
How to Use Asynchronous Callback Functions with GPIB Events for Windows NT/98/95

Patrick Williams, Dan Froelich

Introduction

In many GPIB applications, GPIB events are used to monitor system status or respond to instruments requesting service. Traditional methods for responding to GPIB events involve synchronous waits on the events of interest or polling loops that periodically check the GPIB status variable, `ibsta`, for the events of interest. Asynchronous callback functions offer a new approach for automatically executing blocks of code when one or more GPIB events are detected. Using the `ibnotify` function and the asynchronous event notification technique, you can write applications that eliminate wasted processing time associated with using synchronous waits or the polling technique.

This application note introduces the `ibnotify` function and describes how to use asynchronous callback functions when interfacing with GPIB instruments. It also discusses which GPIB events can be used to trigger asynchronous callback functions and which environments support the asynchronous callbacks. The appendixes contain example applications demonstrating the `ibnotify` function in LabWindows/CVI and Visual C, and the `GpibNotify` ActiveX control in Visual Basic.

Benefits of Asynchronous Callback Functions

In basic GPIB communication, a command is written to the device, and then the response is read back. Both of these operations are generally done synchronously with the read following the write. However, many times, devices may require a period of time to take measurements and prepare a response to a given query. If the application simply calls a synchronous GPIB write and then waits until the response is available to do a read operation, your application will sit idle until the device provides a response.

For example, consider the following sequence of function calls:

<code>ibwrt(ud, command, count)</code>	Write a Command
<code>ibwait(ud, RQS)</code>	Wait For the Device To Request Service
<code>ibrd(ud, data, count)</code>	Read The Data

A command is written to the device specified by the unit descriptor `ud` and then the application waits until the device requests service to read back the desired data. The time spent waiting on the RQS GPIB event is lost.

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

A less wasteful approach for waiting on a GPIB event to occur is to write a command to the device and then periodically poll the device until the desired event has occurred. Between polls to the device, other operations can be carried out by the application. For example,

```
ibwrt(ud, command, count);           Write a Command
while (!(ibsta & RQS)) {              End Loop If Device Requested Service
    Other Processing                  Do Other Program Tasks
    ibwait(ud,0);                    Check For Device Requesting Service
}
ibrd(ud, count);                     Read The Data
```

However, this still requires periodic polling calls to be made from the application. These polls take processor time away from other tasks. Depending on the polling frequency, there will be some delay in how quickly the device is serviced once it is ready to respond to the command.

A more efficient solution to this problem would be to use an asynchronous callback function that is called immediately upon the occurrence of a GPIB event. Win32 GPIB applications can asynchronously receive event notifications using the `ibnotify` function or the `GpibNotify` ActiveX control. These features are useful if you want your application to be notified asynchronously about the occurrence of one or more GPIB events. Using `ibnotify`, your application does not need to check the status of your GPIB device. When your GPIB device requests service, the GPIB software automatically notifies your application that the event has occurred by invoking a callback function. The callback function executes when any of the GPIB events specified in the event mask parameter have occurred.

Software for Asynchronous Callbacks

The `ibnotify` function is available under Windows NT and Windows 95 using the native 32-bit NI-488.2M software. Here is a list of the software versions that include `ibnotify`:

- NI-488.2M, version 1.21 and higher for Windows NT
- NI-488.2M for GPIB-ENET, version 1.0 and higher for Windows NT
- NI-488.2M, version 1.1 and higher for Windows 95

The `GpibNotify` ActiveX control has the same support as the `ibnotify`. Here is the list of the software versions that support `GpibNotify`:

- NI-488.2M, version 1.21 and higher for Windows NT
- NI-488.2M for GPIB-ENET, version 1.0 and higher for Windows NT
- NI-488.2M, version 1.2 and higher for Windows 95

Description of the `ibnotify` Function

The `ibnotify` function notifies the user of one or more GPIB events by invoking the user callback. The following section describes the parameters for both the `ibnotify` function and the callback routine.

C Syntax

```
ibnotify (
    int ud,                // unit descriptor
    int mask,              // bit mask of GPIB events
    GpibNotifyCallback_t Callback, // callback function
    void * RefData        // user-defined reference data
)
```

Visual Basic Syntax

Visual Basic is supported using the `GpibNotify` ActiveX control described later.

Parameter Descriptions

Board/Device Descriptor – `ud`

This parameter specifies a board or device descriptor that was returned by `ibfind` or `ibdev` (or in the case of LabWindows/CVI, `OpenDev`). This descriptor identifies which board or device to use for this operation. You can also pass a board index into the function. For example, use 0 for GPIB0 or 1 for GPIB1.


Event Mask – `mask`


This parameter specifies the GPIB events upon which the callback function is invoked. Table 1 defines the valid event masks for board-level and device-level `ibnotify` calls. Note that the `ibnotify` event mask bits are identical to the `ibwait` mask bits.

Table 1. Valid Events for Asynchronous Callback Functions

Mnemonic	Hex Value	Board/Device	Description
TIMO	4000	Board/Device	Time limit exceeded
END	2000	Board/Device	GPIB board detected END or EOS
SRQI	1000	Board	SRQ asserted
RQS	800	Device	Device requesting service
CMPL	100	Board/Device	I/O completed
LOK	80	Board	GPIB board is in Lockout State
REM	40	Board	GPIB board is in Remote State
CIC	20	Board	GPIB board is CIC
ATN	10	Board	Attention is asserted
TACS	8	Board	GPIB board is Talker
LACS	4	Board	GPIB board is Listener
DTAS	2	Board	GPIB board is in Device Trigger State
DCAS	1	Board	GPIB board is in Device Clear State

The events specified with a bit mask correspond to the bits of the status word `ibsta`. This value reflects a sum of one or more events. If any of the events occur, the callback is invoked. If you want to disable callbacks, use an event mask of 0.

 **Note:** *If one or more of the bits in the event mask is already **TRUE**, the callback is invoked immediately. For example, if you pass **CMPL** as the event mask, and the **CMPL** bit is currently **TRUE** in `ibsta`, the callback is invoked immediately.*

 **Note:** *For **SRQI** event mask, if you want to install a callback for the **SRQI** event, Automatic Serial Polling must be disabled. You can disable Automatic Serial Polling with the following function call:*

```
ibconfig (boardIndex, IbcAUTOPOLL, 0);
```

*For **RQS** event mask, if you want to install a callback for the **RQS** event, Automatic Serial Polling must be enabled for the board. By default, Automatic Serial Polling is enabled. You can enable Automatic Serial Polling with the following function call:*

```
ibconfig (boardIndex, IbcAUTOPOLL, 1);
```

Callback Function – Callback

The callback function, which you register with the `ibnotify` call, is invoked when one or more of the mask bits passed to `ibnotify` is **TRUE**. This function (type `GpibNotifyCallback_t`) takes the form:

```
int __stdcall CallbackFunction (int boardOrDevice,  
    int LocalIbsta, int LocalIberr,  
    long LocalIbent1, void *callbackData);
```

More will be explained on this subject in the detailed description of the `ibnotify` Callback function section of this application note.

Callback Data – RefData

User-defined reference data for the callback. It is a four-byte value that will be passed to the callback function. It may be either a pointer to data or just an integer value.

Return Value of `ibnotify`

The return value of the `ibnotify` function is the GPIB status, `ibsta`, after the call. The GPIB status describes the state of the GPIB and the result of the function call. Any value with the **ERR** bit set indicates an error.

Function Description

If the `mask` passed to the `ibnotify` function is nonzero, `ibnotify` monitors the events specified by the `mask`. When one or more of the events is true, the Callback is invoked.

A given unit descriptor `ud` can have only one outstanding `ibnotify` call at any one time. If a current `ibnotify` is in effect for a `ud`, it is replaced by a subsequent `ibnotify` call. To cancel an outstanding `ibnotify` call for `ud`, issue `ibnotify` for the `ud` in the main thread with a `mask` of 0 and ensure that your Callback function stops rearming `ibnotify` by returning 0.

If an `ibnotify` call is outstanding and one or more of the GPIB events it is waiting on becomes true, the Callback is invoked. Each callback function is set up to become active when any of the GPIB Events included in the callback functions mask occurs. Therefore, although only a single callback may be used for each board or device that callback can become active on one of several events and carry out different actions for each.

Like `ibwait`, `ibstop`, and `ibonl`, the invocation of the `ibnotify` Callback can cause resynchronization of the driver after an asynchronous I/O operation has been completed. In this case, the global variables passed into the Callback after I/O has been completed contain the status of the I/O operation.

The `ibnotify` Callback is invoked in a separate thread from the rest of your application. If your application is performing other GPIB operations while it is using `ibnotify`, you should use the per-thread GPIB globals, that are provided by the Thread functions, `ThreadIbsta()`, `ThreadIberr()`, `ThreadIbcnt()`, and `ThreadIbcntl()`, in these operations. In addition, if your application needs to share global variables with the Callback, you should use a synchronization primitive (for example, semaphore) to protect access to any globals. Alternatively, the issue of data protection can be avoided entirely if the Callback simply posts a message to your application using the Windows `PostMessage()` function. For more information on the use of synchronization primitives, refer to the documentation on using Win32 synchronization objects that came with your development tools.

Possible Errors for `ibnotify`

When you call `ibnotify`, you should always check `ibsta` to see if the error bit, `ERR`, has been set. If so, then check `iberr` for the error code. The following errors that can be generated by a call to `ibnotify` are listed below along with the possible reasons for the error.

EARG A bit set in mask is invalid.

ECAP `ibnotify` has been invoked from within an `ibnotify` Callback function, or the handler cannot perform notification on one or more of the specified mask bits.

EDVR Either `ud` is invalid or the NI-488.2M driver is not installed. `ibcntl` contains a system-dependent error code.

ENEB The interface board is not installed or is not properly configured.

Detailed Description of the `ibnotify` Callback Function

The Callback Function

As explained earlier, the Callback function has the following prototype.

```
int __stdcall CallbackFunction (int boardOrDevice,  
                               int LocalIbsta, int LocalIberr,  
                               long LocalIbcntl, void *callbackData);
```

The parameters are described as follows:

Board/Device – `boardOrDevice`

The Callback function is passed the unit descriptor for the device or board causing the callback function to be invoked.

Status Variables – `LocalIbsta`, `LocalIberr`, `LocalIbcntl`

The current values of the GPIB global variables that are passed into the Callback routine. The normal GPIB global variables, `ibsta`, `iberr`, `ibcnt`, and `ibcntl`, are undefined from within the callback routine. The local status variables, `LocalIbsta`, `LocalIberr`, and `LocalIbcntl`, should be examined, instead of the GPIB global variables, to determine why the callback was invoked. Notice that it is possible that the Callback was invoked because of an error condition rather than because of the setting of one or more of the requested mask bits. Within the Callback, you could continue to use `LocalIbsta` when checking the status condition as shown in the example code in Appendix A. Or for checking any or all of the status variables, you should use the Thread status functions, `ThreadIbsta()`, `ThreadIberr()`, `ThreadIbcnt()`, and `ThreadIbcntl()`. For more information about the Thread functions, please refer to the on-line help or the Writing Multithreaded Win32 GPIB Applications section of the GPIB Programming Techniques chapter in *NI-488.2M User Manual for Windows 95/Windows NT*.

Callback Data – `callbackData`

The user-defined reference data that was passed to the original `ibnotify` call.

Return Value

The value that you return from the callback function is very important. It is the event mask that is used either to rearm the callback or not. If you return 0, the callback is disarmed. That is, it will not be called again until you make another call to `ibnotify`. If you return an event mask different than the one you originally passed to `ibnotify`, the new event mask is used.

If you return an invalid event mask or if there is an operating system error in rearming the callback, the callback is called with the `LocalIbsta` set to `ERR`, `LocalIberr` set to `EDVR`, and `LocalIbcntl` set to `IBNOTIFY_REARM_FAILED`. The callback is called again immediately with these values set. Therefore you should always check the value of the `LocalIbsta` parameter to make sure that one of the requested events has actually occurred.

Callback Details for SRQI

If the callback function is called because of an SRQ (board-level only), the callback function should call the `ibrsp` function. The `ibrsp` function will obtain the status byte, which turns off the SRQ and will prevent the callback from being repeatedly called.

Possible Errors for the Callback

`EDVR` The Callback return failed to rearm the Callback.

C Example Using `ibnotify`

Appendix A contains an example of how you might use `ibnotify` in your application. The example shows how a GPIB application could acquire measurements from a digital multimeter.

LabWindows/CVI and Asynchronous Callbacks

In LabWindows/CVI, you can install asynchronous callback functions for a particular board or device using `ibnotify`. You can also install synchronous callbacks using `ibInstallCallback`. Synchronous callbacks are invoked only when LabWindows/CVI is processing events. LabWindows/CVI processes events when you call `ProcessSystemEvents` or `GetUserEvent`, or when `RunUserInterface` is active and you are not in a callback function. Consequently, the latency between the occurrence of the GPIB event and the invocation of the callback can be large. However, there are no restrictions on what you can do within the callback function.

Using `ibnotify`, you can install asynchronous callbacks that can be called at any time with respect to the rest of your program. This means that LabWindows/CVI does not have to be processing events. Consequently, the latency between the occurrence of the GPIB event and the invocation of the callback is smaller than with synchronous callbacks. However, there are some restrictions on operations when using asynchronous callbacks. You can do the following:

- Call the User Interface Library function `PostDeferredCall`, which schedules a different callback function to be called synchronously.
- Call any GPIB function, except for `ibnotify` or `ibInstallCallback`.
- Call ANSI C functions such as `strcpy` and `sprintf`, which affect only the arguments passed in (that is, have no side effects). You cannot call `printf` or file I/O functions.
- Call `malloc`, `calloc`, `realloc`, or `free`.
- Manipulate global variables, but only if you know that the callback has not been called at a point when the main part of your program is modifying or interrogating the same global variables.

If you need to perform operations that fall outside these restrictions, you can do the following:

1. In your asynchronous callback,
 - Perform the time-critical operations
 - Call `PostDeferredCall` to schedule a synchronous callback.
2. In the synchronous callback, perform the rest of the operations.

Appendix A contains an example written in C of how you might use `ibnotify` in your application, Appendix B contains an example of how you might use `ibInstallCallback` in your application, and Appendix C contains an example of how you might use `PostDeferredCall` and `ibnotify` together in your application. The examples show how a GPIB application could acquire measurements from a digital multimeter. For more extensive examples of using `PostDeferredCall` and `ibnotify` from LabWindows/CVI, please refer to the GPIB Web site in the **Download** section in the following location:

<http://www.natinst.com/gpib/>

Description of the GpibNotify ActiveX Control for Visual Basic

In Visual Basic, asynchronous callback functions cannot be used. However, it is still possible to set up the equivalent of an asynchronous callback function for GPIB events by using an ActiveX control. An ActiveX control that encompasses the same functionality as the C `ibnotify` function has been designed for use with Visual Basic. This ActiveX control can also be used with Visual C versions 4.0 and later. The control is distributed with NI-488.2M, version 1.2 and higher for Windows 95. The `GpibNotify` ActiveX control is available on our GPIB Web site. Please refer to the **Download** section in the following location:

<http://www.natinst.com/gpib/>

Installing the GpibNotify ActiveX Control

If you are using version 1.2 or later of the Windows 95 version of NI-488.2M software, the `GpibNotify` ActiveX control is automatically installed and registered with the operating system when you install the GPIB software. Otherwise, if you obtained the `GpibNotify` ActiveX control from the Web site, you will need to register the `GpibNotify` ActiveX control yourself. The utility used to register the control is called `regsvr32.exe`, which comes with the Visual Basic 4.0 CD-ROM.

Follow the steps below in order to register the `GpibNotify` ActiveX control.

1. Copy the files, `gpibnotify.ocx` and `gpibnotify.tlb`, to the appropriate windows system directory (for Windows 95, usually `c:\windows\system` or for Windows NT, usually `c:\windows\system32`).
2. Open up a DOS shell.
3. Insert the Visual Basic CD-ROM into the CD-ROM drive.
4. Go to the directory called `\tools\pss` on the Visual Basic CD-ROM.
5. On the command line, type in the following:

```
regsvr32 <path>\gpibnotify.ocx
```

where `<path>` is the appropriate windows system directory where the `gpibnotify.ocx` file resides.

Adding the GpibNotify ActiveX Control to the Visual Basic Toolbox

To add the GpibNotify ActiveX control to the toolbox, perform the following steps.

1. Select **Tools** from the menu bar.
2. Select **Custom Controls...** from the drop-down list.
3. In the **Custom Controls** dialog box that appears, put an **X** in the box to the left of the **gpibNotify ActiveX Control module** by clicking on the box itself as shown in Figure 1. Click **OK**.
4. An icon is added to your **Toolbox**. The new icon contains a picture of a bell with the label **NOTIFY**. An example of the updated **Toolbox** is shown in Figure 2. The new **NOTIFY** item is located in the lower right hand corner.

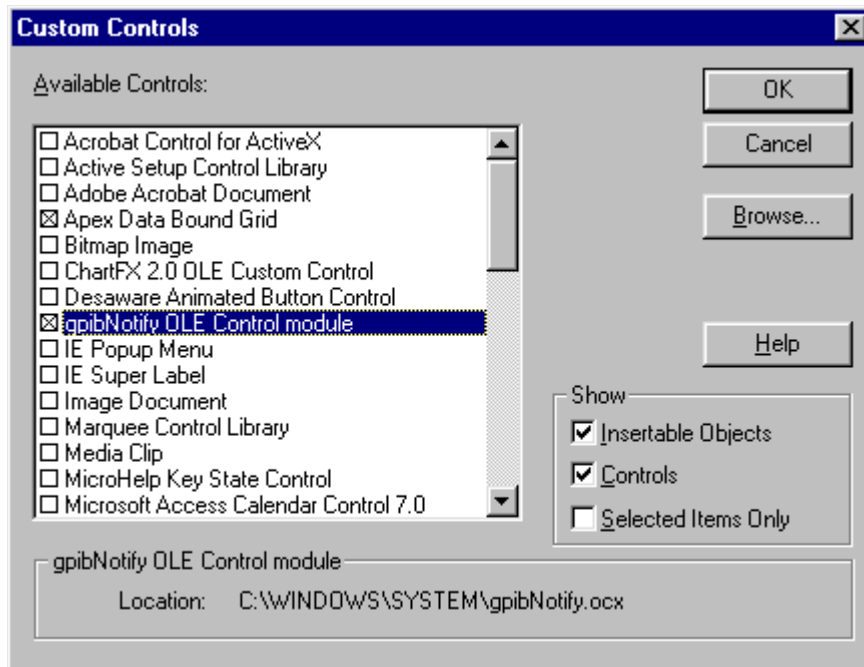


Figure 1.



Figure 2.

Adding a GpibNotify Control to Your Visual Basic Program

Once the `GpibNotify` ActiveX control has been added to the toolbox it can be placed on a Visual Basic form in the same manner as any other control. For each `GpibNotify` ActiveX control that you want to add, follow the instructions below:

1. Double-click on the `NOTIFY` icon in the **ToolBox**. This places the `GPIB notify` icon onto your form as shown in Figure 3.

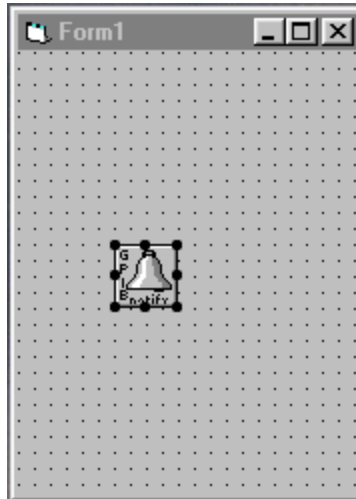


Figure 3.

2. To add code to the callback routine, double-click on the `GPIB notify` icon. The callback routine is named `GpibNotifyX_Notify`, where `X` is the particular numbered `GpibNotify` control. Each `GpibNotify` ActiveX control is numbered. The first control is numbered 1, the next control is numbered 2, and so on.

Setting Up the Event Mask and Installing the Callback Routine

For each `GpibNotify` control, a mask must be set up to determine which events will trigger the callback. There are three different ways that the mask can be set in Visual Basic.

Method 1

The easiest way to set the event mask is to modify the `GpibNotify` Control Properties page. To access that page, follow the steps below:

1. Right-click on the `GPIB notify` icon as it appears on your form.
2. Select **Properties...** from the drop-down list.
3. In the **GpibNotify Control Properties** dialog box, click on the box to the left of the `GPIB` event for each and every `GPIB` event that you wish to monitor.
4. When you have finished selecting all the desired `GPIB` events, click **OK**.

To use the call, enter the following line of code in your application:

```
GpibNotifyX.SetupNotify ud
```

where `X` is the numbered control that you want to invoke and `ud` is the unit descriptor that was obtained by using either `ibdev` or `ibfind`.

Method 2

Another method is to assign a `SetupMask` value to the `SetupNotify` function by declaring an integer value and setting it to the desired GPIB event(s) as shown below:

```
Dim mask As Integer
mask = XXX
```

where `XXX` is the desired GPIB event(s) (for example, `CMPL`, `RQS`, `TIMO`) that you wish to invoke the callback with. The call to `SetupNotify` would look like this:

```
GpibNotifyX.SetupNotify ud, mask
```

where `X` is for the particular numbered control that you wish to invoke and where `ud` is the unit descriptor that was obtained using either `ibdev` or `ibfind`. If you have set up the `GpibNotify` control properties page as in method 1, calling `GpibNotifyX.SetupNotify` with a `mask` parameter will override the choices made on the properties page.

Method 3

The third method is to initialize the `SetupMask` for the desired GPIB events by directly modifying the `SetupMask` property as shown below. This method will also override the control properties page.

```
GpibNotifyX.SetupMask = RQS
```

Then you can call the `SetupNotify` function with only a unit descriptor:

```
GpibNotifyX.SetupNotify ud
```

GpibNotify ActiveX Control Example

The example in Appendix D is intended only to show the basic program flow for using the `GpibNotify` ActiveX control. The example shows how a GPIB application could acquire data.

Appendix A

ibnotify Example in C

The following code is an example of how you might use `ibnotify` in your application. Assume that your GPIB device is a multimeter, which you program to acquire a reading by sending it "SEND DATA." The multimeter requests service when it has a reading ready. Each reading is a floating-point value.

In this example, globals are shared by the Callback thread and the main thread, and the access of the globals is not protected by synchronization. In this case, synchronization of access to these globals is not necessary because of the way they are used in the application, only a single thread is writing the global values and that thread always just adds information (increases the count or adds another reading to the array of floats).

C Source Code Example

```
#include <windows.h>
#include <stdio.h>
#include "decl-32.h"

int __stdcall MyCallback (int ud, int LocalIbsta, int LocalIberr, long LocalIbcntl,
    void *RefData);
int ReadingsTaken = 0;
float Readings[1000];
BOOL DeviceError = FALSE;
BOOL DisableCallback = FALSE;

int main()
{
    int ud;

    // Assign a unique identifier to the device and store it in the
    // variable ud. ibdev opens an available device and assigns it to
    // access GPIB0 with a primary address of 1, a secondary address of 0,
    // a timeout of 10 seconds, the END message enabled, and the EOS mode
    // disabled. If ud is less than zero, then print an error message
    // that the call failed and exit the program.
    ud = ibdev (0, // connect board
        1, // primary address of GPIB device
        0, // secondary address of GPIB device
        T10s, // 10 second I/O timeout
        1, // EOT mode turned on
        0); // EOS mode disabled

    if (ud < 0) {
        printf ("ibdev failed.\n");
        return 0;
    }

    // Issue a request to the device to send the data. If the ERR bit
    // is set in ibsta, then print an error message that the call failed
    // and exit the program.
    ibwrt (ud, "SEND DATA", 9L);
    if (ibsta & ERR) {
        printf ("unable to write to device.\n");
        // Call the ibonl function to disable the hardware and software.
        ibonl(ud, 0);
        return 0;
    }
}
```

```

// set up the asynchronous event notification on RQS
ibnotify (ud, RQS, MyCallback, NULL);
if (ibsta & ERR) {
    printf ("ibnotify call failed.\n");
    // Call the ibonl function to disable the hardware and software.
    ibonl(ud, 0);
    return 0;
}
while ((ReadingsTaken < 1000) && !(DeviceError)) {
    // Your application does useful work here. For example, it
    // might process the device readings or do any other useful work.
}
// Disable notification
DisableCallback = TRUE;
ibnotify (ud, 0, NULL, NULL);
// Call the ibonl function to disable the hardware and software.
ibonl (ud, 0);
return 1;
}
int __stdcall MyCallback (int LocalUd, int LocalIbsta, int LocalIberr, long LocalIbcntl,
void *RefData)
{
    char SpollByte;
    int expectedResponse = 0x40;
    char ReadBuffer[40];
    // If the ERR bit is set in LocalIbsta, then print an error message
    // and return.
    if (LocalIbsta & ERR) {
        printf ("GPIB error %d has occurred. No more callbacks.\n", LocalIberr);
        DeviceError = TRUE;
        return 0;
    }
    // Read the serial poll byte from the device. If the ERR bit is set
    // in ibsta, then print an error message and return.
    LocalIbsta = ibrsp (LocalUd, &SpollByte);
    if (LocalIbsta & ERR) {
        printf ("ibrsp failed. No more callbacks.\n");
        DeviceError = TRUE;
        return 0;
    }
    // If the returned status byte equals the expected response, then
    // the device has valid data to send; otherwise it has a fault
    // condition to report.
    if (SpollByte != expectedResponse) {
        printf("Device returned invalid response. Status byte = 0x%x\n",
            SpollByte);
        DeviceError = TRUE;
        return 0;
    }
}

```

```

// Read the data from the device. If the ERR bit is set in ibsta,
// then print an error message and return.
LocalIbsta = ibrd (LocalUd, ReadBuffer, 40L);
ReadBuffer[ThreadIbcntl()] = '\0';
if (LocalIbsta & ERR) {
    printf ("ibrd failed. No more callbacks.\n");
    DeviceError = TRUE;
    return 0;
}

// Convert the data into a numeric value.
sscanf (ReadBuffer, "%f", &Readings[ReadingsTaken]);
ReadingsTaken += 1;
if ((ReadingsTaken >= 1000) || DisableCallback) {
    return 0;
}
else {
// Issue a request to the device to send the data and rearm
// callback on RQS.
LocalIbsta = ibwrt (LocalUd, "SEND DATA", 9L);
if (LocalIbsta & ERR) {
    printf ("ibwrt failed. No more callbacks.\n");
    DeviceError = TRUE;
    return 0;
}
else {
    return RQS;
}
}
}

```

Appendix B

ibInstall Callback Example in LabWindows/CVI

The following code is an example of how you might use `ibInstallCallback` in your LabWindows/CVI application. Assume that your GPIB device is a multimeter, which you program to acquire a reading by sending it "SEND DATA." The multimeter requests service when it has a reading ready. Assume that the User Interface consists of one panel. On the panel there are two buttons, one labeled **Run**, and the other labeled **Quit**. The panel also has a TextBox where the multimeter measurements are displayed. The GPIB global status variable, `ibsta`, is displayed in its own window. The GPIB global error variable, `iberr`, is displayed in its own window if there is an error.

Assume that clicking the **Run** button causes the program to read five measurements from the multimeter.

To download the complete project, please refer to the GPIB Web site in the Download section in the following location:

<http://www.natinst.com/gpib/>

```
#include <formatio.h>
#include <utility.h>
#include <gpib.h>
#include <ansi_c.h>
// Need the cvirte.h file if linking in external compiler; harmless otherwise
#include <cvirte.h>
#include <userint.h>
#include "IbInstallCallback.h"

static int panelHandle;
static int ReadingsTaken;
static int LineIndex;

void CVICALLBACK Device_Callback (int, int, void *);

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

    // Needed if linking in external compiler; harmless otherwise.
    if (InitCVIRTE (0, argv, 0) == 0) {
        return -1; // out of memory.
    }

    // If debugging is enabled, this function directs LabWindows/CVI not
    // to display a run-time error dialog box when a National Instruments
    // library function reports an error.
    DisableBreakOnLibraryErrors();

    // Assign a unique identifier to the device and store it in the
    // variable device. ibdev opens an available device and assigns it to
    // access GPIB0 with a primary address of 1, a secondary address of 0,
    // a timeout of 10 seconds, the END message enabled, and the EOS mode
    // disabled. If device is less than zero, then print an error message
    // that the call failed and exit the program.
    device = ibdev (0, 1, NO_SAD, T10s, 1, 0);
    if (device < 0) {
        MessagePopup ("ibdev Error", "ibdev failed! Exiting the program.");
        return 0;
    }

    // Turn on AutoPolling for access GPIB0 board.
    ibconfig (0, IbcAUTOPOLL, 1);
```

```

// Install a synchronous callback function for specified device for
// the RQS event.
ibInstallCallback (device, RQS, Device_Callback, NULL);
if (ibsta & ERR) {
    SetCtrlVal (panel, DEVICE_STATUS_LED, 1);
    SetCtrlVal (panel, DEVICE_ERR, iberr);
    ProcessDrawEvents();
}

// Loads a panel into memory from a User Interface Resource (.uir) file.
if ((panelHandle = LoadPanel (0, "IbInstallCallback.uir", DEVICE)) < 0) {
    return -1;
}

// Displays a panel on the screen.
DisplayPanel (panelHandle);

// Setup the Callback Data so that the device is passed into the local function
SetCtrlAttribute (panelHandle, DEVICE_RUN_BUTTON, ATTR_CALLBACK_DATA, &device);
SetCtrlAttribute (panelHandle, DEVICE_QUIT_BUTTON, ATTR_CALLBACK_DATA, &device);
RunUserInterface ();

return 0;
}

// This function is called when the user clicks on the QUIT button.
int CVICALLBACK Quit (int panel, int control, int event, void *callbackData,
    int eventData1, int eventData2)
{
    switch (event)
    {
        case EVENT_COMMIT:
        {
            int device;

            memcpy(&device, callbackData, sizeof(int));

            // Disable the callback.
            ibInstallCallback (device, 0, NULL, NULL);

            // Call the ibonl function to disable the hardware
            // and software.
            ibonl(device, 0);

            QuitUserInterface (0);
            break;
        }
    }
    return 0;
}

// This function is called when user clicks on the RUN button.
int CVICALLBACK Run (int panel, int control, int event, void *callbackData,
    int eventData1, int eventData2)
{
    int device;
    ReadingsTaken = 0;
    LineIndex = 1;

    switch (event)

```

```

    {
        case EVENT_COMMIT:
        {
            int device;
            char string[200];

            memcpy(&device, callbackData, sizeof(int));
            memcpy(string, "SEND DATA", 9);
            string[9] = '\0'; // Terminate the string.

            // Issue a request to the device to send the data. If the
            // ERR bit is set in ibsta, then display the error.
            ibwrt(device, string, strlen(string));
            if (ibsta & ERR)
            {
                SetCtrlVal (panel, DEVICE_STATUS_LED, 1);
                SetCtrlVal (panel, DEVICE_ERR, iberr);
                ProcessDrawEvents();
            }

            // Display the status variable.
            SetCtrlVal (panel, DEVICE_IBSTA, ibsta);
        }
    }
    return 0;
}

void CVICALLBACK Device_Callback (int LocalUd, int mask, void *callbackData)
{
    char SpollByte;
    int LocalIbsta;
    int bytes;
    char ReadBuffer[100];
    char string[100];

    // Read the serial poll byte from the device. If the ERR bit is set
    // in LocalIbsta, then display the error.
    LocalIbsta = ibrsp (LocalUd, &SpollByte);
    if (LocalIbsta & ERR) {
        SetCtrlVal (panelHandle, DEVICE_STATUS_LED, 1);
        SetCtrlVal (panelHandle, DEVICE_ERR, iberr);
        ProcessDrawEvents();
    }

    // If the returned status byte equals the expected response, then the
    // device has valid data to send; otherwise it has a fault condition to
    // report.
    if (SpollByte != 0x50)
    {
        MessagePopup ("ibrsp Error", "Incorrect response from the multimeter!");
        return;
    }

    // Display the status variable.
    SetCtrlVal (panelHandle, DEVICE_IBSTA, ibsta);

    // Read the data from the device. If the ERR bit is set in ibsta, display
    // the error and return.

```



```

ibrd(LocalUd, ReadBuffer, 10);
if (ibsta & ERR) {
    SetCtrlVal (panelHandle, DEVICE_STATUS_LED, 1);
    SetCtrlVal (panelHandle, DEVICE_ERR, iberr);
    ProcessDrawEvents();
    return;
}
ReadBuffer[ibcntl] = '\0';
ReadingsTaken++;
// Display the measurement in the Text Box.
Fmt(string, "Multimeter Measurement #i[w2] = %s", ReadingsTaken, ReadBuffer);
if (ReadingsTaken == 1) {
    ResetTextBox (panelHandle, DEVICE_READ_BOX, string);
}
else {
    InsertTextBoxLine (panelHandle, DEVICE_READ_BOX, LineIndex, string);
    LineIndex++;
}
// Display the status variable.
SetCtrlVal (panelHandle, DEVICE_IBSTA, ibsta);
ProcessDrawEvents();
if (ReadingsTaken < 5) {
    memcpy(string, "SEND DATA", 9);
    string[9] = '\0';

    // Issue a request to the device to send the data.
    ibwrt(LocalUd, string, strlen(string));
    if (ibsta & ERR) {
        SetCtrlVal (panelHandle, DEVICE_STATUS_LED, 1);
        SetCtrlVal (panelHandle, DEVICE_ERR, iberr);
        ProcessDrawEvents();
    }

    // Display the status variable.
    SetCtrlVal (panelHandle, DEVICE_IBSTA, ibsta);
}
}

```

Appendix C

PostDeferredCall Example in LabWindows/CVI

The following code is an example of how you might use `PostDeferredCall` and `ibnotify` in your LabWindows/CVI application. Assume that your GPIB device is a multimeter, which you program to acquire a reading by sending it "SEND DATA." The multimeter requests service when it has a reading ready. Assume that the User Interface consists of one panel. On the panel there are two buttons, one labeled **Run**, and the other labeled **Quit**. The panel also has a `TextBox` where the multimeter measurements are displayed. The GPIB global error variable, `iberr`, is displayed in its own window if there is an error.

Assume that clicking the **Run** button causes the program to read five measurements from the multimeter.

To download the complete project, please refer to the GPIB Web site in the `Download` section in the following location:

<http://www.natinst.com/gpib/>

```
#include <formatio.h>
#include <utility.h>
#include <gpib.h>
#include <ansi_c.h>
// Need the cvirte.h file if linking in external compiler; harmless otherwise
#include <cvirte.h>
#include <userint.h>
#include "Ibnotify.h"

// Bits in the Status Byte
#define RQS_BIT 0x50

static int panelHandle;
static int ReadingsTaken;
static int LineIndex;

int __stdcall My_Callback (int, int, int, long, void *);
void CVICALLBACK Display_ERR_Message (void *);
void CVICALLBACK Display_RD_Message (void *);

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

    // Needed if linking in external compiler; harmless otherwise.
    if (InitCVIRTE (0, argv, 0) == 0) {
        return -1; // out of memory
    }

    // If debugging is enabled, this function directs LabWindows/CVI not
    // to display a run-time error dialog box when a National Instruments
    // library function reports an error.
    DisableBreakOnLibraryErrors();

    // Assign a unique identifier to the device and store it in the
    // variable device. ibdev opens an available device and assigns it to
    // access GPIB0 with a primary address of 1, a secondary address of 0,
    // a timeout of 10 seconds, the END message enabled, and the EOS mode
    // disabled. If device is less than zero, then print an error message
    // that the call failed and exit the program.
    device = ibdev (0, 1, 0, T10s, 1, 0);
    if (device < 0) {
        MessagePopup ("ibdev Error", "ibdev failed! Exiting the program.");
    }
}
```

```

        return 0;
    }

    // Turn on AutoPolling for access GPIB0 board.
    ibconfig (0, IbcAUTOPOLL, 1);
    // Set up the asynchronous event notification on RQS. Notice that the device
    // parameter is passed to the function twice. This is necessary because the
    // Callback Function calls PostDefferedCall which needs to have the device passed
    // in as a parameter that will not be destroyed as soon as the callback function
    // exits.
    ibnotify (device, RQS, My_Callback, &device);
    if (ibsta & ERR) {
        MessagePopup ("ibnotify Error", "ibnotify failed! Exiting the program.");
        // Call the ibonl function to disable the hardware and software.
        ibonl(device, 0);
        return 0;
    }

    // Loads a panel into memory from a User Interface Resource (.uir) file.
    if ((panelHandle = LoadPanel (0, "Ibnotify.uir", PANEL)) < 0) {
        return -1;
    }

    // Displays a panel on the screen.
    DisplayPanel (panelHandle);

    // Setup the Callback Data so that the device is passed into the local function.
    SetCtrlAttribute (panelHandle, PANEL_RUN_BUTTON, ATTR_CALLBACK_DATA, &device);
    SetCtrlAttribute (panelHandle, PANEL_QUIT_BUTTON, ATTR_CALLBACK_DATA, &device);
    RunUserInterface ();

    return 0;
}

// This function is called when the user clicks on the QUIT button.
int CVICALLBACK Quit (int panel, int control, int event,
    void *callbackData, int eventData1, int eventData2)
{
    switch (event) {
        case EVENT_COMMIT:
            {
                int device;

                memcpy(&device, callbackData, sizeof(int));

                // Disable the callback.
                ibnotify (device, 0, NULL, NULL);

                // Call the ibonl function to disable the hardware and
                // software.
                ibonl(device, 0);

                QuitUserInterface (0);
                break;
            }
    }
    return 0;
}

// This function is called when the user clicks on the RUN button.
int CVICALLBACK Run (int panel, int control, int event, void *callbackData,
    int eventData1, int eventData2)
{

```

```

int device;
ReadingsTaken = 0;
LineIndex = 1;
switch (event)
{
    case EVENT_COMMIT:
    {
        int device;
        char string[200];

        memcpy(&device, callbackData, sizeof(int));
        memcpy(string, "SEND DATA", 9);
        string[9] = '\0';

        // Issue a request to the device to send data. If the
        // ERR bit is set in ibsta, then display the error.
        ibwrt(device, string, strlen(string));
        if (ibsta & ERR) {
            SetCtrlVal (panel, PANEL_STATUS_LED, 1);
            SetCtrlVal (panel, PANEL_ERR, iberr);
            ProcessDrawEvents();
        }
        break;
    }
}
return 0;
}

int __stdcall My_Callback (int LocalUd, int LocalIbsta, int LocalIberr,
                          long LocalIbnt1, void *callbackData)
{
    static char SpollByte;
    char ReadBuffer[100];
    char string[100];

    // If the ERR bit is set in LocalIbsta, then print an error message
    // and return.
    if (LocalIbsta & ERR) {
        PostDeferredCall (Display_ERR_Message, NULL);
        // Disarm the callback.
        return 0;
    }

    // Read the serial poll byte from the device. If the ERR bit is set
    // in LocalIbsta, then print an error message and return.
    LocalIbsta = ibrsp (LocalUd, &SpollByte);
    if (LocalIbsta & ERR) {
        PostDeferredCall(Display_ERR_Message, NULL);
        // Disarm the callback.
        return 0;
    }

    // If the returned status byte equals the expected response, then the
    // device has valid data to send; otherwise it has a fault condition
    // to report.
    if (SpollByte & RQS_BIT) {
        // Pass the device LocalUd into the Display_RD_Message through

```

```

        // the use of the callback data
        PostDeferredCall(Display_RD_Message, callbackData);
    }

    // The return value is the mask that is used to rearm the Callback
    return RQS;
}

void CVICALLBACK Display_RD_Message (void *callbackdata)
{
    char ReadBuffer[200];
    int bytes, LocalUd, LocalIbsta;
    char string[100];

    memcpy(&LocalUd, callbackdata, sizeof(int));

    // Read the data from the device. If the ERR bit is set in LocalIbsta,
    // then display the error and return.
    LocalIbsta = ibrd (LocalUd, ReadBuffer, 10);
    ReadBuffer[ibcntl] = '\0';
    if (LocalIbsta & ERR) {
        SetCtrlVal (panelHandle, PANEL_STATUS_LED, 1);
        SetCtrlVal (panelHandle, PANEL_ERR, iberr);
        ProcessDrawEvents();
        return;
    }

    ReadingsTaken++;

    // Display the measurement in the Text Box.
    Fmt(string, "Multimeter Measurement #%i[w2] = %s", ReadingsTaken, ReadBuffer);
    if (ReadingsTaken == 1) {
        ResetTextBox (panelHandle, PANEL_READ_BOX, string);
    }
    else {
        InsertTextBoxLine(panelHandle, PANEL_READ_BOX, LineIndex, string);
        LineIndex++;
    }

    if (ReadingsTaken < 5) {
        memcpy(string, "*TRG; VAL1?", 11);
        string[11] = '\0';

        // Issue a request to the device to send the data.
        LocalIbsta = ibwrt(LocalUd, string, strlen(string));
        if (LocalIbsta & ERR) {
            SetCtrlVal (panelHandle, PANEL_STATUS_LED, 1);
            SetCtrlVal (panelHandle, PANEL_ERR, iberr);
            ProcessDrawEvents();
        }
        return;
    }
}

void CVICALLBACK Display_ERR_Message (void *callbackdata)
{
    SetCtrlVal (panelHandle, PANEL_STATUS_LED, 1);
    SetCtrlVal (panelHandle, PANEL_ERR, iberr);
    ProcessDrawEvents();
}

```

Appendix D

GpibNotify Example in Visual Basic

This example shows the basic program flow for using the GpibNotify ActiveX control. It is written to work with a fictitious device with GPIB address two. The device responds to the string "SEND DATA" by asserting SRQ when the data is available. The data is a floating-point value. The program uses a blank form containing only a GpibNotify ActiveX control. The GpibNotify procedure and the form load procedure are the only defined procedures for the example. They are shown below. This example is designed to be run only once. It could be easily modified to run several times by placing the form load procedure inside a command button click procedure and rearming the GpibNotify mask with RQS instead of zero. For a more complete example program that takes 10 readings from a Fluke 45 and uses a user interface to display the data, refer to the ocxsamp.mak project that comes with the GpibNotify ActiveX control. In addition to registering the ActiveX control a Visual Basic project using GPIB commands must include the niglobal.bas and vbib-32.bas modules. These modules are available with the GPIB Software.

Visual Basic Example

```
Visual Basic Private Sub Form_Load()  
    ' Assign a unique identifier to the device and store  
    ' it in the variable ud. ibdev opens an available  
    ' device and assigns it to access GPIB0 with a primary  
    ' address of 2, a secondary address of 0, a timeout of  
    ' 10 seconds, the END message enabled, and the EOS  
    ' mode disabled. If ud is less than zero, then print  
    ' an error message that the call failed and exit  
    ' the program.  
    Call ibdev(0, 2, 0, T10s, 1, 0, ud)  
    If (ud < 0) Then  
        MsgBox "ibdev error. I'm quitting!", 16  
    End  
End If  
  
    ' Install the GpibNotify callback mechanism for the OLE  
    ' Control by calling its SetupNotify Procedure with an  
    ' event mask for the RQS Event.  
    GpibNotify1.SetupMask = RQS  
    GpibNotify1.SetupNotify ud  
  
    ' Check to see if there was an error invoking the GpibNotify Callback mechanism.  
    If (ThreadIbsta() And EERR) Then  
        msg$ = "Error invoking Notification for RQS!  ibsta = &H" +  
            Hex$(ThreadIbsta())  
        MsgBox msg$, vbCritical, "GPIB Notification Error!"  
        msg$ = "iberr = " + Str$(ThreadIberr())  
        MsgBox msg$, vbCritical, "GPIB Notification Error!"  
        ' Call the ibonl function to disable the hardware and software.  
        Call ibonl(ud, 0)  
    End  
End If  
  
    If (ThreadIbsta() And &H4000) Then  
        MsgBox ("Notification for RQS timed out.")  
    End  
End If
```

```

' Write a command to the instrument to generate data and
' assert the SRQ line when the data is available.
wrtbuf$ = "SEND DATA"
Call ibwrt(ud, wrtbuf$)
If (ibsta And EERR) Then
    MsgBox ("ibwrt Error!")
    ' Call the ibonl function to disable the hardware and software.
    Call ibonl(ud, 0)
    End
End If

End Sub

Private Sub GpibNotify1_Notify(ByVal LocalUd As Long, ByVal LocalIbsta As Long,
ByVal LocalIberr As Long, ByVal LocalIbcntl As Long, RearmMask As Long)
' This is the user-defined callback routine that
' gets invoked when one or more GPIB events in the
' mask that is passed to the SetupNotify method occurs.
'
' For this sample program, a data point will be read and
' displayed from the device.

Dim SPollByte As Integer

' If ERR bit is set in LocalIbsta that is passed into
' this subroutine, then print an error message and
' exit the function.
If LocalIbsta And EERR Then
    MsgBox "Error with GPIB Notify #1- No more callbacks.", 16
    Exit Sub
End If

' NOTE: for the rest of this subroutine, the global
'       version of ibsta is used when checking to see
'       if the error bit is set or not.

' Read the serial poll status byte. If the
' error bit EERR is set in ibsta, display an error
' message and exit.
Call ibrsp(LocalUd, SPollByte)
If (ibsta And EERR) Then
    MsgBox ("ibrsp Error!")
    RearmMask = 0
    Exit Sub
End If

' If the returned status byte is &H50 then the instrument
' has valid data to send
If (SPollByte <> &H50) Then
    MsgBox("Serial poll byte is NOT &H50.")
    RearmMask = 0
    Exit Sub
End If

' Read the data. If the error bit EERR is
' set in ibsta, display an error message
' and exit.
rdbuf$ = Space$(20)
Call ibrd(LocalUd, rdbuf$)

```

```
If (ibsta And EERR) Then
    MsgBox ("ibrd Error!")
    RearmMask = 0
    Exit Sub
End If

' Display the Reading.
MsgBox (rdbuf$)

' Rearm Mask is set to zero to disable the
' Callback mechanism. This example is only
' designed to run once to show flow for using
' GpibNotify Control.
RearmMask = 0

End Sub
```



341255B-01

Dec98