Document Type: Tutorial
NI Supported: Yes
Publish Date: Mar 30, 2009

Developing a LabVIEW Real-Time Driver for a PXI, cPCI or PCI Device

Overview

This document is intended for anyone who is interested in creating a LabVIEW Real-Time driver for a third-party PXI, Compact PCI or PCI device.

Table of Contents

Introduction

The goal of this document is to provide guidelines for creating a LabVIEW Real-Time driver based on the VISA application programming interface. The resulting driver will be a VI library for controlling the device.

This document assumes that the reader has a good understanding of the LabVIEW programming environment and register-level programming techniques.

Background

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 integrator 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 the Windows and LabVIEW 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 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 two application programming interfaces:
Driver API – The set of VIs that the application program uses to control an I/O device.
VISA API – Generally, the set of function calls/VIs defined by the VISA standard for instrument control. Specifically, the VISA API functions related to register-level programming.

Device Drivers for Computer Systems
In the context of computer systems, the term "driver" refers generically to the system software that allows user application software to control the hardware devices that are installed in the system.

The driver exposes a set of high-level, application-oriented functions through the 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 for Test and Measurement Systems
In automated test and measurement systems, the term "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 greatly simplifies system integration and that extends the useful lifetime of instrument drivers for decades.

In Windows-based systems, device drivers are used as instrument drivers. As we will see, however, an instrument driver implementation makes sense for compatibility with LabVIEW Real-Time.

Terminology Used in This Document
The remainder of this document will refer to driver components that run at the kernel level collectively as the "device driver". The driver components that run at the user level will be referred to as the "user driver".

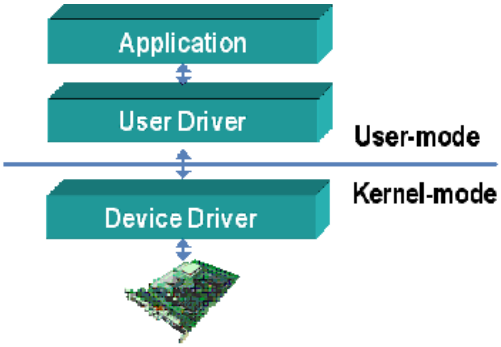

Figure 1. Generic Device Driver Architecture

Each of the driver layers in the previous diagram represents several sub-layers that are specific to the operating system and to the driver implementation.

Unlike Windows drivers, instrument drivers are completely implemented in the user layer. There is no need for kernel-level programming, because VISA abstracts all of the underlying details for the driver developer. VISA provides the low-level API that is used to control a device under LabVIEW Real-Time.

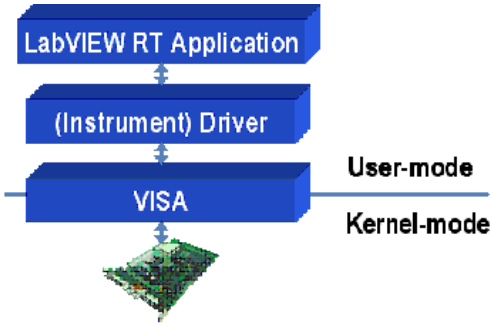

Figure 2. LabVIEW Real-Time Driver Architecture

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 will follow this convention for PCI and PXI devices.

Process Overview

I. Gather Documentation
Having all the documentation at hand will make your work much easier. The following is a list of driver-related documentation that is very helpful when creating a VISA-based driver.

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

3. Example application - When developing a driver, it is very helpful to create a sample application program that you can use to test the driver.

The following paragraphs provide a high-level overview of the steps that must be completed in the process of creating a driver for LabVIEW Real-Time. The remainder of this document provides suggestions and examples for completing these steps.

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 LabVIEW Real-Time, this subject is discussed in detail in the application note Configuring LabVIEW Real-Time to Recognize a Third Party Device.

III. Create the VISA-based Driver
Create the Driver API
It is always a good practice to create a modular driver architecture that leverages each of its components. For example, you could create a low level set of VIs that provide full control of device features for advanced users. On top of that layer, you could create an “Easy API” for less advanced users. The “Easy API” VIs would each consist of common sequences of “advanced” VIs that encapsulate basic application steps for the device.

It is also important to design an API with very well defined states or phases. For example, drivers commonly provide functions for an “Open” phase, a “Configure” phase, a "Run" phase, and a “Close” phase.

Ideally, the VISA functionality will be hidden from the application developer. Programmers should be required to learn the driver functionality, but not VISA.

Develop the Basic Communication Mechanism
In general, the steps required to communicate with the device include:

-

Use VISA to find the device in the system

- Open a VISA session to the device
- Through the VISA API, perform read and write operations to the device's registers
- When all operations are completed, close all the opened VISA sessions.

Develop Interrupt Support
If the device requires interrupt support, develop an interrupt processing mechanism to work within LabVIEW using the functionality provided by VISA.

Develop DMA Support
VISA also provides functionality for implementing Direct Memory Access (DMA) support. The DMA methodology is device-specific, so an in-depth understanding of the device's hardware architecture is required.

Reference Example
The LabVIEW NI-VISA driver for the VMIC 5565 reflective memory card complies with the structure and conventions described in this paper. It can be a useful resource to examine the implementation of complete driver for LabVIEW Real-Time. The driver is available free of charge from the Developer Zone on ni.com. Note that it is not necessary to have the PCI/PXI device to examine the structure of the driver.

Developing Programmed I/O Communication

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
To open a session to the specific device, the driver must call the *VISA VI Open* VI. This VI takes an expression such as PXI1::14::INSTR, which the user must obtain from the *VISA Find Resource* VI. To avoid this, the *VISA Open* VI can use the *VISA Find Resource* VI function to find the appropriate devices and then pass one resource name to the *VISA Open* VI. You can use a user supplied "device number" to select an index into the array of resource names returned from the *VISA Find Resource* VI. The *VMIC Open VI* shows an example of this technique.

The Open VI should return a VISA resource name that can be used as a *refnum* throughout the rest of the VIs. Therefore the driver VIs could simply use a VISA Resource Name control and indicator as *refnum*-type input and output for each VI. However you may want to cluster this resource name with other information or use a LabVIEW variant control as the "refnum." (See Local Copy of Register Map below.) Regardless of the *refnum* type, create a strict typedef control to avoid having to edit each VI should the *refnum* control need to be changed.

It should be noted that distinguishing between multiple PCI/PXI devices of the same type in one system can be difficult. The exact resource name for a particular PXI slot depends on the combination of controller and chassis. Therefore it is not possible to use the slot number directly to identify a device. *VISA Find Resource* VI will list the devices in an arbitrary order.

The Close VI should simply call the *VISA Close* VI to close the session to the device.

VISA Find Resources
One of the powerful features of VISA is a single interface for finding and connecting to devices. The VISA resource manager accomplishes this by assigning unique resource names such as PXI1::14::INSTR, for each device, and by exporting a query-based service for retrieving resource names.

The driver can get the resource name for the device using the *VISA Find Resource* VI. 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}

This query expression allows you to verify that you are opening a session to the correct device in your system.

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

In many cases, the high-level functions *VISA In* (8, 16 or 32 bits) and *VISA Out* (8, 16 or 32 bits) can be used for register read/write access. If the driver repeatedly accesses registers in one address space, however, the low level *VISA Map Address*, *VISA Peek* and *VISA Poke,* may be more efficient.

The VISA Move functions *VISA Move In* and *VISA Move Out* can be used to efficiently transfer large data buffers from memory to memory.

PCI devices can be accessed 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 In* VIs with the address space parameter value *VI_PXI_CFG_SPACE*.

The *VISA Find Resource* VI 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 will normally contain CSR registers for configuring or controlling the device. Some address spaces may contain address windows to on-board memory that can be read or written by the driver.

Interrupt and DMA Support

Interrupt Support

Since LabVIEW does not support the concept of callbacks, LabVIEW cannot respond to interrupts the same way that a Windows driver would. What VISA provides is a mechanism for detecting interrupts. VISA informs LabVIEW that an interrupt occurred by sending an event message. When a VISA event has been received, LabVIEW can proceed to handle the interrupt.

VISA generates an event whenever an interrupt occurs, but there is no way to immediately inform LabVIEW that an event happened. LabVIEW applications must use the *VISA Wait On Event* VI to determine when an interrupt occurs. The interrupt sensing and processing sequences are described in the following sections.

The experienced Windows programmers needs to realize that creating a LabVIEW driver requires a paradigm change. The concept of callbacks or 'event driven programming' needs to be replaced with an approach based on 'data flow programming'.

PCI Interrupt Sequence Summary

In a simplified view, PCI interrupts occur in the following sequence:

1. The operating system loads the device's interrupt configuration information at boot time. For 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 user application should call a function provided by the driver API that programs the CSR registers so that the device will generate 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 the *VISA Enable Event* to enable VISA events for the session before enabling hardware interrupts on the device. The event you need to enable is the *PXI Interrupt Event*. If the device is allowed to generate interrupts before VISA events have been enabled, there will be no mechanism to remove the first interrupt, resulting in a system lockup condition. To enable hardware interrupts, the user application should call a VI provided by the driver API that programs the CSR registers for this purpose. It is good practice to include the *VISA Enable Event* call in this function.

Enabling interrupts is usually performed as part of the device initialization sequence, but may be used (with the Interrupt disable function) 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. For Windows and LabVIEW Real-Time, you can use the VISA Driver Development Wizard to configure these algorithms.

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 Enable Event* VI with an event type parameter of *VI_EVENT_PXI_INTR*.

The size of the VISA event queue can be modified depending on the application. If the interrupt handling routine takes a considerable amount of time processing the interrupt, it may be desirable to increase the event queue size to ensure that no interrupt is lost. The driver can change the event queue size using the VISA *Max Queue Length* property.

The event queue defaults to a value of 50. Visa does not let you dynamically configure queues lengths. That is, you can only modify the queue length for a given session before the first invocation of the *VISA Enable Event* VI.

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.

The following diagram provides an overview of the interrupt processing procedure:

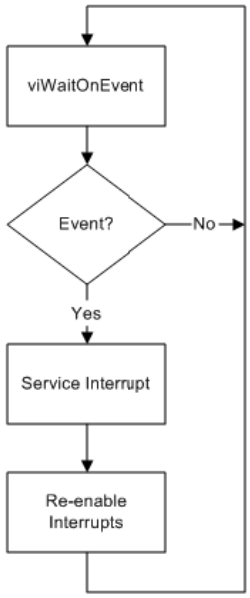

Figure 3. Interrupt Processing Procedure

The methods used to process interrupts vary between devices and even between applications using the same device. Suggested steps for processing an interrupt are described below. These steps take place after the *VISA Wait On Event* VI indicates that an interrupt occurred.

1. Determine the interrupt source. Devices often have multiple sources that can generate an interrupt. It is the driver's responsibility to query the device to determine the source of each interrupt.
2. Service the interrupt. There are 2 main approaches a driver developer can take:

a. Implement a function that services interrupts in a predefined way. This approach may be 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. It is the driver's responsibility to 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.

The interrupt processing mechanism can be exposed as a set of VIs in the driver API. The user application calls the interrupt processing mechanism periodically. When called, it senses interrupt events by calling the *VISA Wait On Event* VI . If no interrupts have occurred, the *VISA Wait On Event* VI and the interrupt processing mechanism should be programmed to return immediately (timeout = 0). This approach allows a real-time application to call the interrupt processing mechanism only when a VISA interrupt event has occurred. Note, however, that 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 *VISA Wait On Event* VI, that waits for an interrupt to occur. By design, 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 the *VISA Wait On Event* VI, you may starve the time critical loop and affect determinism. Your implementation should call the *VISA Wait On Event* VI 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 the *VISA Wait On Event* VI once, and interrogates the interrupt source
- Helper functions for servicing interrupts
- A helper function for re-enabling hardware interrupts on the device.

Disabling 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 the *VISA Disable Event* VI to disable VISA interrupt events for the session.

Example
See the VMIC 5565 driver (previously referenced) for an example interrupt handler implementationi

Tip: As mentioned earlier, the LabVIEW Real-Time mechanism for handling interrupts consists on calling the *WaitOnEvent* VISA function to verify if an interrupt occurred. Because of this, the function/VI that includes the *WaitOnEvent* call will be called periodically in a loop. Make sure that your interrupt service handler is optimized to be called repeatedly. Avoid practices that would slow it down or affect determinism.

Troubleshooting Interrupts
There are numerous interrupt handler implementation errors that can cause unintended system behaviors. Here are some of the more common causes and effects.

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 will 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 will cause 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. VISA functions can be used 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 needs to be locked (non pageable) so that the DMA controller can ensure that the buffer will be available throughout the whole operation. Because of this, DMA operations must be performed using buffers allocated with VISA and not with memory buffers allocated in LabVIEW.

The following steps are required for setting up a DMA operation:
1. Open a session to a VISA Memory Access (MEMACC) Resource by calling the *VISA Open* VI with the VISA resource name PXI0::MEMACC . Note that this VISA session is separate from the device session and will need to 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* VI, to allocate and reserve the memory buffer. This function will return an offset value that can be used to program the DMA controller on the device.
3. Initialize your DMA controller. This step depends on the type of controller. Check the DMA controller's documentation for specific initialization information.

Warning: Make sure 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, however always specifies the buffer size in bytes. When setting up a DMA data transfer, make sure 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 will transfer a total of 4000 bytes (1000 long words). The driver must specify 4000 (and not 1000) as the allocation size using the *VISA Memory Allocation* VI. The DMA controller will continue to increment its memory pointers until the operation is completed, so allocating less memory than required will cause other memory to be overwritten, which in turn may result in corrupted data or a system crash.

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 *VISA Move In* VI. Similarly, before performing a DMA Write operation, the driver must copy the user data into the locked memory using the function *VISA Move Out*.

The following steps are required for a DMA Read operation:
1. Configure the DMA controller to perform a Read operation. The destination offset will be the offset returned by the *VISA Memory Allocation* VI.
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 is completed.
3. When the DMA transfer completes, the data will be located in the locked memory buffer. Copy the data into a user buffer using the *VISA Move In* VI with an address space value of *VI_PXI_ALLOC_SPACE*.

The following steps are required for a DMA Write operation:
1. Configure the DMA controller to perform a 'Write' operation. The source offset will be the offset returned by the *VISA Memory Allocation* VI function.
2. Copy the data from the user buffer into the locked memory using the *VISA Move Out* VI with an address space value of *VI_PXI_ALLOC_SPACE*.
3. 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 is completed.

Cleaning up after a DMA operation
After all DMA operations have completed all allocated memory must release. Failure to deallocate memory buffers and to close the MEMACC session will result in memory leaks.

The following steps are required for cleaning up a DMA operation:
1. Call the *VISA Memory Free* to deallocate the memory
2. Call the *VISA Close* VI to close the VISA MEMACC session.

Recommended Practices

Distinct Device Open and Close 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 modularity, it is a good practice to isolate the VISA calls in a separate layer of the driver. This layer can be implemented as a set of subVIs that includes all the VISA calls. The rest of the driver VIs call this layer to get access to the device. In most cases, the user doesn't need to be aware that the driver is implemented over VISA. All the user should see are VIs that allow communication with the device. Ideally, all the VISA functionality should be hidden from the application programmer. The following diagram depicts this concept.

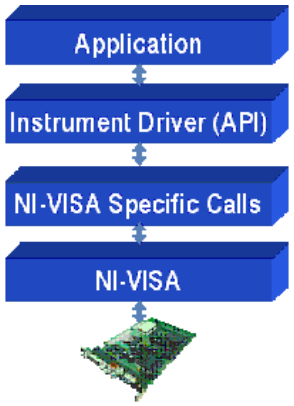

Figure 4. VISA based driver architecture

Hierarchical API
It is good practice to 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 is a good investment because it allows new users to get up and running quickly.

As an 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" *Initialize_Device* configuration VI could be provided that performs these operations using commonly-used default parameter values.

Memory Allocation
The memory allocation strategy 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 will be called repeatedly such as in DMA support code or in the main program loop. If the driver includes functions that dynamically allocate memory, you

should consider implementing helper functions that pre-allocate and release a pool of memory for the driver to use.

Note that failure to release allocated memory will result in a memory leak.

Cached Copy of Device Configuration Information
In many cases, configuration registers are write-only, so the driver must store a copy of the register values in local memory in order to modify parameters that are represented by bit fields within registers. Register values can be cached in several ways:

- A LabVIEW variant can be used for this purpose. Convert the VISA resource name into a variant in the Open VI and use this variant as the refnum in other driver VIs. Additional data can be associated with the variant using the Get/Set Variant Attribute functions. The VMIC 5565 driver DMA functions use this technique.
- Local copies of register values can also be stored in global variables. However, if the driver supports more than one device session, the information will need to be indexed for each device instance.
- Another alternative is to create a cluster control (strictly typed) that contains the VISA resource name control and the additional information. However, this method may be inconvenient if more than a few controls are needed. Adding new data items may also result in the need to resize the front panels of VIs that use the cluster.

Cached parameters can also improve execution performance. It often takes much 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 just read that value from local memory rather than to retrieve the information from the device.

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

Persistent Configuration and Status Information
In many cases, it is desirable to 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.

Windows drivers can store this information in the Windows registry; however LabVIEW Real-Time 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 friendly API is a very important requirement for any driver. There are many approaches to creating an API, and it ultimately becomes a design issue that the developer needs to resolve based on the projects specification. In general, the API should be transparent to the user, regardless of the platform. The final user should not be impacted because a platform change or its underlying implementation. As much as possible, the API of the VISA driver should be equivalent to the API of the Windows driver.

Here are some common approaches 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 will pass 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.
- Another approach, similar to the previous one, relies 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 will pass the session to any subsequent call to the driver.
- In a third approach, 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. Notice that no board number is required for this approach.
- In some cases, you may want to hide the VISA implementation from the user. The API would need to provide some means of representing the board or providing some type of handle that can be internally translated or cast to a VISA session. An earlier section of this document shows an example of this approach using LabVIEW variants.

Tip: Any of the previous approaches may be used, but the recommended approach is to 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). This approach supports fully independent use of multiple boards at the same time, by eliminating race conditions that can cause unpredictable effects.

Driver Implementation
In one simple implementation approach, 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 variant of this approach 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 the *VISA Find Resource* VI with the expression:

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

If this call returns successfully, then the device is compatible.

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

When designing a driver, you must consider whether you want to add the capability of controlling your device remotely. Some devices (like those who map physical memory), remote access is not recommended. If you don't want utilize this capability, just make sure that you configure VISA not to 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 Lock* and *VISA Unlock* VIs 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 Documents

Considerations in Implementing LabVIEW Real-Time Drivers describes some of the factors in choosing a driver implementation approach for LabVIEW Real-Time.

For additional information regarding the VISA standard, visit vxipnp.org. For additional information on NI-VISA, see the NI-VISA User Manual and the NI-VISA Programmer's Reference.

See Also:
National Instruments Product Manuals
vxipnp.org

Conclusion

Developing a LabVIEW Real-Time driver can be a challenging task. Be sure that you understand the requirements, needs and limitations of your device and the target platform(s) before implementing any part of the driver.

This document presents some basic concepts related to this task. As with the design of any software, there are few strict rules and the final architecture will depend on the programmer's style, the type of device, and customer needs.