# Developing Multithreaded GPIB Applications in Windows NT and Windows 95

**Clay Bean, Tony Iglesias, Jim Nagle, Srdan Zirojevic**

## Introduction

Multithreading is a method of programming in which the work of an application is divided into different tasks, or threads. These different threads are created by the main thread of an application and can execute independently of other threads in the system. When a thread completes its task, it exits. The main thread of an application is responsible for closing the application once all tasks have been completed. If your system has more than one processor, you can increase the performance of your application by simultaneously executing individual threads on separate processors. You can also tune single processor systems to yield greater performance by allocating appropriate processor time slices to each thread in your application. By configuring the amount of processor time allocated for each thread, you can improve the performance of your application because the processor can continue to execute one thread while waiting for external events before executing another thread – a common scenario in GPIB applications.

This application note introduces ways you can take advantage of the multithreading features of the NI-488.2 software for Windows NT and Windows 95. Topics include an overview of the multithreaded capability of the NI-488.2 API and examples of multithreaded applications for GPIB-based instrument control.

## Multithreading for Win32 GPIB Applications

NI-488.2M software for Windows NT and Windows 95 delivers complete functionality for multithreaded GPIB applications, including multiple threads in multiple GPIB applications as well as multiple threads in a single GPIB application. NI-488.2M software is designed to efficiently handle calls made from both single and multithreaded GPIB applications. In some multithreaded applications where GPIB calls can be completely independent of each other, the software executes these calls in parallel. Systems that include more than one GPIB interface, where calls are simultaneously made to separate GPIB interfaces, are an example of this situation. In multithreaded applications where calls are made to a single GPIB interface, the calls are executed serially in the order in which they are received. Typically, each GPIB application executes as a single process. Within these processes, the applications can spawn one or more threads for executing a variety of application-related tasks. In multithreaded GPIB applications, the GPIB global variables (`ibsta`, `iberr`, `ibcnt`, `ibcntl`), often used for monitoring GPIB system status, are allocated on a per process basis. Thus, multiple processes each have their own private copies of the GPIB global variables. Within a single process, these GPIB global variables are guaranteed to be valid only when a single thread is making GPIB calls. When GPIB calls are made from multiple threads in a single process, each thread is referencing and updating the same GPIB global variables, making the value of the GPIB global variables unreliable. Managing access to global variables in a multithreaded application is a standard issue that must be addressed. To write applications that spawn multiple threads within a single process, you can use the following calls that have been added to the NI-488.2 API for multithreaded Win32 GPIB applications:

- `ThreadIbsta()` returns the thread-specific `ibsta` value.
- `ThreadIberr()` returns the thread-specific `iberr` value.
- `ThreadIbcnt()` returns the thread-specific `ibcnt` value.
- `ThreadIbcntl()` returns the thread-specific `ibcntl` value.



  

NI-488.2M software for Windows NT and Windows 95 keeps separate copies of the GPIB global variables for each thread. The four calls listed above provide access to the thread-specific GPIB global variables. This ensures the integrity of the GPIB global variables when you have multiple threads within the same process making GPIB calls.

# Multithreaded Instrument Control Applications

Instrument control applications can take advantage of multithreading in a number of ways. Multithreading can help you isolate separate application tasks in order to manage system resources more efficiently. You can use multithreading to isolate time-critical tasks in separate threads so that you can monitor time-critical events, such as a Service Request (SRQ), while background processing acquired data available in memory. For example, an application might have three threads with different responsibilities – one that manages user input, a second that acquires data, and a third responsible for processing and displaying data. Processing and displaying data can be very time consuming and processor intensive. The amount of data and the frequency at which you acquire data can vary. Many times applications may also constantly check for SRQs from instruments. User interaction may be a sporadic event compared to the other ongoing activities in the application. Using multithreading, you can isolate these tasks and address the processing needs of each individual task.

For applications that use more than one GPIB interface, a single thread can be dedicated to manage each GPIB interface and the instrumentation each interface controls. For example, you can configure one interface to manage high-speed GPIB (HS488) instruments and another to manage standard GPIB instruments. An example illustrating how you can use separate threads for each GPIB interface can be found in Appendix A.

As mentioned before, you can use multithreading to perform time-critical or time-consuming operations in separate threads so that the time dependent tasks execute in a manner that enhances the performance of the overall application. Another common example is logging acquired and processed data to disk. Logging data to disk can be a time consuming task relative to the time it takes to acquire and/or process data. An application can use a separate thread and data buffering for logging data, while another thread manages acquisition and processing of data. You can also use multiple threads within an application to respond to emergency situations. A control thread can be tasked with monitoring the system for alarm conditions. Other threads can do the real work of your application, but if an alarm condition develops, the control thread takes responsibility for shutting down the equipment in a timely manner. This frees the other threads from the responsibility of continually checking for the alarm conditions.

# Application Example 1

## Problem Description

Suppose your GPIB system uses a single GPIB interface to communicate with several GPIB instruments. Each instrument is configured to use a different primary address. The GPIB controller acquires and displays a fixed number of readings from each instrument. There are two possible solutions – single-threaded and multithreaded – and the tradeoffs between them are included below. Example code for each approach is included in the appendixes.

## Single-Threaded Solution

In a single-threaded solution, your application might do the following:

1. While more readings to acquire
   a. Tell instrument 1 to send data
   b. Acquire data from instrument 1
   c. Tell instrument 2 to send data
   d. Acquire data from instrument 2
   e. Display acquired data

(See Appendix A for C source code)

## Multithreaded Solution

A multithreaded solution could be implemented much differently. A different thread can manage each GPIB instrument. The thread for each instrument might do the following:

1. While more readings to acquire
   a. Tell instrument to send data
   b. Acquire data from instrument
   c. Display acquired data

After creating a thread for each GPIB instrument, the main thread of the application only has to wait for each of the created threads to exit. The main thread could do the following:

1. Create a thread for each GPIB instrument
2. Wait for all threads to exit

(See Appendix B for C source code)

## Comparison of Solutions

There are strengths and weaknesses to both solutions offered above. The main strength of the multithreaded solution is that the GPIB instruments spend little time in an idle state. Instruments can be serviced immediately when data is available. In the single-threaded solution, each instrument has to wait idly while the application communicates with the other instruments and while the application displays all of the acquired data. The extent of this idle time depends on how long it takes to acquire data from the other instruments and how long it takes to process and display all of the data acquired. These idle times can significantly affect overall application performance.

The types of instruments you use can affect the performance benefits that multithreading has to offer. For example, instruments such as oscilloscopes and spectrum analyzers generally capture large volumes of data compared to a single reading from a digital multimeter (DMM). In addition, users generally perform customized post–processing and use graphs to display the processed waveform data. In situations such as these, multithreading can be used to enhance the performance of the overall application by acquiring measurements in parallel rather than in a serial fashion.

A weakness of any multithreaded solution is the added complexity that multithreading adds to the application. This added complexity is a consequence of the need for synchronization between the daughter threads and the main thread.

# Application Example 2

## Problem Description

Suppose your GPIB system uses a single GPIB instrument. The GPIB controller acquires, processes, and logs to a file a fixed number of readings from the GPIB instrument. There are two possible solutions – single-threaded and multithreaded – and the tradeoffs between them are included in the following discussion. Example code for each approach is included in the appendixes.

## Single-Threaded Solution

In a single-threaded solution, your application might do the following:

1. While more readings to acquire
   a. Tell instrument to send data
   b. Acquire data from instrument
   c. Format, process and log data

(See Appendix C for C source code)

## Multithreaded Solution

A multithreaded solution could be implemented differently. Each type function can be managed in a different thread. One thread can be responsible for communicating with the GPIB instrument and the other thread can be responsible for formatting, processing, and logging the data:

GPIB Thread:

1. While more readings to acquire
   a. Tell instrument to send data
   b. Acquire data from instrument

Log Thread:

1. Format, process, and log data as it becomes available

Main Thread:

1. Create the GPIB and log threads
2. Wait for all threads to exit

(See Appendix D for C source code)

## Comparison of Solutions

As in Example 1, there are strengths and weaknesses to both solutions. The main advantage of the multithreaded solution is that it decouples logging activity and GPIB I/O so that they can execute separately of each other. This approach maximizes the opportunity to have separate operations (i.e., GPIB data acquisition and data logging) occur in parallel. In the single-threaded case, the acquisition and logging of data are serial, potentially reducing the overall throughput of the system.

In this example, a single buffer is used to hold data, but it could be expanded to use multiple buffers or even a circular-buffering scheme. With multiple or circular buffers, you can develop continuous acquisition and logging applications that can quickly acquire, process, and log data on the fly.

A disadvantage of the multithreaded solution is the need for synchronization between separate threads. The data formatting and logging thread has to wait for the buffer to be filled by the GPIB data acquisition thread. Likewise, the GPIB data acquisition thread has to wait for the data formatting and logging thread to finish processing the data before it puts new data into the buffer. These threads must synchronize with each other in order to safely access the shared buffer. Synchronization primitives called “events” are used to accomplish this. Using events adds an additional layer of complexity to the development of the application.

# Conclusions

This application note introduces how multithreading can be incorporated into GPIB instrument control applications. We have discussed a number of situations in which you can use multithreading to divide multiple tasks of an overall application. While multithreading offers an alternative method for implementing your applications, multithreaded applications are more difficult to correctly design and debug than single-threaded applications. If you would like to take advantage of the multithreading capabilities of the NI-488.2M API, you should make a dedicated effort to learn the concepts of multithreading before you begin writing your applications. A possible resource to get you started is *Advanced Windows: The Developer’s Guide to the Win32 API for Windows NT 3.5 and Windows 95*, by Jeffrey Richter.

# Appendix A
# Example 1 – Single-Threaded Sample Code

```
// This sample is a Win32 console application that acquires and displays data
// from two separate GPIB devices that are both connected to a single GPIB
// interface. This sample assumes that the GPIB devices respond to a command of
// "MEAS:DC?" by sending back a measurement in an ASCII string that is less
// than 100 bytes long. The sample is written to acquire and display
// NUMBER_MEASUREMENTS measurements from each GPIB device.
//
// The single-threaded version of this sample serially communicates with each
// of the attached GPIB devices from a loop in main.
//
#include <windows.h>
#include "decl-32.h"
#include <stdio.h>

#define NUMBER_MEASUREMENTS    20

int main ()
{
    int ud1, ud2, counter;
    char measure1[ 102] ;
    char measure2[ 102] ;
    BOOL error = FALSE;

    // Open handle to device 1
    ud1 = ibdev (0,       // connect board is gpib0
                 3,       // device 1 primary address is 3
                 0,       // device 1 doesn't use secondary addressing
                 T10s,    // I/O timeout period - 10 seconds
                 1,       // use EOI to mark the end of I/O writes
                 0);      // don't use an EOS character at the end of reads
    if (ud1 == -1)  {
        printf ("ibdev call failed with ibsta = 0x%x, iberr = %d, ibcntl = %x.\n",
                    ibsta, iberr, ibcntl);
        printf ("exit application\n");
        return -1;
    }
    // Open handle to device 2
    ud2 = ibdev (0,       // connect board is gpib0
                 4,       // device 2 primary address is 4
                 0,       // device 2 doesn't use secondary addressing
                 T10s,    // I/O timeout period - 10 seconds
                 1,       // use EOI to mark the end of I/O writes
                 0);      // don't use an EOS character at the end of reads
    if (ud2 == -1)  {
        printf ("ibdev call failed with ibsta = 0x%x, iberr = %d, ibcntl = %x.\n",
                    ibsta, iberr, ibcntl);
        printf ("exit application\n");
        // Put any open handles offline before exiting
        ibonl (ud1, 0);
        return -1;
    }

    // While fewer than NUMBER_MEASUREMENTS have been acquired and displayed
    // continue getting more measurements.
    for (counter = 0; counter < NUMBER_MEASUREMENTS; counter++) {

        // Tell device 1 to send a measurement
        ibwrt (ud1, "MEAS:DC?", 8);
        if (ibsta & ERR)  {
            printf ("Write of command to device 1 failed!. Exit application.\n");
            error = TRUE;
            break;
        }
        ibrd (ud1, measure1, 100);
        if (ibsta & ERR)  {
```

```
            printf ("Read from device 1 failed!. Exit application.\n");
            error = TRUE;
            break;
        }
        // ibcnt contains the number of bytes in the measurement. Use it to
        // NULL terminate the measurement so it prints right.
        measure1[ibcnt] = 0;

        // Tell device 2 to send a measurement
        ibwrt (ud2, "MEAS:DC?", 8);
        if (ibsta & ERR)  {
            printf ("Write of command to device 2 failed!. Exit application.\n");
            error = TRUE;
            break;
        }
        ibrd (ud2, measure2, 100);
        if (ibsta & ERR)  {
            printf ("Read from device 2 failed!. Exit application.\n");
            error = TRUE;
            break;
        }
        // ibcnt contains the number of bytes in the measurement. Use it to
        // NULL terminate the measurement so it prints right.
        measure2[ibcnt] = 0;

        // Display the measurements
        printf ("Measurement:%s, %s\n", measure1, measure2);

    }

    // Put open handles offline before exiting
    ibonl (ud1, 0);
    ibonl (ud2, 0);
    if (error)  {
        return -1;
    }
    else  {
        return 0;
    }
}
```

# Appendix B
# Example 1 – Multithreaded Sample Code

```
// This sample is a Win32 console application that acquires and displays data
// from two separate GPIB devices located at primary GPIB addresses (pad) 3 and 4.
// Both devices are connected to a single GPIB interface.
// This sample assumes that the GPIB devices respond to a command of
// "MEAS:DC?" by sending back a measurement in an ASCII string that is less
// than 100 bytes long. The sample is written to acquire and display
// NUMBER_MEASUREMENTS measurements from each GPIB device.
//
// The multithreaded version of this sample creates two threads that
// independently communicate with each of the two GPIB devices. While the two
// threads are executing, the main thread just waits for both of them to exit.
//
#include <windows.h>
#include "decl-32.h"
#include <stdio.h>

#define NUMBER_MEASUREMENTS    20
DWORD DeviceThreadFunction(int Pad);

DWORD main (void)
{
    int DevicePads[2] = {3, 4};    // assume the GPIB devices are at pads 3 and 4
    HANDLE ThreadHandles[2];
    DWORD ThreadId;

    // Create the thread for the first device (pad 3)
    ThreadHandles[0] = CreateThread (
                            NULL,               // default security
                            0,                  // default stack size
                             (LPTHREAD_START_ROUTINE)DeviceThreadFunction,
                            DevicePads[0],      // single param is the index
                            0,                  // run thread immediately
                            &ThreadId);

    if (ThreadHandles[0] == NULL)  {
        printf ("Create thread for device 1 failed with error %x. Exiting application.\n",
                        GetLastError());
        return -1;
    }

    // Create the thread for the second device (pad 4)
    ThreadHandles[1] = CreateThread (
                            NULL,               // default security
                            0,                  // default stack size
                             (LPTHREAD_START_ROUTINE)DeviceThreadFunction,
                            DevicePads[1],      // single param is the index
                            0,                  // run thread immediately
                            &ThreadId);

    if (ThreadHandles[1] == NULL)  {
        printf ("Create thread for device 2 failed with error %x. Exiting application.\n",
                        GetLastError());
        return -1;
    }

    // Wait for both of the threads to exit
    WaitForMultipleObjects(2,               // number of handles to wait on
                            ThreadHandles,  // array of thread handles
                            TRUE,           // wait for ***all***
                            INFINITE        // wait forever
                           );

    // Check the exit code of both threads
    {
        DWORD ExitCode;
```

```
        GetExitCodeThread (ThreadHandles[0], &ExitCode);
        if (ExitCode == -1)  {
            printf ("Device 1 failed.\n");
        }
        GetExitCodeThread (ThreadHandles[1], &ExitCode);
        if (ExitCode == -1)  {
            printf ("Device 2 failed.\n");
        }
    }

    CloseHandle (ThreadHandles[0]);
    CloseHandle (ThreadHandles[1]);

    return 0;
}

DWORD DeviceThreadFunction(int Pad)
{
    int ud;               // handle of the device
    int counter;          // used to count the number of measurements taken
    char measure[102];    // string to read measurement into
    BOOL error = FALSE;   // boolean used to indicate an error has occurred

    // Open handle to the device
    ud = ibdev (0,       // connect board is gpib0
                Pad,     // device primary address is 'pad'
                0,       // device doesn't use secondary addressing
                T10s,    // I/O timeout period - 10 seconds
                1,       // use EOI to mark the end of I/O writes
                0);      // don't use an EOS character at the end of reads
    if (ud == -1)  {
        printf ("ibdev call failed with ibsta = 0x%x, iberr = %d, ibcntl = %x.\n",
                    ThreadIbsta(), ThreadIberr(), ThreadIbcntl());
        printf ("exit application\n");
        return -1;
    }

    // While fewer than NUMBER_MEASUREMENTS have been acquired and displayed
    // continue getting more measurements.
    for (counter = 0; counter < NUMBER_MEASUREMENTS; counter++) {

        // Tell device 1 to send a measurement
        ibwrt (ud, "MEAS:DC?", 8);
        if (ThreadIbsta() & ERR)  {
            printf ("Write of command to pad %x device failed!. Exit application.\n", pad);
            error = TRUE;
            break;
        }
        ibrd (ud, measure, 100);
        if (ThreadIbsta() & ERR)  {
            printf ("Read from pad %x device failed!. Exit application.\n", pad);
            error = TRUE;
            break;
        }
        // ibcnt contains the number of bytes in the measurement. Use it to
        // NULL terminate the measurement so it prints right.
        measure[ThreadIbcnt()] = 0;

        // Display the measurement
        printf ("Measurement from pad %x device: %s\n", pad, measure);
    }
    // put the handle offline before exiting
    ibonl (ud, 0);

    if (error)  {
        return -1;
    }
    else  {
        return 0;
    }
}
```

# Appendix C
# Example 2 – Single-Threaded Sample Code

```
// This sample is a Win32 console application that acquires data from a single
// GPIB instrument. The data is processed and then logged to a file. This sample
// assumes that the GPIB device responds to a command of
//        "SOUR:FUNC SIN; SENS:DATA?"
// by sending back a complex reading that is less than 2000 bytes long. The
// sample is written to acquire, format and log a total of NUMBER_MEASUREMENTS
// from the GPIB device.
//
// This is the single-threaded version of this sample.
//

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

#define NUMBER_MEASUREMENTS    20

int main ()
{
    int ud, counter;
    char Measurement[ 2002];
    BOOL error = FALSE;

    // Open handle to device 1
    ud = ibdev (0,       // connect board is gpib0
                3,       // device primary address is 3
                0,       // device doesn't use secondary addressing
                T10s,    // I/O timeout period - 10 seconds
                1,       // use EOI to mark the end of I/O writes
                0);      // don't use an EOS character at the end of reads

    if (ud == -1)  {
        printf ("ibdev call failed with ibsta = 0x%x, iberr = %d, ibcntl = %x.\n",
                    ibsta, iberr, ibcntl);
        printf ("exit application\n");
        return -1;
    }

    // While fewer than NUMBER_MEASUREMENTS have been acquired and displayed
    // continue getting more measurements.
    for (counter = 0; counter < NUMBER_MEASUREMENTS; counter++) {

        // Tell device to send a measurement
        ibwrt(ud, "SOUR:FUNC SIN; SENS:DATA?", 25);
        if (ibsta & ERR)  {
            printf ("Write of command to device failed!. Exit application.\n");
            error = TRUE;
            break;
        }
        ibrd (ud, Measurement, 2000);
        if (ibsta & ERR)  {
            printf ("Read from device failed!. Exit application.\n");
            error = TRUE;
            break;
        }
        // Display the measurement
        FormatAndLogMeasurement(Measurement);
    }
    // Put open handles offline before exiting
    ibonl (ud, 0);
    if (error)  {
        return -1;
    }
    else  {
        return 0;
    }
}
```

# Appendix D
# Example 2 – Multithreaded Sample Code

```
// This sample is a Win32 console application that acquires data from a single
// GPIB instrument. The data is processed and then logged to a file. This sample
// assumes that the GPIB device responds to a command of
//       "SOUR:FUNC SIN; SENS:DATA?"
// by sending back a complex reading that is less than 2000 bytes long. The
// sample is written to acquire, format and log a total of NUMBER_MEASUREMENTS
// from the GPIB device.
//
// The multi-threaded solution creates two worker threads for the application.
// One thread is responsible for acquiring data from the GPIB instrument and
// the other thread is responsible for formatting and logging data as it becomes
// available.
//

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

#define NUMBER_MEASUREMENTS    20
DWORD GpibThreadFunction(LPDWORD Param);
DWORD LogThreadFunction(LPDWORD Param);

// global strings for readings shared by the GpibThread and LogThread
char Measurement[2002];
BOOL MoreDataComing = TRUE;

// handles for the synchronization events shared by the two threads
HANDLE DataReady;
HANDLE BufferFree;

DWORD main (void)
{
    HANDLE GpibThread;
    HANDLE LogThread;
    DWORD ThreadId;

    // Create the two synchronization events (DataReady and BufferFree) used
    // by the GpibThread and LogThread. Note that the initial state of the
    // DataReady event is not signaled because data only becomes available
    // after the GpibThread has read it in. Also note that the initial state
    // of the BufferFree event is signaled because initially the buffer is empty.
    //
    DataReady = CreateEvent(NULL,  // default security attributes
                            FALSE, // automatically reset after wait
                            FALSE, // initial state is ***not*** signaled
                            NULL   // no name for the event
                           );
    BufferFree = CreateEvent(NULL,  // default security attributes
                             FALSE, // automatically reset after wait
                             TRUE,  // initial state IS signaled
                             NULL   // no name for the event
                            );

    // Create the thread for the GPIB instrument
    GpibThread  = CreateThread (
                         NULL,              // default security
                         0,                 // default stack size
                          (LPTHREAD_START_ROUTINE)GpibThreadFunction,
                         NULL,              // NULL parameter to the thread func
                         0,                 // run thread immediately
                         &ThreadId);

    if (GpibThread == NULL)  {
        printf ("Create GpibThread failed with error %x. Exiting application.\n",
                          GetLastError());
       return -1;
   }
```

```
    // Create the thread to format and log the data
    LogThread = CreateThread (
                        NULL,              // default security
                        0,                 // default stack size
                         (LPTHREAD_START_ROUTINE)LogThreadFunction,
                        NULL,              // NULL parameter to the thread func
                        0,                 // run thread immediately
                        &ThreadId);

    if (LogThread == NULL)  {
        printf ("Create LogThread failed with error %x. Exiting application.\n",
                         GetLastError());
        return -1;
    }

    // Wait for both of the threads to exit
    {
        HANDLE ThreadHandles[2];
        ThreadHandles[0]  = GpibThread;
        ThreadHandles[1]  = LogThread;
        WaitForMultipleObjects(2,               // number of handles to wait on
                               ThreadHandles,  // array of thread handles
                               TRUE,           // wait for ***all***
                               INFINITE        // wait forever
                              );
    }

    // Check the exit code of both threads
    {
        DWORD ExitCode;

        GetExitCodeThread (GpibThread, &ExitCode);
        if (ExitCode == -1)  {
            printf ("GPIB thread failed.\n");
        }
        GetExitCodeThread (LogThread, &ExitCode);
        if (ExitCode == -1)  {
            printf ("Log thread failed.\n");
        }
    }

    CloseHandle (GpibThread);
    CloseHandle (LogThread);
    CloseHandle (BufferFree);
    CloseHandle (DataReady);

    return 0;
}

DWORD GpibThreadFunction(LPDWORD Param)
{
    int ud;               // handle of the device
    int counter;          // used to count the number of measurements taken
    BOOL error = FALSE;   // boolean used to indicate an error has occurred

    // Open handle to the device
    ud = ibdev (0,       // connect board is gpib0
                3,       // device primary address is 3
                0,       // device doesn't use secondary addressing
                T10s,    // I/O timeout period - 10 seconds
                1,       // use EOI to mark the end of I/O writes
                0);      // don't use an EOS character at the end of reads
    if (ud == -1)  {
        printf ("ibdev call failed with ibsta = 0x%x, iberr = %d, ibcntl = %x.\n",
                    ThreadIbsta(), ThreadIberr(), ThreadIbcntl());
        printf ("exit application\n");
        MoreDataComing = FALSE;
        return -1;
    }

    // While fewer than NUMBER_MEASUREMENTS have been acquired
    // continue getting more measurements.
```

```
    for (counter = 0; counter < NUMBER_MEASUREMENTS; counter++) {

        // Tell device to send a measurement
        ibwrt(ud, "SOUR:FUNC SIN; SENS:DATA?", 25);
        if (ThreadIbsta() & ERR)  {
            printf ("Write of command to device failed!. Exit application.\n");
           error = TRUE;
           break;
        }
        // Wait forever for the buffer for the measurement to be available
        WaitForSingleObject (BufferFree, INFINITE);

        // Acquire the reading
        ibrd (ud, Measurement, 2000);
        if (ThreadIbsta() & ERR)  {
            printf ("Read from device failed!. Exit application.\n");
           error = TRUE;
           break;
        }
        // Signal the LogThread that a reading is available
        SetEvent (DataReady);
    }

    // put the handle offline before exiting
    ibonl (ud, 0);

    MoreDataComing = FALSE;
    if (error)  {
       return -1;
    }
    else  {
       return 0;
    }
}

DWORD LogThreadFunction(LPDWORD Param)
{
    // While more data is coming from the GpibThread continue formatting
    // and logging.
    while (MoreDataComing) {

        // wait for the next data to be acquired by the GpibThread
        while (MoreDataComing)  {
            // If WaitForSingleObject on DataReady returns WAIT_OBJECT_0,
            // the DataReady event has been signaled by the GpibThread.
            // In this case, the wait is only for 1000 milliseconds = 1 sec
            // instead of an infinite (no timeout) wait because the GPIB
            // instrument could fail and the reading might never be acquired.
            // If this error case happens, you want the LogThread to be able
            // to exit gracefully.
            if (WaitForSingleObject (DataReady, 1000) == WAIT_OBJECT_0)  {
               break;
            }
        }
        // if there isn't any more data coming, break out of the outer loop
        if (!MoreDataComing)  {
           break;
        }

        // Format and display the data as necessary
        FormatAndLogMeasurement (Measurement);
        // Signal the GpibThread that the buffer is available for more data
        SetEvent (BufferFree);
    }
    return 0;
}
```