Document Type: Tutorial
NI Supported: Yes
Publish Date: Feb 02, 2012

Porting a Windows Device Driver to the NI Real-Time Platform

Overview

This document is intended for anyone who is interested in learning how to port an existing Windows PCI device driver to the LabVIEW or LabWindows/CVI Real-Time platform.

Table of Contents

Introduction

This document includes guidelines for creating a Real-Time driver based on the VISA application programming interface, using an existing Windows driver as the starting point. The resulting driver is a C-based DLL and a library of wrapper VIs. While this document emphasizes Windows drivers, the concepts apply to porting any C-based driver to the Real-Time platform. This document assumes that you have a good understanding of the LabVIEW, ANSI C, and register-level programming techniques.

Note: While C-based VISA driver code can be debugged on Windows and should behave correctly in the Real-Time environment, problems may occur due to coding errors, timing issues, or inconsistencies between the Windows and Real-Time run-time support libraries. If this happens, debug the driver directly in the Real-Time environment using LabWindows/CVI's Remote Debugging capabilities.

Background

VISA
VISA is a set of open standards developed and supported by over 50 of the test and measurement industry's leading equipment vendor, user, and system integration organizations. The VISA standards were created to improve the efficiency of integrating and maintaining multi-vendor test and measurement systems based on the VXI and VME hardware form factors or controlled through IEEE-488 (GPIB) and RS-232/422 cabled interfaces. National Instruments' implementation of the VISA software standards is called NI-VISA. In addition to the device types listed above, NI-VISA also supports compact PCI and PXI devices on Windows and the NI Real-Time platforms.

NI-VISA licenses are included with National Instruments system control and instrumentation hardware, as well as with LabVIEW, Measurement Studio, and other National Instruments' application development environment tools. NI-VISA deployment and distribution licenses are also available separately from National Instruments.

Definitions
Application Programming Interface
An application programming interface (or API) is a set of functions that can be called by an application program. This document references three application programming interfaces:

- **Driver API** – The set of function calls/VIs that the application program uses to control an I/O device.
- **Windows API** – The set of function calls provided by Windows to communicate with operating system. Generally, the Win32 API. Specifically, the Win32 API functions typically called by device drivers.
- **VISA API** – Generally, the set of function calls defined by the VISA standard for instrument control. Specifically, the VISA API functions related to register-level programming.

Device Drivers in a Windows PC System
In the context of computer systems, *driver* refers generically to the system software that allows user application software to control the hardware devices that are installed in the system.

A driver exposes a set of high-level, application-oriented functions through its API. As the application program makes API calls, the driver translates them into sequences of low-level register I/O operations that the hardware can understand.

In most operating systems device drivers are partitioned into two layers-the user layer and the kernel layer. Each driver has a driver component for the user layer and a second driver component for the kernel layer. The kernel-layer driver, which is usually linked into the operating system, is technically the *device driver* . The driver API, which resides in the user layer, makes calls into the kernel-layer driver.

Instrument Drivers in a Test and Measurement System
In automated test and measurement systems, an *instrument driver* refers to the software that presents an abstracted interface for controlling programmable instruments. Programmable instruments were originally connected to computers by IEEE-488 bus or through a serial interface such as RS-232 or RS-422. These instruments use a *message-based* protocol consisting of instrument-specific ASCII command strings with limited mechanisms for servicing asynchronous events and reporting status. During the past few decades, modular instruments with *register-based* interfaces have emerged in a variety of form factors, including VME, VXI, PCI, CompactPCI, and PXI form factors.

The VISA standards were created to provide a consistent framework for dealing with the many possible combinations of hardware form factors, software environments, and equipment vendors. VISA provides a foundation that simplifies system integration and extends the useful lifetime of instrument drivers for decades.

In Windows-based systems, "instrument drivers" are typically implemented as device drivers. For compatibility with the NI Real-time platform (as well as other platforms supported by NI-VISA) it makes sense to port the Windows device driver to a VISA-based instrument driver.

Device Driver and Instrument Driver Architectures
This document refers to driver components that run at the kernel level collectively as the *device driver*. The driver components that run at the user level are referred to as the *user driver*.

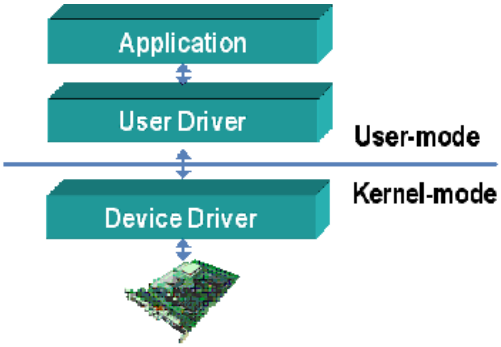

Figure 1. Generic Device Driver Architecture

Unlike device drivers, instrument drivers are implemented in the user-mode layer. There is no need for kernel-level programming because VISA abstracts the underlying details for the software developer. VISA provides the C-based API used to port an existing Windows driver to the NI Real-Time platform.

For the purposes of this document, the Real-Time driver consists of the ported VISA-based driver DLL with a C API, with a LabVIEW API consisiting of wrapper VIs. The remainder of this document focuses on developing the DLL and wrapping the exported DLL functions in LabVIEW VIs.

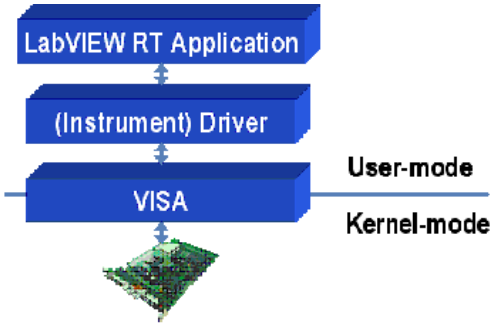

Figure 2. Real-Time Driver Architecture

The device-specific code in a VISA-based driver runs at the user level and the device driver that communicates with the hardware is encapsulated within the VISA layer.

VISA-based drivers for VXI, VME, GPIB and Serial devices are generally referred to as *Instrument Drivers* , so this document follows this convention for PCI and PXI devices.

Process Overview

Complete the following steps to port an existing Windows device driver to a Real-Time instrument driver:

I. Gather Documentation
The following driver-related documentation is very helpful when porting a driver to VISA.

1. Hardware reference manual-Includes the register map and functional descriptions that explain the purpose and read/write properties of the control/status registers (CSRs). The manual should also include the Vendor ID and Model code for the device.

2. Driver software-The API definition and the underlying source code for an existing driver should be used as a starting point whenever possible.

- Driver documentation–Includes information on the API function syntax. It may also describe how the driver works (driver execution flow, interrupt handling and DMA operations).
- Driver source code–It is important to obtain the source code of all the driver layers so that you can analyze exactly which parts need to be ported. In general, a Windows driver may have the following files:
    1. Source files for the High-level driver. This is where most of the driver functionality is implemented.
    2. If applicable, source files that invoke the OS specific calls, such as “DeviceIOControl” in Windows.
    3. If applicable, source files for any filter drivers. Filter drivers are an optional layer within the Windows driver architecture. They allow for pre- or post-processing of data in between driver layers. Filter drivers are common for data encryption and data manipulation between driver layers. You also can use them to call other drivers to expand the driver functionality in ways not previously anticipated by the driver developer. For more information about filter drivers, refer to the appropriate Windows documentation.
    4. Source code for the device driver. These are the files for the code that runs at the kernel level.
- Example application-When developing a driver, use a sample application program to test the VISA driver to confirm that its behavior is consistent with the Windows driver.

3. VISA user manual and programming reference manual–Includes all necessary VISA definitions and how to use the VISA API.

II. Configure VISA to Recognize the PCI Device
The first step in developing the instrument driver is to configure VISA to recognize the PCI device. This can be accomplished by creating an .INF file for the device using the PXI Driver Development Wizard.

For more information on this topic, refer to Configuring the NI Real-Time Environment to Recognize a Third Party Device.

III. Port the Windows Driver to VISA

Replace the Operating System-Specific I/O Function Calls

For a driver to work in the Real-Time environment, the OS-specific I/O functions that program the CSR registers and transfer data to and from the device must be translated to VISA calls. An understanding of how the driver is organized, how each driver layer relates to the others, and how data is transferred between them is helpful in locating and identifying OS-specific I/O calls.

If the device supports bus master DMA operation, the DMA controller programming operations for the must be translated to VISA.

If the driver is designed for portability, the I/O function calls may be located in a single source file. If the driver is not organized this way, you must search throughout the source code to locate and replace each of the I/O calls.

Replace Operating System Services
If the driver depends on the operating system for certain services, you must replace them or stubbed them out with enough functionality that the driver can operate properly. Typical OS-specific services used by a drivers include:

- Persistent data storage–Storing device configuration and status for future sessions
- Instance management–Support for multiple devices of the same type
- Resource management–Support for multitasking environments

The instrument driver and any software modules that it calls must be free of any OS-specific function calls.

Tip: You can use LabWindows CVI to build your DLL since it specifically provides a build option for the Real-Time environment. If the source code contains Windows-specific calls, the LabWindows/CVI compiler generates an error.

Develop the Interrupt Processing Mechanism
If the device requires interrupt support, use VISA functionality to develop an interrupt processing mechanism.

Develop DMA Support
If the device requires DMA support, use VISA functionality to develop a DMA programming mechanism. DMA engines are device-specific, so an in-depth understanding of the hardware architecture is required.

IV. Create LabVIEW Wrapper VIs
The final step of the process is to create LabVIEW wrapper VIs around functions exported by the VISA-based driver DLL.

Windows Device Driver Architecture

The Windows driver architecture provides an extensible interface between the user and kernel layers called the Windows I/O manager. The I/O Manager API includes functions for communicating with device drivers using data structures called I/O Request Packets (IRPs).

To improve portability, some implementations isolate the I/O Manager calls in a separate layer as illustrated below, while other implementations make I/O manager calls throughout the driver code.

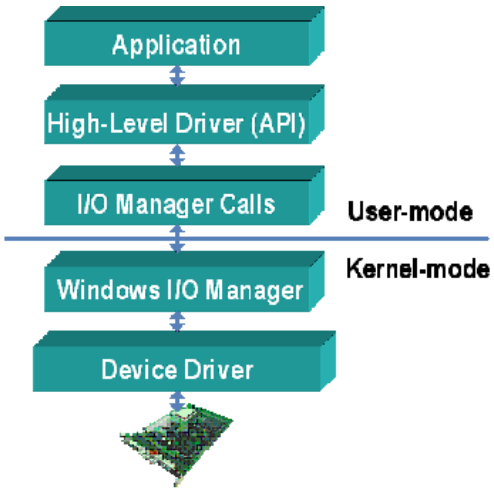

Figure 3. Simplified Windows driver architecture

The communication process between the user driver and the device driver includes the following steps:

1. The user driver opens a reference to the device by using the Windows API function *CreateFile*.
2. The user driver sends a control code (command) to the device driver through the Windows API function *DeviceIOControl.*
3. The Windows I/O manager receives the control code, creates an IRP, and routes the IRP to the device driver.
4. The device driver receives the IRP. Based on the control code, it performs the operation requested by the user-level driver. The device driver accesses the hardware to complete the requested operation. Any data returned by the device is inserted in the IRP and sent back to the user driver via the I/O manager.
5. When the application is finished, it requests the user driver to close the device. The user driver closes the device reference by calling the Windows API function *CloseHandle*.

Note: In the Windows driver architecture, devices are treated as file resources, providing a common abstraction model for the user driver. Simple direct access to the device driver can be achieved by calling the Windows API functions *ReadFile* and *WriteFile* (instead of *DeviceIOControl)*. There are many additional Windows API functions that can be used to communicate with device drivers. Refer to the appropriate Microsoft documentation for more information.

Porting the Windows Driver to VISA

VISA Device Sessions

A VISA session serves as a handle for a device. Each device session stores the information used to configure the communication channel to the device, as well as information about the device itself, as a collection of session attributes. The driver can configure a session by setting attributes and can query session parameters by reading attributes. Any operation related to a device is controlled through a VISA session.

VISA Open and Close
Before opening a session for a device, the driver must first open a session to the VISA resource manager (RM). This is accomplished by calling the VISA function “viOpenDefaultRM”.

The driver can open a session for a device by calling the VISA function viOpen using the resource name for that device.

The driver should open a new RM session for each device session, and close the corresponding RM session as it closes each device session. Opening an RM session per device session does not add significant overhead.

When the user application closes a device, the driver must close the associated device and Resource Manager sessions by calling the VISA function viClose. The driver queries the attribute VI_ATTR_RM_SESSION to retrieve the resource manager session associated with the current device session.

Failure to close any opened device and RM sessions creates a memory leak.

VISA Find Resources
VISA includes a single interface for finding and connecting to devices. The VISA resource manager assigns unique resource names such as “PXI1::14::INSTR”, for each device and exports a query-based service for retrieving resource names.

The driver uses the VISA viFindRsrc function to get the resource name for the device. An example query expression for a device of a particular make and model is shown below:

PXI?*INSTR{VI_ATTR_MANF_ID ==0x35BC && VI_ATTR_MODEL_CODE ==0x0241}

The query expression verifies that you are opening a session to the correct device in the system.

PCI Register-Level Programming
After a VISA device session is open, you can read and write the CSR registers and memory using VISA functions.

In many cases, you can use the high-level functions VISA In (viIn8/16/32) and VISA Out (viOut8/16/32) for register read/write access. If the driver repeatedly accesses registers in one address space, however, the low level functions viMapAddress, viPeek, and viPoke, may be more efficient.

Use the VISA Move functions viMoveIn and viMoveOut to efficiently transfer large data buffers from memory to memory.

Access PCI devices via seven PCI address spaces:

- PCI configuration space
- Six Base Address Register spaces (BAR0 – BAR5).

PCI Configuration Space
The device’s configuration registers are located in the PCI configuration space. Configuration registers contain system configuration information for the device and are usually read-only. Most of a device’s configuration information can be read as VISA attributes. However, the device configuration registers can also be accessed directly using the VISA viIn functions with the address space parameter value VI_PXI_CFG_SPACE.

The VISA Find Resources function uses the PCI configuration space values device ID, vendor ID, subsystem ID, and subsystem vendor ID for comparison with the manufacturer ID and model code expression parameters. If Subsystem ID and Subsystem Vendor ID are defined for the device, then VISA populates the manufacturer ID (VI_ATTR_MANF_ID) and model code (VI_ATTR_MODEL_CODE) attributes with these values. Otherwise, VISA returns the PCI Device ID and Vendor ID values for these attributes.

BAR0-5 Address Spaces
The PCI Configuration Space also contains information, such as the base address for each of the BAR address spaces used by the device. The driver does not need to read this information from the configuration registers since VISA automatically stores and uses the base address of each BAR space. VISA I/O calls only need to specify the appropriate address space (VI_PXI_BAR0_SPACE to VI_PXI_BAR5_SPACE).

The content of each BAR address space is unique for each type of device. One or more address spaces normally contains CSR registers for configuring or controlling the device. Some address spaces contain address windows to on-board memory that can be read or written by the driver.

Compiling and Debugging the Driver
The initial compile target should be a debuggable DLL. After the driver development and testing is complete, you can create a release version.

Tip: Develop the driver on Windows on a dual-boot PXI controller. Periodically boot into LabVIEW Real-Time to test the driver’s functionality or real-time performance.

Interrupt and DMA support

Interrupt support
Because LabVIEW does not support the concept of callbacks, LabVIEW cannot respond to interrupts the same way a Windows driver would. However VISA provides a mechanism for detecting interrupts. VISA informs LabVIEW that an interrupt occurred by sending an event message. When a VISA interrupt event is received, the application can handle the interrupt. Real-Time applications must use the VISA function viWaitOnEvent to determine when an interrupt occurs.

PCI Interrupt Sequence Summary
PCI interrupts occur in the following sequence:
1. The operating system loads the device’s interrupt configuration information at boot time. On Windows and LabVIEW Real-Time, the interrupt configuration is stored in the .INF file.
2. The instrument driver configures the device to generate interrupts under device-specific conditions.
3. The device driver handles each interrupt as it occurs with the following sequence of events:

A. The device generates an interrupt that matches the interrupt detection algorithm defined in the INF file.
B. The operating system routes the interrupt to the VISA kernel driver, which polls each device until it locates the device that is generating the interrupt.
C. The VISA kernel driver then disables the interrupt by masking it on the device, using the interrupt removal sequence defined in the INF file.
D. The VISA kernel driver generates a VISA *VI_Event_PXI_INTR* event.
E. The *VI_Event_PXI_INTR* event causes an interrupt service handler provided by the instrument driver to be executed.
F. The interrupt service handler processes the interrupt, clears the interrupt condition, and re- enables interrupts for that device.

4. When the application is no longer interested in receiving interrupts, the instrument driver disables interrupts for the device.

Configuring Interrupts
To configure interrupts the API should provide a function that programs the CSR registers so that the device generates interrupts under certain conditions if hardware interrupts are enabled. This function should not enable hardware interrupts on the device.

Enabling Interrupts
The driver must call *viEnableEvent* to enable VISA events for the session before enabling hardware interrupts on the device. The should enable the *VI_EVENT_PXI_INTR* event. If the device generates interrupts before VISA events are enabled, there is no mechanism to remove the first interrupt, resulting in a system lockup condition. To enable hardware interrupts, the user application should call a function provided by the API that programs the CSR registers for this purpose. You should include the viEnableEvent call in this function.

Interrupts are often enabling as part of the device initialization sequence, but you also can use interrupt enable/disable functions to control interrupt generation at any point in the user application.

Interrupt Detection and Removal
Interrupt detection and removal consists of a sequence of register read/write operations performed by the VISA kernel driver that detects whether the device is generating an interrupt by reading values from the CSR registers, and disables hardware interrupts on the device by writing specific values to the CSR registers.

The interrupt detection and removal sequences are defined in an .INF file associated with each device type. You can configure these algorithms as you create the .INF file using the VISA Driver Development Wizard as described above.

Configuring the VISA Event Queue
VISA places events in the event queue every time an interrupt is detected. The driver must enable PXI Interrupt events by calling the VISA function *viEnableEvent* with an event type parameter of *VI_EVENT_PXI_INTR*.

You can modify the size of the VISA event queue based on the application. If the interrupt handling routine takes a considerable amount of time processing the interrupt, you can increase the event queue size to ensure that no interrupt is lost. The driver can change the event queue size using the *viSetAttribute* function using the *VI_ATTR_MAX_QUEUE_LENGTH* parameter.

The event queue defaults to a value of 50. VISA does not let you dynamically configure queues lengths. You can only modify the queue length for a given session before the first invocation of the *viEnableEvent* operation.

Interrupt Processing
In many cases, the interrupt processing requirements are application-specific, so the driver API should include functions commonly required by user applications for this purpose.

Figure 4 provides an overview of the interrupt processing procedure:

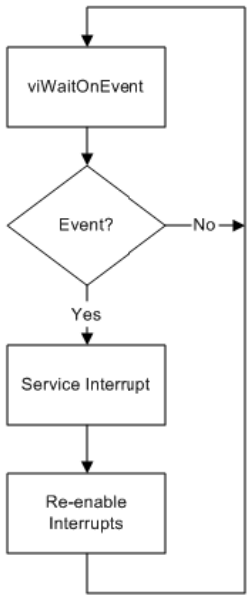

Figure 4. Interrupt Processing Procedure

The methods for processing interrupts vary between devices and applications using the same device. The following steps for processing interrupts take place after the VISA function *viWaitOnEvent* indicates that an interrupt occurred.

1. Determine the interrupt source. Some devices have multiple sources that can generate an interrupt. The driver must query the device to determine the source of each interrupt.
2. Service the interrupt. There are two main approaches a driver developer can take:

a. Implement a function that services interrupts in a predefined way. This approach is useful when interrupt processing is not application-specific. You could wrap this type of interrupt processing function together with the functionality of the previous step to provide the user with a one-step interrupt processing implementation.
b. Allow the user application to provide code that services the interrupt in an application-specific way. This approach is useful when interrupt servicing is application specific, and is similar to a Windows driver in the sense that the driver allows for user-defined functions to be called whenever an interrupt occurs. The driver may need to provide additional helper functions so that the user application can directly interact with the device.

3. Re-enable interrupts on the device. When a device generates an interrupt, VISA automatically disables hardware interrupts on that device. The driver must re-enable the device's interrupts after the interrupt has been serviced. If you are using the approach described in 2b, one of your helper functions should be a function that re-enables hardware interrupts on the device. The user application is responsible for calling this function after the application-specific code completes execution.

You can expose the interrupt processing mechanism as a set of functions in the driver API. The user application calls the interrupt processing mechanism periodically. When called, it senses interrupt events by calling the VISA *viWaitOnEvent* function. If no interrupts have occurred, *viWaitOnEvent* and the interrupt processing mechanism should return immediately (timeout = 0) so a real-time application calls the interrupt processing mechanism only when a VISA interrupt event has occurred. However, the worst-case interrupt latency using this approach depends on how often the application calls the interrupt processing mechanism.

The driver implementation of the interrupt processing mechanism should not include a loop around *viWaitOnEvent* that waits for an interrupt to occur. Most LabVIEW applications include a loop and

in some cases, this is a time-critical loop. The application developer must include the driver functions to handle interrupts inside their application loop. If your interrupt processing implementation already contains a loop on *viWaitOnEvent*, you may starve the time critical loop and affect determinism. Your implementation should call *viWaitOnEvent* once and return immediately whether an interrupt occurred or not. The application's execution flow is the responsibility of the application developer, not the driver developer.

In summary, a good implementation practice is to create several driver functions for processing interrupts that include:

- An interrupt event sensing function that calls *viWaitOnEvent* once, and interrogates the interrupt source
- Helper functions for servicing interrupts
- A helper function for re-enabling hardware interrupts on the device.

Disable Interrupts
Interrupts must be disabled on the device before the device session is closed.

After interrupts are disabled (but before the session is closed), the driver must also call *viDisableEvent* to disable VISA interrupt events for the session.

Example
The following example demonstrates an interrupt processing implementation. The example includes four modular functions that perform distinct and well-defined operations. Functions that service interrupts and enable/disable interrupts on the device are not included here because they are device-specific.

This example shows the approach that requires the user application to provide code that processes interrupts in an application-specific manner. These functions are wrapped in VIs and presented to the application developer as part of the driver API. The user application is responsible for calling these functions in an order that achieves the desired functionality.

```
ViStatus my_Enable_Interrupts(ViSession session)
{
ViStatus status = 0;

//enable VISA events before enabling hardware interrupts!
status = viEnableEvent (session, VI_EVENT_PXI_INTR,
VI_QUEUE, VI_NULL);

/* Add here the function call to enable hardware interrupts on the device*/

return status;
}


ViStatus my_Interrupt_Identifier(ViSession session,
ViUInt32 timeoutValue, ViBoolean *timedOut,
ViUInt32 *interruptSource)
{
ViStatus status = 0;
ViEvent eventHandle; //use it for extra info of the event

status = viWaitOnEvent (session, VI_EVENT_PXI_INTR,
timeoutValue, VI_NULL, &eventHandle);


//verify if an interrupt occurred or viWaitOnEvent just timed out
if (status == VI_ERROR_TMO)
{
*timedOut = TRUE;
return status;
}

//if no error, get the interrupt source
if (status == 0)
{
*timedOut = FALSE;

/* Add here the code to determine the interrupt source */

//returning the interrupt source to the caller
//*interruptSource = 'some value'
}


return status;
}

ViStatus my_ReEnable_Interrupts(ViSession session)
{
ViStatus status = 0;

/* Add here the function call to enable interrupts on the
device*/

return status;
}

ViStatus my_Disable_Interrupts(ViSession session)
{
ViStatus status = 0;

/* Add here the function call to disable interrupts on the
```

```
device */

//disable VISA events after disabling hardware interrupts!
status = viDisableEvent (session, VI_EVENT_PXI_INTR, VI_QUEUE);

return status;
}
```

Tip: The LabVIEW Real-Time mechanism for handling interrupts consists of calling the WaitOnEvent VISA function to verify if an interrupt occurred. As a result, the function or VI that includes the WaitOnEvent call should be called periodically in a loop. Ensure that the interrupt service handler is optimized to be called repeatedly. Avoid practices that slow it down or affect determinism.

Tip: The example code for interrupt handling contains platform-specific calls (VISA). To be consistent with a layered architecture, you should implement these functions in the platform-specific layer of your driver. You can create wrapper functions that do not contain any VISA calls and place them in the high-level layer of your driver (API). Keeping all platform-specific calls together makes a more modular and maintainable driver.

Troubleshooting Interrupts
There are numerous interrupt handler implementation errors that can cause unintended system behaviors.

Behavior	Possible Cause	Effect
Spurious interrupt service handler calls	Incorrect interrupt detection sequence – False positive.	Detection algorithm in INF file senses that the device is generating interrupts when it is not. Another device is generating the interrupts.
Erratic or no interrupt service handler calls (unexpected Wait on Event timeouts, etc.)	Incorrect device interrupt configuration.	The device does not generate interrupts at the right time or at all.
Erratic or no interrupt service handler calls (unexpected Wait on Event timeouts, etc.)	Interrupt service handler is not re-enabling interrupts on the device.	After the first interrupt, the device does not generate further interrupts.
System lockup	Incorrect interrupt detection sequence – False negative.	Detection algorithm does not sense interrupts generated by the device, so the removal sequence is not executed. This causes in an unsquelched interrupt condition.
System lockup	Incorrect interrupt removal sequence.	Removal algorithm does not successfully disable the interrupt on the device, resulting in an unsquelched interrupt condition.
System lockup	Driver enables hardware interrupts on the device before calling *viEnableEvent*.	Interrupt detection and removal sequence for the device are not active, resulting in an unsquelched interrupt condition.
System lockup	Driver calls *viDisableEvent* before disabling hardware interrupts on the device.	Interrupt detection and removal algorithms for the device are not active, resulting in an unsquelched interrupt condition.

An unsquelched interrupt condition causes the system to lock up because the VISA kernel driver never returns from the interrupt handler.

Bus Master DMA Support
DMA operations are useful for transferring large amounts of data between the device and the host without consuming CPU time. There are two basic DMA operations-DMA Read (transfers data from the device to the host) and DMA Write (transfers data from the host memory to the device).

When using VISA, DMA transfers occur between the devices memory and a buffer of locked memory on the host computer. You can use VISA functions to allocate contiguous, locked memory buffers on the host controller for use in bus master DMA operations.

Preparing for DMA Operations
DMA operations require contiguous memory since the DMA controller increments the memory pointers sequentially. The memory also must be locked (non-pageable) so that the DMA controller can ensure that the buffer is available throughout the whole operation. As a result, DMA operations must be performed using buffers allocated with VISA and not with memory buffers allocated with the ANSI C malloc function.

The following steps are required for setting up a DMA operation:
1. Open a session to a VISA Memory Access (MEMACC) Resource by calling viOpen with the VISA resource name “PXI0::MEMACC”. This VISA session is separate from the device session and must be passed to all the subsequent functions that handle the VISA memory allocation. This opens a session to memory in the host controller.
2. Call the VISA Memory Allocation function, viMemAlloc, to allocate and reserve the memory buffer. This function returns an offset value that you can use to program the DMA controller on the device.
3. Initialize the DMA controller. Refer to the DMA controller’s documentation for specific initialization information.

Warning: Ensure that the driver treats the DMA transfer size and the allocated buffer size consistently. Depending on the device, the DMA transaction units maybe specified as byte, short, or long words. The VISA Memory Allocation VI, always specifies the buffer size in bytes. When setting up a DMA data transfer, ensure that the driver allocates enough memory for the whole transaction. For example, if the DMA controller transfers data as long words (4 bytes) and the transfer length is set to 1000, the DMA controller transfers 4000 bytes (1000 long words). The driver must specify 4000 (and not 1000) as the allocation size when using the VISA Memory Allocation VI. The DMA controller continues to increment memory pointers until the operation is completed, so allocating less memory than required causes other memory to be overwritten, which may result in corrupted data or a system crash.

Note: Contiguous locked memory buffers are a limited resource, so they should only be used for DMA operations. If the driver must allocate memory for other purposes, use the ANSI C malloc function instead of viMemAlloc.

Performing DMA Read and Write Operations
After performing a DMA Read operation, the driver must copy the data placed in the locked buffer into a user buffer using the function viMoveIn. Before performing a DMA Write operation, the driver must copy the user data into the locked memory using the function *viMoveOut*.

Complete the following steps for a DMA Read operation:
1. Configure the DMA controller to perform a read operation. The destination offset is the offset returned by the *viMemAlloc* function.
2. Start the DMA operation. Depending on your DMA controller, you may need to poll some registers or wait for an interrupt to know that the operation completed.
3. When the DMA transfer completes, the data is located in the locked memory buffer. Copy the data into a user buffer using the *viMoveIn* function with an address space value of *VI_PXI_ALLOC_SPACE*.

Complete the following steps for a DMA Write operation:
1. Configure the DMA controller to perform a write operation. The source offset is the offset returned by the *viMemAlloc* function.
2. Copy the data from the user buffer into the locked memory using the viMoveOut function with an address space value of *VI_PXI_ALLOC_SPACE*.

3. Start the DMA operation. Depending on the DMA controller, the driver may need to poll a register or wait for a DMA interrupt to confirm that the operation completed.

Cleaning Up After a DMA Operation
After DMA operations are complete, any memory allocated for DMA must be released. Failure to deallocate memory buffers and close the *MEMACC* session results in memory leaks.

Complete the following steps to clean up after a DMA operation:
1. Call the VISA function, viMemFree, to deallocate the memory.
2. Call the VISA Close function, viClose, to close the VISA MEMACC session.

Creating LabVIEW Wrapper VIs

After you create the API and built the DLL, you must create wrapper VIs for LabVIEW users. Creating these wrappers can be tedious because you must create a new VI for each exported function in the driver DLL. Integrated tools in both LabVIEW and LabWindows/CVI can automate this process. LabWindows/CVI 7.0 includes a scripting utility that automatically creates a function panel file from a header file (*.h). In the .h file, you specify which functions to include in the function panel and any required input and output parameters. Refer to the LabWindows/CVI documentation for more information.

After you create the function panels, create the LabVIEW VIs. LabVIEW includes a utility for automatically converting LabWindows/CVI function panel (.fp) files to LabVIEW instrument drivers. In LabVIEW, select **Tools»Instrumentation»Import CVI Instrument Driver** to launch the import tool. The tool creates a wrapper VI for each C function included in the function panel file and creates the appropriate parameter linkages to the driver DLL.

After you create the basic VIs, you can add standard LabVIEW error handling(using the LabVIEW error clusters and case structures) to each VI for your drivers. Integrated error handling in instrument drivers is a standard LabVIEW practice. Your API should also provide functions that follow an Open, Perform Action, Close architecture, which is familiar to LabVIEW users. You may want to split the functionality of your driver, creating a DLL function for each of these actions, and then create a wrapper VI around them. The reference value is passes from VI to VI to inform the driver which instance of the current action it should invoke.

If you did not create a LabWindows/CVI front panel file, you must manually create the wrapper VIs for your driver API functions. For wiring consistency and easy visual identification within LabVIEW programs, use a common icon and connector pane style for each of the VIs belonging to the instrument driver. You can create a VI template to use as a starting point for each VI. Select the standard 4-in/4-out or 6-in/6-out connector pane to make the wiring connections consistent. In most cases, each VI should have a reference in (VISA session or handle) input on the upper-left connector, a duplicate reference out output on the upper-right connector, an error in cluster input on the lower-left connector and an error out cluster output on the lower-right connector.

The resulting LabVIEW API should use concepts familiar to LabVIEW programmers so they can use the driver just as they would use native LabVIEW libraries. The following example uses a typical LabVIEW API.

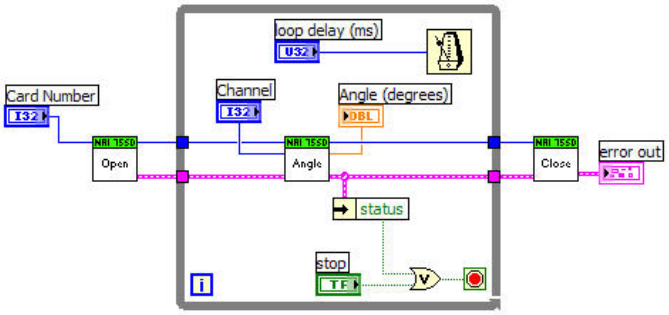

Figure 5. Driver API in LabVIEW

Notice how every function from this API conforms to typical LabVIEW practices, such as wiring the error cluster in and out of each VI call. Remember that the final user of this API is not a C programmer but a LabVIEW programmer. The following example implements a wrapper VI.

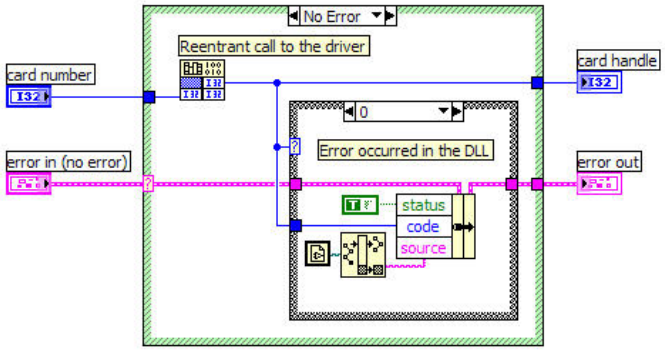

Figure 6. LabVIEW Wrapper to a driver DLL call

When calling DLLs in LabVIEW Real-Time, you must configure the Call Library Node to be reentrant. A non-reentrant call to a DLL runs in the user interface thread, significantly affecting performance and introducing jitter. VIs created automatically from a LabWindows/CVI function panel call the DLL in the user interface thread. Ensure you revise each VI to set the DLL call to be reentrant.

In LabVIEW 8.0 or later, refer to the **Fundamentals»Calling Code Written in Text-Based Programming Languages** book in the *LabVIEW Help* (linked below) for more information.

Recommended Practices

Distinct Open Device and Close Device Phases
Encapsulate all initialization and cleanup code in *Open Device* and *Close Device* driver API functions that are the first and last calls the application must make to the driver, respectively. This will greatly reduce the risk of system lockups and memory leaks.

Layered Structure
For driver portability, it is common practice to isolate the platform-specific I/O calls in a separate layer of the device driver. It is also common practice to implement the platform-specific layer as a separate source file.

In the following example, a high-level driver implements the API and most of the driver's functionality. When the driver code needs to access the hardware, it calls generic I/O functions that are implemented in the platform-specific layer. The high-level driver includes a function that configures the board by writing to two registers. To accomplish this, the function *Configure_Board* calls the helper function *Write_32b_Register*, which is defined in the platform-specific layer.

High-level driver.c

```
#include Platform_Specific_Calls.h

void Configure_Board (HANDLE board) {

int offset = 0x32;
int data = 1;
// initializing the board
Write_32b_Register (board, offset, data);

//initializing the interrupts
offset = 0x80;
Write_32b_Register (board, offset, data);
}
```

Platform_Specific_Calls_Layer.c

```
void Write_32b_Register (HANDLE board, int offset, int data) {

/* necessary platform-specific code to write a 32-bit value to a register */
}
```

Helper functions in the platform-specific layer are implemented as wrappers around equivalent functions supported by the target platform. In this example, the platform-specific layer includes the function *Write_32b_Register* that writes a 32-bit value to a device register on the target platform.

VISA presents an API that directly supports virtually all types of register-based I/O, so VISA calls can be directly substituted for the platform-specific code in the previous example, creating an NI-VISA-specific calls layer. The following diagram illustrates the organization of a layered VISA driver.

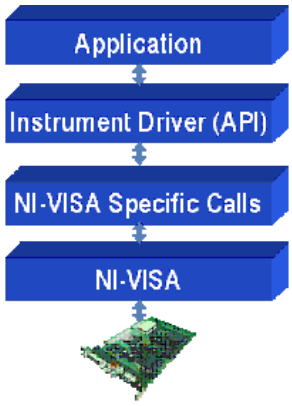

Figure 7. Layered Driver Architecture

The following example implements the *Write_32b_Register* function in the NI-VISA-specific calls layer.

NI-VISA_Specific_Calls_Layer.c

```
#include "visa.h"

void Write_32b_Register (HANDLE board, int offset, int data)
{
//local variable definitions
ViSession session;

//retrieve the ViSession
session = *(ViSession *) board;

viOut32 (session, ADDRESS_SPACE, offset, data);
}
```

Write_32b_Register receives the board handle parameter as a Handle data type (void *) and casts it to a ViSession, which might not be possible or necessary in all cases. The example includes the Handle data type because it is commonly used in Windows applications. This example demonstrates that the application does not need to be aware of VISA or VISA data types. You should

avoid changing the syntax or functionality of the driver API. This will help guarantee that existing example and user applications continue to execute correctly after the driver has been ported into VISA. In this example, VISA calls are only present within the NI-VISA-specific calls layer.

Note: It is a good practice to keep the highest layer of your API unchanged. However for future supportability, every function in the driver API should receive a VISA session as a parameter. This approach is a tradeoff between supportability and compatibility with existing applications. When porting a driver into VISA, the driver developer should analyze the best possible driver architecture.

Hierarchical API
You should package common sequences of driver functions under a second set of API functions that are provided for ease of use. Typically, this alternate interface is marketed as a basic API for new users. The driver API also includes the foundation functions as an advanced API. This hierarchical implementation is optional, but it allows new users to get started quickly.

For example, a hierarchical API might include a basic function for initializing the device. To initialize the hardware, the driver might require separate calls to open a session to the device, initialize a DMA controller, and enable interrupts. The advanced API exports these functions so that users can have full control of the configuration parameters. For new users, a basic configuration function, *Initialize_Device,* could be provided that performs these operations using commonly-used default parameter values.

Memory Allocation

Memory allocation is important in a real-time environment. Memory allocation is a non-deterministic operation, but pre-allocating memory can preserve driver determinism.

Avoid allocating memory in functions that are called repeatedly, such as in DMA support code or in the main program loop. If the driver includes functions that dynamically allocate memory, consider implementing helper functions that pre-allocate and release a pool of memory for the driver to use.

Failure to release allocated memory results in a memory leak.

Cached Copy of Device Configuration Information
In many cases configuration and status registers are write-only, so the driver must store a copy of the device configuration parameters in local memory in order to modify parameters that are represented by bit fields within registers.

Cached parameters also improve execution performance. It often takes longer to read or write device registers than it takes for the data to flow between the various layers of a driver. A local copy allows the driver to bypass the relatively slow step of reading the information from the device registers. For example, if the application queries the driver to find out if interrupts are enabled, the driver can read that value from local memory rather than retrieving the information from the device.

If the Windows driver does not provide a local store mechanism, you can create a data structure for the configuration parameters and store a pointer to the structure as the session attribute *VI_ATTR_USER_DATA.* Use the memory management functions *malloc* and *free* to allocate and deallocate storage memory for the device configuration data.

The following example opens a session, stores the configuration parameters as user data, and closes the session.

```
#include <ansi_c.h>
#include "visa.h"

#define MY_MAX_DESCRIPTOR_SIZE 1000
#define TRUE -1
#define FALSE 0

typedef struct {
/* Define all the necessary fields */
ViBoolean interruptEnabled;
ViInt32 controlRegister;
} MY_INFO;

ViStatus my_Initialize(ViSession *session)
{
ViStatus status = 0;
ViSession defaultRM;
ViChar instrDescriptor[MY_MAX_DESCRIPTOR_SIZE];
MY_INFO *deviceInfo = NULL;

//open a VISA resource manager session
status = viOpenDefaultRM (&defaultRM);

//find the device in the system.
status = viFindRsrc (defaultRM,
"PXI?*INSTR{VI_ATTR_MANF_ID == 0x35BC &&
VI_ATTR_MODEL_CODE == 0x0241}",
VI_NULL,VI_NULL,instrDescriptor);


//open a device session
status = viOpen (defaultRM, instrDescriptor, VI_NULL,
VI_NULL, session);

//allocate memory for the MY_INFO structure for the current //session
deviceInfo = malloc (sizeof(MY_INFO));

//initialize the user data
deviceInfo->interruptEnabled = FALSE;

//store the info structure in the current session
status = viSetAttribute (*session, VI_ATTR_USER_DATA,
(ViAttrState)deviceInfo);


/* add here the code to initialize your device */

return status;
}
```

```
ViStatus my_check_Interrupt_State(ViSession session, ViBoolean
*intEnabled)
{
ViStatus status = 0;
MY_INFO *info = NULL;

//accessing local copy instead of reading data from hardware
status = viGetAttribute (session, VI_ATTR_USER_DATA, &info);
*intEnabled = info->interruptEnabled;

return status;
}

ViStatus my_Close(ViSession session)
{

ViStatus status = 0;
ViSession currentRscManager;
MY_INFO *deviceInfo = NULL;


/* add here the code to close your device */


//releasing the memory allocated for the MY_INFO structure
status = viGetAttribute (session, VI_ATTR_USER_DATA, &deviceInfo);

if (deviceInfo)
free(deviceInfo);

//getting the default resource manager for this session
status = viGetAttribute (session, VI_ATTR_RM_SESSION,
¤tRscManager);

status = viClose (session);
status = viClose (currentRscManager);

return status;

}
```

Warning: Do not cache volatile registers. Caching a register that can change its contents without software control can cause intermittent problems that are difficult to debug.

Persistent Configuration and Status Information

In most cases, you should store the device-specific information in a form that persists between device sessions. For example, some drivers store user-defined configuration parameters and status information for utilization during the next device session. The storage mechanism may depend on the platform.

Windows drivers can store this information in the Windows registry; however the Real-Time environment does not support this approach. Storing persistent information in a file is a method commonly used to store this type of data.

Multiple Instance Support

There are number of possible strategies for implementing multiple instance support. The following sections describe a typical approach.

API Implementation

Creating a user-friendly API is an important requirement for any driver. In general, the API should be transparent to the user, regardless of the platform. The final user should not be impacted because of a platform change or underlying implementation. Whenever possible, the API of your VISA driver should be equivalent to the API of your Windows driver.

The following common approaches are used to implement multiple instance support:

- Pass a board number to an initialization routine. Based on this number, use the VISA viFindRsrc and viOpen functions to index the corresponding instrument descriptor and open a session to it. The driver is responsible for keeping track of the VISA session. To operate the device, the user passes the board number to any subsequent driver call. The driver receives the board number and retrieves the corresponding VISA session to operate on the device.
- Rely on the application program to keep track of the VISA session. The user passes a board number to an initialization routine. The driver opens a session to the device and returns the session by reference. The user passes the session to any subsequent call to the driver.
- The user can pass an empty pointer to a session to the initialization routine. The initialization routine inserts the address of the current session into the user's parameter. A board number is not required for this approach.
- Hide the VISA implementation from the user. The API must provide a means of representing the board or a type of handle that can be internally translated or cast to a VISA session.

Tip: While the previous approaches are valid, National Instruments recommends that you allow the user to specify the device in each call to the driver (either by specifying the board number or the VI Session to each API call). By eliminating race conditions that can cause unpredictable effects, this approach supports independent use of multiple boards at the same time.

Driver Implementation

You can use the VISA function *viFindRsrc* to retrieve resource names for a device supported by the driver. The *viFindRsrc* output parameter *findList* returns a handle to the current resource name query. You can use this handle as an input for *viFindNext*, which returns the resource name for the next device instance. When the query is exhausted, *viFindNext* returns the error code *VI_ERROR_RSRC_NFOUND*.

In one implementation, the resource names for all compatible devices can be stored as a list, and the "device number" value in the Open Device function call can be used to index into the list of resource names.

Alternatively, the driver could open a session for each resource name and compare the attributes of the associated device to some other selection criteria. For example, the driver could look for a match between the "device number" and the PXI slot number of each compatible device.

In another implementation, the “device number” could be interpreted as a slot number to directly test if a device in a particular PXI slot is compatible with the driver. For example, to test the compatibility of the device in the slot specified by “device number”, call viFindRsrc with the expression:

PXI?*INSTR{VI_ATTR_MANF_ID ==0x35BC && VI_ATTR_MODEL_CODE ==0x0241 && VI_ATTR_SLOT==<Device Number>}

If this function returns successfully, the device is compatible.

A complete search string also includes the chassis number since it is possible to have a multi-chassis system with the same device in the same slot of each chassis. VISA can also be configured to automatically scan any remote computer configured in your network. If this is the case, the *viFindRsrc* command may return information of devices located in separate computers. The resource string for a remote device includes its IP address, for example: //100.4.23.12/PXI0::16::INSTR.

When designing a driver, consider if you want to add the capability of controlling your device remotely. For some devices, such as those who map physical memory, remote access is not recommended. If you don’t want utilize this capability, ensure that you configure VISA to not scan remote devices.

Note: The slot number attribute depends on geographical addressing features defined by the PXI system standards, and is only valid for PXI controllers and devices installed in a PXI chassis.

Note: The slot number attribute is supported only by VISA 3.0 and later versions.

Multitasking Support
The driver can use the VISA *viLock* and *viUnlock* functions to prevent simultaneous access to the device by more than one user application.

The driver should not lock the device session for longer than required. Failure to unlock a session may introduce unexpected behavior such as priority inversion and thread starvation.

Related Links

Considerations in Implementing Real-Time Drivers describes factors in choosing a driver implementation approach for the NI Real-Time programming environment.
Developing a LabVIEW Real-Time Driver for a PXI, cPCI or PCI Device describes creating a native LabVIEW driver using the NI-VISA LabVIEW API for LabVIEW Real-Time.

For more information about the VISA standard, visit vxipnp.org. For more information about NI-VISA, refer to the *NI-VISA User Manual* and the *NI-VISA Programmer’s Reference*.

See Also:
vxipnp.org
LabWindows/CVI
Calling Code Written in Text-Based Programming Languages