KPCI-3108 or DAS-1800AO Series Stimulus/Response Test Using DriverLINX® and Excel

by
Matt Holtz
Keithley Instruments, Inc.

Introduction

It is often necessary to measure how a given device responds to a known input signal. To do this, an input signal (stimulus) is applied to the device. The output of the device (response) is then measured in real time, and the data is stored to be analyzed later. Since the analysis of such data can be complex, it should be handled by a computer. Additionally, the software should be automated, so that the user need not perform tedious tasks to handle this data.

Tests of this type must be able to measure the response at precisely the same time as the stimulus signal in order to accurately measure phase shifts, gains, rise times, and other characteristics. For this type of application, a Keithley computer-based data acquisition (DAQ) board is ideal. This is because DAQ boards have relatively high frequency sampling rates, and the DAQ board's analog input (AI) and analog output (AO) channels can be set to use the same clock (the pacer clock on the DAQ board), so that each pulse of AO corresponds to a simultaneous acquisition.

This application note describes how to set up and use a KPCI-3108 or a DAS-1800AO Series DAQ board to measure the frequency response of an amplifier. Specifically, it explains the example program's interface and how to use the example program with its Microsoft® Excel automation that records and analyzes data from the amplifier. The conclusion deals with the differences between the KPCI-3108 (a 16-bit, 100 Ksamples/second (KS/s) board), and the DAS-1800AO (a 12-bit, 333KS/s board). The following is an outline of the contents of this application note:

- Test Description
- Test System Configuration
- Program Overview
- Performing the Test
- Equipment List
- Conclusions
- Program Walkthrough
  - Initialization
    1. Initialize Driver
    2. Initialize Device
    3. Set up Service Requests

  - Acquisition

    1. Starting Data Acquisition
    2. Trapped Service Request Events
    3. Terminating Data Acquisition

  - Analysis

    1. Opening Excel with OLE Automation
    2. Writing Data to and Updating Excel
    3. Closing Excel and Displaying File in OLE Window

Please note that throughout the rest of this document, the DAS-1800AO Series DAQ boards will simply be referred to as the DAS-1800AO.

Test Description

The most important characteristic of an amplifier is its bandwidth. A typical amplifier acts as a low-pass filter, and will attenuate frequencies above its cutoff frequency (its -3dB point). To measure the bandwidth, the frequency response of the amplifier must be measured. Since the DAS-1800AO has a maximum sampling rate of 333KS/s, and the defined sine wave consist of 20 samples per cycle, this puts an upper frequency constraint of approximately 16KHz (333KS/s/20 samples) on the measurement. The KPCI-3108, however, is slower, because of its better bit resolution. Its maximum frequency is 100KS/s, so this limits the bandwidth to about 5KHz (100KS/s/20 samples). To increase the bandwidth, a lower number of samples per cycle could be used as a stimulus.

To measure the bandwidth of an amplifier, its frequency response over several decades must be measured. Thus, the DAS-1800AO should be measured at 1Hz, 10Hz, 100Hz, 1KHz, and 10KHz, while the KPCI-3108 should be measured at 1Hz, 10Hz, 100Hz, 1KHz, and 5KHz. The sample program included in this application note is designed to output these waveforms on the analog output channel, as well as record the response and analyze the data.

Test System Configuration

In this application, two channels from the DAQ board are used: an analog output (AO) channel and an analog input (AI) channel. Any channel can be used, as long as the software code controlling the board is modified to use that channel. *Figure 1* illustrates the physical wiring setup.

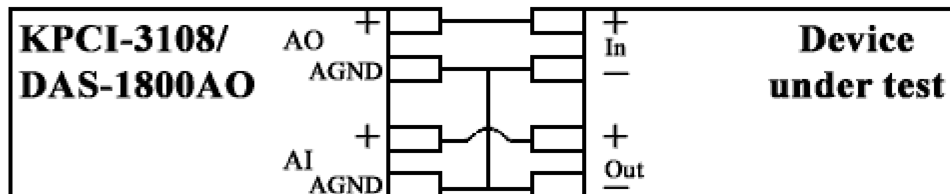


Figure 1. Wiring the Board for this Application

Here, AO can be either output channel (there are two AO channels on both the DAS-1800AO and the KPCI-3108 boards). Likewise, AI can be any of the input channels on the board (both the DAS-1800AO and the KPCI-3108 have 16 input channels in single-ended mode or 8 in differential mode). AGND refers to either LL_GND (DAS-1800AO) or AGND (KPCI-3108).

This test will be set up to use voltages well above the millivolt range, so noise should not be a problem. Therefore, the DAQ board should be configured in single-ended mode, with the AO and AI channels sharing a common ground (LL_GND or AGND).

Program Overview

Before running the test, it is useful to look at the software that controls it.

This program was written in Visual Basic using DriverLINX. However, the program was written with simplicity in mind, so it does not trap errors and may terminate when incorrect values are used. For example, if you set the name of the Excel Input

file to a file which does not exist, the program will terminate.

The program itself provides an interface to the AO and AI channels. The program first initializes the data acquisition board, then waits for the user to perform an action. When running the program, the screen in *Figure 2* displays.

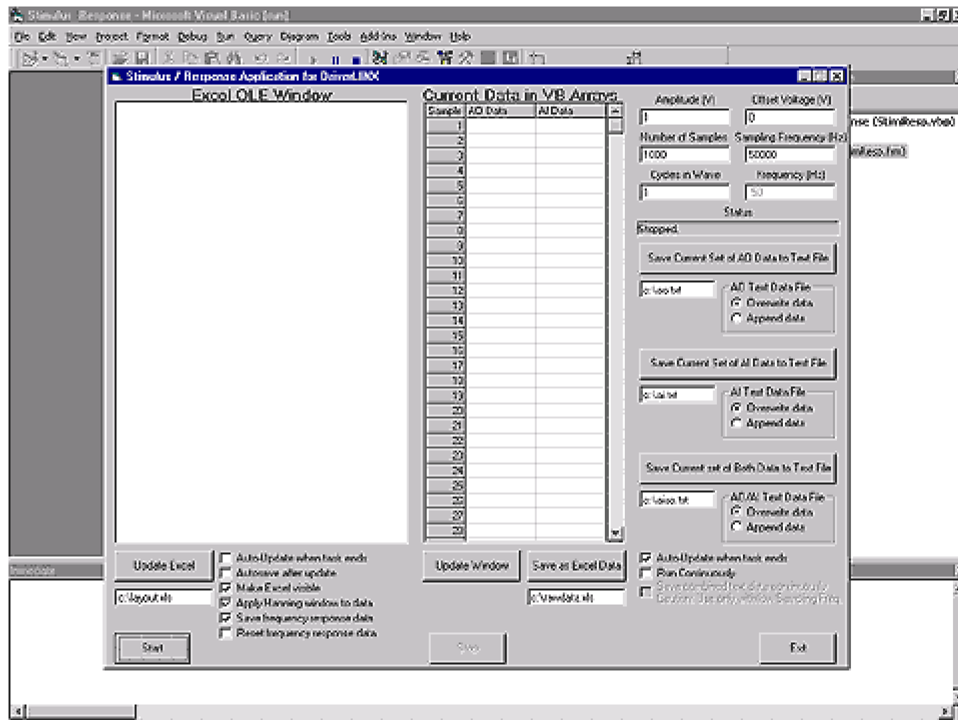


Figure 2. Example User Interface

The first things to notice are the two large windows. Both windows display data taken from the DAQ board. The window on the right, titled Current Data in VB Arrays, displays data stored in the Visual Basic arrays. If this window is empty or contains old data, it can be updated via the Update Window button below it.

The window to its left is an Excel OLE window. It will display a link to the filename listed below it. This window will display the Excel spreadsheet containing the data analysis.

The next thing to notice on this form is the output wave characteristics: amplitude, offset voltage, number of samples, etc. The wave parameters are displayed, ready for editing, in the upper right corner. The only parameter that is not modifiable is the frequency. This value is a function of the sampling frequency (Sf), the number of samples (Ns), and the number of cycles per waveform (fc). The frequency (f) is computed as:

$$f = \frac{S_f}{N_s / f_c}$$

Immediately below the AO wave characteristics is a status box, which displays information as the program is running. It will, for example, say what portion of the Excel worksheet it is updating, or, if an error occurs, will return the DriverLINX error number.

Below the status box are three areas that relate to file I/O. All three will output the current data in the Visual Basic arrays to an ASCII file with the name given in the text box. The format of the file is ideal for input into a spreadsheet program, since it is comma delimited. The user can choose to either overwrite the given file (no warning is given if a file exists) or append the current data to it. Note that if it is simpler or more convenient to save the current data directly to an Excel spreadsheet, click the "Save as Excel Data" button.

There are three checkboxes in the lower right area of the form:

- **The Auto-Update When Task Ends checkbox**

  When this checkbox is selected, the following occurs. If new data is acquired and DriverLINX has just completed the data acquisition task, then the program will automatically update the Current Data in VB Arrays window with the data it received. This is useful to see whether a circuit is stable. The user should click the Start button a number of times (with this box checked) to see whether the values are changing or staying the same.

- **The Run Continuously checkbox**

  When this checkbox is selected, the program will execute continuous service requests, so that the wave is outputted continuously (not periodically, as if the user were just quickly hitting the Start button). While this occurs, the status box will display information about which buffer it is receiving data from. The Current Data in VB Arrays window will *not* update itself (to do so would take far too much processor time), but the window can be manually updated at any time. If *Auto-Update when task ends* is checked, then the new data will be displayed when the Stop button is clicked.

- **The Save Combined Text Data Continuously checkbox**

  Selecting this checkbox causes the program to stream the data it takes to an ASCII text file. The ASCII text file is comma delimited, and the filename is the same as the one used by the save AI/AO data command (under the Status box).

  Be very cautious when using this function, for several reasons. First, even at what seems like relatively low sampling rates (50KS/s, for example), it is possible that the system will become busy and not respond immediately. Also, there is a point at which Visual Basic cannot completely output its data before another SR_BufferFilled() event occurs. At this point, the written file will have gaps in data. Lastly, at high sampling rates, files can easily become several Mbytes in a few seconds.

There are six checkboxes in the lower left area of the form:

- **The Auto-Update When Task Ends checkbox**

  When this checkbox is selected, the following occurs. If new data is acquired and DriverLINX has just completed the data acquisition task, then the program will automatically update the Excel OLE window with the data it received. This is useful to see whether a circuit is stable. The user should see the data, along with various analyses performed on it, in the window.

  Excel will be updated when a task has completed. Since the Excel update is not fast, this option is unchecked by default.

- **The AutoSave After Update checkbox**

  When selected, this checkbox instructs the program to save the Excel file after automatically updating. When this box is not checked, Excel asks the user to save changes (if bad data was recorded, or if formatting errors occurred, the user can say no and update again). When it is checked, it executes File -> Save, just as if the user had done so.

- **The Make Excel Visible checkbox**

  The user can elect to watch as the Excel automation progresses, or simply wait while it occurs in the background. If the user elects to wait, useful information about what is happening is displayed in the Status box. Bear in mind that when the Excel window is minimized or in the background, the task will take longer, because the application is not active.

- **The Apply Hanning Window to Data checkbox**

  In layout.xls, Excel is instructed to perform an FFT on the acquired data (and the sent data). Since this acquisition has a finite time, the data is actually windowed by the Heaviside (step) function. This means that in the frequency domain, a convolution will occur.

  The net result is that the FFT will have ripples that don't actually occur in the data. One way to combat this is to use

an alternate windowing function on the data, such as the Hanning. It convolves much more nicely in the frequency domain, and the data's spectrum will look better. The data for this function is stored to the right of the graphs on Sheet1.

- **The Save Frequency Response Data checkbox**

  Selecting this checkbox causes the program to save the gain and phase information into the appropriate location on Sheet2 and update the graphs to match. If a wide enough range of frequencies are chosen and enough data points are acquired, a good quality Bode plot is created. (See *Figure 3*.)

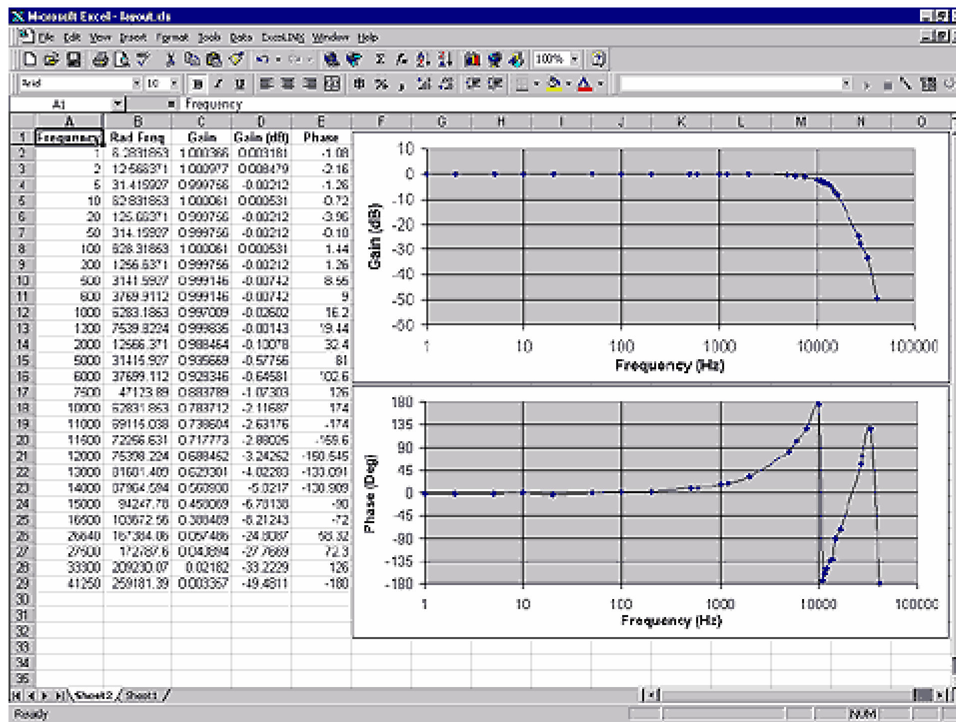
- **The Reset Frequency Response Data checkbox**

  Selecting this option will cause all current frequency response data in Sheet2 to be deleted, so be careful when using this option with autosave. If important frequency response data is stored there, it will be lost when the spreadsheet saves automatically.

NOTE: The spreadsheet file that the program uses to analyze its data in is much like a template. All the formatting and columns are already set up. The program knows exactly where to write its data beforehand and does so.

Performing the Test

To calculate the transfer function H(jw) of the amplifier, different frequency waves are sent and the program records the resulting waveforms. Sheet2 in the template file layout.xls contains the frequency response data that has been recorded. Testing over a wide range of frequencies yields data as shown in *Figure 3*.



Frequency	Rad Freq	Gain	Gain (dB)	Phase
1	6.2831853	1.000366	0.003181	-1.08
2	12.566371	1.000977	0.008479	-2.16
5	31.415927	0.999756	-0.00212	-1.26
10	62.831853	1.000061	0.000531	-0.72
20	125.66371	0.999756	-0.00212	-3.96
50	314.15927	0.999756	-0.00212	-0.10
100	628.31853	1.000061	0.000531	1.44
200	1256.6371	0.999756	-0.00212	1.26
500	3141.5927	0.999146	-0.00742	8.56
600	3769.9112	0.999146	-0.00742	9
1000	6283.1853	0.997009	-0.02602	16.2
1200	7539.8224	0.999835	-0.00143	19.44
2000	12566.371	0.988464	-0.10078	32.4
5000	31415.927	0.935669	-0.57756	81
6000	37699.112	0.928346	-0.64581	102.6
7500	47123.89	0.883789	-1.07303	126
10000	62831.853	0.783712	-2.11687	174
11000	69115.038	0.738604	-2.63176	-174
11500	72256.631	0.717773	-2.88026	-159.6
12000	75398.224	0.688452	-3.24262	-150.545
13000	81681.409	0.629301	-4.02283	-133.091
14000	87964.594	0.560900	-5.0217	-130.909
15000	94247.78	0.450069	-6.78138	-90
16500	103672.56	0.388409	-8.21243	-72
26640	167384.06	0.057486	-24.8087	58.32
27500	172787.6	0.040894	-27.7669	72.3
33300	209230.07	0.02182	-33.2229	126
41250	259181.39	0.003357	-49.4811	-180

Figure 3. Example of a Test with a Wide Range of Frequencies

Equipment List

The following equipment is required to set up a Stimulus-Response test using the sample program:

- KPCI-3108 or DAS-1800AO Series data acquisition board
- Physical device to test (this example uses an amplifier)
- Microsoft Windows¨ 95, 98, or NT
- DriverLINX software with Visual Basic support functions installed
- Microsoft Excel 98 or later
- Microsoft Visual Basic 6.0 or later

Conclusions

The major requirement for this application is to have a measurement device that has a larger bandwidth than the bandwidth of the device under test. Thus, any slower-sampling data acquisition PC board can be used, as long as it is fast enough for the device under test. Additionally, the DAQ board must have analog output channels. Thus, devices such as the DAS-1701AO or DAS-1702AO will work with about one half the bandwidth.

The tradeoff between the KPCI-3108 and the DAS-1800AO in this application is essentially between speed and accuracy. The KPCI-3108 has 16-bit DACs (Digital-to-Analog Converters) and 16-bit ADCs (Analog-to-Digital Converters), with a 100KS/s throughput. The DAS-1800AO, on the other hand, has 12-bit DACs and ADCs, but with a 333KS/s throughput. Thus, when using a gain of 1 in the ±5V range, the KPCI-3108 has a voltage resolution of 152.6 μV, while the DAS-1800AO has a resolution of 2.44mV. In this application, the speed is advantageous, because small quantization errors from inputs are not really an issue, whereas bandwidth is an issue.

It is also good to note that the source code given here is device independent. That is, as long as DriverLINX is used, and the DAQ board in the computer supports the required functions, *any* Keithley DAQ board (not just the DAS-1800AO or KPCI-3108) can be used without changing the source code at all. (In this example, the same exact code was used with the DAS-1800AO then the KPCI-3108). The code is device independent and therefore highly portable. This represents a huge savings in software engineering and test development time.

Program Walkthrough

This section of the document is intended to explain how DriverLINX operates within this downloadable example program (stimresp.zip), as well as how to automate Excel from Visual Basic. It is divided into 4 sections: initialization, acquisition, analysis, and cleanup.

Initialization

1. Initialize the Driver

In any Visual Basic code, the first code to execute is form_load(). Within this particular application, that code is the following:

```
Call Initialize
Call CreateSineWave(OutputWave, OutputWaveData(), NumberOfSamples)
Call SetupInitialServiceRequests
```

The first line calls the code that executes all of the initialization. This makes the code far cleaner. Within Initialize(), the first important bit of initialization code (without comments) is:

```
StatusLabel.Caption = "Initializing DriverLINX. . ."
DriverName = OpenDriverLINXDriver(SR1, "", True)
```

The function OpenDriverLINXDriver (contained in DLVBLib.bas) has three arguments. The first refers to a service request. The second argument is a string containing the name of the DriverLINX driver you wish to open. If the string is null, DriverLINX will prompt the user for a driver. The third is a boolean informing DriverLINX whether it should prompt the user or not.

In this case, the service request's name is SR1, and it can be "created" by dragging the Service Request object onto the form you are designing. (This can also be done for the Logical Device Descriptor (LDD)). See the red arrows in the screen capture in *Figure 4*. In this particular program, there are two service requests: one for AO, and one for the AI. However, there is only one LDD, since we are using the same logical device.

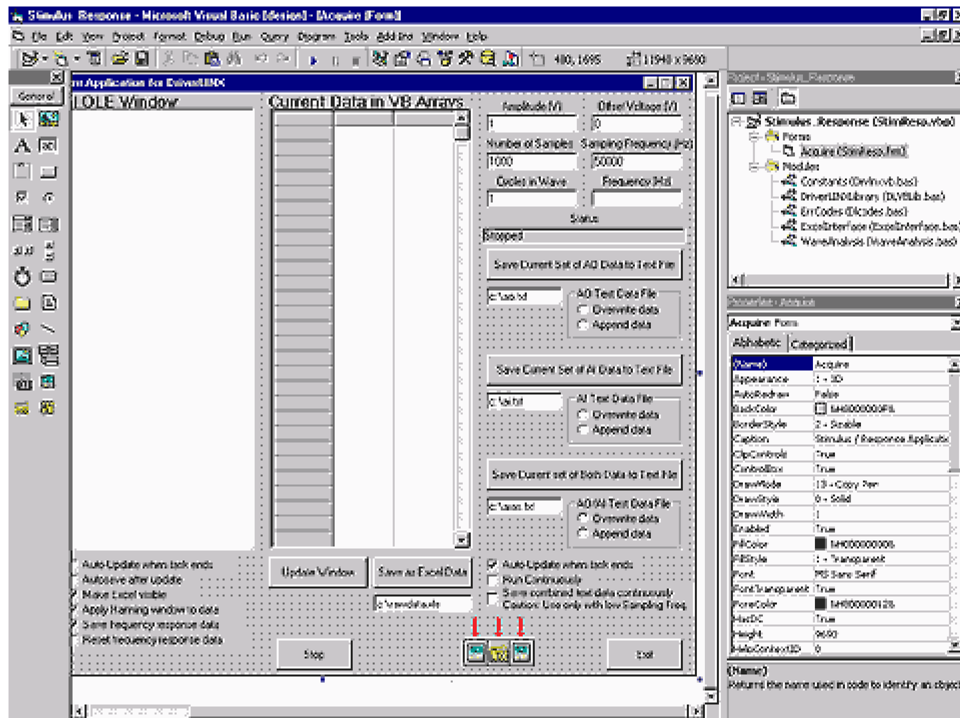


Figure 4. Application's form at design time

This routine will return either the name of the driver it opened or null if no driver was opened. We cannot proceed without an open driver, so we trap the device name.

```
If DriverName = "" Then
   MsgBox "No driver opened.", vbOKOnly, "Error Opening Driver"
   End
End If
```

Next, we need to make sure that both service requests refer to the same driver. OpenDriverLINXDriver changed SR1 to refer to the driver it opened, and we will manually change SR2 to the same driver name.

```
SR2.Req_DLL_name = SR1.Req_DLL_name
```

Incidentally, you can view all the properties of a DriverLINX service request from within Visual Basic by clicking on the service request you created on the form.

2. Initialize the Device

The next part of the initialization is actually initializing the device. This is somewhat involved and has several loops in case the default driver 0 is not correct, but only the important function calls will be shown.

```
Call InitDevice(DriverName)

StatusLabel.Caption = "Stopped."

Private Sub InitDevice(ByVal DriverName As String)

. . .

  If ValidDevice Then   'Try to (re)initialize
    result1 = InitializeDriverLINXDevice(SR1, DeviceNumber)
    result2 = InitializeDriverLINXDevice(SR2, DeviceNumber)

      If result1 <> DL_NoErr Or result2 <> DL_NoErr Then
        ValidDevice = False
      Else  'There was no error initializing.  Exit this loop. (The only way out)
        Exit Do
      End If
```

The first time this example program executes, it will always execute the block of code above. The key function call is InitializeDriverLINXDevice(). The function returns a value. This value is defined in the DLCodes module to make for easier reading. The first parameter to the function is, of course, the service request. The second is the device number you wish to open. In this program, that value is initialized to 0 (this is where numbering starts).

3. Setup Service Requests

After the device has opened, we need to set up the service requests to perform the data acquisition task. The

form_load() routine calls the function that performs this, SetupInitialServiceRequests():

```
Private Sub SetupInitialServiceRequests()
        If chkContinuous.value Then
                Call SetupDriverLINXContinuousBufferedIO(SR1, LDD,
                        DeviceNumber, DL_AI, LogicalChannelInput, _ChannelGain,
                        SamplingFrequency, NumberOfSamples, NumberOfBuffers,
                        _BackGroundForeGround)

                Call SetupDriverLINXContinuousSynchronizedBufferedIO(SR2,
                        SR1, LDD, DeviceNumber, DL_AO, _LogicalChannelOutput,
                        ChannelGain, NumberOfSamples, _NumberOfBuffers,
                        BackGroundForeGround)
        Else
                Call SetupDriverLINXBufferedIO(SR1, LDD, DeviceNumber,
                        DL_AI, LogicalChannelInput,     _ChannelGain,
                        CSng(SamplingFrequency), NumberOfSamples,
                        _NumberOfBuffers, BackGroundForeGround)

                Call SetupDriverLINXBufferedIO(SR2, LDD, DeviceNumber,
                        DL_AO, LogicalChannelOutput, _ChannelGain,
                        CSng(SamplingFrequency), NumberOfSamples,
                        _NumberOfBuffers, BackGroundForeGround)

                AddTimingEventSyncIO SR2, GetSubSystemsDefaultClock(SR1,
                        LDD, SR1.Req_subsystem)
        End If
End Sub
```

There are three important functions used in this code segment:

1. SetupDriverLINXBufferedIO(),
2. SetupDriverLINXContinuousBufferedIO(), and
3. SetupDriverLINXContinuousSynchronizedBufferedIO().

The first function, SetupDriverLINXBufferedIO(), does exactly what it says. Since we want both AO and AI to be buffered, we set up both to do so. We also need to make sure that the two service requests are set to use the same clock (in this case, the board's internal Pacer clock). AddTimingEventSyncIO() does this.

The second function, SetupDriverLINXContinuousBufferedIO(), is exactly the same as the first with the exception that the service request is set to operate continuously (i.e., repeat itself until the software says stop). This means that DriverLINX will continually read data from the AI channel into its memory buffer until told not to.

The third function, SetupDriverLINXContinuousSynchronizedBufferedIO(), is the same as the second except that it takes care of synchronizing the two service requests. Actually, all this function does differently than the second is that it performs the AddTimingEventSyncIO().

The file DLVBLib.bas defines many functions. Generally, there will be a function for any basic data acquisition task you wish to do. For more complex tasks, you may have to add options directly to your service request (for example, AddTimingEventSyncIO()).

Once the user clicks on the Start button, but before the data acquisition begins, there is one other function that needs to be called in order to write the data created (from CreateSineWave()) to the output buffers:

```
. . .
   Call PutDriverLINXAIBuffer(SR2, 0, NumberOfSamples, OutputWaveData())
```

```
   Call PutDriverLINXAIBuffer(SR2, 1, NumberOfSamples, OutputWaveData())
. . .
```

This function copies the data that we have written to the OutputWaveData() array to SR2's memory buffer in order to begin output.

Acquisition

1. Starting Acquisition

Within this application, data acquisition begins when the user clicks on the Start button:

```
Private Sub cmdStart_Click()
  . . .

  SR2.Refresh 'Output
  SR1.Refresh 'Input

End Sub
```

These two methods of the DriverLINXSR type are what actually set things in motion. They execute the service request, using all of the modified properties as its parameters. In this case, we want to output before we input so that we can excite the circuit first; therefore, SR2 is before SR1.

2. Trapped Service Request Events

Remember that the Windows environment is entirely event driven. That is, programs will wait for certain things (events) to happen before some things execute. DriverLINX, since it is written for Windows, follows this same design paradigm.

There are nine events that a DriverLINX service request can execute, but something will happen only if you attach code to the event you wish to trap. For example, if you want to do something when a critical error occurs, then you have to write a function named SR1_CriticalError() (or whatever your service request is named) with code consistent with your goal.

The following lists the nine service request events (they are explained in the DriverLINX manual in detail):

- BufferFilled
- CriticalError
- DataLost
- ServiceDone
- ServiceStart

- StartEvent
- StopEvent
- TimerTic
- UserBreak

Only some of these events matter to us in our particular application. Since we are buffering AI data, we need to know when our input buffer is full. SR1 (our input service request) will execute this routine every time its buffer is full (as long as sel_buf_notify is set to DL_NOTIFY; otherwise it sends no event messages). We trap a full buffer as follows:

```
Private Sub SR1_BufferFilled(task As Integer, device As Integer, subsystem As
        Integer, _mode As Integer, bufIndex As Integer)

                Dim samples As Long

                samples = GetDriverLINXAIBuffer(SR1, bufIndex, InputWaveData())
                StatusLabel.Caption = "Reading buffer" & Str(bufIndex)

                If chkSaveContinuous.value Then
                        'SR1.Sel_buf_notify = DL_NOEVENTS
                        Call WriteAIAOData
                        'SR1.Sel_buf_notify = DL_NOTIFY
                End If

        End Sub
```

The routine above calls the subroutine GetDriverLINXAIBuffer with arguments such that InputWaveData() comes back with our data. This routine also writes the data it just retrieved to the AO/AI text file if that option was chosen on the form. Notice the lines that are commented out. These lines would disable SR1 from calling the BufferFilled event (while the computer is writing data to the hard disk). If your program saves the data without disabling buffer messages, then there is no need to uncomment these lines. They are just in case disk access is slow.

When programming, you may find it useful to use several of the parameters that DriverLINX will give when calling the BufferFilled event. It is a lot of information, and it could be quite useful. This application used only bufIndex.

Another event that is necessary to trap in this application is ServiceStart:

```
Private Sub SR1_ServiceStart(task As Integer, device As Integer, subsystem As
Integer, mode As Integer)

        cmdStart.Enabled = False
        cmdStop.Enabled = True
        . . .
        NumberOfSamplesTextBox.Enabled = False
        SamplingFrequencyTextBox.Enabled = False
        chkSaveContinuous.Enabled = False
        StatusLabel = "Running. . ."

        If chkSaveContinuous.value Then
Open FileBoxBoth.Text For Output As #1
        End If

End Sub
```

This function just disables a few buttons so that it cannot be clicked twice. The function then updates the Status box. It also opens an ASCII file to allow for continuous streaming of data to the disk, if that option is checked on the form. The ServiceStop function is very similar to this function.

The last event of interest that we trap is DataLost. Any application you write should trap this error, as data overruns are possible, especially at high sampling rates with few buffers. In this case, we need to trap for *both* input and output.

```
Private Sub SR1_DataLost(task As Integer, device As Integer, subsystem
  As Integer, mode As Integer, bufIndex As Long, bufElement As Long)

        cmdStop_Click
        MsgBox "Data lost!", vbOKOnly, "Error"
        cmdStart.Enabled = True
End Sub

. . .
(SR2_DataLost())
. . .
```

This routine just does the bare minimum. It lets the user know of data lost events and quits the program. But this routine has the potential to do a lot of bug reporting. DataLost could report the taskid, device, subsystem, buffer, and even the buffer element in question. This example program's CriticalError handler performs about the same function.

3. Terminating Data Acquisition

Our example program stops acquiring data on software command or error. The way a user stops the data acquisition is by clicking on the Stop button. The code executed when this happens is:

```
Private Sub cmdStop_Click()
Dim Status As Integer
        Status = StopDriverLINXIO(SR1)
        StatusLabel.Caption = "DriverLINX returned " + Str(Status) + "."

        If Status = DL_NoErr Then
Do
        DoEvents
 Loop Until cmdStart.Enabled
        End If

        Status = StopDriverLINXIO(SR2)
        StatusLabel.Caption = "DriverLINX returned " + Str(Status) + "."

        If Status = DL_NoErr Then
Do
        DoEvents
 Loop Until cmdStart.Enabled
        End If
```

```
        cmdStop.Enabled = False
        AmplitudeTextBox.Enabled = True
        CyclesTextBox.Enabled = True
        OffsetTextBox.Enabled = True
        NumberOfSamplesTextBox.Enabled = True
        SamplingFrequencyTextBox.Enabled = True
End Sub
```

The critical DriverLINX function executed here is StopDriverLINXIO. The return code of this function is a standard DriverLINX error code. This function just makes sure that, if there is no error, the respective SR_ServiceStop has been executed.

Analysis

1. Opening Excel With OLE Automation

OLE Automation with Visual Basic follows the same object model that the rest of Windows follows. To open a new document (as is necessary when saving our data to a new file), the following lines need to be executed:

```
 Dim xlApp As Excel.Application
 Dim xlBook As Excel.Workbook
 Dim xlSheet As Excel.Worksheet
 Dim xlChart As Excel.Chart

 Set xlApp = New Excel.Application
 Set xlBook = xlApp.Workbooks.Add
 Set xlSheet = xlBook.Worksheets("Sheet1")
```

It is very important to remember the "New" keyword. If this word is not present, Visual Basic may open a copy of Excel, which is already open. "Add" is a method of the Workbooks object. You can (and will probably have to) view the objects that you are referencing in Visual Basic using the object browser.

Opening an existing file is somewhat different:

```
Dim xlApp As Excel.Application
Dim xlBook As Excel.Workbook
Dim xlSheet As Excel.Worksheet
Dim xlSheet2 As Excel.Worksheet
Dim xlChart As Excel.Chart
Dim xlFourierAmplitudeChart As Excel.Chart

frm.StatusLabel.Caption = "Opening Excel. . ."
Set xlApp = New Excel.Application
Set xlBook = xlApp.Workbooks.Open(frm.FileBoxExcelLink.Text)
Set xlSheet = xlBook.Worksheets("Sheet1")
Set xlSheet2 = xlBook.Worksheets("Sheet2")
Set xlChart = xlSheet.ChartObjects(1).Chart
Set xlFourierAmplitudeChart = xlSheet.ChartObjects(2).Chart
```

This time we must use the Open method instead of the Add method. Otherwise it is very similar to when we dimensioned the variables and set them to new objects.

2. Writing Data To and Updating Excel

This part can be difficult, especially if there is little organization in the code. However, this example program was written with modules and functions wherever needed, so it should be somewhat easy to read. The entire process of writing data to Excel is very procedural. The first thing the example program does is update the sample number column (column A in layout.xls), as well as the AO and AI columns (C and D). This is all done quite simply:

```
With xlSheet
        For i = 0 To NumberOfSamples - 1
                .Cells(i + 2, 1) = i + 1
                .Cells(i + 2, 3) = OutputWaveData(i)
                .Cells(i + 2, 4) = InputWaveData(i)
        Next I
End With
```

An interesting thing to note here is that the cells() property of an excel.worksheet takes (row, col) designation, which is suitable for "For" loops. However, the range property can take standard Excel designations (for example, "A2").

It is also interesting to note that Excel may be slow on some systems. Sometimes other open copies of Excel can cause update delays, so make sure that all other copies of Excel are closed. Also, updating seems to take longer when Excel is not visible, so that option is enabled by default.

The next thing to do in Excel is to update individual cells with individual functions and values:

```
.Range("$B$2") = "=($A2*$K$19)/$K$20"
. . .
.Range("$J$2") = "=DEGREES(ATAN2(IMREAL($F2),IMAGINARY($F2)))"
.Range("$A$" & (FourierInput + 2) & ":$T$37768").Clear
. . .
```

These are just like the sort of values that you would type into cells when editing a worksheet, except that some parts will vary with each execution (FourierInput, for example). Note that some of the Excel functions used here require "Analysis Toolpak" and "Analysis Toolpak - VBA" to be installed. The application will attempt to load them, but they must be available from the "Add-Ins" menu.

The rest of the program proceeds as shown above, making updates on columns, charts, and whatnot. If the

Frequency Response box is checked, then that portion of the code will execute as well.

3. Closing Excel and Displaying File in OLE Window

It is of the utmost importance that the application correctly releases the objects it was using, or else it will just sit resident in memory, even after the program has terminated. Correct closing routines are:

```
xlBook.Close
xlApp.Quit

Set xlBook = Nothing
Set xlSheet = Nothing
Set xlChart = Nothing
Set xlApp = Nothing
```

The "set x = nothing" lines are especially important, because they release the objects. The "xlBook.Close" and "xlApp.Quit" lines just perform the "File ->Close" and "File->Quit" functions-the same functions found in Excel's File menu.

Cleanup

The last thing to do after completing our data acquisition and analysis is to close the driver:

```
Private Sub cmdExit_Click()
        cmdStop_Click

        Call CloseDriverLINXDriver(SR2)
        Call CloseDriverLINXDriver(SR1)

        Close
        End
End Sub
```

The CloseDriverLINXDriver() function takes a service request and closes the driver associated with it.