VISA Events in NI-VISA

Introduction

The VISA operations Read, Write, In and Out allow basic I/O operations from the controller to instruments. These operations make up the basics of instrument I/O, but there are many cases where these simple communication commands are not sufficient to utilize all the functionality built into GPIB and VXI devices.

Consider the following situation. Suppose there is a system which contains a sophisticated message based VXI device that can perform a variety of different measurement operations. Some of these measurements require as much as several seconds for the device to obtain a stable reading. With the VISA operations mentioned above there is not a good way to handle this situation.

Once the command to obtain a measurement is written to the device there is no way of knowing when the measurement is available. After doing the write command, the program could continuously attempt to do VISA Reads from the device until one is successful. However, there may be other operations that the application needs to perform at the same time. What if the program needs to control other instruments, write data to disk, or update the screen display? If obtaining the results of the measurement from the instrument is not urgent, it may be possible to wait a long time to see if the result is available. Alternatively the application might attempt to read the result periodically until it is available, but if it is performing a lot of other operations this may not be possible. Also, if speed is important this is not efficient.

Clearly there is a need for some other means of communication between VXI resources and a VISA application. In VISA, **events** provide this alternate means of communication. Events are notifications to the application that a certain condition has occurred in the system. Additionally, these notifications occur through a channel independent of the data path. This independence allows the notifications to occur **while** data is being transferred.

This application note discusses VISA events and how to use them. It goes over some of the options the programmer has in dealing with an event. It also discusses what attributes are associated with each event and how these attributes can be used in a VISA program. At the end of the document, there are three examples of programming with VISA events. The first example demonstrates the synchronous method of waiting for a GPIB SRQ. The example is written with LabWindows/CVI but the code would be the same for any C compiler and is very similar in Visual Basic. The second example is a LabVIEW example also showing synchronous SRQ events. The third example, also written in LabWindows/CVI demonstrates the use of asynchronous I/O. It uses VISA events to notify the program that the asynchronous I/O operation has completed.

Events and Event Attributes

VISA events are objects, and in being objects, they can have attributes associated with them.Currently there are five events defined for instrument control. Listed below are the defined events and their corresponding attributes.

VISA Event	Associated Attributes
VI_EVENT_SERVICE_REQ – notification of a service request from a device on the bus (valid for VXI or GPIB)	• None
VI_EVENT_VXI_SIGP - notification of a VXIbus signal or VXIbus interrupt (valid only for VXI)	• **VI_ATTR_SIGP_STATUS_ID** – the 16 bit Status/ID value retrieved during the IACK cycle or from the signal register
VI_EVENT_TRIG – notification of a VXIbus Trigger (valid only for VXI)	• **VI_ATTR_RECV_TRIG_ID** – the trigger line on which the trigger was received
***VI_EVENT_IO_COMPLETION** – notification that an asynchronous I/O operation has completed (valid for all interfaces)	• **VI_ATTR_STATUS** – status information about the operation • **VI_ATTR_RET_COUNT** – contains the value for the number of bytes (or elements) transferred
VI_EVENT_VXI_VME_INTR – this is notification of a VXIbus or VMEbus interrupt (valid only for VXI)	• **VI_ATTR_INTR_STATUS_ID** – the 32 bit Status/ID value retrieved during the IACK cycle • **VI_ATTR_RECV_INTR_LEVEL** – the VXI interrupt level on which the interrupt was received

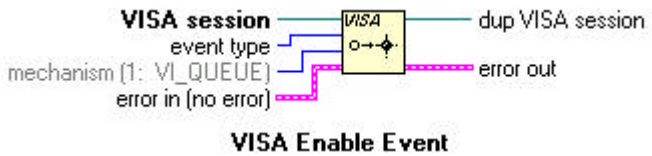
* not supported in LabVIEW

Preparing for an Event

For an event to be detected in VISA, the event must first be enabled. Enabling an event simply configures the driver up to detect it and tells the driver what mechanism to use in the handling of the event for the given session. It is important to realize that this is required for all event types, and if an event is not enabled it will not be detected by VISA. The function used to enable events is `viEnableEvent()`.

```
status = viEnableEvent (ViSession vi, viEventType eventType,
ViUInt16 mechanism ,ViEventFilter context);
```

Note: the **context** parameter is reserved for future use and currently the constant VI_NULL should be used for this parameter.



In handling events, there are two possible mechanisms that can be employed. These two mechanisms are the queuing mechanism (synchronous) and the callback mechanism (asynchronous). The queuing mechanism is generally used when the servicing of the event is not time-critical. The callback mechanism is used for more time-critical handling of events and is invoked immediately on every occurrence of the specified event type. The mechanism of the event handling is also specified in the method `viEnableEvent()`.

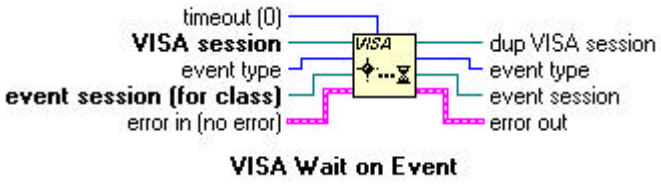
To specify which mechanism is to be used, the parameters VI_QUEUE and/or VI_HNDLR are sent to `viEnableEvent` as the third parameter. For example, to set up a queuing mechanism to handle GPIB SRQ events the syntax would be:

```
viEnableEvent (instr, VI_EVENT_SERVICE_REQ, VI_QUEUE, VI_NULL)
```

Handling an Event Synchronously– The Queuing Method

In the queuing method of handling events, the driver keeps track of all the events that occur and stores them in a queue allowing the programmer the flexibility of receiving events only when requested. Events are "dequeued" with the `viWaitOnEvent()` method.

```
ViStatus viWaitOnEvent (ViSession vi, ViEventType inEventType, ViUInt32
timeout, ViPEventType outEventType, ViPEvent outContext)
```



VISA Wait on Event

If one or more events are available, the function will pull an event off of the queue, oldest event first, and immediately continue. If no event is available in the queue when the program reaches this call, the program will halt until an event is received or until the timeout period is exceeded.

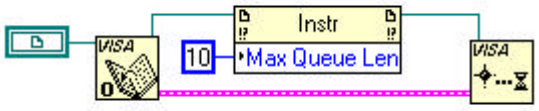
To check the queue without waiting for the next event to occur, a call can be made to `viWaitOnEvent`() with the timeout set to VI_TMO_IMMEDIATE in C or Visual Basic, or by wiring a zero for the timeout value in LabVIEW. In this case, if an event exists, it is pulled of the queue and goes directly into the handling routine. If no event is waiting, the function returns immediately and the code continues.

By default, the VISA driver will queue up to 50 events per session. If additional events occur, the new events will be discarded. However, VISA does allow the

programmer to explicitly specify the size of the event queue if he or she wishes to do so. This is done with the VISA attribute VI_ATTR_MAX_QUEUE_LENGTH. The one stipulation is that this call must be made **before** the event is enabled for a given session.

Example:

```
status = viSetAttribute(instr, VI_ATTR_MAX_QUEUE_LENGTH, 10);
status = viEnableEvent (instr, VI_EVENT_SERVICE_REQ, VI_QUEUE,
VI_NULL);
```



Each event type in a session has an individual queue. In other words there is an event queue per session as well as per event type. For example, if a queue size of 10 events is defined for a VXI-DAQ card, this means that VISA would be able to queue up to 10 VI_EVENT_IO_COMPLETION events **and** 10 VI_EVENT_TRIG events provided that both of these events have been enabled.

It is important to remember at this point that every occurrence of an event is accessed via a handle to it. This is automatically taken care of by `viWaitOnEvent()`. When an event is picked off the queue by viWaitOnEvent(), a handle to that specific event gets returned. This handle can be used to reference the particular occurrence of an event when examining attributes and should be passed to `viClose`() to close the handle to the event once it has been handled.

Example A demonstrates the queuing (synchronous) mechanism of event handling in a C based program.

Example B demonstrates the queuing (synchronous) mechanism of event handling in LabVIEW.

Handling an Event Asynchronously – The Callback Mechanism

Another option that VISA provides for handling events is the callback mechanism. The callback mechanism is sometimes referred to as the handler mechanism. The concept behind the callback mechanism is the following: Before an event is enabled, a handler function is installed. This is a function that will automatically execute every time an event is received. With this mechanism, very time critical events can be handled immediately. Currently VISA allows only one handler callback to be installed per session, but this may change in later versions of VISA. The handler callback is installed

by the method `viInstallHandler()` and should be called before `viEnableEvent()`.

```
ViStatus viInstallHandler(ViSession vi, ViEventType eventType,
ViHndler handler, ViAddr userHandle)
```

One nice feature of the callback handler is that the driver automatically handles the closing of each event handle. No additional calls need to be made in the code to take care of these occurrences.

Example C demonstrates how to use a callback (asynchronous) handler to handle events.

It is important to note that because of the dataflow programming structure of LabVIEW it is not possible to use the callback mechanism of event handling in LabVIEW. There is a **Wait On Event Asynchronously.vi** which uses LabVIEW programming techniques to simulate asynchronous event detection. With this function, any code running parallel to the dataflow of the VISA sequence will continue to execute. This vi is not truly asynchronous but rather polls for an event to occur.

Enabling Both handling Mechanisms Simultaneously

It is also possible to use both the queuing and the callback methods of event handling simultaneously. After an initial call to `viEnableEvent()` specifying one mechanism, a subsequent call specifying the other mechanism will enable both mechanisms of handling at the same time. It is important to note that this successive calling will NOT undo the first mechanism.

status = viEnableEvent (instr, VI_EVENT_SERVICE_REQ, **VI_HNDLR**, VI_NULL)
status = viEnableEvent (instr, VI_EVENT_SERVICE_REQ, **VI_QUEUE**, VI_NULL)

To set up both mechanisms to be enabled simultaneously with one function call the following syntax can be used:

status = viEnableEvent (instr, VI_EVENT_SERVICE_REQ, **VI_QUEUE|VI_HNDLR**, VI_NULL)

What To Do With An Event Once It Is Received

Notification that an event has occurred isn't much good if the program doesn't react with the proper response. Often times just knowing that an event has occurred isn't enough. More information generally needs to be known such as why the event has occurred. VISA provides a means for gathering this information via a concept called attributes. An attribute is basically a characteristic of any object. In this discussion, the object is the handle to the particular occurrence of an event and the attribute is a value that gives information unique to that particular occurrence of the event. Common

attributes for each event were mentioned on page 1 of this application note, but an example is discussed below.

A VXI system has multiple instruments and a VXI-device at logical address 10 asserts an interrupt because it is ready to pass data to the controller. The interrupt is received by the VISA driver and is passed to the VISA code (either asynchronously with a handler or synchronously with `viWaitOnEvent()`). The method `viGetAttribute`() is then called to get the value of VI_ATTR_SIGP_STATUS_ID. This attribute is a 16 bit value in which the lower 8 bits contain the logical address of the device asserting the interrupt (this is always the case with all VXI devices) and the upper 8 bits contain information as to why the interrupt was asserted. The driver decodes the low bits and determines that the device at logical address 10 asserted the interrupt; the application can determine this as well, but the session used to get the event already contains this information. It is up to the program to interpret the upper 8 bits as the message "Ready to send data." Now the program can handle this information accordingly.

Currently, the event VI_EVENT_SERVICE_REQ does not have an attribute defined for it other than VI_ATTR_EVENT_TYPE. Because of this fact, `viGetAttribute`() cannot be used to get the status information. Instead, a call must immediately be made to `viReadSTB()`. In fact, if this call is not made, it will prevent further service requests from being detected.

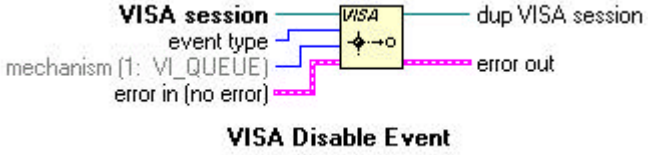
Another attribute that is important to mention when discussing VISA events is VI_ATTR_TRIG_ID. This is a session attribute (as opposed to an event attribute) which is used to determine which trigger events a session will receive. By default it is set to software triggers. This attribute can only be changed **before** trigger events are enabled. After they are enabled it is a read only value. The following line of code shows how to set the ID to detect TTL triggers on trigger line 3.

```
status = viSetAttribute (instr, VI_ATTR_TRIG_ID, VI_TRIG_TTL3);
```

Disabling Events

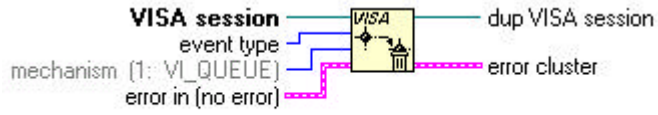
Once the program reaches a point where it is no longer necessary to detect events, a call should be made to `viDisableEvent()`.

```
ViStatus viDisableEvent(ViSession vi, ViEventType eventType,
ViUInt16 mechanism)
```



viDisableEvent() notifies the driver that it is no longer necessary to detect events and report them to the application. This method will only disable the specified event on the specified session. Using viDisableEvent() will prevent a queue from receiving additional events, but it will not clear out any events that are already present in the queue. To flush the queue, a program can call `viDiscardEvents()` which will safely clean out the queue and close each event in the queue. The C prototype and LabVIEW vi are listed below:

```
ViStatus viDiscardEvents (ViSession vi, ViEventType eventtype, ViUInt16
mechanism)
```



VISA Discard Events

In an earlier section it was mentioned that an event could be set for both synchronous and asynchronous event handling. If this is the setup, a program can disable one of these and leave the other in place by calling `viDisableEvent()` and specifying the method to be disabled. To disable both methods two calls must be made to `viDisableEvent`() or else the methods can be OR-ed together as discussed for `viEnableEvent()`.

Conclusion

VISA events provide a means of reporting information about the status of an instrumentation bus to the program. VISA also provides the programmer with the flexibility to decide how to receive and respond to the event. This ability to use events, along with the programming ease of use of VISA, make VISA a very powerful solution for any instrumentation system, whether it be VXI, GPIB, serial or parallel.

Other VISA Resources

NI-VISA User Manual
NI-VISA Programmer Reference Manual

Example A

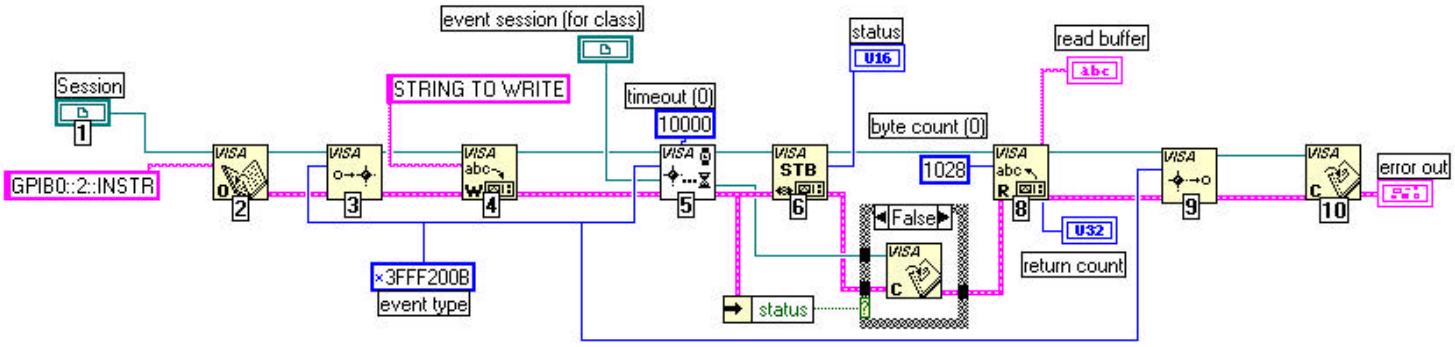
```
/*****************************************************************************/
/*  This example shows how to enable VISA to detect SRQ events
/*  The program writes a command to a device and then waits to receive
/*  an SRQ event before trying to read the response.
/*  General Program Flow:
/*  Open A Session To The Visa VISA Resource Manager
/*  Open A Session To A GPIB Device
```

```
/*  Enable SRQ Events
/*  Write A Command To The Instrument
/*  Wait to receive an SRQ event
/*  Read the instrument's status byte
/*  Use the data
/*  Close the handle to the event
/*  Disable SRQ Events
/*  Close The Instrument Session
/*  Close The Resource Manager Session
/*********************************************************************/

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

static ViUInt32 timeout = 30000; //Timeout in milliseconds
static ViEvent ehandle;
static ViEventType etype;
static ViStatus status;
static ViSession inst,dfltRM;
static ViUInt16 stb;
static ViUInt32 rcount, bytes;

int main (int argc, char *argv[])
{

        /*  First we open a session to the VISA resource manager.  We are
         *  returned a handle to the resource manager session that we must
         *  use to open sessions to specific instruments.
         */

        status = viOpenDefaultRM (&dfltRM);

        if (status < VI_SUCCESS)
        {
                printf("The session to the Resource Manager Could Not Be Opened");
                exit (EXIT_SUCCESS);
        }

        /*
         *  Next we use the resource manager handle to open a session to a
         *  GPIB instrument at address 2.  A handle to this session is
         *  returned in the handle inst.
         */

        status = viOpen (dfltRM, "GPIB::2::INSTR", VI_NULL, VI_NULL, &inst);
        if (status < VI_SUCCESS)
        {
                printf("The session to the device simulator");
                viClose(dfltRM);
                exit (EXIT_SUCCESS);
        }

        /*  Now we must enable the service request event so that VISA
         *  will receive the events.  Note: one of the parameters is
         *  VI_QUEUE indicating that we want the events to be handled by
         *  a synchronous event queue.  The alternate mechanism for handling
```

```
 *  events is to set up an asynchronous event handling function using
 *  the VI_HNDLR option.  The events go into a queue which by default
 *  can hold 50 items.  This maximum queue size can be changed with
 *  an attribute but it must be called before the events are enabled.
 */

status = viEnableEvent (inst, VI_EVENT_SERVICE_REQ, VI_QUEUE, VI_NULL);

if (status < VI_SUCCESS)
{
        printf("The SRQ event could not be enabled");
        viClose(inst);
        viClose(dfltRM);
        exit (EXIT_SUCCESS);
}

/*
 *  Now the VISA write command is used to send a request to the
 *  Instrument to generate a sine wave and assert the SRQ line
 *  When it is finished.
 */

status = viWrite (inst, "command string",  bytes, &rcount);

if (status < VI_SUCCESS)
{
        printf("Error writing to the instrument");
        viClose(inst);
        viClose(dfltRM);
        exit (EXIT_SUCCESS);
}

/*
 *  Now we wait for an SRQ event to be received by the event queue.
 *  The timeout is in milliseconds and is set to 30000 or 30 seconds.
 *  Notice that a handle to the event is returned by the wait on
 *  event call.  This event handle can be used to obtain various
 *  attributes of the event.  It should also be closed in the program
 *  to release memory for the event.
 */

status = viWaitOnEvent (inst, VI_EVENT_SERVICE_REQ, timeout, &etype,
                                                &ehandle);

if (status >= VI_SUCCESS)
{
        status = viReadSTB (inst, &stb);

        if (status < VI_SUCCESS)
        {
                printf("There was an error reading the status byte");
                viClose(inst);
                viClose(dfltRM);
                exit(VI_SUCCESS);
        }
}
```

```
        else
        {
                        printf("There was an error waiting on the event");
                        viClose(inst);
                        viClose(dfltRM);
                        exit(VI_SUCCESS);
        }

        /*
         *  If an SRQ event was received we first read the status byte with
         *  the viReadSTB function.  This should always be called after
         *  receiving an SRQ event or subsequent events will not be
         *  received properly.  Then the data is read and the event is closed
         *  and the data is displayed.  Otherwise sessions are closed and the
         *  program terminates.
         */

        /* At this point the command to tell the instrument to send back data is issued.
         * Perhaps even a function to print the data to the screen
         * When finished with the event, the handle must be closed
         */

        status = viClose (ehandle);
        if (status < VI_SUCCESS)
        {
                printf("There was an error closing the event handle");
                viClose(inst);
                viClose(dfltRM);
                exit(VI_SUCCESS);
        }

         /* Now we should disable the previously enabled event for completeness */

        status = viDisableEvent (inst, VI_EVENT_SERVICE_REQ, VI_QUEUE);

        if (status < VI_SUCCESS)
        {
                printf("There was a error closing one of the sessions");
                viClose(dfltRM);
                exit(VI_SUCCESS);
        }

        /*Now we must close the session to the instrument and the session
         to the resource manager.
         */
        status = viClose (inst);
        status = viClose (dfltRM) + status;

        if (status < VI_SUCCESS)
        {
                printf("There was a error closing one of the sessions");
        }

        return 0;
}
```

Example B



1. **VISA Session**:
   Tells VISA to create a data structure of the necessary format (in this case a generic instrument type is used). This is created by dropping a VISA Session Control on the front panel. This is located in the Path/Refnum Controls Sub-Palette. To change the session type, pop up on the control and select the appropriate type from the VISA Class menu.
2. **VISA Open**
   Open a session to the GPIB instrument at primary address 2.
3. **VISA Enable Event**
   Prepares VISA to detect SRQ's. This is done with the value of hex 3FFF200B.
4. **VISA Write**
   Send a command to the device to prepare data and issue SRQ when ready.
5. **VISA Wait On Event Asynchronously**
   Waits for the SRQ to occur. This vi will return each event that occurs. This VISA "Session Control" is set to Generic Event. Using this asynchronous function will allow any vi's that are running parallel to the above data flow to execute while waiting for the event.
6. **VISA Read Status Byte**
   Reads the gpib status byte after a serial poll. This MUST be done when waiting on SRQ events.
7. **VISA Close**
   Closes the "session" to the event.
8. **VISA Read**
   Reads the data previously requested from the instrument (if necessary).
9. **VISA Disable Events**
   Tells VISA to stop keeping track of specified events. Allows VISA to deallocate resources.
10. **VISA Close**
    Closes the session opened to the GPIB Device.

NOTE: Another VI, **Wait For RQS**, incorporates steps 5,6 and 7 into one VI. For your application you may choose to use **Wait for RQS**.vi instead of what is depicted here.

Example C

```
/*****************************************************************************/
/*  This example shows how to set up an asynchronous callback function
/*  that is called when an asynchronous input/output operation completes.
/*  The code uses VISA functions and sets a flag in the callback for the
/*  completion of an asynchronous read from a GPIB device to break out of
/*  a loop.
/*  The flow of the code is as follows:
/*  Prototype the handler
/*  Define the handler function
/*  Open A Session To The Visa Resource Manager
/*  Open a Session to a GPIB Device
/*  Install A Handler For Asynchronous IO Completion Events
/*  Enable Asynchronous IO Completion Events
/*  Write A Command To The Instrument
/*  Call The Asynchronous Read Command
/*  Start A Loop That Can Only Be Broken By A Handler Flag Or Timeout
/*  Perform any necessary operations on the data
/*  Disable Asynchronous I/O Completion Events
/*  Close The Instrument Session
/*  Close The Resource Manager Session
/*****************************************************************************/


#include <userint.h>
#include <utility.h>
#include <ansi_c.h>
#include <visa.h>

static ViUInt32 writecount = 30;
static ViUInt32 readcount = 30;
static ViBoolean stopflag = VI_FALSE;
static ViUInt16 tcount;
static ViUInt32 rcount;
static ViJobId job;
static ViChar data[1028];
static ViAddr uhandle;
static ViStatus status;
static ViSession inst, dfltRM;

//  Prototype for the handler for asynchronous i/o completion

ViStatus _VI_FUNCH Ahandler(ViSession vi, ViEventType etype, ViEvent event, ViAddr userHandle);

/*
 * The handler function. The instrument session, the type of event, and a
 * handle to the event are passed to the function along with a user handle
 * which is basically a label that could be used to reference the handler.
 * The only thing done in the handler is to set a flag that allows the
 * program to finish.  This is the most simple of the operations possible
 * in an event handler for demonstation.  Operations that occur in an event
 * handler should be limited to quick operations.  No file i/o or screen
 * updates are recommended because an event handle should take very little
```

```
 * time to execute.
 */

ViStatus _VI_FUNC Ahandler(ViSession vi, ViEventType etype, ViEvent event, ViAddr userHandle)
{
        stopflag = VI_TRUE;
        return VI_SUCCESS;
}


int main (int argc, char *argv[])
{
        /*
         *  First we open a session to the VISA resource manager.  We are
         *  returned a handle to the resource manager session that we must
         *  use to open sessions to specific instruments.
         */

        status = viOpenDefaultRM (&dfltRM);

        if (status < VI_SUCCESS)
        {
                printf("The session to the resource manager could not be opened");
                exit (EXIT_SUCCESS);
        }

        /*
         *  Next we use the resource manager handle to open a session to a
         *  GPIB instrument at device 2.  A handle to this session is
         *  returned in the handle inst.
         */

        status = viOpen (dfltRM, "GPIB::2::INSTR", VI_NULL, VI_NULL, &inst);

        if (status < VI_SUCCESS)
        {
                printf("The session to the instrument could not be opened");
                viClose (dfltRM);
                exit (EXIT_SUCCESS);
        }

        /*
         *  Now we install the handler for asynchronous I/O completion events.
         *  We must pass the handler our instrument session, the type of
         *  event to handle, the handler function name and a user handle
         *  which is not used often but acts as a handle to the handler
         *  function.
         */

        status = viInstallHandler (inst, VI_EVENT_IO_COMPLETION, Ahandler, uhandle);

        if (status < VI_SUCCESS)
        {
                printf("The handler did not successfully install");
                viClose (inst);
                viClose (dfltRM);
```

```
        exit (EXIT_SUCCESS);
}

/*  Now we must actually enable the I/O completion event so that our
 *  handler will detect the events.  Note one of the parameters is
 *  VI_HNDLR indication that we want the events to be handled by
 *  an asynchronous callback.  The alternate mechanism for handling
 *  events is to queue them and read events off of the queue when
 *  ever you want to check them in your program.
 */

status = viEnableEvent (inst, VI_EVENT_IO_COMPLETION, VI_HNDLR,
                                                        VI_NULL);

if (status < VI_SUCCESS)
{
        printf("The event was not successfully enabled");
        viClose (inst);
        viClose (dfltRM);
        exit (EXIT_SUCCESS);
}

/*
 *  Now the VISA write command is used to send a request to the
 *  Instrument to generate a sine wave.
 */


status = viWrite (inst, "command string",  writecount, &rcount);

if (status < VI_SUCCESS)
{
        printf("The command was not successfully sent");
        viClose (inst);
        viClose (dfltRM);
        exit (EXIT_SUCCESS);
}

/*
 *  Next the asynchronous read command is called to read back the
 *  data from the instrument.  Immediately after this is called
 *  the program goes into a loop which will terminate
 *  on an I/O completion event triggering the asynchronous callback.
 *  or after thirty seconds pass.   Note that the asynchronous read command
 *  returns a job id that is a handle to the asynchronous command.
 *  We can use this handle to terminate the read if too much time has passed.
 */


status = viReadAsync (inst, data, readcount, &job);


while(!stopflag && tcount < 30)
{
        tcount++;
        Delay(1);
```

```
                printf("%i seconds have passed\n", tcount);
        }

        /*
         *  If the asynchronous callback was called and the flag was set
         *  we print out the data read back otherwise we terminate the
         *  asynchronous job.
         */

        if (stopflag)

        {
        //perform any necessary operation on the data
        }

        else
        {
                status = viTerminate (inst, VI_NULL, job);
                printf("The asynchronous read did not complete");
        }

    /* Now we should disable the previously enabled event for completeness */

        status = viDisableEvent (inst, VI_EVENT_IO_COMPLETION, VI_HNDLR);

        if (status < VI_SUCCESS)
        {
                printf("There was a error disabling the event");
                viClose (inst);
                viClose (dfltRM);
                exit (EXIT_SUCCESS);
        }

          /*
         *  Now we close the instrument session and the resource manager
         *  session to free up resources.
         */

        status = viClose(inst);
        status = viClose(dfltRM)+status;

        if (status < VI_SUCCESS)
        {
                printf("Error closing one of the sessions\n");
        }

        return 0;
}
```