximum number of PpiDevInfo elements in the user-allocated array.
deviceIdArray [in, out]	A user-allocated array for holding the device IDs. For each element in the array:
	Fourth Word (most sig) Interface
	Third Word Bus Number
	Second Word PCI Device Number
	First Word (least sig) PCI Function Number
isPrimaryArray[in, out]	For each device returned in deviceIdArray, this specifies whether this plug-in is the actual driver for it; can be NULL if *includeNonPrimary* is VI_FALSE.
deviceCount [in, out]	The count of device IDs detected by this plug-in.

Return Value

If the operation completes successfully, the return value is VI_SUCCESS.

If the operation fails the return value is less than zero (VI_SUCCESS). If the operation fails an appropriate VISA error code is returned.

If the plug-in finds more devices than arrayElementCount, it must return VI_ERROR_INV_LENGTH and set the number of devices actually found in the deviceCount output parameter. The plug-in **SHALL NOT** write any values to the deviceIdArray or isPrimaryArray outputs in this case.

3.3 PpiOpen

Open a handle to a modular instrument. The combination of intfc, bus, device, and function uniquely identify the modular instrument.

A plug-in **SHALL** do whatever is necessary to ensure any cache is up-to-date, so the results of this call are always valid at the time the call is made. This does not necessarily mean that the plug-in must always re-initialize the cache even if a configuration file was updated in another process; as long as the device existed before in the cache and still exists, then as long as the plug-in can access it, this function can succeed. For example, if a plug-in supports hot-swap devices, it must be able to access a device that was plugged in after the cache was initially established. But across multiple calls to this function where the configuration does not change, it is reasonable to not update the cache unnecessarily, for performance reasons.

```
ViStatus PpiOpen(
  __in        ViInt32       intfc,
  __in        ViInt32       bus,
  __in        ViInt32       device,
  __in        ViInt32       function,
  __out       PpiHandle*    handle
);
```

Parameters

intfc [in]	This is the 0-based PCI/PCIe interface # for the device.
bus [in]	Bus number for the device.
device [in]	Device number for the device.
function [in]	Function number for the device.
handle [in, out]	A pointer to the handle to be returned. If the function succeeds, *handle is a session handle to the device to be used in VISA PXI Plug-in API calls. If the function fails, the plug-in **SHALL** ensure *handle = 0.

Return Value

If the operation completes successfully, the return value is VI_SUCCESS.

If the operation fails the return value is less than zero (VI_SUCCESS). If the operation fails an appropriate VISA error code is returned.

3.4 PpiGetSpaceInfo

Get the PCI device information for the device corresponding to this *handle*.

If called for a space not used by this device, the implementation **SHALL** set the three output parameters to 0 and return VI_SUCCESS. If the space does not correspond to a valid BAR index (for example, the configuration space) then an error **SHALL** be returned.

```
ViStatus PpiGetSpaceInfo(
  __in        PpiHandle     handle,
  __in        PpiSpace      space,
  __out       ViPInt16      spaceType,
  __out       ViPUInt64     spaceBase,
  __out       ViPUInt64     spaceSize
);
```

Parameters

handle [in]	Handle to the modular device, returned from PpiOpen().
space [in]	The space on the device for which to read information.
spaceType	The type of this BAR space using values as follows: *0 None* *1 Memory* *2 I/O*
spaceBase	The base address for the designated BAR.
spaceSize	The size in bytes of the region of memory assigned to this BAR.

Return Value

If the operation completes successfully, the return value is VI_SUCCESS.

If the operation fails the return value is less than zero (VI_SUCCESS). If the operation fails an appropriate VISA error code is returned.

3.5 PpiGetDeviceAttribute

Get the specified information (such as Manufacturer and Model) for the device corresponding to this *handle*.

A plug-in is required to implement the following VISA attributes :

VI_ATTR_MANF_ID	ViUInt16
VI_ATTR_MODEL_CODE	ViUInt16
VI_ATTR_MANF_NAME	ViChar [256]
VI_ATTR_MODEL_NAME	ViChar [256]
VI_ATTR_PXI_ALLOW_WRITE_COMBINE	ViBoolean
VI_ATTR_DMA_ALLOW_EN	ViBoolean

A plug-in may optionally implement the following VISA attributes :

VI_ATTR_PXI_SLOTPATH	ViChar [256]

A plug-in is allowed to implement additional VISA attributes, such as vendor-specific attributes.

If the property VI_ATTR_PXI_ALLOW_WRITE_COMBINE or VI_ATTR_DMA_ALLOW_EN is queried, the plug-in **SHALL**return whether enabling write combining or DMA, respectively, is supported.

Any properties declared as optional are things a client could easily get elsewhere, such as from pxi*sys.ini. If a plug-in returns an error instead of implementing optional properties, the client is responsible for handling this appropriately.

```
ViStatus PpiGetDeviceAttribute(
  __in      PpiHandle   handle,
  __in      ViAttr      attributeID,
  __out     void *      attributeValue
);
```

Parameters

handle [in]	Handle to the modular device, returned from PpiOpen().
attributeID [in]	*VISA-defined or vendor-specific attribute ID.*
attributeValue [out]	*Value of the requested attribute, on success. The type of data that this parameter points to depends on the attribute ID. The caller is responsible for passing a buffer large enough to contain the attribute value.*

Return Value

If the operation completes successfully, the return value is VI_SUCCESS.

If the operation fails the return value is less than zero (VI_SUCCESS). If the operation fails an appropriate VISA error code is returned.

3.6 PpiMapMemory

Map base address register memory on a PCIe device into user space and make it available to the calling process.

```
ViStatus PpiMapMemory (
  __in        PpiHandle     handle,
  __in        PpiSpace      space,
  __in        ViUInt64      offset,
  __in        PpiLength     length,
  __out       void **       userSpaceMem
);
```

Parameters

handle *[in]*	Handle to the modular instrument, returned from PpiOpen().
space [in]	The base address register to map. PpiSpace is an enum type. Note that PCI config space and I/O space cannot be mapped using this method.
offset [in]	The offset in bytes within the base address register to begin to map.
length [in]	The number of bytes to map. This is either a 32 bit value or a 64-bit value, depending on the application.
userSpaceMem [out]	A pointer (virtual address) to the mapped memory, available to the calling process. If the function fails, *userSpaceMem is 0x0.

Return Value

If the operation completes successfully, the return value is VI_SUCCESS.

If the operation fails the return value is less than zero (VI_SUCCESS). If the operation fails an appropriate VISA error code is returned.

3.7 PpiUnmapMemory

Unmap previously mapped memory. Note that if the implementation requires the BAR, offset, or length to Unmap, it should be saved when the PpiMapMemory is done.

```
ViStatus PpiUnmapMemory (
  __in        PpiHandle     handle,
  __in        ViAddr        userSpaceMem
);
```

Parameters

handle [in] Handle to the modular device, returned from PpiOpen().

userSpaceMem [in] A pointer to the mapped memory to unmap.

Return Value

If the operation completes successfully, the return value is VI_SUCCESS.

If the operation fails the return value is less than zero (VI_SUCCESS). If the operation fails an appropriate VISA error code is returned.

3.8 PpiBlockWrite

Write a block of memory to a device.

If the address space is configuration space, and the offset points to a register that is managed by the operating system and/or BIOS such that it is potentially unsafe to change its value, the implementation is allowed to return an error. For example, the registers at offsets 0-63 are typically managed by the operation system and/or BIOS.

```
ViStatus PpiBlockWrite (
  __in        PpiHandle     handle,
  __in        ViInt32       flags,
  __in        PpiSpace      space,
  __in        ViUInt64      offset,
  __in        ViUInt32      width,
  __in        ViBoolean     increment,
  __in        void *        writeBuffer,
  __in        PpiLength     count,
  __in        ViUInt32      timeoutMilliseconds
);
```

Parameters

handle [in] Handle to the modular device, returned from PpiOpen().

flags [in] Flags to control write behaviors. These flags are used as hints, and may be ignored if the device being accessed does not support the mode. A plug-in **SHALL** ignore any bit that it does not recognize. It may flag an error for non-sensical combinations. The most significant 16 bits are reserved for vendor definition. Bits defined:

```
#define USE_DMA             0x1
#define USE_WRITE_COMBINE   0x2
```

space[in] The space on the device to which to write.

offset [in] A byte offset within *space*.

width[in] Transfer width in bytes. Possible values are 1, 2, 4, and 8.

increment [in] If true, increments the destination address by *width* between each individual write.

writeBuffer [in] Pointer to the data to write.

count[in] Number of elements of *width* size to write.

timeoutMilliseconds[in] The amount of time to wait for the function to complete. 0xFFFFFFFF indicates infinity.

Return Value

If the operation completes successfully, the return value is VI_SUCCESS.

If the operation fails the return value is nonzero (not VI_SUCCESS). If the operation fails an appropriate VISA error code is returned.

3.9 PpiBlockRead

Read a block of memory from a device.

```
ViStatus PpiBlockRead (
  __in        PpiHandle     handle,
  __in        ViInt32       flags,
  __in        PpiSpace      space,
  __in        ViUInt64      offset,
  __in        ViUInt32      width,
  __in        ViBoolean     increment,
  __in, out   void *        readBuffer,
  __in        PpiLength     count,
  __in        ViUInt32      timeoutMilliseconds
);
```

Parameters

handle [in]	Handle to the modular device, returned from PpiOpen().
flags [in]	Flags to control read behaviors. These flags are to be used as hints, and may be ignored if the device being accessed does not support the mode. A plug-in **SHALL** ignore any bit that it does not recognize. It may flag an error for non-sensical combinations. The most significant 16 bits are reserved for vendor definition. See *flags* documented in PpiBlockWrite.
space [in]	The space on the device from which to read.
offset [in]	A byte offset within *space*.
width[in]	Transfer width in bytes. Possible values are 1, 2, 4, and 8.
increment [in]	If true, increments the source address by width between each individual read.
readBuffer [in, out]	Pointer to a user-allocated buffer that will receive the data.
count[in]	Number of elements of *width* size to read.
timeoutMilliseconds [in]	The amount of time to wait for the function to complete. 0xFFFFFFFF indicates infinity.

Return Value

If the operation completes successfully, the return value is VI_SUCCESS.

If the operation fails the return value is less than zero (VI_SUCCESS). If the operation fails an appropriate VISA error code is returned.

3.10 PpiEnableInterrupts

Enables the client to receive interrupts from this device.

The client should call *PpiEnableInterrupts* before calling *PpiWaitInterrupt*. If an interrupt occurs after calling *PpiEnableInterrupts*, but before *PpiWaitInterrupt* is called, that *PpiWaitInterupt* will return without having to wait.

The specification does not specify whether any interrupts occurring before *PpiEnableInterrupts* are buffered or ignored. If there are any existing buffered interrupts, this function does not flush them.

PXI-4 (PXI Module DescriptionFile Specification) Section 2.4, defines an interrupt enable sequence that is placed in the peripheral description file. If no interrupt enable sequence is defined, this call may generate an error.

```
ViStatus PpiEnableInterrupts (
  __in     PpiHandle   handle,
  __in     ViUInt16    queueLength
);
```

Parameters

handle [in]	Handle to the modular device, returned from PpiOpen().
queueLength [in]	The number of interrupts a plug-in must be able to buffer.

Return Value

If the operation completes successfully, the return value is VI_SUCCESS.

If interrupts are already enabled because the caller has previously called *PpiEnableInterrupts* on this session, the return value is VI_SUCCESS_EVENT_EN.

If the operation fails the return value is less than zero (VI_SUCCESS). If the operation fails an appropriate VISA error code is returned.

3.11 PpiWaitInterrupt

If there is at least one buffered interrupt on this session, regardless of whether interrupts are currently enabled, this function **SHALL** return immediately with success.

Otherwise, if interrupts are already enabled by *PpiEnableInterrupts*, block this thread until an interrupt occurs. Note that interrupts are initially disabled.

See *PpiEnableInterrupts* for additional information about how this is configured.

PpiWaitInterrupt may be called with a *timeoutMilliseconds* of zero to discard a single buffered interrupt. Calling it successively until a timeout error occurs will clear all queued interrupts.

If interrupts are not enabled (meaning *PpiEnableInterrupts* has not been called), this function **SHALL** return immediately rather than waiting the specified timeout. The return value (success or error) in this case depends on whether there are any buffered interrupts.

Does not return until either:

- A general purpose (non data transfer) PCIe interrupt arrives from the device. This returns VI_SUCCESS.
- The specified timeout has elapsed. This returns VI_ERROR_TMO.
- A PpiDisableAndAbortWaitInterrupt is called. This returns VI_ERROR_ABORT.
- PpiClose is called. The return value is an error but is not required to be a specific value.

```
ViStatus PpiWaitInterrupt (
  __in     PpiHandle   handle,
  __in     ViUInt32    timeoutMilliseconds,
  __out    ViPInt16    interruptSequence,
  __out    ViPUInt32   interruptData
);
```

Parameters

handle [in] Handle to the modular device, returned from PpiOpen().

timeoutMilliseconds [in] The amount of time to wait for the function to complete. 0xFFFFFFFF indicates infinity.

interruptSequence [out] The index of the interrupt sequence (defined in PXI-4) that detected the interrupt condition. Driver not based on PXI-4 are permitted to return an arbitrary value indicating the reason for the interrupt. VISA provides this value to the client in the VI_ATTR_PXI_RECV_INTR_SEQ attribute on the event context.

interruptData[out] The interrupt detection sequence (defined in PXI-4) may include a read. This interrupt data is the value from the first successful read. Drivers not based on PXI-4 are permitted to return an arbitrary value to the VISA client using this. VISA provides this value to the client in the VI_ATTR_PXI_RECV_INTR_DATA attribute on the event context.

Return Value

If the operation completes successfully, the return value is VI_SUCCESS.

If interrupts are disabled and there are no buffered interrupts, the return value is VI_ERROR_NENABLED.

If the operation fails the return value is less than zero (VI_SUCCESS). If the operation fails an appropriate VISA error code is returned.

3.12 PpiDisableAndAbortWaitInterrupt

Unblocks a *PpiWaitInterrupt* call. After calling this function, clients should not call *PpiWaitInterrupt* without first calling *PpiEnableInterrupts* again.

This specification does not specify whether any interrupts occuring after *PpiDisableAndAbortWaitInterrupt* are buffered or ignored.

```
ViStatus PpiDisableAndAbortWaitInterrupt (
  __in        PpiHandle handle
);
```

Parameters

handle [in] Handle to the modular device, returned from PpiOpen().

Return Value

If the operation completes successfully, the return value is VI_SUCCESS.

If the operation fails the return value is less than zero (VI_SUCCESS). If the operation fails an appropriate VISA error code is returned.

3.13 PpiTerminateIO

Abort a block I/O transfer that is running in the background. The plug-in is permitted to ignore this operation and wait for the transfer to complete normally. The plug-in **SHALL** return VI_ERROR_NIMPL_OPER if the operation is ignored.

```
ViStatus PpiTerminateIO (
  __in      PpiHandle handle,
  __in      void*     buffer
);
```

Parameters

handle [in] Handle to the modular device, returned from *PpiOpen*().

buffer[in] The buffer that the I/O was initiated on (readBuffer or writeBuffer).

Return Value

If the operation completes successfully, the return value is VI_SUCCESS.

If the operation fails the return value is less than zero (VI_SUCCESS). If the operation fails an appropriate VISA error code is returned.

3.14 PpiClose

Close the handle to a device. This will abort all pending I/O and unblock any threads waiting for an interrupt.

```
ViStatus PpiClose (
  __in        PpiHandle handle
);
```

Parameters

handle [in] Handle to the modular device, returned from *PpiOpen*().

Return Value

If the operation completes successfully, the return value is VI_SUCCESS.

If the operation fails the return value is less than zero (VI_SUCCESS). If the operation fails an appropriate VISA error code is returned.

3.15 PpiFinalizePlugin

A client should call this plug-in function last, for example, immediately before unloading the plug-in. A plug-in should implement this by doing whatever one-time (per-process) cleanup is needed. Since a plug-in may be loaded by multiple clients in the same process, it **SHALL** reference count how many times PpiInitializePlugin() was called, so it does not prematurely clean up when this function is called. In particular, the plug-in **SHALL NOT** do any work on any calls to PpiFinalizePlugin() except the last one.

```
ViStatus PpiFinalizePlugin();
```

Parameters

None.

Return Value

If the operation completes successfully, the return value is VI_SUCCESS.

If the operation fails the return value is less than zero (VI_SUCCESS). If the operation fails an appropriate VISA error code is returned.

4 Data types in include file

```
typedef ViBusSize      PpiLength;

typedef ViAddr         PpiHandle ;

typedef enum
{
      Bar0 = 0,
      Bar1,
      Bar2,
      Bar3,
      Bar4,
      Bar5,
      Config
} PpiSpace;
```