

KEITHLEY

DriverLINX Programming

Tutorial Manual

A GREATER MEASURE OF CONFIDENCE



DriverLINX Programming Tutorial Manual

©2001, Keithley Instruments, Inc.
All rights reserved.
Cleveland, Ohio, U.S.A.
First Printing, October 2001
Document Number: KPCI-904-01 Rev. A

Manual Print History

The print history shown below lists the printing dates of all Revisions and Addenda created for this manual. The Revision Level letter increases alphabetically as the manual undergoes subsequent updates. Addenda, which are released between Revisions, contain important change information that the user should incorporate immediately into the manual. Addenda are numbered sequentially. When a new Revision is created, all Addenda associated with the previous Revision of the manual are incorporated into the new Revision of the manual. Each new Revision includes a revised copy of this print history page.

Revision A (Document Number KPCI-904-01A)October 2001

Safety Precautions

The following safety precautions should be observed before using this product and any associated instrumentation. Although some instruments and accessories would normally be used with non-hazardous voltages, there are situations where hazardous conditions may be present.

This product is intended for use by qualified personnel who recognize shock hazards and are familiar with the safety precautions required to avoid possible injury. Read the operating information carefully before using the product.

The types of product users are:

Responsible body is the individual or group responsible for the use and maintenance of equipment, for ensuring that the equipment is operated within its specifications and operating limits, and for ensuring that operators are adequately trained.

Operators use the product for its intended function. They must be trained in electrical safety procedures and proper use of the instrument. They must be protected from electric shock and contact with hazardous live circuits.

Maintenance personnel perform routine procedures on the product to keep it operating, for example, setting the line voltage or replacing consumable materials. Maintenance procedures are described in the manual. The procedures explicitly state if the operator may perform them. Otherwise, they should be performed only by service personnel.

Service personnel are trained to work on live circuits, and perform safe installations and repairs of products. Only properly trained service personnel may perform installation and service procedures.

Exercise extreme caution when a shock hazard is present. Lethal voltage may be present on cable connector jacks or test fixtures. The American National Standards Institute (ANSI) states that a shock hazard exists when voltage levels greater than 30V RMS, 42.4V peak, or 60VDC are present. **A good safety practice is to expect that hazardous voltage is present in any unknown circuit before measuring.**

Users of this product must be protected from electric shock at all times. The responsible body must ensure that users are prevented access and/or insulated from every connection point. In some cases, connections must be exposed to potential human contact. Product users in these circumstances must be trained to protect themselves from the risk of electric shock. If the circuit is capable of operating at or above 1000 volts, **no conductive part of the circuit may be exposed.**

As described in the International Electrotechnical Commission (IEC) Standard IEC 664, this instrument is Installation Category I, and signal lines must not be directly connected to AC mains.

For rack mounted equipment in which the power cord is not accessible, in the event of fire or other catastrophic failure, the user must provide a separate power disconnect switch.

Do not connect switching cards directly to unlimited power circuits. They are intended to be used with impedance limited sources. NEVER connect switching cards directly to AC mains. When connecting sources to switching cards, install protective devices to limit fault current and voltage to the card.

Before operating an instrument, make sure the line cord is connected to a properly grounded power receptacle. Inspect the connecting cables, test leads, and jumpers for possible wear, cracks, or breaks before each use.

For maximum safety, do not touch the product, test cables, or any other instruments while power is applied to the circuit under test. ALWAYS remove power from the entire test system and discharge any capacitors before: connecting or disconnecting cables or jumpers, installing or removing switching cards, or making internal changes, such as installing or removing jumpers.

Do not touch any object that could provide a current path to the common side of the circuit under test or power line (earth) ground. Always make measurements with dry hands while standing on a dry, insulated surface capable of withstanding the voltage being measured.

The instrument and accessories must be used in accordance with its specifications and operating instructions or the safety of the equipment may be impaired.

Do not exceed the maximum signal levels of the instruments and accessories, as defined in the specifications and operating information, and as shown on the instrument or test fixture panels, or switching card.

When fuses are used in a product, replace with same type and rating for continued protection against fire hazard.

Chassis connections must only be used as shield connections for measuring circuits, NOT as safety earth ground connections.

If you are using a test fixture, keep the lid closed while power is applied to the device under test. Safe operation requires the use of a lid interlock.

If a  screw is present, connect it to safety earth ground using the wire recommended in the user documentation.

The  symbol on an instrument indicates that the user should refer to the operating instructions located in the manual.

The  symbol on an instrument shows that it can source or measure 1000 volts or more, including the combined effect of normal and common mode voltages. Use standard safety precautions to avoid personal contact with these voltages.

The **WARNING** heading in a manual explains dangers that might result in personal injury or death. Always read the associated information very carefully before performing the indicated procedure.

The **CAUTION** heading in a manual explains hazards that could damage the instrument. Such damage may invalidate the warranty.

Instrumentation and accessories shall not be connected to humans.

Before performing any maintenance, disconnect the line cord and all test cables.

To maintain protection from electric shock and fire, replacement components in mains circuits, including the power transformer, test leads, and input jacks, must be purchased from Keithley Instruments. Standard fuses, with applicable national safety approvals, may be used if the rating and type are the same. Other components that are not safety related may be purchased from other suppliers as long as they are equivalent to the original component. (Note that selected parts should be purchased only through Keithley Instruments to maintain accuracy and functionality of the product.) If you are unsure about the applicability of a replacement component, call a Keithley Instruments office for information.

To clean an instrument, use a damp cloth or mild, water based cleaner. Clean the exterior of the instrument only. Do not apply cleaner directly to the instrument or allow liquids to enter or spill on the instrument. Products that consist of a circuit board with no case or chassis (e.g., data acquisition board for installation into a computer) should never require cleaning if handled according to instructions. If the board becomes contaminated and operation is affected, the board should be returned to the factory for proper cleaning/servicing.

Table of Contents

1 Introduction

About this manual	1-2
Manual conventions	1-2

2 Using the DriverLINX Manual Set

Manuals supplied with DriverLINX	2-2
Using the DriverLINX manuals	2-4

3 Accessing the DriverLINX Tools

Introduction	3-2
Microsoft Visual Basic 6.0	3-3
Microsoft Visual C++ 5.0	3-4
Creating the application	3-5
Building the dialog box	3-10
Adding code	3-15
Running the application	3-18
Sample code fragments	3-19
Borland Delphi	3-24

4 Learning DriverLINX

DriverLINX design philosophy	4-2
Devices, subsystems, and channels	4-3
Logical devices	4-3
Logical subsystems	4-3
Logical channels	4-4
Understanding the Service Request	4-4
Overview	4-4
Control Group	4-6
Request Group	4-6
Events Group	4-6
Select Group	4-7

Results Group	4-8
Auditing the Service Request	4-8
Learning to use the Service Request	4-9
Displaying the Edit Service Request dialog with LearnDL	4-10
Displaying the Edit Service Request dialog from within an application	4-14
Visual Basic	4-14
C/C++	4-15
DriverLINX source file examples	4-15
Analog input subsystem	4-16
Analog output subsystem	4-17
Analog I/O subsystems	4-17
Digital input subsystem	4-17
Digital output subsystem	4-18
Digital input/output subsystems	4-19
Counter/timer subsystem	4-19

5 Common DriverLINX Tasks

Introduction	5-2
Basic sequence of operations	5-2
Opening the DriverLINX driver	5-3
Visual Basic	5-3
C/C++	5-4
Initializing a device	5-4
Visual Basic	5-5
C/C++	5-5
Initializing a subsystem	5-6
Visual Basic	5-6
C/C++	5-6
Closing the DriverLINX driver	5-7
Visual Basic	5-7
C/C++	5-7
Using more than one subsystem on a board	5-8
Using more than one data acquisition board	5-8
Visual Basic	5-8
C/C++	5-9

C/C++ special considerations	5-9
Variable declarations and include files	5-10
Clearing buffers	5-10
Error Check	5-11
Allocate Buffers	5-11
Channel Gain List	5-12
Clearing a Channel/Gain List	5-12

6 Analog and Digital I/O Programming

Introduction	6-2
Basic steps for data acquisition	6-2
Windows messaging	6-4
Using events	6-5
Software command events	6-5
Visual Basic	6-6
C/C++	6-6
Terminal count events	6-7
Visual Basic	6-7
C/C++	6-7
Rate events	6-8
Visual Basic	6-8
C/C++	6-8
Analog events	6-9
Visual Basic	6-10
C/C++	6-10
Digital events	6-11
Visual Basic	6-11
C/C++	6-11
Event delays	6-12
Visual Basic	6-12
C/C++	6-13
Reading a single digital value	6-13
Visual Basic	6-14
C/C++	6-15
Reading a single analog value	6-15
Visual Basic	6-16
C/C++	6-17

Reading a series of digital values	6-17
Visual Basic	6-18
C/C++	6-19
Writing a single digital value	6-20
Visual Basic	6-21
C/C++	6-22
Writing a series of digital values	6-23
Visual Basic	6-23
C/C++	6-25
Reading analog values on several channels	6-26
Visual Basic	6-27
C/C++	6-28
Using messages in a background task	6-30
Visual Basic	6-31
C/C++	6-32
Writing a single analog value	6-35
Visual Basic	6-35
C/C++	6-36
Writing a series of analog values	6-36
Visual Basic	6-37
C/C++	6-38
Writing analog values on several channels	6-40
Visual Basic	6-41
C/C++	6-42
Foreground and background operation	6-43
Visual Basic	6-44
C/C++	6-45
Using an external digital trigger	6-47
Visual Basic	6-48
C/C++	6-50
Analog Input Events	6-52
VB	6-53
C/C++	6-55

7 Alternate API for Digital I/O Boards

Introduction	7-2
Configuration	7-3
Direct I/O using Visual Basic	7-5
Direct I/O using C/C++	7-5
Sample console application	7-5
GUI applications	7-8

8 Counter/Timer Programming

Introduction	8-2
Task vs. group mode	8-2
Task mode	8-2
Group mode	8-3
Brief Hardware Review	8-3
Event counting	8-5
Visual Basic	8-6
C/C++	8-6
Frequency measurement	8-7
Visual Basic	8-9
C/C++	8-9
Pulse width measurement	8-10
Visual Basic	8-11
C/C++	8-12
Square wave generator	8-13
Visual Basic	8-13
C/C++	8-14
One shot pulse generator	8-15
Visual Basic	8-15
C/C++	8-16
Time interval measurement	8-16
Visual Basic	8-17
C/C++	8-23

9 Troubleshooting

Introduction	9-2
LearnDL application	9-2
About LearnDL	9-2
Using LearnDL	9-3
Preparing LearnDL	9-4
Acquiring data with LearnDL	9-7
Error messages	9-8
Displaying error messages	9-9
Visual Basic	9-10
C/C++	9-10
Common error messages	9-10
Interpreting error messages	9-12
Decoding error messages	9-12
Visual Basic	9-12
C/C++	9-12
Responding to error messages	9-13
Visual Basic	9-13
C/C++	9-14
Errors with Windows messaging	9-14

List of Illustrations

3 Accessing the DriverLINX Tools

Figure 3-1	Components dialog	3-3
Figure 3-2	Visual Basic tool box	3-4
Figure 3-3	New projects dialog	3-5
Figure 3-4	MFC AppWizard - step 1	3-6
Figure 3-5	MFC AppWizard - step 2	3-7
Figure 3-6	New project information dialog	3-8
Figure 3-7	Microsoft Development Studio File tab	3-9
Figure 3-8	Starting the dialog box	3-10
Figure 3-9	Adding the buttons	3-11
Figure 3-10	Setting button properties	3-12
Figure 3-11	Edit properties dialog	3-13
Figure 3-12	Adding button functions	3-14
Figure 3-13	aiODlg.cpp file cope addition points	3-16
Figure 3-14	aiODlg.h file code addition points	3-17
Figure 3-15	Running the application	3-18
Figure 3-16	Borland Delphi 5 Import ActiveX dialog	3-25

4 Learning DriverLINX

Figure 4-1	Edit Service Request dialog	4-5
Figure 4-2	Edit Service Request dialog	4-9
Figure 4-3	LearnDL Device menu	4-10
Figure 4-4	Select Device window	4-11
Figure 4-5	Edit Service Request dialog	4-12
Figure 4-6	Completed Edit Service Request dialog	4-12
Figure 4-7	Oscilloscope display	4-13

5 Common DriverLINX Tasks

Figure 5-1	Open DriverLINX dialog	5-3
------------	------------------------------	-----

6 Analog and Digital I/O Programming

Figure 6-1	Burst generator	6-41
------------	-----------------------	------

8 Counter/Timer Programming

Figure 8-1	Basic counter elements	8-4
Figure 8-2	16-bit frequency measurement	8-7

9 Troubleshooting

Figure 9-1	OpenDriverLINX dialog	9-4
Figure 9-2	LearnDL main window	9-5
Figure 9-3	Select device dialog	9-6
Figure 9-4	Edit Service Request dialog	9-7
Figure 9-5	DriverLINX modal dialog	9-9

1 Introduction

About this manual

This manual is provided as an introduction to programming with DriverLINX. This manual focuses mainly on programming DriverLINX for the KPCI series boards using the 32-bit driver, such as the KPCI and PIO series, and the DAS 800, 1700, and 1800 series. Some modification of the examples and lessons provided may be necessary when using other data acquisition boards such as the DAS-8, DAS-16, or DAS-1600, and these products are not specifically addressed in this manual.

Programming languages covered in this manual include Visual Basic, Delphi, and C/C++. Examples in the tutorials are provided for Microsoft Visual Basic 6 and for Microsoft Visual C++ 6.0. With the exception of minor syntactical differences, the programming examples provided for Visual Basic may also be used with the Delphi programming language. The C++ source code examples will work with both the C and C++ programming languages.

This manual assumes a basic familiarity with the chosen programming language. This manual does not provide instruction on the selected programming language. It provides DriverLINX programming instruction, with examples and source code in various programming languages. There are many good instructional manuals and courses available for basic programming instruction.

Manual conventions

This manual uses the following conventions when presenting material.

- This Courier font denotes sample code that is presented as it would appear in your program.
- Manual titles, such as the *Analog I/O Programming Guide*, are presented in italics.

The following icons are used to denote the programming language to which the description or sample code applies.

VB Visual Basic/Delphi

C/C++ C/C++

2 Using the DriverLINX Manual Set

Manuals supplied with DriverLINX

There are several manuals supplied with the data acquisition board and DriverLINX. Knowing how to use the manuals is a key step in learning to program DriverLINX applications. The DriverLINX installation program provides an option to install these manuals on your computer. If the manuals are installed, they can be viewed using Adobe Acrobat Reader by selecting the On-line Manuals icon in the DriverLINX program group. A copy of the free Acrobat Reader is located on the DriverLINX installation CD-ROM.

The following manuals are provided with DriverLINX:

- *Analog I/O Programming Guide* — This manual contains information regarding the analog input and output functions of DriverLINX. Some information regarding basic digital I/O and counter/timer operations is also provided. This manual is a good starting point for new DriverLINX programmers.
- *Digital I/O Programming Guide* — This manual contains information on programming the digital input and output functions of DriverLINX. Information is also provided for configuring the digital I/O hardware.
- *Counter/Timer Programming Guide* — This manual provides information for programming the counter/timer functions of DriverLINX.
- *DriverLINX Technical Reference Manual* — The Technical Reference Manual is a reference manual for the DriverLINX API. This manual focuses on the C/C++ programming language. The manual contains a complete description of the service request, logical device descriptor, and DriverLINX support and messaging functions. This manual is intended for the more experienced DriverLINX programmer. Users new to DriverLINX should refer to the Analog I/O Programming Guide.

- *DriverLINX/VB Technical Reference Manual* — This manual is the reference manual that describes the DriverLINX ActiveX control that is used when programming with VB or Delphi. The manual contains a complete description of the service request, logical device descriptor, and DriverLINX support and event functions. This manual is intended for the more experienced DriverLINX programmer. Users new to DriverLINX should refer to the Analog I/O Programming Guide.
- *Using DriverLINX With Your Hardware* — Unlike other DriverLINX manuals, this manual is written specifically to apply to the data acquisition board with which DriverLINX has been supplied. This manual contains information on configuring the features of your board, and the support for those features provided by DriverLINX.
- *Data Acquisition Board User's Manual* — Keithley Instruments supplies a user's manual written specifically for each data acquisition board. This manual focuses on the board itself, and not on DriverLINX. Information provided in this manual includes hardware and software installation, capabilities of the card, calibration, troubleshooting, and specifications.

Using the DriverLINX manuals

The DriverLINX manuals are written to support the hardware independent nature of DriverLINX. To accomplish this, the software manuals contain no hardware-specific information. All hardware-specific information has been placed in the *Using DriverLINX With Your Hardware* manuals. These manuals are written specifically for the data acquisition board with which they are supplied.

Due to the hardware independent nature of the manuals, they contain information which may not be applicable to the data acquisition board you are using. To be sure that any individual function is supported by your data acquisition board, check the *Using DriverLINX With Your Hardware* manual.

When writing new DriverLINX applications, the best manual for new users to reference is the *Analog I/O Programming Guide*. This manual contains basic information on how DriverLINX functions, as well as basic steps for programming simple analog I/O functions. In addition to information on programming analog I/O tasks, this manual contains information on basic digital I/O and counter/timer functions.

For more in-depth information about programming digital I/O functions, refer to the *Digital I/O Programming Guide*. This manual describes how DriverLINX maps the hardware and logical channels for the digital I/O device, how to perform digital I/O tasks, and contains some programming examples. This manual also contains information regarding hardware configuration.

The *Counter/Timer Programming Guide* provides in-depth descriptions of the different counter/timer chips, channel configuration, and task programming. This manual also provides a description of the operating modes for the AMD Am9513 and Intel 8254 timing chips.

3 Accessing the DriverLINX Tools

Introduction

After the DriverLINX drivers are installed on the computer they must be added to the application programming environment before they can be used. There are two components which need to be added: the Service Request control and the Logical Device Descriptor control. For Visual Basic or Delphi, these are separate ActiveX controls. For C/C++, the controls are DLLs.

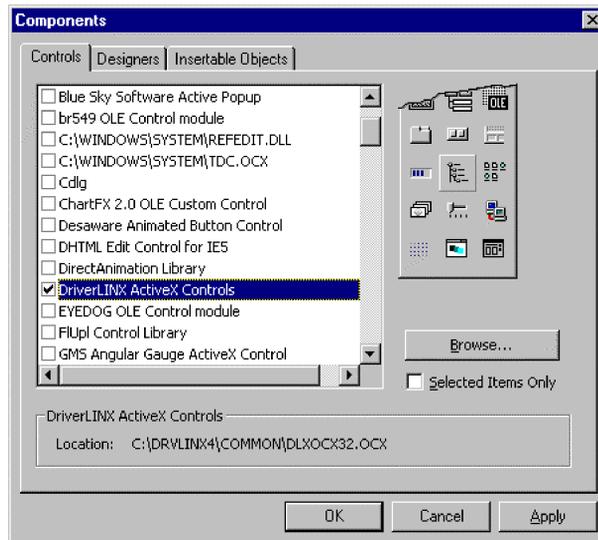
The following sections detail the procedures necessary to use the DriverLINX controls in your application programming environment.

Microsoft Visual Basic 6.0

Adding the DriverLINX ActiveX controls to the Visual Basic tool box is a simple task. Perform the following procedure to add the ActiveX controls.

1. Start Visual Basic and select a project type of Standard Executable.
2. Open the Components dialog, shown in [Figure 3-1](#), using one of the following methods:
 - Right click on the toolbox and select Components from the pop-up menu.
 - From the Project menu, select Components.
 - Use the keyboard shortcut CTRL+T.
3. In the Controls tab of the Components dialog, scroll down the components list to find the DriverLINX ActiveX Controls item. Place a check mark in the box to the left of this item and click the OK button.

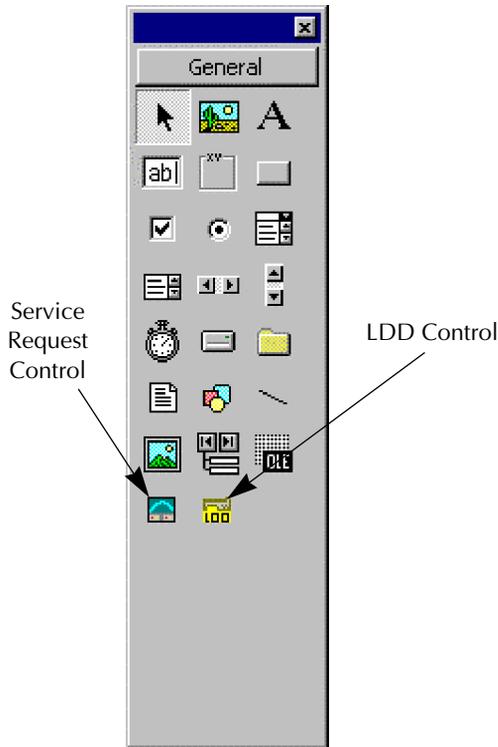
Figure 3-1
Components dialog



- Two new controls are added to the tool box. These are the Service Request Control and the LDD Control, as shown in [Figure 3-2](#).

Figure 3-2

Visual Basic tool box



Microsoft Visual C++ 5.0

C and C++ use the DLL API of DriverLINX. It is possible to use the ActiveX control with C/C++, but this is not covered in this manual. For an example, see the source code provided in the `drvlinx4\source\cpp\AIBuffX` directory.

This sample program uses the DLL API to create a C++ dialog based application using the MFC Application Wizard. This application demonstrates analog input and analog output functions using a KPCI-3108 data acquisition card. This card has a D/A memory buffer to store the wave form data for the analog output function. The

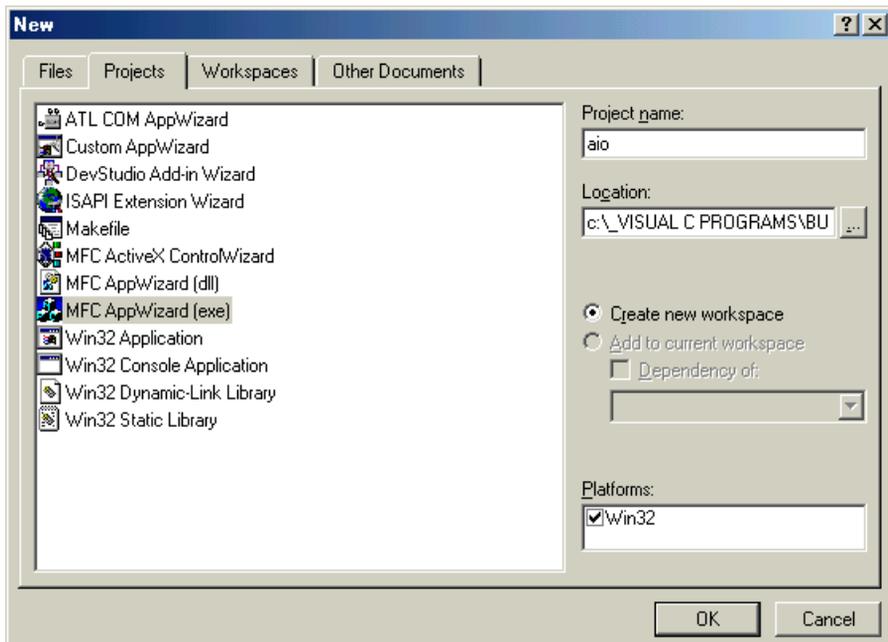
KPCI-3108 oscillator will clock the D/A system at speed. Due to the latency of Windows from sample to sample, it would not be possible to achieve waveform generation while storing the data in a Windows environment. While Windows can burst large amounts of data at speed, the bursting operation can be 20 mS or longer between bursts if there is no plug-in memory card to receive the data. That translates to 50 or less updates per second.

Creating the application

Begin building the application by starting Microsoft Visual C++ 5.0 and select New from the File menu. Perform the following procedure:

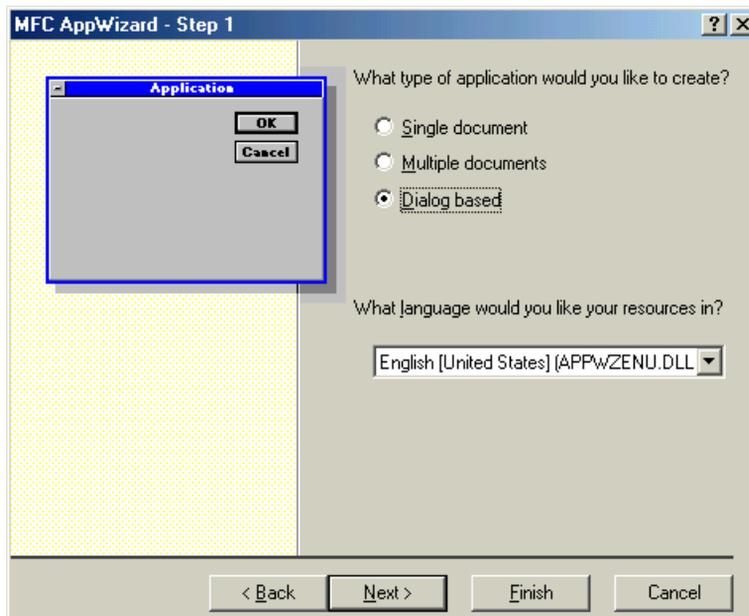
1. From the Projects Tab, select MFC AppWizard (exe). Use the Location: box to specify the path for the new application build, and give the project a name. For this example we will use “aio”. These selections are shown in [Figure 3-3](#). The Platforms: is Win32 and should already be checked. After completing all selections, click OK.

Figure 3-3
New projects dialog



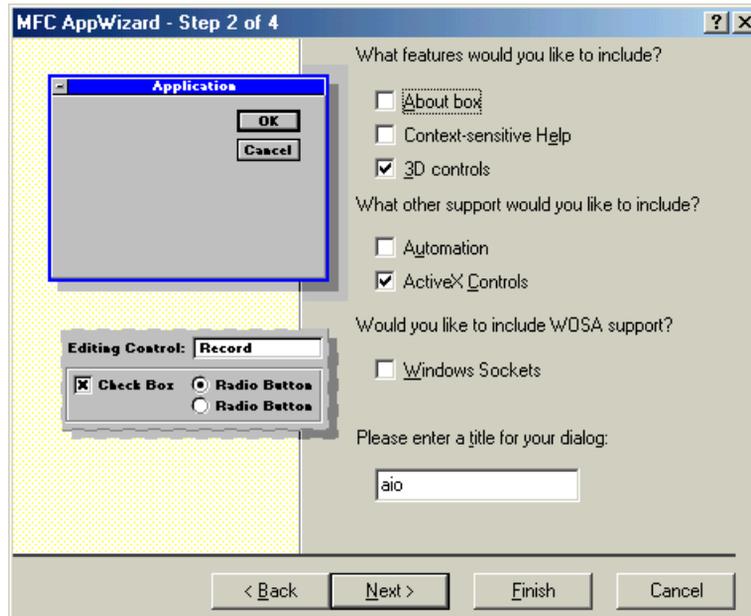
2. The MFC AppWizard, shown in [Figure 3-4](#), provides a choice of application types: Single Document, Multiple Document, or Dialog Based. The sample application code provided with DriverLINX is written using the default Single Document Model. For this tutorial, we will develop an application using the Dialog based model. This will reduce the overall complexity of the application development. Choose Dialog Based, then select Next.

Figure 3-4
MFC AppWizard - step 1



3. For the Analog I/O example developed here does not include an About Box, keeping the code simpler. This requires that the About Box be unchecked. In the MFC AppWizard step 2, shown in [Figure 3-5](#), uncheck the About box check box and click Finish. Although additional options are available in steps 3 and 4, they are not related to this DriverLINX application, and can be left at their default.

Figure 3-5
MFC AppWizard - step 2

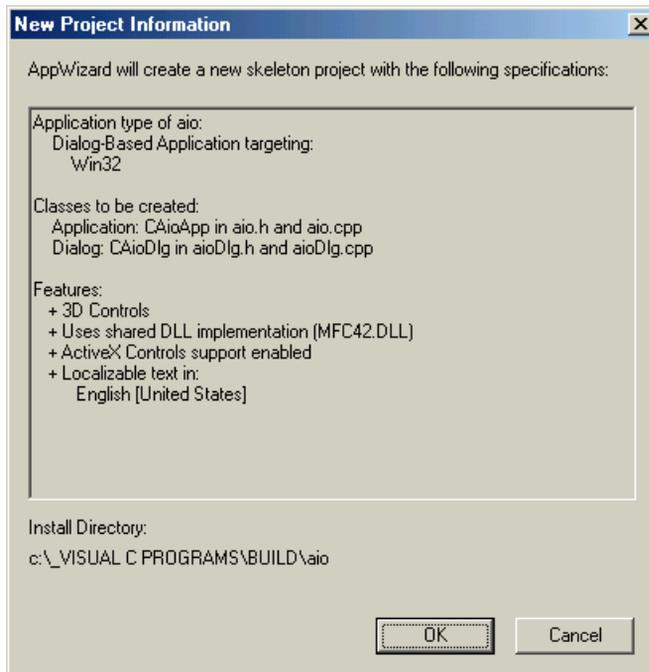


Note:

If your project uses the alternate API for digital I/O as described in [Section 7](#), you must check the Automation check box. However, this is not used in this example program, and this box should be left unchecked.

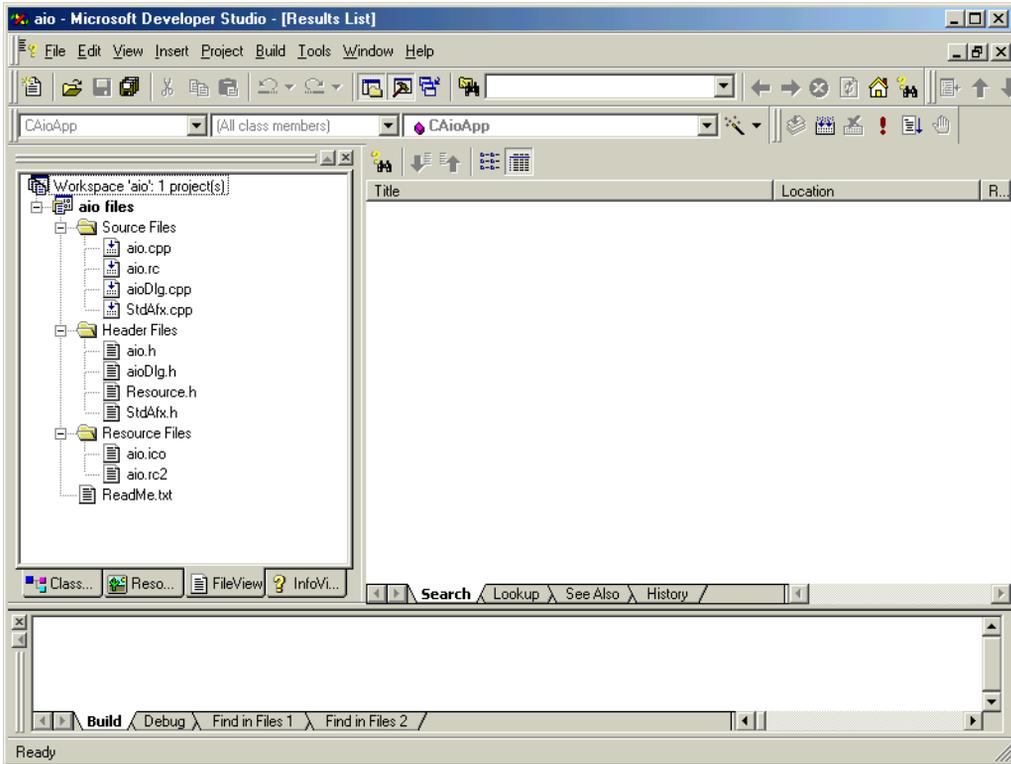
4. The MFC AppWizard will generate a dialog as shown in [Figure 3-6](#). This box lists a summary of the project that will be created. Click OK. The MFC AppWizard will build the foundation code.

Figure 3-6
New project information dialog



5. [Figure 3-7](#) shows the Visual C++ Development Studio project for the aio program with the File tab selected. Note the Source, Header and Resource files, these will be edited to build the application.

Figure 3-7
Microsoft Development Studio File tab

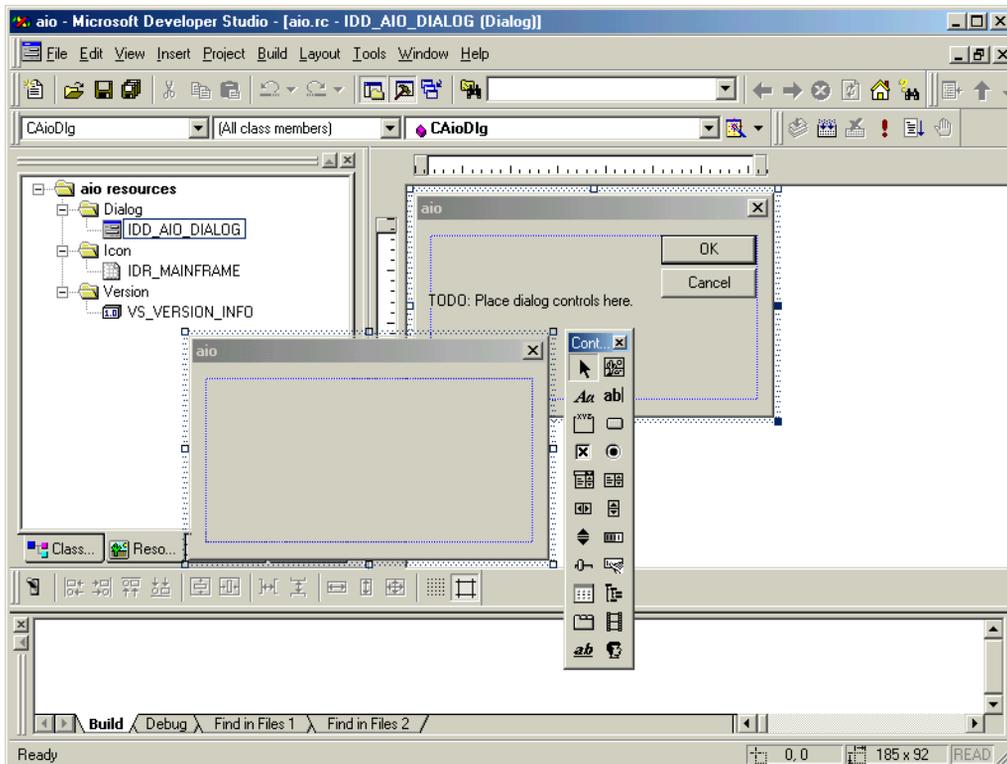


Building the dialog box

The first step of code development is building the visual panel, the 3-D controls that will initiate and report program activity.

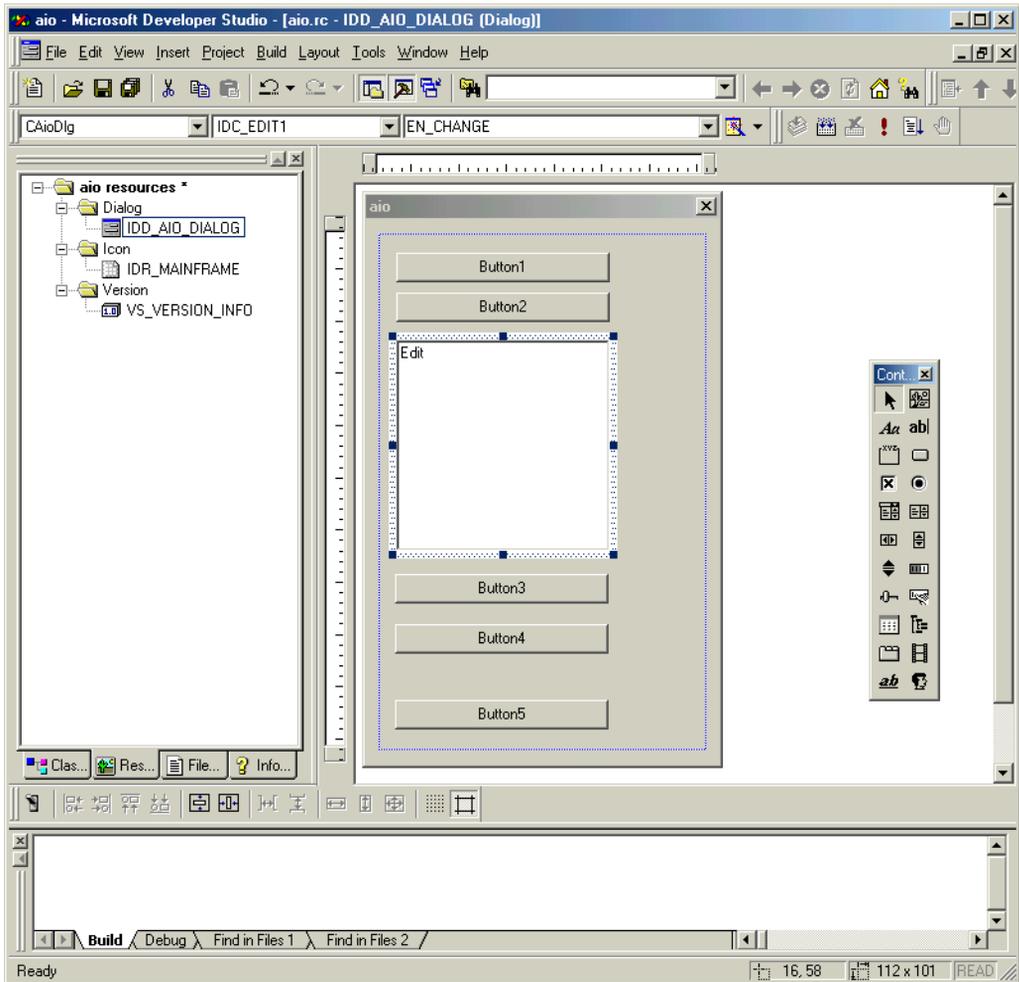
1. Choose Resource View.
2. If the dialog is not already visible, open the folders and double click the `IDD_AIO_DIALOG` icon. The (TODO:Place dialog controls here) panel will display along with the controls pallet, as shown in [Figure 3-8](#).

Figure 3-8
Starting the dialog box



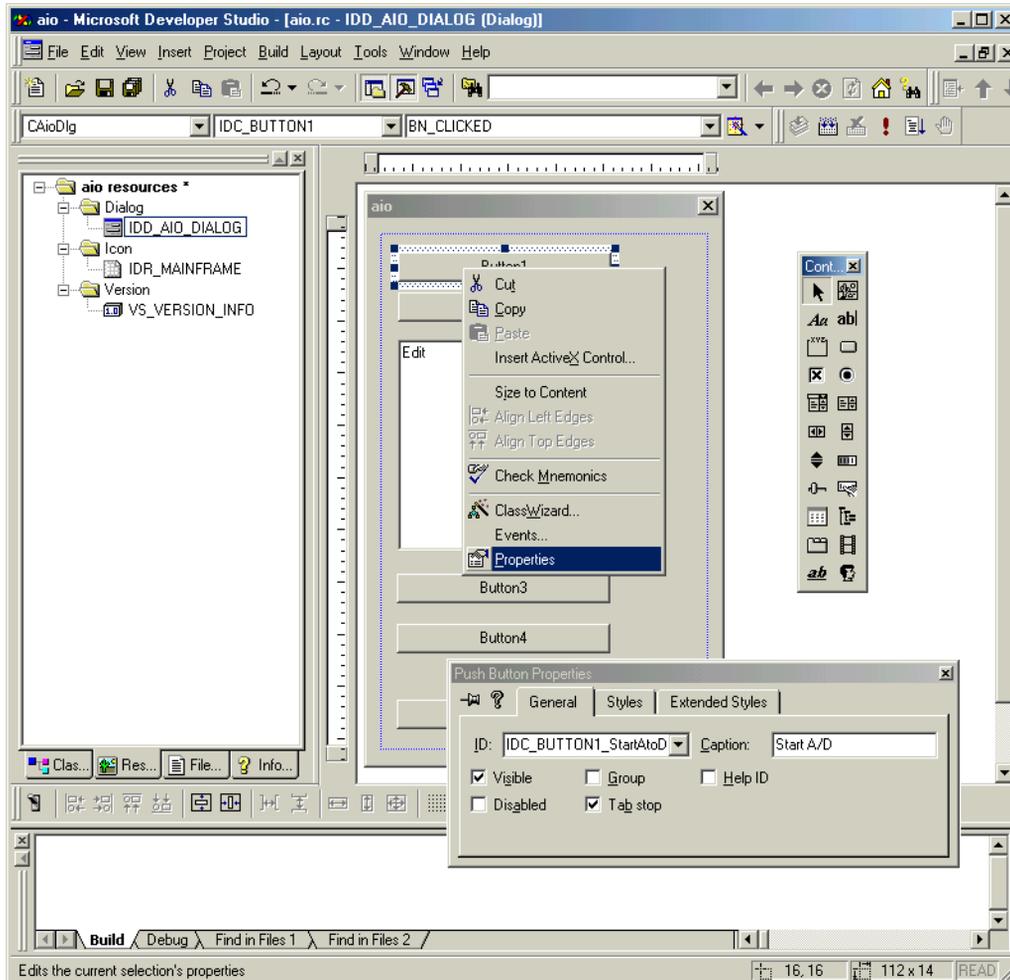
3. Highlight the OK Button in the aio dialog box and choose delete. Do the same for the Cancel Button and the Static Text so we have a clean dialog box to start from.
4. Expand the dialog box to make it larger. From the Controls pallet, add five Push Buttons, one Edit Box, and the Group Box as shown in Figure 3-9.

Figure 3-9
Adding the buttons



5. Right Click on Button1. Choose Properties and edit the PushButton Properties ID and Caption options as shown in Figure 3-10.

Figure 3-10
Setting button properties

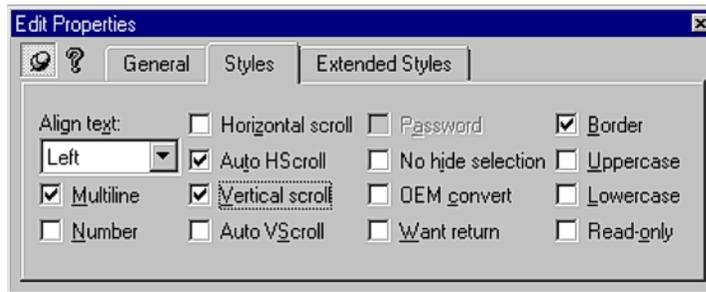


6. Repeat step 5 for PushButtons 2 through 5, using the properties shown below:

Initial Caption	ID	New Caption
Button1	IDC_Button1_StartAtoD	Start A/D
Button2	IDC_Button2_ClearEditBox	Clear
Button3	IDC_Button3_StartDtoA	Start D/A
Button4	IDC_Button4_StopDtoA	Stop D/A
Button5	IDC_Button5_Quit	Quit
Static Text	IDC_STATIC	ai0

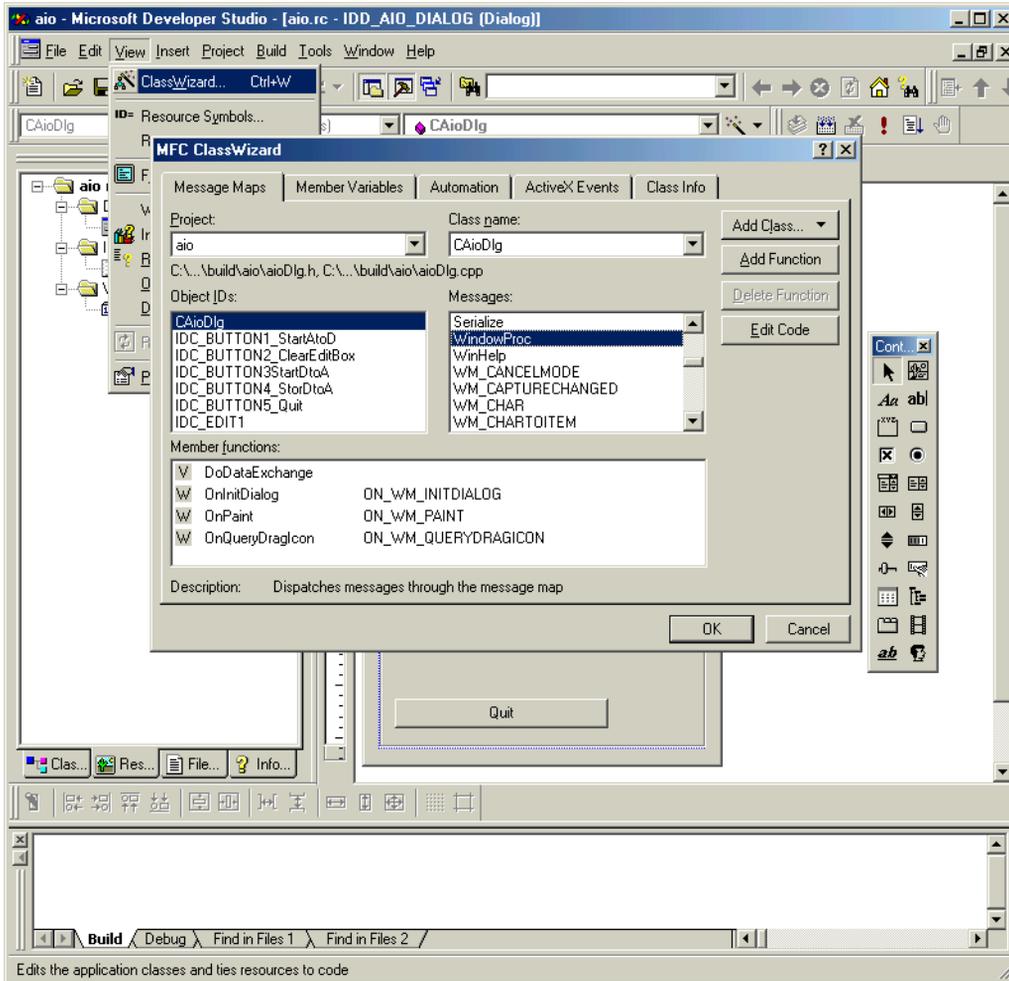
7. For the Edit Box properties, check the Multiline and Vertical scroll check boxes so multiple values can be displayed in the edit box, as shown in [Figure 3-11](#).

Figure 3-11
Edit properties dialog



8. Next, use the ClassWizard to place code for these controls in the program. The ClassWizard is also used to set up Windows Messaging and assign a variable name to the edit box. Refer to [Figure 3-12](#) and perform steps 9 to 11.

Figure 3-12
Adding button functions



Adding windows messaging

9. From the View Tab, Select ClassWizard, note that we are in the Message Maps TAB of the ClassWizard. The CaidDlg Object ID should be highlighted. In the Messages window, scroll down and select WindowProc. Choose Add Function. WindowProc will appear in the Member Functions window. Choose OK. This added function dispatches messages through the message map, an extremely important part of the program.

Adding code blocks for the push buttons

10. From the View Tab, select ClassWizard and highlight the IDC_Button1_StartAtoD selection in the Object ID's box. Highlight BN_Clicked in the Messages Box and choose Add Function, then OK. Repeat this for each of the Push Buttons listed in the Object ID's box.

Adding a variable for the edit box

11. Before closing the MFC ClassWizard, choose the Member Variables Tab. Highlite IDC_Edit2 in the Controls ID window and select Add Variable. Assign a member variable name of m_editBox, and choose OK. Choose OK once more to close the MFC ClassWizard.

Adding code

Figure 3-13 and Figure 3-14 show the aioDlg.cpp and aioDlg.h files as they exist in the application at this time. These are the only files that need to be edited to add all required code for the Service Request. The code to be added is broken into 13 parts. This demonstrates how to build an application after the AppWizard portion is complete.

The 13 code fragments are provided at the end of this example in “[Sample code fragments](#).”

Code part 14 is not a code fragment that must be added, but is simply a reminder to add the DriverLINX library file, DRVLNX32.lib, to the project. This file is normally located in C:\DrvLINX4\DLApi\DRVLNX32.lib. To add this file, select Project > Add to Project > Files from the menu bar.

Refer to Figure 3-13 and Figure 3-14 and place the 13 code parts in the appropriate locations in the AIODgl.cpp and AIODlg.h files.

Figure 3-13
aiODlg.cpp file cope addition points

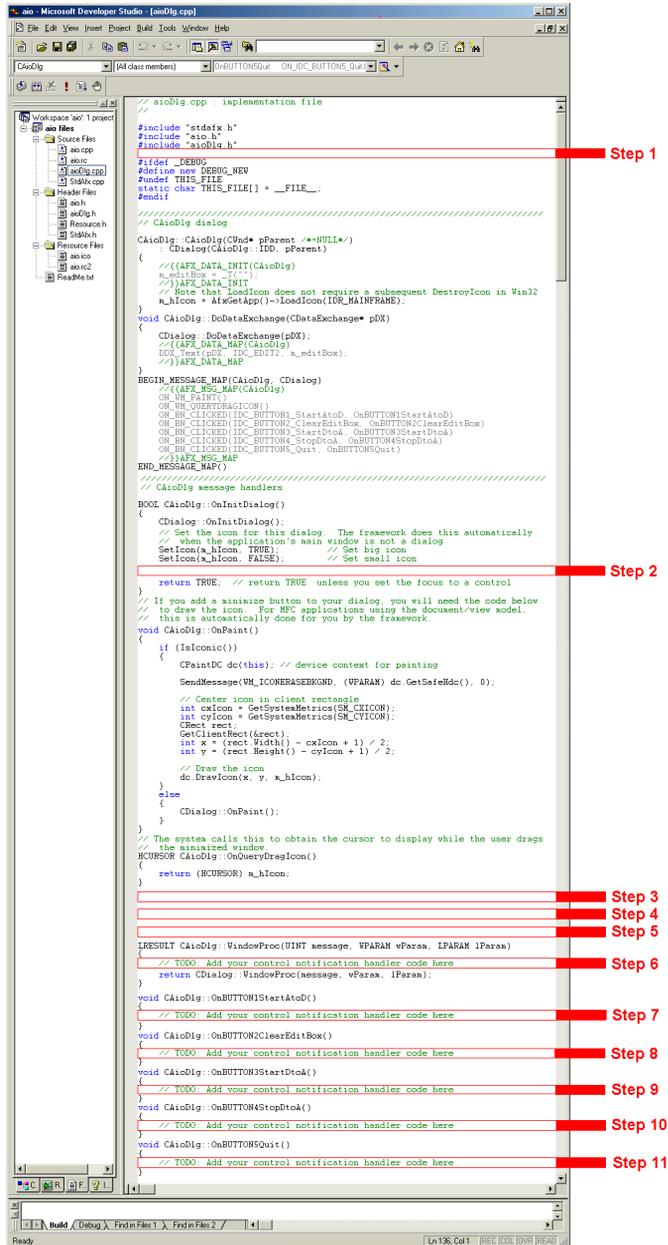
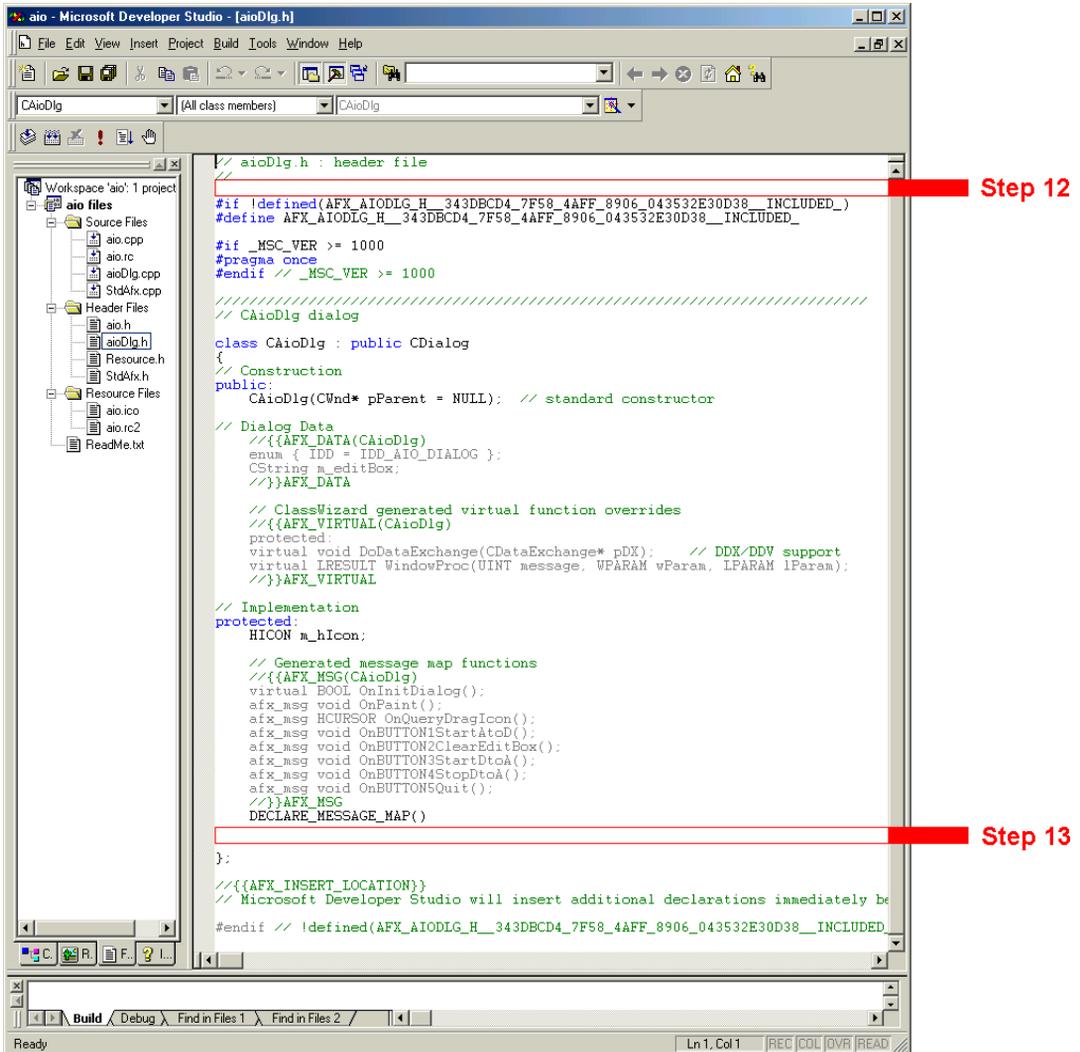


Figure 3-14
aioDlg.h file code addition points

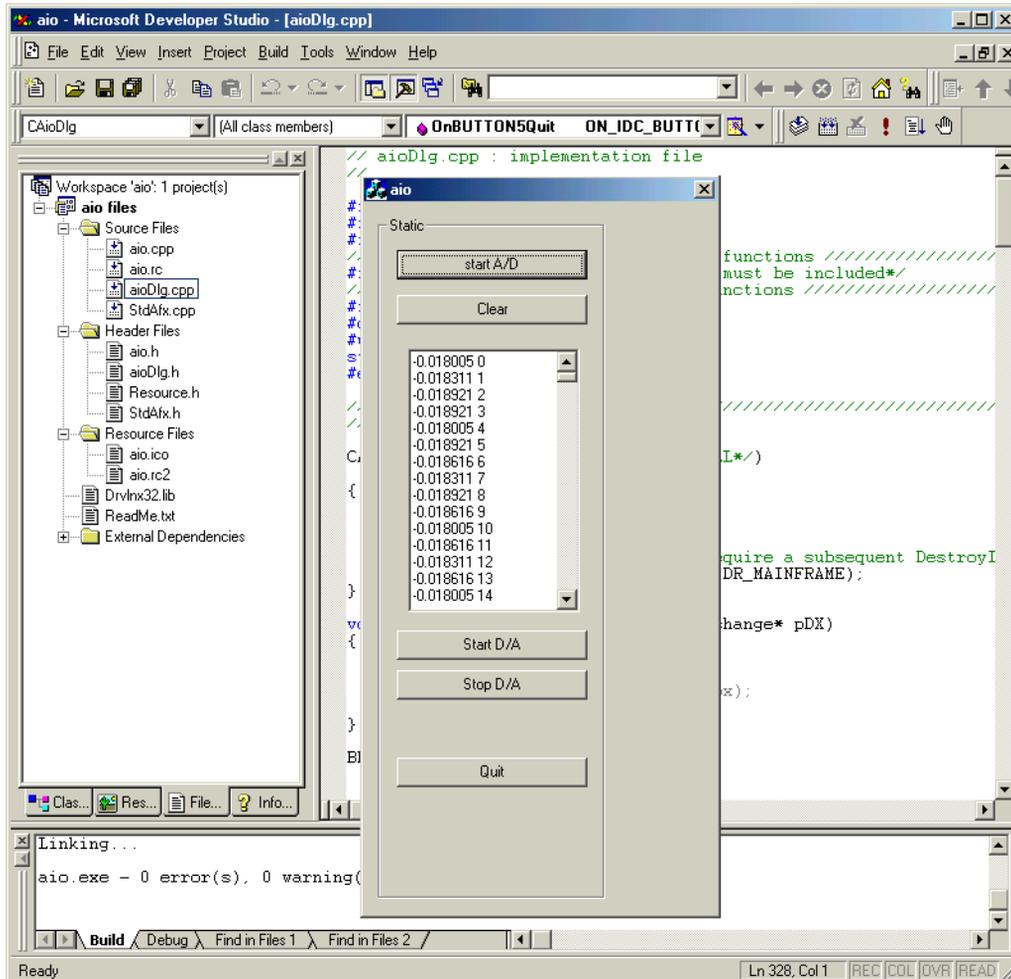


Note that the *DRVLNX32.lib* file has been placed in the project. Once the project has been compiled, there is an additional folder, External Dependencies, in the *aio* files. For the data to report vertically in the Edit Box, you must edit the properties: Multiline and Vertical scroll bars.

Running the application

Figure 3-15 shows the final application running.

Figure 3-15
Running the application



Sample code fragments

Listed below are the program code parts that must be added to the locations shown in [Figure 3-13](#) and [Figure 3-14](#).

```
//AIODLG.cpp//
//Start Step 1 Code: adding the math functions //
#include<math.h>          /*math functions must be included*/
//End Step 1 Code: adding the math functions //

//Start Step 2 Code: initialization//
    m_logicalDevice=0;
    m_samples=1000;
    m_driverInstance=OpenDriverLINX(m_hWnd,"KPCI3108");
    m_AOsr=(DL_ServiceRequest*) new DL_ServiceRequest;
    m_AIsr=(DL_ServiceRequest*) new DL_ServiceRequest;
    memset(m_AIsr,0,sizeof(DL_ServiceRequest));
    DL_SetServiceRequestSize(*m_AIsr);
    memset(m_AOsr,0,sizeof(DL_ServiceRequest));
    DL_SetServiceRequestSize(*m_AOsr);
    m_AOsr->operation=INITIALIZE;
    m_AOsr->device=m_logicalDevice;
    m_AOsr->mode=OTHER;
    m_AOsr->hWnd=m_hWnd;
    DriverLINX(m_AOsr);
    showMessage(m_AOsr);
    m_DLmsg=RegisterWindowMessage(DL_MESSAGE);
//End Step 2 Code: initialization//

//Start Step 3 Code://
void CAioDlg::showMessage(DL_ServiceRequest *SR)
{
    SR->operation=MESSAGEBOX;
    DriverLINX(SR);
}
//End Step 3 Code://

//Start Step 4 Code://
void CAioDlg::clearBuffers(DL_ServiceRequest *SR)
```

```
{
    if (SR!=NULL)
    {
        if (SR->lpBuffers!=NULL)
        {
            if (SR->lpBuffers->BufferAddr [0] !=NULL)
            {
                BufFree (SR->lpBuffers->BufferAddr [0]);
                SR->lpBuffers->BufferAddr [0]=NULL;
            }
            delete SR->lpBuffers;
            SR->lpBuffers=NULL;
        }
    }
}
```

```
//End Step 4 Code://
```

```
//Start Step 5 code://
```

```
void CAioDlg::done()
{
    float *readings;
    readings = new float [m_samples];
    CString temp,fullstr;
    int index;

    m_AIsr->operation=CONVERT;
    m_AIsr->mode=OTHER;
    m_AIsr->start.typeEvent=DATACONVERT;
    m_AIsr->start.u.dataConvert.startIndex=0;
    m_AIsr->start.u.dataConvert.nSamples=m_samples;
    m_AIsr->start.u.dataConvert.numberFormat=tSINGLE;
    m_AIsr->start.u.dataConvert.offset=0.0f;
    m_AIsr->start.u.dataConvert.scaling=0.0f;
    m_AIsr->start.u.dataConvert.wBuffer=0;
    m_AIsr->start.u.dataConvert.lpBuffer=readings;
    DriverLINX (m_AIsr);
    showMessage (m_AIsr);
    m_editBox="";
    fullstr="";
    for (index=0;index<m_samples;index++)
```

```
{
    temp.Format ("%f %d\r\n", readings[index], index);
    fullstr+=temp;
}
m_editBox=fullstr;
UpdateData (FALSE);
delete [] readings;
}
//End Step 5 Code://

//Start Step 6 Code://
if (message==m_DLmsg)
{
    switch (wParam)
    {
        case DL_BUFFERFILLED:
            done();
            break;
    }
}
//End Step 6 Code://

//Start Step 7 Code: Start the Analog Acquisition//
clearBuffers (m_AIsr);
m_AIsr->operation=START;
m_AIsr->subsystem=AI;
m_AIsr->mode=INTERRUPT;
m_AIsr->start.typeEvent=COMMAND;
m_AIsr->timing.typeEvent=RATEEVENT;
m_AIsr->stop.typeEvent=TCEVENT;
m_AIsr->channels.nChannels=1;
m_AIsr->channels.chanGain[0].channel=0;
m_AIsr->channels.chanGain[0].gainOrRange=Gain2Code (m_logicalDevice, AI, -1.0);
m_AIsr->channels.numberFormat=tNATIVE;
m_AIsr->lpBuffers=(DL_BUFFERLIST*) new BYTE [DL_BufferListBytes(1)];
m_AIsr->lpBuffers->notify=NOTIFY;
m_AIsr->lpBuffers->nBuffers=1;
m_AIsr->lpBuffers->bufferSize=Samples2Bytes (m_logicalDevice, AI, 0, m_samples);
```

```
m_AIsr->lpBuffers->BufferAddr[0]=BufAlloc(GBUF_INT,m_AIsr->lpBuffers->bufferSize);
m_AIsr->timing.u.rateEvent.channel=DEFAULTTIMER;
m_AIsr->timing.u.rateEvent.mode=RATEGEN;
m_AIsr->timing.u.rateEvent.clock=INTERNAL1;
m_AIsr->timing.u.rateEvent.gate=DISABLED;
m_AIsr->timing.u.rateEvent.period=Sec2Tics(m_logicalDevice,AI,INTERNAL1,0.00001f);
m_AIsr->timing.u.rateEvent.pulses=0;
m_AIsr->hWnd=m_hWnd;
DriverLINX(m_AIsr);
showMessage(m_AIsr);
//End Step 7 Code: Start the Analog Acquisition//

//Start Step 8 Code: Clear the Edit Box//
m_editBox="";
UpdateData(FALSE);
MessageBeep((WORD)-1);
//End Step 8 Code: Clear the Edit Box //

//Start Step 9 Code:Start the Analog Output//
clearBuffers(m_AOsr);
float *sinebuf;
sinebuf=new float[m_samples];
int i;
for(i=0;i<m_samples;i++)
{
    sinebuf[i]=(float) sin(i*2*3.14159/m_samples);
}
m_AOsr->operation=CONVERT;
m_AOsr->device=m_logicalDevice;
m_AOsr->subsystem=AO;
m_AOsr->mode=OTHER;
m_AOsr->start.typeEvent=DATAACVERT;
m_AOsr->lpBuffers=(DL_BUFFERLIST*) new BYTE[DL_BufferListBytes(1)];
m_AOsr->lpBuffers->nBuffers=1;
m_AOsr->lpBuffers->bufferSize=Samples2Bytes(m_logicalDevice,AO,0,m_samples);
m_AOsr->lpBuffers->BufferAddr[0]=BufAlloc(GBUF_INT,m_AOsr->lpBuffers->bufferSize);
m_AOsr->lpBuffers->notify=NULL;
m_AOsr->channels.nChannels=2;//1;
m_AOsr->channels.chanGain[0].channel=0;
m_AOsr->channels.numberFormat=tNATIVE;
```

```
m_AOsr->start.u.dataConvert.startIndex=0;
m_AOsr->start.u.dataConvert.nSamples=m_samples;
m_AOsr->start.u.dataConvert.numberFormat=tSINGLE;
m_AOsr->start.u.dataConvert.scaling=0.0f;
m_AOsr->start.u.dataConvert.offset=0.0f;
m_AOsr->start.u.dataConvert.wBuffer=0;
m_AOsr->start.u.dataConvert.lpBuffer=sinebuf;
DriverLINX(m_AOsr);
showMessage(m_AOsr);
m_AOsr->operation=START;
m_AOsr->device=m_logicalDevice;
m_AOsr->subsystem=AO;
m_AOsr->mode=INTERRUPT;
m_AOsr->start.typeEvent=COMMAND;
m_AOsr->timing.typeEvent=RATEEVENT;
m_AOsr->timing.u.rateEvent.channel=DEFAULTTIMER;
m_AOsr->timing.u.rateEvent.mode=RATEGEN;
m_AOsr->timing.u.rateEvent.clock=INTERNAL1;
m_AOsr->timing.u.rateEvent.gate=DISABLED;
m_AOsr->timing.u.rateEvent.period=Sec2Tics(m_logicalDevice,AO,INTERNAL1,0.00001f);
m_AOsr->timing.u.rateEvent.pulses=0;
m_AOsr->stop.typeEvent=COMMAND;
DriverLINX(m_AOsr);
showMessage(m_AOsr);
//End Step 9 Code:Start the Analog Output//

//Start Step 10 Code: Stop the Analog Output//
m_AOsr->operation=STOP;
DriverLINX(m_AOsr);
showMessage(m_AOsr);
//End Step 10 Code: Stop the Analog Output//

//Start Step 11 Code: Code block for the QUIT Button//
clearBuffers(m_AOsr);
clearBuffers(m_AIsr);
delete m_AOsr;
m_AOsr=NULL;
delete m_AIsr;
m_AIsr=NULL;
CloseDriverLINX(m_driverInstance);
```

```
m_driverInstance=NULL;
OnOK();
//End Step 11 Code: Code block for the QUIT Button//

//AIODlg.h://
//Step 12 Code://
#include"c:\drvlinx4\dlapi\drvlinx.h"
#include"c:\drvlinx4\dlapi\dlcodes.h"

//Step 13 Code://

private:
    int m_samples;
    void done();
    WORD m_DLmsg;
    void clearBuffers(DL_ServiceRequest* SR);
    int m_logicalDevice;
    void showMessage(DL_ServiceRequest* SR);
    DL_ServiceRequest* m_AIsr; //Declare SS Structure
    DL_ServiceRequest* m_AOsR;
    HINSTANCE m_driverInstance; //To contain Instance Handle

//Step 14 Code://
//Add the DriverLINX library file to the project, C:\DrvLINX4\DLApi\DRVLINX32.lib
```

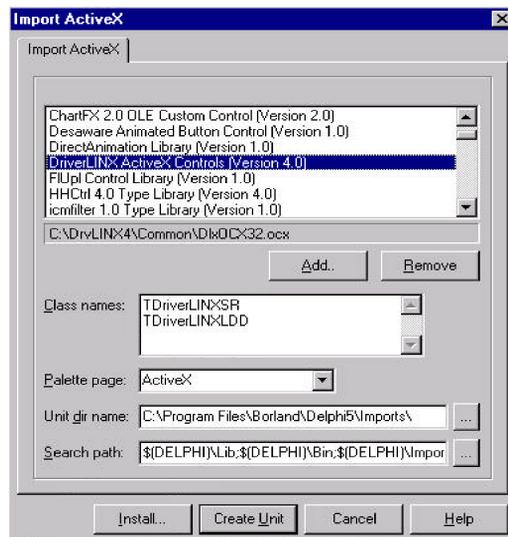
Borland Delphi

Borland programming environments interact with DriverLINX through an ActiveX control. Sample programs written in Delphi are installed when the DriverLINX driver is installed. These are located in the C:\DrvLINX4\Source directory by default.

Adding the ActiveX controls is very similar to the procedure for Visual Basic, but there are some differences. Perform the following procedure to install the ActiveX controls.

1. Start the Borland Delphi program.
2. From the Component menu select Import ActiveX. The Import ActiveX dialog shown in [Figure 3-16](#) will appear.
3. In the alphabetical list of controls, select the DriverLINX ActiveX Controls.
4. Note the names listed in the Class names box. Delphi 5.0 will typically use the names TDriverLINXSR and TDriverLINXLDD. Any class name can be used. However, when moving applications to another computer, the class names must be the same between the two computers to share the application. The sample programs provided on the Keithley Instruments web site are created with the class names TDriverLINXSR and TDriverLINXLDD. To change the name of an installed control, it must be removed and reinstalled with the new name.
5. Click the Install button to close the Import ActiveX dialog.
6. Click the OK button to close the Install Components dialog.

Figure 3-16
Borland Delphi 5 Import ActiveX dialog



4 Learning DriverLINX

DriverLINX design philosophy

DriverLINX is designed as a high-performance data acquisition device driver for 32-bit version of Microsoft Windows. The DriverLINX API is device independent, leading to the development of device independent portable code. DriverLINX is multi-user and multi-tasking, allowing both foreground and background tasks. It allows for multiple programs to access the data acquisition boards simultaneously, if supported by the board.

Due to the device independent nature of DriverLINX, it implements more functions than are available in any single data acquisition board. As a result, any individual board can only use a subset of the DriverLINX full function set. To find the specific functions of DriverLINX supported by an individual data acquisition board, refer to the *Using DriverLINX with Your Hardware* manual for that board.

To aid in developing device independent code, DriverLINX provides a control called the Logical Device Descriptor (LDD). The LDD contains information regarding the physical capabilities of the hardware, including the number and type of channels, operating modes, and others. Applications written with DriverLINX can dynamically obtain information about the data acquisition hardware from the LDD, rather than hard coding for a specific board. The application can then be moved to a different environment with different hardware and still function, provided that the installed board meets the minimum requirements of your application.

DriverLINX does not require the use of the Logical Device Descriptor when creating applications. It is provided as an aide for developing portable applications. When learning to program DriverLINX applications, it is a good idea to avoid the LDD. As you learn more about DriverLINX and grow more comfortable with the DriverLINX API, you can add the LDD capabilities to create more flexible, robust applications. The sample programs provided with DriverLINX are created using the LDD to take advantage of the full capabilities of whatever data acquisition hardware is installed on the system.

Devices, subsystems, and channels

There are three logical levels used to control data acquisition boards when programming with DriverLINX. These are the logical device, subsystem, and channel. When performing any task with DriverLINX, all three of these must be specified.

For information on how DriverLINX assigns logical devices, subsystems, and channels for a particular data acquisition board, refer to the *Using DriverLINX with Your Hardware* manual specific to that board. This manual will list all logical subsystems and channels defined for that board, along with the corresponding physical channels, connectors, and signals.

Logical devices

The logical device corresponds to the physical data acquisition board itself. Systems with more than one data acquisition board will have more than one logical device. The logical device is used to tell DriverLINX which data acquisition board will be used.

There is a maximum of six devices per family, indicated by the driver name, installed at one time. The device numbers should start at zero and increment by one. It is legal to have more than one device numbered zero as long as they correspond to boards from different board families (driver names).

Logical subsystems

The logical subsystem refers to the general function that will be performed. For example, when performing analog input operations, the analog input subsystem is used. DriverLINX considers each data acquisition board to have the following subsystems:

- System – Refers to the actual data acquisition board itself.
- Analog input – Refers to the A/D converters, multiplexers, and associated hardware.
- Analog output – Refers to the D/A converters and associated hardware.
- Digital input – Refers to the digital input ports and associated hardware.
- Digital output – Refers to the digital output ports and associated hardware.
- Counter/timer – Refers to the counter/timer channels and associated hardware.

Logical channels

Logical channels refer to the addressable hardware channels of each subsystem. For most analog and digital subsystems, the logical channel assignment is very straightforward. Logical channel assignments usually match the physical channel assignments. For example, physical analog input channel 0 is assigned to logical channel 0.

DriverLINX may assign a logical channel to items which are not normally thought of as channels. This is most noticeable with counter/timer functions. For example, DriverLINX may assign a 1-bit digital logical channel to external hardware trigger input lines.

For counter/timer subsystems, the logical channel assignments can be more complex. In some cases, multiple logical channels may be assigned to a single physical channel. This may occur when a hardware channel has multiple operating modes. A logical channel may be assigned to each individual counter/timer when operating independently, and another logical channel may be assigned when two counters are used in a cascaded mode. (KPCI-3108 series boards implement this for the counter/timer subsystem.)

In some cases, DriverLINX may assign multiple physical channels to a single logical channel. This occurs when the physical channels cannot be operated independently. In this case, DriverLINX will assign all such physical channels to a single logical channel for ease of operation.

Understanding the Service Request

Overview

A Service Request is a data structure or control created by an application that provides DriverLINX with all the information it needs to perform a task.

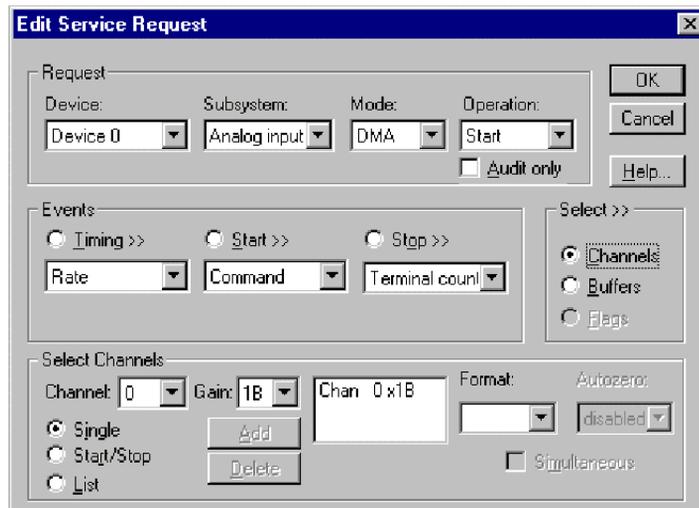
DriverLINX uses a data, or table, oriented approach to tasks, as opposed to a procedural approach. Using DriverLINX, it is not necessary to specify all the steps to be performed to complete a task. Rather, all the required data needed by the task is assigned to the table or control. The data is then sent to DriverLINX all at once. Depending on the programming language, this is done either with a single function call or using the control's refresh method. DriverLINX takes care of the rest.

Not all DriverLINX tasks will use all properties of the Service Request. The properties used are determined by the individual task being performed. Detailed information on the Service Request can be found in the *DriverLINX Technical Reference Manual* in *Section 6 - Service Requests*, and in the *DriverLINX/VB Technical Reference Manual* in *Section 6 - Service Request Control*. All unused properties should be initialized to a defined value, such as zero. This will prevent problems in case updated version of the DriverLINX drivers use previously undefined properties.

The Service Request is depicted graphically by DriverLINX using the Edit Service Request dialog, shown in [Figure 4-1](#). Not all properties of the Service Request dialog are displayed in the Edit Service Request dialog. The properties that are not displayed are assigned dynamically at run-time.

Two ways to display the Edit Service Request dialog are by using the LearnDL application, and by calling it directly from an application. Both of these methods are discussed later in this section.

Figure 4-1
Edit Service Request dialog



The properties of the Service Request are broken into five separate groups. These groups are the Control, Request, Events, Select, and Results groups. The following is a brief description of these groups.

Control Group

VB The Control group is only used when using the ActiveX controls of the DriverLINX. This group contains standard control properties, such as the control name, window handle, and control design-time coordinates. These properties do not directly affect data acquisition tasks. The Control group is not displayed in the Edit Service Request dialog.

Request Group

The Request group contains data specifying the device and subsystem being used, the data acquisition mode, and the operation being performed. When programming in C/C++ the Request group also contains the window handle, since C/C++ does not use the Control group. The Request group is displayed at the top of the Edit Service Request dialog.

Of all the groups in a Service Request, only the Request group is required for all services. The fields of the Events and Select groups are used selectively depending on the particular task requested of DriverLINX.

Events Group

The Events group contains properties controlling the timing of the data acquisition, and the start and stop events for the task. This group is displayed in the center portion of the Edit Service Request dialog. There are several classes of events that can be assigned to the timing, start, and stop properties. These classes are simple, analog, digital, rate, and setup.

The simple events are events that do not require additional properties. These include null events, command events, and terminal count events. Command events are software generated start and stop events. Terminal count events are used when the task should automatically stop after one trip through the data buffers.

Analog events perform real-time (board level hardware/firmware) monitoring of an analog input channel for the occurrence of the specified signal condition. Depending on the features of your hardware, analog events can be used as start and stop events, and may use a delay.

Digital events perform real-time monitoring of a digital input channel and signal when the channel meets the specified condition. Digital events use a mask to specify which bits to monitor, and a pattern to which they are compared. The comparison can be an equals or not equals condition. The mask and pattern properties control rising and falling edge sensitivities.

Rate events can be used as timing events or counter/timer setup events. Timing events pace or clock the data acquisition task. Counter/timer setup events can specify frequency and time interval measurements and event counters.

Setup events configure digital I/O ports. Configuration events are hardware specific. Refer to the *Using DriverLINX with Your Hardware* manual for information on configuration events supported by a specific data acquisition board.

DriverLINX supports many start and stop events. To determine which events are supported by a particular data acquisition board, refer to the *Using DriverLINX with Your Hardware* manual.

Select Group

The Select group defines the channels and data buffers to be used for the task. In addition, a flag property is provided to allow the disabling of some DriverLINX message notifications in specific circumstances.

The channel properties of the Select group specify the logical channel to be used, the gain for the channel, and the number format of the data. Depending on the hardware capabilities, it may be possible to specify a single channel, a range of channels or a channel-gain list. If the hardware does not directly support a channel-gain list, this feature may not be used.

The data buffers properties of the Select group specify the number and size of the data buffers to be used for the data acquisition task. The actual number of buffers supported by DriverLINX varies with the specific version. The exact number may be obtained from the Logical Device Descriptor (LDD). For information on the LDD, refer to the *DriverLINX Technical Reference Manual in Section 7 - Logical Device Descriptors*, or the *DriverLINX/VB Technical Reference Manual in Section 7 - Logic Device Descriptor Control*.

The flags properties of the Select group allow the disabling of event flags. These flags can be enabled to improve performance of DriverLINX in certain circumstances. This

may be helpful when repeatedly polling for a small number of samples. For more information on these flags, refer to the *DriverLINX/VB Technical Reference Manual*.

The Select group is represented in the Service Request dialog, shown in [Figure 4-1](#), in a different manner than the other groups. The right side of the center of the window contains the first portion of the Select group, labeled Select>>. This series of three radio buttons controls which properties are displayed in the bottom of the Edit Service Request dialog. Selecting the Channels, Buffers, or Flags radio button displays the respective properties in the bottom of the dialog. To display a different set of properties, select the appropriate radio button.

Results Group

The Results group contains the results of the task returned by DriverLINX. Using the Results group, DriverLINX returns the status of the task and various information regarding the buffers and the data in the buffers. The actual information returned depends on the type of task performed. The Results group is not displayed in the Edit Service Request dialog.

Auditing the Service Request

DriverLINX automatically performs an audit, or error checking process, on all Service Requests. This process verifies that all the values passed to DriverLINX are valid. The values are verified against the device information stored in the Logical Device Descriptor. The audit process helps prevent problems resulting from executing Service Requests with invalid data.

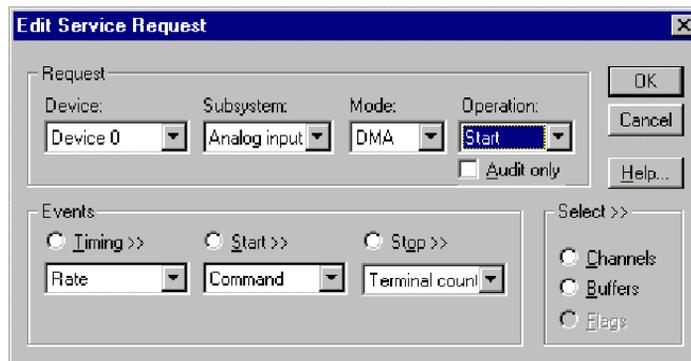
Service Requests can be audited without being executed. Setting the property `Req_op_auditOnly` to `DL_True` when calling a Service Request will audit, but not execute, the Service Request. If the audit process detects any problems with the Service Request, DriverLINX returns an error code. For information on displaying and interpreting error codes, refer to [Section 9, Troubleshooting](#), of this manual.

Learning to use the Service Request

DriverLINX provides some tools to help learn to use the Service Request. These tools include the Edit Service Request dialog and the LearnDL application.

The Edit Service Request dialog, shown in [Figure 4-2](#), is an interactive method of filling out a Service Request. As choices are made, the dialog changes to show the appropriate controls available for the current selections. Controls are added and removed in response to selections. The Edit Service Request dialog will only show controls and choices valid for the selected device. If a given set of selections can be made in the Edit Service Request dialog, then the same configuration may be achieved when programming.

Figure 4-2
Edit Service Request dialog



Displaying the Edit Service Request dialog with LearnDL

The easiest way to display and learn to use the DriverLINX Edit Service Request dialog is through the LearnDL application. LearnDL is installed when the DriverLINX driver itself is installed. A shortcut to LearnDL is created in the DriverLINX program group. The LearnDL application is designed to step you through all the actions required to acquire data using DriverLINX.

The LearnDL application can be very useful for a new DriverLINX programmer. LearnDL provides access to almost all of DriverLINX's features. This allows you to test the data acquisition hardware and ensure DriverLINX is installed and functioning properly. In some cases LearnDL may be used by technical support to assist in setting up and troubleshooting the data acquisition hardware and DriverLINX. More information regarding the LearnDL application can be found in [Section 9, Troubleshooting](#), of this manual.

The following procedure demonstrates using LearnDL to perform a simple analog input operation using the Edit Service Request dialog.

1. From the LearnDL main window, click the Device menu. The menu shown in [Figure 4-3](#) appears. Notice that the only menu choice enabled is Select. You must first tell DriverLINX what device you will be using before you can do anything with the device. Click the Select option.

Figure 4-3
LearnDL Device menu



2. When the Select Device window appears, as shown in [Figure 4-4](#), select the desired logical device. Unless more than one data acquisition board is installed, the logical device should be zero. Click the OK button to select the device and close the window.

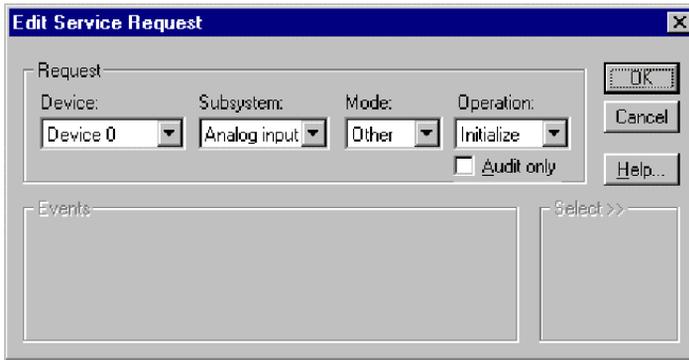
Figure 4-4
Select Device window



3. From the LearnDL main window, click the Device menu. Notice that the Configure and Initialize choices are now available. It is not necessary to configure the device before using it unless the data acquisition task requires a new configuration. Click the Initialize option to set the device to a known state. The Initialize option of the Device menu can also be used to abort any current task.
4. From the Analog Input menu, select Initialize. This will perform a software reset of the subsystem on the selected device. Notice that before the Initialize option is selected, all other options for that subsystem are disabled.

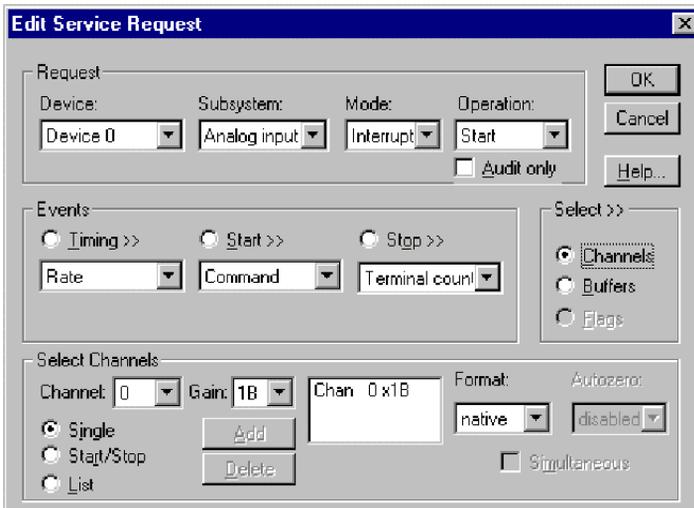
- From the Analog Input menu, select Edit/execute. This will open the Edit Service Request dialog shown in [Figure 4-5](#). Note that there are very few selections available when the dialog opens. As choices are made, additional options will become enabled as appropriate.

Figure 4-5
Edit Service Request dialog



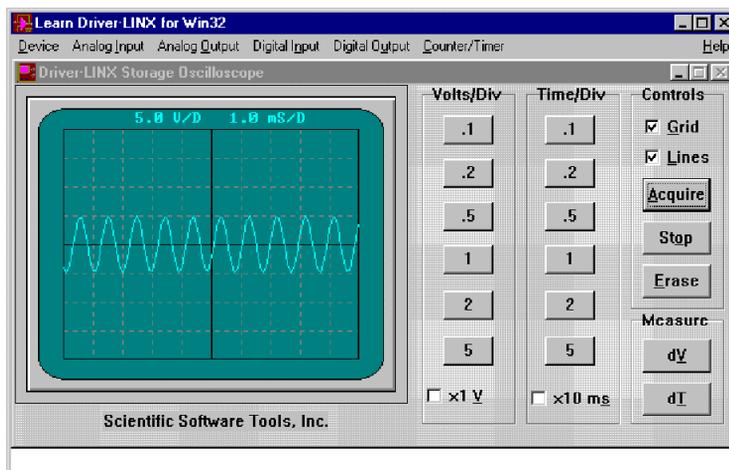
- Fill out the Edit Service Request dialog as shown in [Figure 4-6](#). These settings instruct DriverLINX to take an analog reading on channel 0.

Figure 4-6
Completed Edit Service Request dialog



- To instruct DriverLINX to execute the Service Request, click the OK button. This will close the Edit Service Request dialog and execute the Service Request. When the Service Request is complete, the LearnDL window will update to show the results of the Service Request. This specific Service Request performs a task that acquires 500 data points from analog input channel 0. The data points are graphed on a digital oscilloscope display, as shown in Figure 4-7.

Figure 4-7
Oscilloscope display



- The Service Request can be executed again by selecting Start from the Analog Input menu, or clicking the Acquire button.
- If the Service Request takes an extended period of time to execute, it can be stopped by clicking the Stop button, or selecting Stop from the Analog Input menu. If a task is not currently running when the Stop button is clicked, a DriverLINX error will be displayed.

Displaying the Edit Service Request dialog from within an application

DriverLINX allows an application to display the Edit Service Request dialog at run-time. This feature of DriverLINX is very useful when programming. Several properties can be assigned to a Service Request, then the Edit Service Request box can be displayed to check how DriverLINX applied the properties to the Service Request.

Displaying the Edit Service Request dialog at run-time requires the setting of at least two properties of the Service Request. The first property, Req_DLL_name must be set to the specific DLL that controls the device to which the Service Request will be sent. The second property is the Req_op_edit property. A call to DriverLINX using the Refresh method with Req_op_edit set to True will display the Edit Service Request dialog. When the user dismisses the dialog, Req_op_edit is set to False and DriverLINX returns a result code. If the user canceled the dialog, the Service Request is not updated and DriverLINX returns an error. If no error was returned, the Service Request is updated and can be executed by another call to DriverLINX with the Refresh method.

The following source code examples demonstrate how to display the Edit Service Request dialog.

Visual Basic

VB

```
' Assumes the SR.Req.DLL.Name property is already set to the driver name
' Use this procedure to show the Edit Service Request dialog
Function ShowEditSR (SR As DriverLINXSR) As Integer
    ' Caller should set up a Service Request
    SR.Req_op_edit = DL_True
    SR.Refresh
    ' DriverLINX automatically removes the Req_op_edit flag
    ' Caller can execute SR if there are no errors
    ShowEditSR = SR.Res_result
End Function
```

C/C++

```

C/C++ //*****
// Use this procedure to show the Edit Service Request dialog
//*****
UINT ShowEditSR (LPServiceRequest pSR)
{
    // Caller sets up Service Request
    pSR->operation = (Ops) (pSR->operation | EDIT);
    // DriverLINX automatically removes the EDIT flag
    // Caller can execute SR if there are no errors
    return DriverLINX(pSR)
}

```

DriverLINX source file examples

During the installation of the DriverLINX drivers, a wide selection of sample programs are installed. These programs are provided in several different programming languages, including VB, Delphi, and C++. Not all examples are provided in all programming languages.

The sample programs are designed to be very robust programs. Through the use of the DriverLINX Logical Device Descriptor, or LDD, these sample programs will function on any supported data acquisition board. The LDD control contains information regarding the capabilities of the data acquisition boards installed on the computer. A DriverLINX application can query the LDD to retrieve the hardware capabilities of the board. The use of the LDD is not covered in this tutorial manual. For information regarding the LDD, refer to the *Analog I/O Programming Guide*.

When learning to program DriverLINX the sample programs are generally not a good place to start. These samples are written from a very high level and assume a strong background in programming. The examples also use numerous function calls which can hide what is happening when the example executes. In addition, the extensive use of the LDD further obscures what is really happening.

The sample programs are separated according to the subsystem to which they apply. Some examples illustrate the integration of more than one subsystem. The samples provided may change with each version of DriverLINX. By default, the DriverLINX source code examples are installed in the C:\DrvLINX4\Source folder.

Analog input subsystem

AISingle — Demonstrates how to acquire a single sample of analog input data.

AIScan — Demonstrates how to acquire a single scan of analog input data from a list of analog input channels.

AIBuffer — Demonstrates how to acquire a single buffer of analog input data from an analog input channel.

AIMultch — Demonstrates how to acquire a single buffer of analog input data that contains data from multiple analog input channels. Note that this example demonstrates how to implement a channel gain list as opposed to a start-stop list.

AINonstp — Demonstrates how to acquire a single channel of analog input data continuously by using multiple buffers.

AIDigTrg — Similar to AIBuffer except that it demonstrates how to implement a digital start event.

AIDStpTg — Similar to AIBuffer except that it demonstrates how to implement a digital stop event.

AIBurst — Demonstrates how to acquire a single buffer of analog input data that contains data from multiple analog input channels. Note that this program demonstrates how to implement a burst-mode timing event.

AIATrig — Similar to AIBuffer except that it demonstrates how to implement an analog start event. Note that very few boards support this option. Consult your hardware manual for more information concerning the implementation of this feature.

AIStpTrg — Similar to AIBuffer except that it demonstrates how to implement an analog stop event. Note that very few boards support this option. Consult your hardware manual for more information concerning the implementation of this feature.

AISstream — Demonstrates how to continuously stream analog input data to disk.

TCGains — Demonstrates how to set up specialized gain codes that set the type, engineering units and other properties of a DAS-TC/B channel.

Analog output subsystem

AOSingle — Demonstrates how to write a single sample to an analog output.

AOBuffer — Demonstrates how to write a single buffer of analog data to an analog output channel.

AOMultch — Demonstrates how to write a single buffer of analog data to multiple analog output channels.

AONonstp — Demonstrates how to write continuously to a single analog output channel by using multiple buffers.

AODigTrg — Similar to AOBuffer except it demonstrates how to implement a digital start event.

Analog I/O subsystems

AISyncAO — Demonstrates how to synchronize acquiring analog input while outputting analog output. This task is accomplished by the analog output subsystem using the clock source as the analog input subsystem.

Digital input subsystem

DISingle — Demonstrates how to acquire a single sample of digital input data.

DIBuffer — Demonstrates how to acquire a single buffer of digital input data from a digital input channel.

DISysclk — Similar to DIBuffer except it uses the system clock as the pacer clock for acquiring a buffer of digital input data from the digital input subsystem. Note that not all boards support this functionality.

DIMultch — Demonstrates how to acquire a single buffer of digital input data that contains data from multiple digital input channels.

DINonstp — Demonstrates how to acquire a single channel of digital input data continuously by using multiple buffers.

DISimult — Demonstrates how to acquire a single buffer of data from multiple channels simultaneously by using your board's digital input subsystem.

DIExtend — Demonstrates how to acquire a single digital sample of data from your board's digital input subsystem using extended I/O addressing. This allows the user to isolate channels into sizes other than their native size.

Digital output subsystem

DOSingle — Demonstrates how to write a single sample to a digital output.

DOBuffer — Demonstrates how to write a single buffer of digital data to a digital output channel.

DOSysclk — Similar to DOBuffer except it uses the system clock as the pacer clock for writing a buffer of digital output data from the digital output subsystem. Note that not all boards support this functionality.

DOMultch — Demonstrates how to write a single buffer of digital data to multiple digital output channels.

DONonstp — Demonstrates how to write continuously to a single digital output channel by using multiple buffers.

DOSimult — Demonstrates how to write a single buffer of data to multiple channels simultaneously by using your board's digital output subsystem.

DOExtend — Demonstrates how to write a single digital sample of data to your board's digital output subsystem using extended IO addressing. This allows the user to isolate channels into sizes other than their native size.

Digital input/output subsystems

DIOCfg — Demonstrates how to configure a digital channel as either a digital input channel or a digital output channel. Note that this feature is not available to all boards.

Counter/timer subsystem

EventCtr — Demonstrates event counting of a single input signal using two counter/timer channels. This program has only been developed for VB. Note that not all boards support this functionality.

PerMeas — Demonstrates both period and pulse width measurement using two counter/timer channels. This program has only been developed for VB. Note that not all boards support this functionality.

FreqGen — Demonstrates rate, square-wave, and variable duty cycle frequency generation. This program has only been developed for VB. Note that not all boards support this functionality.

FreqMeas — Demonstrates frequency measurement using two or three counter/timer channels. Requires an external connection from the interval timer channel's output to the cycle counter channel's gate input. This program has only been developed for VB. Note that not all boards support this functionality.

5 Common DriverLINX Tasks

Introduction

When creating a DriverLINX application, there are several tasks that need to be performed regardless of the purpose of the application. These tasks include opening and closing the DriverLINX driver and initializing devices and subsystems.

Basic sequence of operations

All applications must follow the same basic sequence of operation to perform data acquisition with DriverLINX. The following is the basic sequence of operations required to perform a data acquisition task.

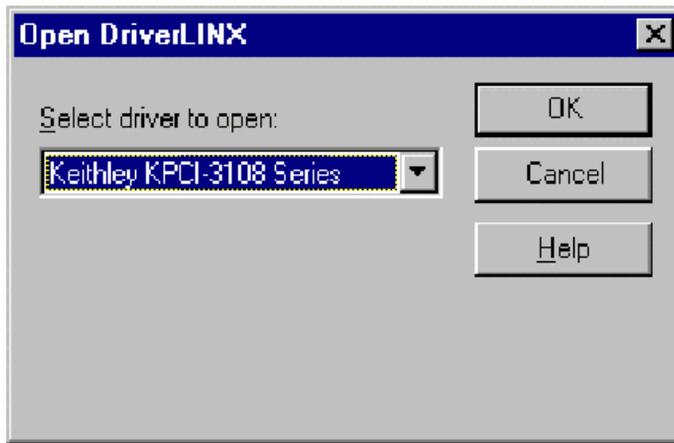
- Open the DriverLINX driver
- Initialize the device
- Perform the data acquisition task
- Close the DriverLINX driver

Complex applications may use additional steps, such as initializing subsystems, or may perform some of these steps multiple times. The remainder of this section discusses the common basic steps, such as opening and closing the DriverLINX driver, and initialization. [Section 6, Analog and Digital I/O Programming](#), discusses the steps involved in performing the data acquisition task itself.

Opening the DriverLINX driver

Before an application can communicate with a data acquisition board, it must specify the driver to be used. If a driver is not specified, or if the specified driver is invalid, the Open DriverLINX dialog shown in [Figure 5-1](#) will appear at run-time. The user may then pick a driver from a list of installed drivers.

Figure 5-1
Open DriverLINX dialog



Visual Basic

- VB** Specifying the driver when using Visual Basic requires specifying the name of the driver. A separate instance of the control should be created for each driver opened. When calling functions, the control will automatically select the appropriate driver.

```
DriverLINXSR_object.Reg_DLL_name = "driver"
```

C/C++

C/C++ Use the following with driver name to avoid the OpenDriverLINX dialog.

```
// For KPCI-3101/2/3/4 or KPCI-3110/3116, driver name is kpci3100
// For KPCI-3107/8, driver name is kpci3108
// For KPCI-3160, driver name is kpci3160
// For PCI PIO products, driver name is kpcipio
// For ISA PIO products, driver name is kmbpio

HINSTANCE m_driverInstance;
m_driverInstance = OpenDriverLINX(m_hWnd, "driver");

m_pSR = (DL_ServiceRequest*) new DL_ServiceRequest;
//allocate memory for the service request

memset(m_pSR,0,sizeof(DL_ServiceRequest));
//reset the members of the service request to defaults

DL_SetServiceRequestSize(*m_pSR);
```

Initializing a device

Initializing a device is a way of resetting the device and all subsystems to a known state. This should be done immediately after opening the DriverLINX driver.

Due to the multi-tasking and multi-device nature of DriverLINX, the exact outcome of initializing a device can vary. In the simplest sense, where a single process is using a single device, initializing that device will cancel all active Service Requests and perform a software reset of all subsystems. All outputs will be set to zero if this was the first initialization after loading the driver, or to the last known value otherwise.

Performing an initialization when multiple processes are using the same device will have one of the following effects:

- If another process is executing a Service Request on the device, DriverLINX performs all portions of the initialization that do not interfere with the other process. DriverLINX then returns a Device Busy error.
- If multiple processes are sharing the device, DriverLINX will not perform the initialization.

Visual Basic

VB The following code example demonstrates initializing the device.

```
Function InitDriverLINXDevice(SR As DriverLINXSR, ByVal Device As Integer)
    As integer
    With SR
        .Req_device = Device
        ' Device = integer number assigned in DriverLINX configuration
        .Req_subsystem = DL_DEVICE
        .Req_mode = DL_OTHER
        .Req_op = DL_INITIALIZE
        ' No events, buffers, channels needed
        .Evt_Tim_type = DL_NULLEVENT
        .Evt_Str_type = DL_NULLEVENT
        .Evt_Stp_type = DL_NULLEVENT
        .Sel_buf_N = 0
        .Sel_chan_N = 0
        .Refresh
        InitDriverLINXDevice = .Res_result
    End With
End Function
```

C/C++

C/C++ The following code example demonstrates initializing the device.

```
m_pSR->device=m_LogicalDevice;
// Device = integer number assigned in DriverLINX configuration
m_pSR->operation=INITIALIZE;
m_pSR->mode=OTHER;
m_pSR->subsystem=DEVICE;
m_pSR->hWnd=m_hWnd;
DriverLINX(m_pSR);
showMessage(m_pSR); //Show DriverLINX errors, if any
```

Initializing a subsystem

Initializing a subsystem resets the subsystem to a known state. This procedure is not normally needed, since DriverLINX performs an initialization of all subsystems when the device itself is initialized. Initializing a subsystem may be useful if another process is using another subsystem on the same data acquisition board. When a subsystem is initialized, all Service Requests being processed by the process that called the initialization are canceled.

Visual Basic

VB The following code example demonstrates initializing the subsystem.

```
Function InitSubsystem(SR As DriverLINXSR, ByVal Device As Integer,
    Subsystem As Integer) As integer
    With SR
        .Req_device = Device
        .Req_subsystem = Subsystem
        .Req_mode = DL_OTHER
        .Req_op = DL_INITIALIZE
        ' No events, buffers, channels needed
        .Evt_Tim_type = DL_NULLEVENT
        .Evt_Str_type = DL_NULLEVENT
        .Evt_Stp_type = DL_NULLEVENT
        .Sel_buf_N = 0
        .Sel_chan_N = 0
        .Refresh
        InitSubsystem = .Res_result
    End With
End Function
```

C/C++

C/C++ The following code example demonstrates initializing the subsystem.

```
/******
// Use this procedure to initialize a subsystem
//*****

UINT InitSubsystem (LPServiceRequest pSR, UINT Device, Subsystems subsystem)
{
```

```

memset(pSR, 0, sizeof(DL_SERVICEREQUEST));
DL_SetServiceRequestSize(*pSR);

pSR->hWnd = m_hWnd;
pSR->device = Device;
pSR->subsystem = Subsystem;
pSR->mode = OTHER;
pSR->operation = INITIALIZE;

return DriverLINX(pSR);
}

```

Closing the DriverLINX driver

When an application is done using a DriverLINX driver, the driver should be closed. Closing a driver stops all active tasks using the driver and removes the driver from memory.

Visual Basic

VB Closing the DriverLINX driver is accomplished by setting the Req_DLL_name property to a zero-length string. The following sample code demonstrates closing the driver.

```
DriverLINX_object.Req_DLL_name = ""
```

C/C++

C/C++ The following code can be used to close DriverLINX and free up the memory used by the Service Request.

```

if(m_DriverInstance != NULL)
//Close the DriverLINX driver, if it is open
{
    CloseDriverLINX(m_DriverInstance);
    m_DriverInstance=NULL;
}
if(m_pSR != NULL) //Free the memory used by the service request
{
    delete m_pSR;
    m_pSR=NULL;
}

```

Using more than one subsystem on a board

Using multiple subsystems on a single data acquisition board is a simple matter of submitting a Service Request with a new value for the Req_subsystem or subsystem property.

To create more robust code, the application can be created to allow for the easy transfer of a subsystem from one data acquisition board to another. For example, an application uses both a digital input and an analog output on the same board. It is later decided to add another data acquisition board and move the digital input to the new board, leaving the analog output on the old board. If the code has been properly set up, moving the digital input to the new board requires only a few minor changes.

To accomplish this, the application is created using a separate control or Service Request structure for each subsystem used. If the subsystem is later moved to a different board, the application only needs to point the appropriate control or structure to the new hardware. This is done by opening a new driver. When programming with C/C++, make sure to include the necessary SelectDriverLINX function calls.

Using more than one data acquisition board

The multi-tasking nature of DriverLINX allows an application to access more than one data acquisition board at a time. This allows the use of boards with different capabilities, or using additional boards with the same capabilities to increase the number of available channels. Regardless of whether the multiple data acquisition boards are of the same or different models, the multiple boards are addressed in the same way. Additional copies of the control or driver must be opened, and the additional boards must be initialized.

Visual Basic

- VB** When using multiple boards with Visual Basic, simply open additional instances of the control with new names. The additional boards are controlled exactly as the first. When programming Service Requests, make sure to use the appropriate control name. DriverLINX will automatically execute the Service Request on the appropriate board based on the control used.

C/C++

C/C++ Using multiple data acquisition boards with the C/C++ programming language is slightly more complex than with Visual Basic. As each copy of the DriverLINX driver loads, it returns an instance handle. This instance handle is used to select which copy of the driver will receive the Service Requests from the application.

DriverLINX automatically sends all function calls to the last driver that was used. This is ideal for configurations using a single data acquisition board, as the driver does not need to be repeatedly specified. When using multiple data acquisition boards, however, multiple drivers will be loaded at the same time. DriverLINX needs to be told when to send new commands to a different driver. To select the driver, and thereby the device to which the function call is sent, use the SelectDriverLINX function. Once a SelectDriverLINX function call is used, all further functions calls will be sent to that driver until another SelectDriverLINX function call is used.

```
HINSTANCE DLLAPI SelectDriverLINX (const HINSTANCE hDLL);
```

This code segment demonstrates opening two drivers, and selecting one of them.

```
HINSTANCE m_drv1;  
HINSTANCE m_drv2;  
  
m_drv1 = OpenDriverLINX(m_hWnd, "kpci3108");  
m_drv2 = OpenDriverLINX(m_hWnd, "kpcipio");  
SelectDriverLINX(m_drv1);
```

If the SelectDriverLINX function call is successful, it returns the instance handle of the selected driver. If it fails, it returns zero.

C/C++ special considerations

When using C/C++ to create DriverLINX applications, the following special considerations need to be observed.

Variable declarations and include files

C/C++ The following sample code shows the necessary include and variable declarations needed. For simple dialog based implementations, these declarations are found in the dialog's header file.

```
#include "c:\drvlinx4\dlapi\drvlinx.h"
#include "c:\drvlinx4\dlapi\dlcodes.h"
private:
short m_LogicalDevice;
void showMessage(DL_ServiceRequest *SR);
// function prototype for error check
HINSTANCE m_DriverInstance; // Used with open, select, close DriverLINX
                               functions
DL_ServiceRequest * m_pSR; // Declare the SR structure
void clearBuffers (DL_ServiceRequest *SR);
```

NOTE *You must also add the Drvlnx32.lib to the project. This library file is located in the c:\drvlinx4\dlapi folder.*

Clearing buffers

C/C++ The following code can be used to clear the buffers.

```
void CmyDlgClassDlg::clearBuffers (DL_ServiceRequest *SR)
{
    if (SR!=NULL)
    { if (SR->lpBuffers!=NULL)
      {
          // Expand as necessary, to clear more than 1 buffer
          if (SR->lpBuffers->BufferAddr [0] !=NULL)
          {
              BuffFree (SR->lpBuffers->BufferAddr [0] );
              SR->lpBuffers->BufferAddr [0] =NULL;
          }
          delete (SR->lpBuffers);
          SR->lpBuffers=NULL;
      }
    }
}
```

Error Check

C/C++ The code below shows how to use the messagebox operation of DriverLINX. If no error condition is present, then no message box will result. Other methods of error checking are described in [Section 9, Troubleshooting](#).

```
void CmyDlgClassDlg::showMessage(DL_ServiceRequest *SR)
{
    SR->operation=MESSAGEBOX;
    DriverLINX(SR);
    return;
}
```

Allocate Buffers

C/C++ The code below shows the procedure used to allocate buffers.

```
int m_buffers=3;
int m_samples=100;

//make a buffer list for three buffers
m_pSR->lpBuffers=(DL_BUFFERLIST*) new BYTE[DL_BufferListBytes
(m_buffers)];
m_pSR->lpBuffers->nBuffers=m_buffers; //Set number of buffers = 3
m_pSR->lpBuffers->bufferSize=Samples2Bytes(m_device,AI,0,m_samples);
//Set size of buffers in bytes
m_pSR->lpBuffers->BufferAddr[0]=BufAlloc(GBUF_INT,m_pSR->lpBuffers->
bufferSize);//allocate buffer 0
m_pSR->lpBuffers->BufferAddr[1]=BufAlloc(GBUF_INT,m_pSR->lpBuffers->
bufferSize);//allocate buffer 1
m_pSR->lpBuffers->BufferAddr[2]=BufAlloc(GBUF_INT,m_pSR->lpBuffers->
bufferSize);//allocate buffer 2
```

Channel Gain List

C/C++ The Service Request structure has four members for specification of the start channel, the gain to be used with this start channel, the stop channel, and the gain to be used with all channels between the start channel and the stop channel.

If supported by the hardware, a channel gain list can be used instead. Use of channel gain list permits the channels to be sampled in any order, even repetitively, and permits the gain to be specified on a channel by channel basis. The maximum length of the channel/gain list is dictated by the hardware specification. For the KPCI-3108, the channel gain list can contain as many as 256 entries.

When using a channel gain list, memory for the list must be allocated. The code below demonstrates the process.

```
int m_channels=4; //Must be 3 or higher to use channel/gain list

m_pSR->channels.nChannels=m_channels;
m_pSR->channels.chanGainList=new CHANGAIN[m_channels]; //allocate
channel gain list
// now populate the list with channels and gains
for(i=0;i<m_channels;i++)
{
    m_pSR->channels.chanGainList[i].channel=i;
    m_pSR->channels.chanGainList[i].gainOrRange=Gain2Code
    (m_logicalDevice,AI,-1.0);
}
```

Clearing a Channel/Gain List

C/C++ The following code can be used to clear a channel/gain list.

```
if(m_pSR->channels.chanGainList!=NULL)
{
    delete [] m_pSR->channels.chanGainList;
    m_pSR->channels.chanGainList=NULL;
}
```

6 Analog and Digital I/O Programming

Introduction

This section provides a series of progressively more complex programming samples. Each sample introduces additional techniques that build upon the techniques presented in previous samples. For this reason, it is important that you review each sample in order.

***NOTE** The examples presented in this tutorial assume the driver has been opened and the board initialized as discussed in [Section 5](#). The examples provided are brief code segments, not stand alone programs.*

Each programming sample is organized into four parts. The first part provides a brief description of the sample program. The second part provides a list of the key concepts presented in that sample. Each concept is described in detail. If applicable, references are made to the appropriate DriverLINX manual for more information. The third and fourth parts of the sample contain sample program code for Visual Basic and C/C++.

Before moving on to the samples, review the following information regarding the basic steps for a data acquisition task, and basic information on Windows messaging.

Basic steps for data acquisition

After the DriverLINX driver has been opened and the device initialized, the data acquisition task itself can be performed. Opening the driver and initializing the device are covered in [Section 5, Common DriverLINX Tasks](#). This section describes the steps involved in performing the data acquisition task.

The following is a list and basic description of the basic steps required for a data acquisition task.

- **Get hardware information**

An application can use the Logical Device Descriptor, LDD, to extract specific information about hardware capabilities at run-time. This advanced feature can aid in creating device-independent, portable applications. However, due to the complexity required when using the LDD, its use is not covered in this tutorial. For information on using the LDD, refer to Understanding the Logical Device Descriptor in the *Analog I/O Programming Guide* and Chapter 7 Logical Device Descriptors in the *DriverLINX Technical Reference Manual*.

- **Display user interface**

The user interface allows the user to input information regarding the data acquisition task. This information can be anything from specifying the task itself, to specifying how much data should be acquired and over what period of time. The sample programs provided here do not include a user interface.
- **Set up the Service Request**

When setting up the Service Request, data gathered from the user interface is combined with known information about the hardware. A Service Request is generated to actually perform the data acquisition task. All details of the Service Request do not need to be entered.
- **Pre-load buffers for output tasks**

If the data acquisition task involves output, the data buffers must be pre-loaded. How many buffers need to be preloaded depends on the buffering method used and the buffer notify flag. If linear buffering is used with a terminal count stop event, then all buffers may be pre-loaded. If the buffer notify flag is used, then only two buffers need to be pre-loaded. When the first buffer is empty, the second buffer will be used, and the buffer notify flag is set. This allows another buffer to be loaded.
- **Start the task**

When using the ActiveX control, the task is started by using the Refresh method on the control. To start a task using C/C++, call the DriverLINX function.
- **Monitor the task**

A running task can be monitored one of three ways. The first and easiest way is to wait for DriverLINX to return a ServiceDone event. When DriverLINX finishes processing the last data buffer, the task is terminated and a ServiceDone event is sent. The second way to monitor a task is to monitor for a BufferFilled event. As DriverLINX finishes processing each buffer, the Buffer Filled event is sent. The third way to monitor the task is to submit a Status Service Request. This procedure is valid only for background tasks. When a Status service Request is submitted, the data returned is dependent on the type of task being performed. For buffered, interrupt, or DMA I/O tasks DriverLINX returns the number of the buffer being processed and the position of the next sample to transfer. For counter/timer tasks, DriverLINX returns the current count.

Windows messaging

DriverLINX uses the Windows messaging system to inform the application submitting a Service Request of an event. When DriverLINX needs to communicate to the application, DriverLINX submits a Windows message to the applications message queue. A basic understanding of Windows messaging will allow you to better use the message and event functions of DriverLINX and help prevent some possible problems.

There are four levels of messages in Windows messaging. The highest level of messages have priority over all other types of messages. When this type of message is sent, the sending program relinquishes control to the receiver of the message. DriverLINX does not use this type of message. However, this is an example of how Windows multi-tasking can affect data acquisition tasks.

The second level of messages is the type of message used by DriverLINX for all events, such as buffer filled events. These second level messages are handled on a first in, first out basis, and do not transfer control. These messages are only processed when no first level messages are present. As you can see, DriverLINX messages may be delayed by first level messages generated by other programs.

The third level of messages is used for mouse, keyboard, serial, and other such messages. All keypresses and mouse clicks are processed at this level. These messages are only processed when no first and second level messages are present.

The fourth level of messages are used by Windows for events such as screen redraws and signal timer tics. These messages are processed only when no other messages of any kind are present. In certain circumstances, such as when many messages of a higher priority are present, these messages may not be processed at all.

It is important to understand the interaction between the second and third level messages. DriverLINX generates all of its events as second level messages. These are processed before all third level messages, such as keystrokes and mouse clicks. Due to this fact, it is possible for a program causing a high volume of events to effectively lock out the user from making any input. This is especially significant in a task that depends on user input to stop. It may be effectively impossible to stop such a runaway task short of resetting the computer.

A common place for this to occur is during buffering of high speed acquisition. If the buffer is set too small, it can fill very rapidly. When the first buffer is full, DriverLINX will send a BufferFilled event and begin filling the second buffer. If the application can not process the buffer fast enough, the second buffer will fill too soon. This will

cause another BufferFilled event before the application is ready for it. At the least, this will corrupt the data in the buffer currently being processed. Depending on the buffer size and the rate of data acquisition, the task may effectively flood the system with messages.

Another situation in which Windows messaging can cause problems with data acquisition tasks is for tasks which depend on the Windows timer. Due to the fact that timer ticks are a fourth level message, Windows can only process timer ticks when no other messages are present. Any event that arrives after a timer event but before the application returns to an idle state, will arbitrarily and indefinitely postpone status-polling loops that depend on Windows timer events.

Using events

Events tell DriverLINX when to start or stop a task, or how to pace a task. DriverLINX supports many different events. Events can also be used to synchronize tasks between multiple channels, and even multiple devices. The events supported by a specific data acquisition board are discussed that board's *Using DriverLINX with your Hardware* manual.

Software command events

Software command events are the simplest trigger available in DriverLINX. Software triggered events wait for the application to tell DriverLINX when to start or stop the task. Setting a software command start events tells DriverLINX to set the task up to start immediately when the Service Request is submitted. Setting the stop event to software command tells DriverLINX to repeat the task until the application submits a stop event.

The following example shows a Service Request set up to start on a software command.

Visual Basic

VB

```
With DriverLINXSR1
    .Req_subsystem = DL_DI
    .Req_mode = DL_POLLED
    .Req_op = DL_START
    .Evt_Str_type = DL_COMMAND ' Start on command
    .Sel_chan_format = DL_tNATIVE
    .Sel_chan_N = 1
    .Sel_chan_start = 0
    .Sel_chan_startGainCode = 0
    .Sel_buf_N = 0
    .Refresh
End With
```

C/C++

C/C++

```
memset(m_pSR, 0, sizeof(DL_ServiceRequest));
DL_SetServiceRequestSize(*m_pSR);
m_pSR->hWnd=m_hWnd;
m_pSR->device=m_LogicalDevice;
m_pSR->operation=START;
m_pSR->subsystem=DI;
m_pSR->mode=POLLED;
m_pSR->start.typeEvent=COMMAND; //Start on command
m_pSR->channels.nChannels=1;
m_pSR->channels.chanGain[0].channel=0;
m_pSR->status.typeStatus=IOVALUE;
DriverLINX(m_pSR);
```

Terminal count events

Terminal count stop events set up the task to run a fixed number of times. These events can be used to process a task until all buffers have been processed once, or until a maximum number of counts has been reached.

The following examples show a Service Request set up to use a terminal count stop event.

Visual Basic

VB

```
With DriverLINXSR1
    .Req_subsystem = DL_DI
    .Req_mode = DL_POLLED
    .Req_op = DL_START
    .Evt_Str_type = DL_COMMAND
    .Evt_Stp_type = DL_TCEVENT      ' Stop on terminal count
    .Evt_Tim_type = DL_NULLEVENT
End With
```

C/C++

C/C++

```
memset(m_pSR, 0, sizeof(DL_ServiceRequest));
DL_SetServiceRequestSize(*m_pSR);
m_pSR->hWnd=m_hWnd;
m_pSR->device=m_logicalDevice;
m_pSR->operation=START;
m_pSR->mode=POLLED;
m_pSR->subsystem=DI;
m_pSR->timing.typeEvent=NULLEVENT;
m_pSR->start.typeEvent=COMMAND;
m_pSR->stop.typeEvent=TCEVENT;      //Stop on terminal count
```

Rate events

Rate events are used as timing events to pace a data acquisition input or output task. Rate events are also used to set up counter/timer measurements or a pulse output Service Request. Rate events can be simple or complex depending on what is being accomplished. Some of the tasks that can be accomplished with rate events include the following:

- Rate and square wave generators
- Variable duty cycle generators
- Burst generators
- Frequency dividers for external clocking inputs
- Frequency and time interval measurement
- Event counters

The following samples show a Service Request set up to use a simple rate generator to pace an analog task.

Visual Basic

VB

```
With DriverLINXSr1
    .Req_subsystem = DL_AO
    .Req_mode = DL_DMA
    .Req_op = DL_START
    .Evt_Str_type = DL_COMMAND
    .Evt_Stp_type = DL_TCEVENT

    .Evt_Tim_type = DL_RATEEVENT
    .Evt_Tim_rateChannel = DL_DEFAULTTIMER ' Default of requested subsystem
    .Evt_Tim_rateMode = DL_RATEGEN
    .Evt_Tim_rateClock = DL_INTERNAL1
    .Evt_Tim_rateGate = DL_DISABLED
    .Evt_Tim_ratePeriod = .DLSecs2Tics(DL_DEFAULTTIMER, 1 / 100)
End With
```

C/C++

```
C/C++ m_pSR->operation=START;
        m_pSR->subsystem=AO;
        m_pSR->mode=DMA;
```

```
m_pSR->start.typeEvent=COMMAND;
m_pSR->stop.typeEvent=TCEVENT;
m_pSR->timing.typeEvent=RATEEVENT;
m_pSR->timing.u.rateEvent.channel=DEFAULTTIMER;
m_pSR->timing.u.rateEvent.mode=RATEGEN;
m_pSR->timing.u.rateEvent.clock=INTERNAL1;
m_pSR->timing.u.rateEvent.gate=DISABLED;
m_pSR->timing.u.rateEvent.period=
    Sec2Tics(m_device,AI,INTERNAL1,0.01f);
m_pSR->timing.u.rateEvent.pulses=0;
```

Analog events

Analog events are used to monitor an analog input channel and signal when a certain condition is met. This type of event is very configurable, and can be complex. Analog events are used as start, stop, and timing events. The following are some of the ways an analog event can be configured:

- Positive or negative level trigger
- Positive or negative edge trigger
- Inside or outside both thresholds
- Positive or negative band crossing

These different analog event methods can be used to perform many tasks. For example, a complete cycle of a periodic wave can be captured by using start and stop events that detect a zero crossing. Over- and under-range alarms can be created by using background tasks and waiting for a completion message.

The sample code fragments below shows the use of an analog input event to stop a task.

Visual Basic

VB

```

With DriverLINXSR1
.Req_subsystem = DL_AI
.Req_mode = DL_DMA
.Req_op = DL_START
.Evt_Str_type = DL_COMMAND

.Evt_Stp_type = DL_AIEVENT ' Set to stop on AI event
.Evt_Stp_aiChannel = 0 ' Set AI channel
.Evt_Stp_aiGainCode = .DLGain2Code(-1)

Dim threshold as long
threshold = .DLVolts2Code(2.0)
' the method returns a LONG...2 volts passed in
If threshold > 32767 Then ' need to cast as integer
    threshold = threshold - 65535
End If

.Evt_Stp_aiLowerThreshold = threshold ' This property is an integer
.Evt_Stp_aiUpperThreshold = threshold
.Evt_Stp_aiSlope = AnaTrgNegInside ' Trigger on falling edge
End With

```

C/C++

C/C++

```

m_pSR->operation= START;
m_pSR->subsystem=AI;
m_pSR->mode=DMA;
m_pSR->start.typeEvent= COMMAND;

m_pSR->stop.typeEvent=AIEVENT; //Stop on Analog Trigger
m_pSR->stop.u.aiEvent.channel = 0; //Use AI channel 0
m_pSR->stop.u.aiEvent.gain = Gain2Code(0,AI,-1);
m_pSR->stop.u.aiEvent.flags = AnaTrgNegInside; // negative slope
m_pSR->stop.u.aiEvent.upperThreshold = Volts2Code(m_logicalDevice,AI,2.0f);
m_pSR->stop.u.aiEvent.lowerThreshold = Volts2Code(m_logicalDevice,AI,2.0f);

```

Digital events

Digital events monitor a board's digital input channels and signal when a specified condition is met. The following four items need to be specified when setting a digital event:

- The channel to monitor
- A mask to specify which bits to monitor
- A pattern to which to compare the input
- Whether the input should match or not match the pattern

DriverLINX selects the input bits to monitor by combining the input with the mask using the logical AND operator. The result is compared to the pattern using either an “equals” or “not equals” comparison.

Digital events are data acquisition board hardware/firmware features such as a TTL Trigger. In some cases, the board generates an interrupt when the digital port receives new data or an external interrupt line sees an edge. Consult the Using DriverLINX With Your Hardware manual for your specific data acquisition board for more information.

Visual Basic

VB

```
With DriverLINXSR1
.Req_subsystem = DL_AI
.Req_mode = DL_DMA
.Req_op = DL_START

.Evt_Str_type = DL_DIEVENT      ' start on Digital Event
.Evt_Str_diChannel = DL_DI_EXTTRG ' Use external trigger
.Evt_Str_diMask = 1
.Evt_Str_diMatch = DL_NotEquals ' Input does not equal pattern
.Evt_Str_diPattern = 0 ' Pattern to match

End With
```

C/C++

C/C++

```
m_pSR->operation=START;
m_pSR->subsystem=AI;
m_pSR->mode=DMA;
```

```
m_pSR->start.typeEvent= DIEVENT; // start on Digital Event
m_pSR->start.u.diEvent.channel=DI_EXTTRG; // Use external trigger
m_pSR->start.u.diEvent.mask=1;
m_pSR->start.u.diEvent.match=0; // Input does not equal pattern
m_pSR->start.u.diEvent.pattern=0; // Pattern to match
```

Event delays

Analog and digital events can include a delay property. Delays are typically used to either capture or avoid transients associated with applying a signal. The delay property of the Service Request is specified as a number of samples. DriverLINX will either discard or buffer the specified number of samples.

For example, a start event with a positive delay causes DriverLINX to discard the specified number of samples before beginning to buffer data. Positive delays with stop events cause DriverLINX to continue the task after the stop event occurs until the number of samples specified by the delay have been buffered.

Negative delays can be used with a start event to capture data before the event occurs. Negative delays require hardware support. Before using a negative delay, consult the *Using DriverLINX With Your Hardware* manual to ensure that your hardware supports negative delays. When using a negative delay, data is continuously overwritten in the buffer until the start event occurs, then the data is no longer overwritten. Data points equaling the specified delay are kept.

To avoid the transient delay associated with applying a signal, a positive delay should be used with the start event. This will avoid the transient and capture the steady state readings. To capture the transient, a positive delay with a stop event can be used. Alternately, a negative delay can be used with a start event, if supported by hardware.

Visual Basic

VB

```
With DriverLINXSR1
.Req_subsystem = DL_AI
.Req_mode = DL_DMA
.Req_op = DL_START
.Evt_Str_type = DL_COMMAND
.Evt_Stp_type = DL_AIEVENT
.Evt_Stp_aiChannel = 0
.Evt_Stp_aiGainCode = .DLGain2Code(-1)
```

```

Dim threshold as long
threshold = .DLVolts2Code(2.0) ' 2 volt threshold
If threshold > 32767 Then
    threshold = threshold - 65535
End If

.Evt_Stp_aiLowerThreshold = threshold
.Evt_Stp_aiUpperThreshold = threshold
.Evt_Stp_aiSlope = AnaTrgNegInside
.Evt_Stp_delay = 100 ' take 100 samples after the trigger occurs
End With

```

C/C++

```

C/C++ m_pSR->operation= START;
m_pSR->subsystem=AI;
m_pSR->mode=DMA;
m_pSR->start.typeEvent= COMMAND;

m_pSR->stop.typeEvent=AIEVENT;
m_pSR->stop.u.aiEvent.channel = 0; //Use AI channel 0
m_pSR->stop.u.aiEvent.gain = Gain2Code(0,AI,-1);
m_pSR->stop.u.aiEvent.flags = AnaTrgNegInside; // negative slope
m_pSR->stop.u.aiEvent.upperThreshold = Volts2Code(m_logicalDevice,AI,2.0f);
m_pSR->stop.u.aiEvent.lowerThreshold = Volts2Code(m_logicalDevice,AI,2.0f);
m_pSR->stop.delay = 100;
// acquire 100 samples after the Trigger occurs

```

Reading a single digital value

Single sample operations are the simplest data acquisition task to perform using DriverLINX. DriverLINX will return the results of the task using the `Res_Sta_ioValue` property of the Service Request. No buffers need to be allocated or read. The sample presented below illustrates reading a single value from a digital input channel. The following key concepts are used:

- No buffers used — This example uses the `ioValue` property of the Service Request to read the single result value.
- Polled mode operation — This task uses polled mode operation. The task executes synchronously, or in the foreground, and returns control to the application

only after the task is complete. This means there is no danger of attempting to read the results too soon. However, the application cannot do anything else until the task is completed.

- Software command start event — This task starts immediately when the Service Request is submitted.
- Terminal count stop event — A terminal count stop event tells DriverLINX to stop the task as soon as the value is read. No separate stop event is required.
- Error checking — DriverLINX returns an error code with every Service Request using the Res_result property. If an error occurs, this value will be non-zero. The Visual Basic example performs error checking by performing a not-equals operation on the Res_result property. If Res_result <> 0, the message box function of DriverLINX is called to display the error. For more information on error messages, refer to [Section 9, Troubleshooting](#).

Visual Basic

VB

The following sample code illustrates a single value digital input task. This sample assumes the driver is open and the hardware initialized.

```
With DriverLINXSR1
.Req_subsystem = DL_DI
.Req_mode = DL_POLLED
.Req_op = DL_START
.Evt_Str_type = DL_COMMAND
.Evt_Stp_type = DL_TCEVENT
.Evt_Tim_type = DL_NULLEVENT
.Sel_chan_format = DL_tNATIVE
.Sel_chan_N = 1      ' one channel only
.Sel_chan_start = 0
.Sel_chan_startGainCode = 0
.Sel_buf_N = 0      ' No buffers
.Refresh
End With
If DriverLINXSR1.Res_result <> 0 Then
Label1.Caption = DriverLINXSR1.Message      ' Display error
Else
Label2.Caption = Str(DriverLINXSR1.Res_Sta_ioValue)      ' Display digital input
                                                         value
```

C/C++

C/C++ The following sample code illustrates a single value digital input task.

```
memset(m_pSR,0,sizeof(DL_ServiceRequest));
DL_SetServiceRequestSize(*m_pSR);
m_pSR->hWnd=m_hWnd;
m_pSR->device=m_LogicalDevice;
m_pSR->operation=START;
m_pSR->subsystem=DI;
m_pSR->mode=POLLED;
m_pSR->channels.nChannels=1;
m_pSR->channels.chanGain[0].channel=0;
m_pSR->status.typeStatus=IOVALUE;
DriverLINX(m_pSR); //Execute the service request
showMessage(m_pSR); //Call user defined error check function
m_value=m_pSR->status.u.ioValue; //set the value field the value read
UpdateData(FALSE); //Update the textbox, MFC function
```

NOTE *m_value is a variable assigned using ClassWizard to a text box control on the dialog (form). C++ does not permit direct interaction with controls on forms, eg, label.caption is not permitted. You must assign variables to these controls, and the code uses these assigned variable names.*

Reading a single analog value

Reading a single analog value is very similar to reading a single digital value. However, there are a few extra items that need to be considered. First, analog channels need to have a gain specified when reading values. DriverLINX requires that the gain specified be in the form of a hardware code. Second, the value returned by DriverLINX for an analog reading is a hardware code. This code is normally converted to an actual voltage value. DriverLINX provides methods to aid in converting to and from hardware codes for both gain and voltage.

This example demonstrates many of the same concepts used in the previous example. In addition, this examples illustrates the following key concepts:

- No buffers used — This example uses the ioValue property of the Service Request to read the single result value.

- Gain conversion to hardware codes — The method `DLGain2Code` converts a gain for an analog channel to the hardware code specific to the data acquisition board being used. The gain is passed to the method, and the hardware specific gain code is returned. Specifying a negative gain tells DriverLINX that the gain is bipolar. The `DLGain2Code` and other useful methods are described in detail in the *DriverLINX/VB Technical Reference Manual* in Section 9, Service Request Control Support Methods, and in the *DriverLINX Technical Reference Manual* in Section 9, Support Functions.
- Conversion from voltage codes to voltage values — The `DLCode2Volts` method converts the hardware specific voltage code returned by DriverLINX to a voltage value. This routine should only be used with hardware gain codes of zero, which corresponds to a bipolar gain of 1.

Visual Basic

VB

The following sample code demonstrates reading a single analog value.

```
With DriverLINXSR1
    '-----Request Group-----
    .Req_subsystem = DL_AI
    .Req_mode = DL_POLLED
    .Req_op = DL_START

    '-----Event Group-----
    .Evt_Tim_type = DL_NULLEVENT
    .Evt_Str_type = DL_COMMAND
    .Evt_Stp_type = DL_TCEVENT

    '-----Select Group-----
    .Sel_chan_format = DL_tNATIVE
    .Sel_chan_N = 1
    .Sel_chan_start = 0
    .Sel_chan_startGainCode = DriverLINXSR1.DLGain2Code(-1)

    .Sel_buf_N = 0
    '-----Result Group-----
    .Res_Sta_typeStatus = DL_IOVALUE 'default value not necessary but calls
    out it's existence; contrasts with later use for other use
End With
DriverLINXSR1.Refresh 'execute the acquisition
```

```
lblResult.Caption = Str(DriverLINXSR1.DLCode2Volts  
    (DriverLINXSR1.Res_Sta_ioValue))  
lblStatus.Caption = "Acquisition Complete"
```

C/C++

C/C++ The following sample code demonstrates reading a single analog value.

```
m_pSR->operation=START;  
m_pSR->mode=POLLED;  
m_pSR->subsystem=AI;  
m_pSR->channels.nChannels=1; //Only reading one channel  
m_pSR->channels.chanGain[0].channel=0; //Read channel 0  
    //Use bipolar unity gain for channel 0  
m_pSR->channels.chanGain[0].gainOrRange=Gain2Code(0,AI,-1.0);  
m_pSR->status.typeStatus=IOVALUE; //The status member wanted is the ioValue  
DriverLINX(m_pSR); //Execute the service request  
showMessage(m_pSR); //show any errors  
float volts;  
int result;  
result=Code2Volts(0,AI,m_pSR->status.u.ioValue,&volts); //Convert to volts  
m_reading.Format("%f",volts); //Display the voltage using the Edit box  
UpdateData(FALSE); //Update the dialog, MFC function
```

NOTE *m_reading is a variable assigned using ClassWizard to a text box control on the dialog (form). C++ does not permit direct interaction with controls on forms, eg, label.caption is not permitted. You must assign variables to these controls, and the code uses these assigned variable names.*

Reading a series of digital values

Reading a series of digital values is very similar to reading a single value. The most significant difference is the use of a buffer. When transferring only a single value, we used the ioValue property of the Service Request to transfer our single result from the task to the application. This technique will not work when transferring more than a single value. In this case, we must use a buffer.

The programming sample uses a single buffer to transfer the values of three digital input channels. Once the channels have been read, the buffer is transferred to a VB array. This programming sample demonstrates the following key concepts:

- Start/stop channel list — The sample code uses a channel start/stop list to read all channels between logical channels 0 and 2. The Service Request properties `Sel_chan_start` and `Sel_chan_stop` set the beginning and ending channels.
- Buffered data transfer — This sample code uses the `Sel_buf_N` and `Sel_buf_samples` properties to allocate memory to be used as buffers. The `Sel_buf_N` property tells DriverLINX how many buffers will be used. This sample code uses only one buffer. The `Sel_buf_samples` tell DriverLINX how many data points each buffer should be able to hold. In this case, 3 data points will be stored in each buffer. DriverLINX calculates the amount of memory needed to hold the specified number of data points, and automatically allocates the required memory.
- Data conversion and transfer to an array — After a task completes filling a buffer, the data must be transferred to a location where it can be used by the application. The data must also be converted to a usable format. The data acquisition format for digital channels is 8 bit, but DriverLINX uses 16 bit buffer elements. The `VBArryBufferXfer` method sorts the 16 bit buffer elements into their 8 bit port results and places them into a VB array.

Visual Basic

VB

The following sample code illustrates reading a series of digital channels.

```
With DriverLINXSR1
.Req_subsystem = DL_DI
.Req_mode = DL_POLLED
.Req_op = DL_START
.Evt_Str_type = DL_COMMAND
.Evt_Stp_type = DL_TCEVENT
.Evt_Tim_type = DL_NULLEVENT
.Sel_chan_format = DL_tNATIVE ' for digital channels, native is 8bit wide
.Sel_chan_N = 2
.Sel_chan_start = 0
.Sel_chan_stop = 2
.Sel_buf_N = 1
.Sel_buf_samples = 3
.Refresh
End With
```

```

Dim convert As Single
Dim digitalData(3) As Byte
convert = DriverLINXSRL.VBArrayBufferXfer(0, digitalData,
DL_BufferToVBArray)

Dim i As Integer
For i = 0 To 2
Debug.Print digitalData(i)
Next i

```

C/C++

C/C++ The following sample code illustrates reading a series of digital channels.

```

void CDIBufferDlg::OnStart ()
{
    memset(m_pSR,0,sizeof(DL_ServiceRequest));
    DL_SetServiceRequestSize(*m_pSR);
    m_pSR->hWnd=m_hWnd;
    m_pSR->device=m_logicalDevice;
    m_pSR->operation=START;
    m_pSR->mode=POLLED;
    m_pSR->subsystem=DI;
    m_pSR->timing.typeEvent=NULLEVENT;
    m_pSR->start.typeEvent=COMMAND;
    m_pSR->stop.typeEvent=TCEVENT;
    //Stop when the number of samples has been read

    m_pSR->channels.nChannels=2;
    m_pSR->channels.chanGain[0].channel=0;
    m_pSR->channels.chanGain[1].channel=2;
    m_pSR->channels.numberFormat=tNATIVE;
    m_pSR->lpBuffers=(DL_BUFFERLIST*)new
        BYTE[DL_BufferListBytes(1)];

    //Create a list of buffers; in this case, just one
    m_pSR->lpBuffers->nBuffers=1; //Only one buffer will be used
    m_pSR->lpBuffers->bufferSize=Samples2Bytes(0,DI,0,m_samples);
    // m-samples=3
    m_pSR->lpBuffers->BufferAddr[0]=BufAlloc(GBUF_INT,m_pSR->lpBuffers->
bufferSize);

```

```
//Allocate enough memory in the buffer for the number of samples
DriverLINX(m_pSR);
showMessage(m_pSR);
done();
}

void CDIBufferDlg::done()
{
    m_readlist.ResetContent(); //Clear the listbox to show the samples
    m_pData=(BYTE*)m_pSR->lpBuffers->BufferAddr[0];
    //Copy the contents of BufferAddr[0] to the local pointer,
    //and cast as an array of bytes
    //It is necessary to specify byte, because the channel is only 8
    // bits wide, but the buffer elements are 32 bits wide
    int index;
    CString str;
    for(index=0;index<m_samples;index++)
    {
        str.Format("%d",m_pData[index]);
        //Format the sample as a string to display it in the listbox
        m_readlist.AddString(str); //Add the current sample to the listbox
    }
    UpdateData(FALSE); //Update the listbox display
}
```

Writing a single digital value

Many data acquisition boards have configurable digital I/O ports. This allows the ports to be used for either input or output. By default, programmable digital I/O ports are configured as input ports. Before they can be used for digital output, they must be configured as output ports. This can be accomplished either through the DriverLINX Configuration Panel or the application. The DriverLINX Configuration Panel is available from the DriverLINX program group. This programming sample demonstrates the following key concepts:

- Reconfiguring digital ports with code — Reconfiguring ports with code inside the application allows the application to function properly regardless of the port configuration set before the application started. It is not necessary to manually configure the hardware before running the application.

- No buffers used — This example uses the `Res_Sta_ioValue` property of the Service Request to pass the output value to the Service Request. This is a simple way to pass a single data value to an output task without using buffers.
- Polled mode operation — This task uses polled mode operation. The task executes synchronously, or in the foreground, and returns control to the application only after the task is complete.
- No start event needed — When setting a single digital output value, no start event is needed. The `Evt_Str_Type` is set to `DL_NULLEVENT`.
- No stop event needed — This operation does not require a stop event. When the output has been set, the task ends. The `Evt_Stp_Type` is set to `DL_NULLEVENT`.

Visual Basic

VB

The following sample code illustrates configuring a digital channel to be used as an output, and outputting a single digital value. Note that the statement `SR_DO.Res_Sta_ioValue = doVal` assigns the result status of the Service Request to the variable `doVal`.

```
' code below will Configure Channel 0 for output
SR_DO.Req_subsystem = DL_DO ' the subsystem to assign the channel
SR_DO.Req_mode = DL_OTHER
SR_DO.Req_op = DL_CONFIGURE ' it is a configuration type operation
SR_DO.Evt_Tim_type = DL_DIOSETUP
SR_DO.Evt_Tim_dioChannel = 0 ' configure channel 0 or Port A as output
SR_DO.Evt_Tim_dioMode = DL_DIO_BASIC
SR_DO.Evt_Str_type = DL_NULLEVENT
SR_DO.Evt_Stp_type = DL_NULLEVENT
SR_DO.Sel_chan_N = 0
SR_DO.Refresh ' carry out the request
lblStatus.Caption = SR_DO.Message 'display result message

'code below will set up SR for single value DO
SR_DO.Req_subsystem = DL_DO
SR_DO.Req_mode = DL_POLLED
SR_DO.Req_op = DL_START
SR_DO.Evt_Str_type = DL_NULLEVENT
SR_DO.Evt_Stp_type = DL_NULLEVENT
SR_DO.Evt_Tim_type = DL_NULLEVENT
SR_DO.Sel_chan_format = DL_tNATIVE
```

```

SR_DO.Sel_chan_N = 1
SR_DO.Sel_chan_start = 0   'Channel 0 is already configured as an output
SR_DO.Sel_chan_startGainCode = 0
SR_DO.Sel_buf_samples = 1
SR_DO.Sel_buf_N = 0
SR_DO.Res_Sta_ioValue = doVal   'For an 8-bit port, doVal = 0 to 255
SR_DO.Refresh

```

C/C++

C/C++ The following sample code illustrates configuring a digital channel to be used as an output, and outputting a single digital value.

```

// code to dynamically configure a digital channel for output mode
// this example will configure digital channel 1 for output mode

m_pSR->hWnd=m_hWnd;
m_pSR->device=m_LogicalDevice;
m_pSR->operation=CONFIGURE;
m_pSR->subsystem=DO;
m_pSR->mode=OTHER;
m_pSR->timing.typeEvent = DIOSETUP;
m_pSR->timing.u.diSetup.channel = 1;
m_pSR->timing.u.diSetup.mode = DIO_BASIC;
DriverLINX(m_pSR); //Execute the service request
showMessage(m_pSR); //and show the result

// write value from text box to the DO channel
UpdateData(TRUE); //Get the current textbox values
memset(m_pSR,0,sizeof(DL_ServiceRequest));
DL_SetServiceRequestSize(*m_pSR);
m_pSR->hWnd=m_hWnd;
m_pSR->device=m_LogicalDevice;
m_pSR->operation=START;
m_pSR->subsystem=DO;
m_pSR->mode=POLLED;
m_pSR->channels.nChannels=1;
m_pSR->channels.chanGain[0].channel=1;
m_pSR->status.typeStatus=IOVALUE;
m_pSR->status.u.ioValue=m_DOValue; //Set the i/o value to the textbox
value
DriverLINX(m_pSR);
showMessage(m_pSR);

```

NOTE *m_DOvalue* is a variable assigned using *ClassWizard* to a text box control on the dialog (form). For an 8-bit digital channel, it can be a range of 0 to 255.

Writing a series of digital values

This programming sample builds upon some of the concepts presented in the previous example. This sample uses a subroutine to configure a series of digital I/O ports for output. An array is created containing the data to be written to the output ports, and a buffer is created to hold the data. The data is then transferred from the array to the buffer. Submitting the Service Request instructs DriverLINX to write the data in the buffer to the output ports.

This sample demonstrates the following key concepts:

- Reconfiguring digital ports with code — Similar to the previous example of writing a single digital value, this example also uses code to reconfigure the digital I/O ports for output. However, this time we have created a subroutine called from a loop to simplify reconfiguring multiple channels.
- Buffered operation — This example uses a small buffer to hold the data being passed to DriverLINX for output to the digital ports. The data is loaded into the buffer from a previously created array. The transfer process is accomplished with the use of the `VBAArrayBufferXfer` method. The last parameter passed to the method, `DL_VBAArrayToBuffer`, controls the direction of data transfer. In this case, the data is being transferred from a VB array to a DriverLINX buffer.

Visual Basic

VB

The following sample code illustrates writing a digital value to a series of digital output channels.

```
' subroutine to do channel configuration (uses generic name for the SR control):
Private Sub ConfigChan_for_Output(SR As Control, chan As Integer)
' code below will Configure a Channel for Output
SR.Req_subsystem = DL_DO ' the subsystem to assign the channel to
SR.Req_mode = DL_OTHER
SR.Req_op = DL_CONFIGURE ' it is a configuration type operation
SR.Evt_Tim_type = DL_DIOSETUP
```

```
SR.Evt_Tim_dioChannel = chan    ' configure the channel
SR.Evt_Tim_dioMode = DL_DIO_BASIC
SR.Evt_Str_type = DL_NULLEVENT
SR.Evt_Stp_type = DL_NULLEVENT
SR.Sel_chan_N = 0
SR.Refresh            ' carry out the request
Debug.Print SR.Message ' display result message
End Sub

' configure 3 channels for digital output
Dim i As Integer
For i = 0 To 2
  ConfigChan_for_Output DriverLINXSR1, i ' pass to subroutine which SR object
  to use and which channel
Next i

' setup DO Service Request for polled mode Output operation
With DriverLINXSR1
  .Req_subsystem = DL_DO
  .Req_mode = DL_POLLED
  .Req_op = DL_START
  .Evt_Str_type = DL_COMMAND
  .Evt_Stp_type = DL_TCEVENT
  .Evt_Tim_type = DL_NULLEVENT
  .Sel_chan_format = DL_tNATIVE
  .Sel_chan_N = 2
  .Sel_chan_start = 0
  .Sel_chan_stop = 2
  .Sel_buf_samples = 3
  .Sel_buf_N = 1
End With

' move contents of an array to a DriverLINX buffer
Dim digitalData(3) As Byte
digitalData(0) = 255 ' all bits on...channel 0 data
digitalData(1) = 0   ' all bits off...channel 1 data
digitalData(2) = 127 ' all but MSB on..channel 2 data

Dim convert As Single
convert = DriverLINXSR1.VBArrayBufferXfer(0, digitalData,
  DL_VBArrayToBuffer)
'now execute the task
```

```
DriverLINXSR1.Refresh
Debug.Print DriverLINXSR1.Message
```

C/C++

C/C++ The following sample code illustrates writing a digital value to a series of digital output channels.

```
//Assumes the channels are configured for output mode already
//per the code in the previous example

memset(m_pSR,0,sizeof(DL_ServiceRequest));
DL_SetServiceRequestSize(*m_pSR);
m_pSR->hWnd=m_hWnd;
m_pSR->device=m_logicalDevice;
m_pSR->operation=START;
m_pSR->mode=POLLED;
m_pSR->subsystem=DO;
m_pSR->timing.typeEvent=NULLEVENT;
m_pSR->start.typeEvent=COMMAND;
m_pSR->stop.typeEvent=TCEVENT;
m_pSR->channels.nChannels=2;
m_pSR->channels.chanGain[0].channel=0;
m_pSR->channels.chanGain[1].channel=2;
m_pSR->channels.numberFormat=tNATIVE;
m_pSR->lpBuffers=(DL_BUFFERLIST*)new BYTE[DL_BufferListBytes(1)];
//Create a list of buffers; in this case, just one
m_pSR->lpBuffers->nBuffers=1; //Only one buffer will be used
m_pSR->lpBuffers->bufferSize=Samples2Bytes(0,DO,0,m_samples);
// m_samples = 3
m_pSR->lpBuffers->BufferAddr[0]=BufAlloc(GBUF_INT,m_pSR->lpBuffers
->bufferSize);
//Allocate
// Now to load the buffer with 3 values

BYTE DigitalBuf[3];
DigitalBuf[0] = 255;
DigitalBuf[1] = 0;
DigitalBuf[2] = 127;
m_pSR->lpBuffers->BufferAddr[0] = &DigitalBuf; // point memory buffer at
our data array
DriverLINX(m_pSR);
showMessage(m_pSR);
```

Reading analog values on several channels

Reading multiple values in a single task requires specifying which channels to read, and the use of buffers to return the data from DriverLINX to the application. There are two ways to set up the Service Request to read multiple channels. The first way is to use a start/stop list, providing the logical number of the first and last channels. DriverLINX will perform the requested operation on all channels between the start and stop channels. The second way is to use a channel/gain list. A channel/gain list allows the selection on a non-contiguous set of channels, each with its own gain. Channel/gain lists may not be supported on all data acquisition boards. Consult the *Using DriverLINX With Your Hardware* manual for information on whether or not your board supports a channel gain list.

The following sample demonstrates using a start/stop list to read several analog inputs. While the procedure is relatively simple, the sample illustrates several good concepts for programming DriverLINX. The key concepts demonstrated in this sample include:

- **Buffered data transfer** — This sample code uses the `Sel_buf_N` and `Sel_buf_samples` properties to allocate memory to be used as buffers. The `Sel_buf_N` property tells DriverLINX how many buffers will be used. This sample code uses only one buffer. The `Sel_buf_samples` tell DriverLINX how many data points each buffer should be able to hold. In this case, 16 data points will be stored in each buffer. DriverLINX calculates the amount of memory needed to hold the specified number of data points, and automatically allocates the required memory.
- **Buffer notification** — The `Sel_buf_notify = DL_NOTIFY` property instructs DriverLINX to send a `BufferFilled` event when the last sample has been taken. If this task were being set up as a background task, this event could be used to notify the application that task processing was complete and the buffer is full. Since this task is set up as a polled mode task, the event is not needed. However, it is good programming practice with DriverLINX to set all properties of a Service Request to a known value to minimize possible interference from properties with an unknown value.
- **Start/stop channel list** — The sample code uses a channel start/stop list to read all channels between logical channels 0 and 15. The Service Request properties `Sel_chan_start` and `Sel_chan_stop` set the beginning and ending channels.

Sel_chan_startGainCode specifies the gain code for the first channel.

Sel_chan_stopGainCode sets the gain for all other channels.

- Gain code conversion — When specifying a channel gain in a Service Request, the hardware specific gain code must be used. This sample code uses a DriverLINX method named Gain2Code to convert a gain to the required hardware gain code. For more information on this and other methods provided by DriverLINX, refer to Chapter 9, Service Request Control Support Methods, in the *DriverLINX/VB Technical Reference Manual*, or Chapter 9, Support Function, in the *DriverLINX Technical Reference Manual*.
- Data conversion and transfer to an array — After a task completes filling a buffer, the data must be transferred to a location where it can be used by the application. The data must also be converted from the hardware-returned format to volts. The sample code uses the DriverLINX method VBArrayBufferConvert to perform both tasks at the same time. The data is read from the buffer, converted to volts, and loaded into the specified array.

Visual Basic

VB

The following example demonstrates reading an analog value from a series of analog input channels.

```
With DriverLINXSR1
  .Req_subsystem = DL_AI
  .Req_mode = DL_POLLED
  .Req_op = DL_START
  .Evt_Str_type = DL_COMMAND
  .Evt_Stp_type = DL_TCEVENT
  .Evt_Tim_type = DL_NULLEVENT
  .Sel_chan_format = DL_tNATIVE
  .Sel_chan_N = 2
  .Sel_chan_start = 0
  .Sel_chan_startGainCode = .DLGain2Code(-1)
  .Sel_chan_stop = 15
  .Sel_chan_stopGainCode = .DLGain2Code(-1)
  .Sel_buf_N = 1
  .Sel_buf_samples = 16
  .Sel_buf_notify = DL_NOTIFY
  .Refresh
End With
```

```
Dim convert As Single
Dim dataArray(16) As Single
convert = DriverLINXSRL.VBArrayBufferConvert(0, 0, 16, dataArray,
    DL_tSINGLE, 0, 0)
Debug.Print dataArray(0) 'Display first value to immediate window of VB
```

C/C++

C/C++ The following example demonstrates reading an analog value from a series of analog input channels.

```
m_pSR->operation=START; //Start the acquisition
m_pSR->subsystem=AI; //using the AI subsystem
m_pSR->mode=POLLED; //use polled mode
m_pSR->start.typeEvent=COMMAND; //Start on command
m_pSR->timing.typeEvent=NULLEVENT;
m_pSR->stop.typeEvent=TCEVENT;
m_pSR->channels.nChannels=2;
m_pSR->channels.chanGain[0].channel=0; //start on channel 0
m_pSR->channels.chanGain[0].gainOrRange=
    Gain2Code(m_logicalDevice,AI,-1.0); //Use bipolar unity gain
m_pSR->channels.chanGain[1].channel=7; // stop on channel 7
m_pSR->channels.chanGain[1].gainOrRange=
    Gain2Code(m_logicalDevice,AI,-1.0);
m_pSR->channels.numberFormat=tNATIVE;
    //use the native format (integer counts)
m_pSR->lpBuffers=(DL_BUFFERLIST*) new BYTE[DL_BufferListBytes(1)];
    //create a buffer list pointer for one buffer
m_pSR->lpBuffers->notify=NOTIFY; //enable the buffer filled message
m_pSR->lpBuffers->nBuffers=1; //use only one buffer
m_pSR->lpBuffers->bufferSize=Samples2Bytes
    (m_logicalDevice,AI,m_logicalChannel,m_samples); //set the size of the
    buffer (in bytes) to hold the number of samples
m_pSR->lpBuffers->BufferAddr[0]=BufAlloc(GBUF_INT,m_pSR->lpBuffers->
    bufferSize);
    //Allocate Buffer 0 based on the size we just specified

DriverLINX(m_pSR);
    //Execute the service request to start the acquisition
showMessage(m_pSR); //show any errors
done();
```

```

    }

void CAIbufferDlg::done()
{
float *readings;
readings = new float[m_samples];
    //make a temporary array to hold the converted readings
CString str;
//The CString class includes a format method to convert float to
//CString to display them in the listbox

int index;
/* The convert operation used here is taking advantage of the fact
that the logical device number and the subsystem (AI) have already
been set in the start request */
m_pSR->operation=CONVERT;
//Use the convert operation to convert the raw counts in the
//buffer to voltages
m_pSR->mode=OTHER;
    //Convert is not a polled, interrupt, or DMA operation
m_pSR->start.typeEvent=DATACONVERT;
    //Set the start type to convert the data
m_pSR->start.u.dataConvert.startIndex=0;
    //start at index 0 of the buffer
m_pSR->start.u.dataConvert.nSamples=m_samples;
    //set the number of samples to convert
m_pSR->start.u.dataConvert.numberFormat=tSINGLE;
    //convert the counts to tSINGLE (float)
m_pSR->start.u.dataConvert.scaling=0.0f; //no scaling will be used
m_pSR->start.u.dataConvert.offset=0.0f; //no offset will be applied
m_pSR->start.u.dataConvert.wBuffer=0;
//convert DriverLINX buffer 0, since that's the only one being used
m_pSR->start.u.dataConvert.lpBuffer=readings;
//put the converted readings in the temporary buffer
DriverLINX(m_pSR); //Execute the conversion
showMessage(m_pSR); //show any errors
m_readList.ResetContent(); //clear the listbox
for(index=0;index<m_samples;index++)
{
    str.Format("%f",readings[index]);
    //format the float reading into a string for each reading
    m_readList.AddString(str); //Add the string to the listbox
}
}

```

```
}  
UpdateData(FALSE); //Update the listbox display  
delete readings; //clear the temporary buffer so we don't have memory  
    leaks when we run it again  
}
```

Using messages in a background task

When performing a task in background mode, DriverLINX can provide messages to inform the application of the status of the task. These messages can provide information to the application such as when the task starts, when it stops, and when the buffer is full. Applications can act on these messages to begin other tasks, process buffers, or synchronize multiple tasks.

This sample task obtains one sample on each of 16 channels at a rate of 10 Hz. While this operation will take 1.6 seconds to complete, it is not recommended to submit the Service Request, wait the 1.6 seconds for the task to finish, and then process the buffer. Windows overhead issues and other factors may cause the task to not be completed in 1.6 seconds. Since easy communication from DriverLINX to the application is available, coordination between DriverLINX and the application is simple. This sample uses a buffer filled message to tell the application that DriverLINX has filled the requested buffer. The application then processes the buffer and displays the collected data.

The key concepts demonstrated in this sample include:

- **Background operation** — This task uses the Interrupt mode. This operating mode allows the task to be performed in the background. The application that submitted the Service Request continues to run in the foreground, and can perform other operations. The data acquisition board will notify DriverLINX via an interrupt when it is ready to transfer data. DriverLINX then transfers the data from the board to the buffer.
- **Paced task** — The DLSecs2Tics method converts a given value from seconds to hardware specific clock tics for the specified channel. The sample specifies that 1/10 seconds should be converted to clock tics for the DL_DEFAULTTIMER channel. When assigned to the Evt_Tim_ratePeriod property of the Service Request, the data will be paced at 10 Hz.

- Buffer notification — The `Sel_buf_notify = DL_NOTIFY` property instructs DriverLINX to send a BufferFilled event when the last sample has been taken. The event notifies the application that the buffer is full.
- Edit Service Request dialog display — The `Req_op_edit = True` statement instructs DriverLINX to display the Edit Service Request dialog. This allows the user to check the Service Request and make any changes prior to executing the Service Request. Note that when displaying the Edit Service Request, the Refresh method must be used twice. When the first Refresh is used, DriverLINX displays the dialog and sets the `Req_op_edit` property to False. The second Refresh executes the Service Request.

Visual Basic

VB The following sample code illustrates using messages with a background task.

```

With DriverLINXSR1
    .Req_subsystem = DL_AI
    .Req_mode = DL_INTERRUPT
    .Req_op = DL_START

    .Evt_Str_type = DL_COMMAND

    .Evt_Stp_type = DL_TCEVENT

    .Evt_Tim_type = DL_RATEEVENT
    .Evt_Tim_rateChannel = DL_DEFAULTTIMER
    .Evt_Tim_rateMode = DL_RATEGEN
    .Evt_Tim_rateClock = DL_INTERNAL1
    .Evt_Tim_rateGate = DL_DISABLED
    .Evt_Tim_ratePeriod = .DLSecs2Tics(DL_DEFAULTTIMER, 1 / 10)

    .Sel_chan_format = DL_tNATIVE
    .Sel_chan_N = 2
    .Sel_chan_start = 0
    .Sel_chan_startGainCode = .DLGain2Code(-1)
    .Sel_chan_stop = 15
    .Sel_chan_stopGainCode = .DLGain2Code(-1)
    .Sel_buf_N = 1
    .Sel_buf_samples = 16
    .Sel_buf_notify = DL_NOTIFY
    .Req_op_edit = True

```

```

.Refresh
.Refresh
End With

Private Sub DriverLINXSR1_BufferFilled(task As Integer, device As Integer,
    subsystem As Integer, mode As Integer, bufIndex As Integer)
Dim convert As Single
Dim dataArray(16) As Single
convert = DriverLINXSR1.VBArrayBufferConvert(bufIndex, 0, 16, dataArray,
    DL_tSINGLE, 0, 0)
Debug.Print dataArray(0) ' Display first point
End Sub

Private Sub DriverLINXSR1_ServiceDone(task As integer, device As Integer,
    subsystem As Integer, mode As Integer)
Debug.Print "Service is done"
End Sub

```

C/C++

C/C++ The following sample code illustrates using messages with a background task.

```

WORD m_DLmsg;
m_DLmsg=RegisterWindowMessage(DL_MESSAGE);
//need to register DriverLINX messages !!!!!

m_pSR->operation=START; //Start the acquisition
m_pSR->subsystem=AI; //using the AI subsystem
m_pSR->mode=INTERRUPT; //use interrupt mode
m_pSR->start.typeEvent=COMMAND; //Start on command
m_pSR->timing.typeEvent=RATEEVENT; //timing will be determined by the rate
    generator
m_pSR->stop.typeEvent=TCEVENT; //Stop on command
m_pSR->channels.nChannels=1; //Acquire on 1 channel
m_pSR->channels.chanGain[0].channel=m_logicalChannel;
//use the channel defined by m_logicalChannel
m_pSR->channels.chanGain[0].gainOrRange=Gain2Code(m_logicalDevice,AI,-1.0);
//Use bipolar unity gain
m_pSR->channels.numberFormat=tNATIVE; //use the native format (integer
    counts)
m_pSR->lpBuffers=(DL_BUFFERLIST*) new BYTE[DL_BufferListBytes(1)];
//create a buffer list pointer for one buffer

```

```

m_pSR->lpBuffers->notify=NOTIFY; //enable the buffer filled message
m_pSR->lpBuffers->nBuffers=1; //use only one buffer
m_pSR->lpBuffers-
    >bufferSize=Samples2Bytes(m_logicalDevice,AI,m_logicalChannel,m_samples);
    //set the size of the buffer (in bytes) to hold the number of samples
m_pSR->lpBuffers->BufferAddr[0]=BufAlloc(GBUF_INT,m_pSR->lpBuffers-
    >bufferSize); //Allocate Buffer 0 based on the size we just specified
m_pSR->timing.u.rateEvent.channel=DEFAULTTIMER;
//DEFAULTTIMER is a symbol representing the default counter/timer channel
used for pacing.
m_pSR->timing.u.rateEvent.mode=RATEGEN;
m_pSR->timing.u.rateEvent.clock=INTERNAL1; //External clocking will be used
m_pSR->timing.u.rateEvent.gate=DISABLED; //no gating will be used
m_pSR-
    >timing.u.rateEvent.period=Sec2Tics(m_logicalDevice,AI,INTERNAL1,0.01f);
DriverLINX(m_pSR); //Execute the service request to start the acquisition
showMessage(m_pSR); //show any errors

// the WindowProc function, often called "message pump", is used to respond
to messages
LRESULT CAIbufferDlg::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
    if(message==m_DLmsg)
//Did DriverLINX post a message? We only want to act on the DriverLINX
buffer filled message
    {
        switch(wParam)
        {
            case DL_BUFFERFILLED: //Was the DriverLINX message the buffer filled
message?
                done(); //if so, call the done() function to process the results
                break;
        }
    }

    return CDialog::WindowProc(message, wParam, lParam);
} // end WindowProc

// the function called in response to a buffer filled message from
DriverLINX
void CAIbufferDlg::done()
{
    float *readings;

```

```
readings = new float[m_samples]; //make a temporary array to hold the
converted readings
CString str; //The CString class includes a format method to convert
float to CString to display them in the listbox
int index;
/*The convert operation used here is taking advantage of the fact
that the logical device number and the subsystem (AI) have already
been set in the start request*/
m_pSR->operation=CONVERT; //Use the convert operation to convert the raw
counts in the buffer to voltages
m_pSR->mode=OTHER; //Convert is not a polled, interrupt, or DMA operation
m_pSR->start.typeEvent=DATACONVERT; //Set the start type to convert the
data
m_pSR->start.u.dataConvert.startIndex=0; //start at index 0 of the buffer
m_pSR->start.u.dataConvert.nSamples=m_samples; //set the number of samples
to convert
m_pSR->start.u.dataConvert.numberFormat=tSINGLE; //convert the counts to
tSINGLE (float)
m_pSR->start.u.dataConvert.scaling=0.0f; //no scaling will be used
m_pSR->start.u.dataConvert.offset=0.0f; //no offset will be applied
m_pSR->start.u.dataConvert.wBuffer=0;
//convert DriverLINX buffer 0, since that's the only one being used
m_pSR->start.u.dataConvert.lpBuffer=readings;
//put the converted readings in the temporary buffer
DriverLINX(m_pSR); //Execute the conversion
showMessage(m_pSR); //show any errors
m_readList.ResetContent(); //clear the listbox
for(index=0;index<m_samples;index++)
{
    str.Format("%f",readings[index]); //format the float reading into a
string for each reading
    m_readList.AddString(str); //Add the string to the listbox
}
UpdateData(FALSE); //Update the listbox display
delete readings; //clear the temporary buffer so we don't have memory
leaks when we run it again
} // end done
```

Writing a single analog value

Writing an analog value to single analog output channel is very similar to reading a single analog input value. The differences are simply changing from the AI subsystem to the AO subsystem, and specifying the `Res_Sta_io_Value` rather than reading it. Since DriverLINX requires that the output value be specified as a hardware-specific voltage code rather than an actual voltage, the `DLVolts2Code` method is used for the conversion.

The following key concepts are demonstrated in this sample:

- No buffers used — Single value transfers do not require the use of the buffer. For this sample, the value to be transferred to DriverLINX is assigned to the `Res_Sta_ioValue` property of the Service Request.
- Voltage conversion to hardware code — DriverLINX expects all voltage values passed to be in the form of hardware-specific codes. The method `DLVolts2Code` performs the conversion. This method accepts a voltage as an input and returns the corresponding hardware specific code to pass to DriverLINX.

Visual Basic

VB

The following sample code demonstrates outputting a single analog value.

```
With DriverLINXSR1
    .Req_subsystem = DL_AO
    .Req_mode = DL_POLLED
    .Req_op = DL_START
    .Evt_Str_type = DL_COMMAND
    .Evt_Stp_type = DL_TCEVENT
    .Evt_Tim_type = DL_NULLEVENT
    .Sel_chan_format = DL_tNATIVE
    .Sel_chan_N = 1
    .Sel_chan_start = 0
    .Sel_chan_startGainCode = .DLGain2Code(-1)
    .Sel_buf_N = 0
    .Res_Sta_typeStatus = DL_IOVALUE
    .Res_Sta_ioValue = .DLVolts2Code(4.345) ' put 4.345 volts on AO channel 0
    .Refresh
End With
Debug.Print DriverLINXSR1.Message
```

C/C++

C/C++ The following sample code demonstrates outputting a single analog value.

```
UpdateData (TRUE);
m_pSR2->status.u.ioValue = Volts2Code(0,AO,m_volts);
//move value from screen to ioValue member
m_pSR->operation=START;
m_pSR->subsystem=AO;
m_pSR->mode=POLLED;
m_pSR->channels.nChannels=1;
m_pSR->channels.chanGain[0].channel=0;
m_pSR->status.typeStatus=IOVALUE;
DriverLINX(m_pSR);
showMessage(m_pSR);
```

Writing a series of analog values

This example introduces several more complex concepts of DriverLINX. This example uses a background task to output two series of analog voltages to a single analog output channel paced by an external clock. Once the Service Request has been submitted, DriverLINX will begin outputting the analog values defined in the first buffer, paced by the input from the external clock. After processing the first buffer, DriverLINX processes the second buffer. Since this is a terminal count event, the task ends when processing of the second buffer is complete.

The following key concepts are demonstrated by this sample:

- **Background task** — This sample uses the DMA transfer mode. This is a background processing mode that returns control back to the application immediately. DriverLINX sets up the buffers and the task. The data acquisition board reads the data values directly from the buffer without needing DriverLINX to transfer the information to the board.
- **No stop event needed** — The sample sets up the task as a terminal count task. When DriverLINX has completed processing all buffers a single time, the task ends.
- **External clock input** — The task is paced based on an input from an external clock. Setting the Service Request property `Evt_Tim_rateClock` to

DL_EXTERNAL tells DriverLINX that the clock input for the rate event will be taken from an external clock source.

- Paced task — This sample paces the task using a rate event. DriverLINX will set the task to use the default timer channel and set up a rate generator using the external clock input as the clock source. Note that even though we are pacing the task using an external source, we still need to specify the period using the Evt_Time_ratePeriod property. DriverLINX uses this property as an estimate to make decisions about how to set up the task internally. This property should be set as close as possible to the expected clocking frequency, but does not need to be exact.
- Buffers — This example uses two buffers to store the analog output values. The buffers are loaded from VB arrays using the VBArrayBufferConvert method. This method converts the values of the array into hardware-specific codes and loads them into the specified buffer.

Visual Basic

VB

The following sample code demonstrates outputting a series of analog values.

```
Dim arrayZero(100) As Single
Dim arrayOne(100) As Single
Dim convert As Single
Dim i As Integer

For i = 0 To 99
    arrayZero(i) = i / 100      ' load arrays with VOLTAGE values
    arrayOne(i) = i * 5 / 100
Next i

With DriverLINXSR1
    .Req_subsystem = DL_AO
    .Req_mode = DL_DMA
    .Req_op = DL_START
    .Evt_Str_type = DL_COMMAND
    .Evt_Stp_type = DL_TCEVENT

    .Evt_Tim_type = DL_RATEEVENT
    .Evt_Tim_rateChannel = DL_DEFAULTTIMER
    .Evt_Tim_rateMode = DL_RATEGEN
    .Evt_Tim_rateClock = DL_EXTERNAL
    .Evt_Tim_rateGate = DL_DISABLED
```

```

.Evt_Tim_ratePeriod = .DLSecs2Tics(DL_DEFAULTTIMER, 1 / 100)
.Sel_chan_format = DL_tNATIVE
.Sel_chan_N = 1
.Sel_chan_start = 0
.Sel_chan_startGainCode = .DLGain2Code(-1)
.Sel_buf_N = 2
.Sel_buf_samples = 100
' convert and move the arrays to buffers
convert = .VBArryBufferConvert(0, 0, 100, arrayZero, DL_tSINGLE, 0, 0)
convert = .VBArryBufferConvert(1, 0, 100, arrayOne, DL_tSINGLE, 0, 0)
.Refresh ' don't forget to apply external clock signal
End With
Debug.Print DriverLINXSR1.Message

```

C/C++

C/C++ The following sample code demonstrates outputting a series of analog values.

```

memset(m_pSR,0,sizeof(DL_ServiceRequest));
DL_SetServiceRequestSize(*m_pSR);
clearBuffers(); //clear any existing buffers
// generate an array to write a sine wave on the AO channel
float sinebuf[100]; //create an array of float for the sine wave
int i;
for(i=0;i<100;i++)
{
    sinebuf[i]=(float)(5*sin(i*2*3.14159/100));
}
// convert this data into hardware specific DAC units and move the data to
DLinx memory buffer
m_pSR->operation=CONVERT; //Use the convert operation to translate float
into counts
m_pSR->device=m_logicalDevice;
m_pSR->subsystem=AO;
m_pSR->mode=OTHER;
m_pSR->start.typeEvent=DATACONVERT;
m_pSR->lpBuffers=(DL_BUFFERLIST*)new BYTE[DL_BufferListBytes(2)];
//Create a buffer list for two buffers
m_pSR->lpBuffers->nBuffers=2; //Two buffers will be used
m_pSR->lpBuffers->bufferSize=
    Samples2Bytes(m_logicalDevice,AO,m_logicalChannel,100);
//allocate enough bytes for 100 samples

```

```

m_pSR->lpBuffers->BufferAddr[0]=BufAlloc(GBUF_INT,m_pSR->lpBuffers-
>bufferSize);
//allocate one 100 point buffer
m_pSR->lpBuffers->BufferAddr[1]=BufAlloc(GBUF_INT,m_pSR->lpBuffers-
>bufferSize);
m_pSR->channels.nChannels=1; //only using one channel
m_pSR->channels.chanGain[0].channel=m_logicalChannel;
// Set the appropriate channel in the channel/gain list
m_pSR->channels.numberFormat=tNATIVE;
m_pSR->start.u.dataConvert.startIndex=0; //start at index 0 for the
conversion
m_pSR->start.u.dataConvert.nSamples=100; //Convert 100 samples
m_pSR->start.u.dataConvert.numberFormat=tSINGLE; //the input buffer is of
type float
m_pSR->start.u.dataConvert.scaling=0.0f; //no scaling will be used
m_pSR->start.u.dataConvert.offset=0.0f; //no offset will be applied
m_pSR->start.u.dataConvert.wBuffer=0; //put the converted samples into
buffer 0
m_pSR->start.u.dataConvert.lpBuffer=sinebuf; //use the sinebuf array as the
input buffer
m_pSR->hWnd=m_hWnd;
DriverLINX(m_pSR); //execute the conversion
showMessage(m_pSR); //show any errors
m_pSR->start.typeEvent=DATA_CONVERT;
// Put converted samples into buffer 1
m_pSR->start.u.dataConvert.wBuffer=1;
DriverLINX(m_pSR);
ShowMessage(m_pSR);

//Setup the service request for interrupt analog output
m_pSR->operation=START;
m_pSR->device=m_logicalDevice;
m_pSR->subsystem=A0;
m_pSR->mode=DMA;
m_pSR->start.typeEvent=COMMAND;
m_pSR->channels.nChannels=1;
m_pSR->channels.chanGain[0].channel=0;
m_pSR->channels.chanGain[0].gainOrRange=
Gain2Code(m_logicalDevice,AI,-1.0);

```

```
//Use bipolar unity gain
m_pSR->channels.numberFormat=tNATIVE;
//use the native format

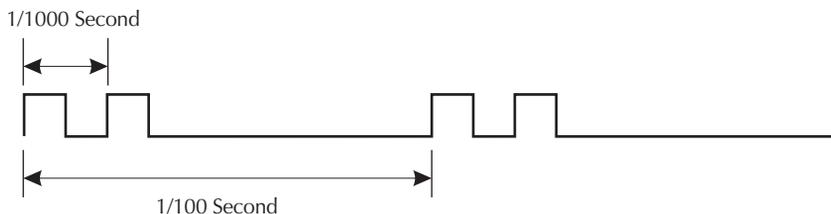
m_pSR->timing.typeEvent=RATEEVENT;
m_pSR->timing.u.rateEvent.channel=DEFAULTTIMER;
//DEFAULTTIMER will pick the appropriate timing channel for a rate event
m_pSR->timing.u.rateEvent.mode=RATEGEN; //use the rate generator mode to
time the waveform
m_pSR->timing.u.rateEvent.clock=EXTERNAL;
m_pSR->timing.u.rateEvent.gate=DISABLED; //don't use gating
m_pSR->timing.u.rateEvent.period=
    Sec2Tics(m_logicalDevice,AO,INTERNAL1,0.01f); // 100Hz
m_pSR->stop.typeEvent=TCEVENT; //Stop at end of buffer
DriverLINX(m_pSR); //Start the waveform
showMessage(m_pSR); //show any errors
```

Writing analog values on several channels

The previous example used a single analog output channel to output two series of voltages paced by an external clock. Building on that example, we can modify our program to use two separate analog output channels at the same time. Each channel will output one of the two series of voltages. Rather than use a simple rate generator, this example switches to a burst generator paced by an internal timer.

A burst generator requires three properties to be set to specify how the burst generator will function. The first property is the period at which the bursts will occur. The property `Evt_Tim_ratePeriod` in the example uses the `DLSecs2Tics` method to set the period to 100 Hz. The second property is the number of pulses for each burst. Since this example uses two channels, we will use two pulses. The property `Evt_Time_ratePulses` is set to 2. The third property is the rate at which the pulses will be burst. This example uses a rate of 1000 Hz, or 1 kHz. When combined, these properties produce a burst generator that will generate bursts at 100 Hz. Each burst will consist of two 1 kHz pulses. This can be illustrated as shown in [Figure 6-1](#).

Figure 6-1
Burst generator



The following key concepts are presented in this example:

- Internal timer — This task is paced using the internal clock on the data acquisition board.
- Burst generator — This task uses a burst generator style clock to pace the task. A burst generator requires a period, burst count, and burst rate to be specified. The burst generator is selected with the `Evt_Tim_rateMode = DL_BURSTGEN` property.
- Start/stop channel list — A start/stop list is used to specify the channels to be used for output. This example uses analog output channels 0 and 1.

Visual Basic

VB

The following example demonstrates writing a series of analog values to two analog output channels.

```
Dim arrayZero(100) As Single
Dim arrayOne(100) As Single
Dim convert As Single
Dim i As Integer
```

```
For i = 0 To 99
arrayZero(i) = i / 100 ' load arrays with VOLTAGE values
arrayOne(i) = i * 5 / 100
Next i
```

```
['now that we're multichannel data points are interleaved....(0) to chan0,
(1) to chan1, (2) to chan0, back ,forth, back, forth, etc.]
```

```
With DriverLINXSRL
.Req_subsystem = DL_AO
```

```

.Req_mode = DL_DMA
.Req_op = DL_START
.Evt_Str_type = DL_COMMAND
.Evt_Stp_type = DL_TCEVENT

.Evt_Tim_type = DL_RATEEVENT
.Evt_Tim_rateChannel = DL_DEFAULTTIMER
.Evt_Tim_rateMode = DL_BURSTGEN
.Evt_Tim_rateClock = DL_INTERNAL1
.Evt_Tim_rateGate = DL_DISABLED
.Evt_Tim_ratePeriod = .DLSecs2Tics(DL_DEFAULTTIMER, 1 / 100)
.Evt_Tim_ratePulses = 2 ' one pulse for each channel in burst
.Evt_Tim_rateOnCount = .DLSecs2Tics(DL_DEFAULTTIMER, 1 / 1000)
    ' how fast to burst
.Sel_chan_format = DL_tNATIVE
.Sel_chan_N = 2
.Sel_chan_start = 0
.Sel_chan_startGainCode = .DLGain2Code(-1)
.Sel_chan_stop = 1
.Sel_chan_stopGainCode = .DLGain2Code(-1)
.Sel_buf_N = 2
.Sel_buf_samples = 100
.Sel_buf_notify = DL_NOTIFY
' convert and move the arrays to buffers
convert = .VArrayBufferConvert(0, 0, 100, arrayZero, DL_tSINGLE, 0, 0)
convert = .VArrayBufferConvert(1, 0, 100, arrayOne, DL_tSINGLE, 0, 0)
.Refresh
End With
Debug.Print DriverLINXSR1.Message

```

C/C++

C/C++ The following example demonstrates writing a series of analog values to two analog output channels.

```

m_pSR->operation=START;
m_pSR->device=m_logicalDevice;
m_pSR->subsystem=AO;
m_pSR->mode=DMA;
m_pSR->start.typeEvent=COMMAND;
m_pSR->channels.nChannels=2; //Write on start stop range of channels

```

```
m_pSR->channels.chanGain[0].channel=0; //start on channel 0
m_pSR->channels.chanGain[0].gainOrRange=
    Gain2Code(m_logicalDevice,AI,-1.0);
//Use bipolar unity gain
m_pSR->channels.chanGain[1].channel=1; // stop on channel 1
m_pSR->channels.chanGain[1].gainOrRange=
    Gain2Code(m_logicalDevice,AI,-1.0);
m_pSR->channels.numberFormat=tNATIVE;
m_pSR->timing.typeEvent=RATEEVENT;
m_pSR->timing.u.rateEvent.channel=DEFAULTTIMER;
m_pSR->timing.u.rateEvent.mode=BURSTGEN;
m_pSR->timing.u.rateEvent.clock=INTERNAL1;
m_pSR->timing.u.rateEvent.gate=DISABLED; //don't use gating
m_pSR->timing.u.rateEvent.period=
    Sec2Tics(m_logicalDevice,AO,INTERNAL1,0.01f); //100Hz
m_pSR->timing.u.rateEvent.onConvert = .Sec2Tics(m_logicalDevice, AO,
    Internal1, 0.001f); // 1000Hz
m_pSR->timing.u.rateEvent.Pulses = 2;
m_pSR->stop.typeEvent=TCEVENT; //Stop at end of buffer
DriverLINX(m_pSR); //Start the waveform
showMessage(m_pSR); //show any errors
```

Foreground and background operation

DriverLINX takes advantage of the multi-tasking nature of the Windows operating system. Multi-tasking allows an application to do more than one thing at a time. The application which has control is known as the foreground task. All other applications are said to be running in the background. The operating mode of a task is determined when the Service Request properties are specified. Polled mode tasks are executed in the foreground. Interrupt and DMA mode tasks are executed in the background.

When an application starts a DriverLINX task in the foreground, it must wait until the task is complete before it can continue. If the application starts the DriverLINX task in the background, it can continue on performing other operations while DriverLINX processes the task. This difference is an important distinction. Applications starting a foreground mode task can count on the task being complete when control is returned and the application continues. Applications that start background mode tasks will immediately be returned control while the task is processed in the background by DriverLINX. In this case, the application will either have to wait for DriverLINX to

return an event, or poll the task with a status request. Background tasks can run continuously for long periods of time. It is important, when starting a background task, to provide a method of stopping the task.

It is also important to determine a way to handle all the data that can be generated by a background task. If DriverLINX is running a terminal count task, it will automatically stop the task as soon as all buffers have been processed a single time. If a different type of stop event is used, DriverLINX will recycle the buffers. When the last buffer has been processed, the first buffer will be reused. Unless the data has already been used by the application, the original contents of the buffer will be lost. Sufficient buffer space must be allocated to allow the application time to process the data before the buffer is reused.

The following example application starts a background task that continually reads analog input channel 0 at a rate of 100 Hz. The task will continually cycle through three buffers until a stop command is received. To stop the task, a DL_STOP operation is attached to a button on a form. When the button is clicked, a Service Request is sent to DriverLINX to stop the task.

The following key concepts are presented in this example:

- Continuous background operation — Once the Service Request is submitted, the task runs continuously in the background. The task will not stop until a Service Request is submitted to stop the task.
- Large buffers — This task allocates three buffers, each containing space for 100 samples. At the requested rate of 100 Hz, each buffer will take one second to fill. This allows the application sufficient time to process the buffer data before the task begins to recycle the buffers. A rule of thumb that can be used when determining how much buffer space to allocate is to use at least three buffers comprising at least one second worth of total buffering. Allocating less buffer space for a continuously running task can cause Data Lost messages.
- Stop on command — The example task will continue to cycle through all available buffers until a stop event is received. In this example, the stop event is generated by clicking on a button that generates a Service Request.

Visual Basic

VB

The following sample code demonstrates a continuously running background task.

```
With DriverLINXSR1
.Req_subsystem = DL_AI
```

```

.Req_mode = DL_DMA      ' or DL_INTERRUPT
.Req_op = DL_START
.Evt_Str_type = DL_COMMAND
.Evt_Stp_type = DL_COMMAND
.Evt_Tim_type = DL_RATEEVENT
.Evt_Tim_rateChannel = DL_DEFAULTTIMER
.Evt_Tim_rateMode = DL_RATEGEN
.Evt_Tim_rateClock = DL_INTERNAL1
.Evt_Tim_rateGate = DL_DISABLED
.Evt_Tim_ratePeriod = .DLsecs2Tics(DL_DEFAULTTIMER, 1 / 100)
.Sel_chan_format = DL_tNATIVE
.Sel_chan_N = 1
.Sel_chan_start = 0
.Sel_chan_startGainCode = .DLGain2Code(-1)
.Sel_buf_N = 3
.Sel_buf_samples = 100  ' 100 samples at 100Hz rate means this buffer = 1
    second worth of data
.Sel_buf_notify = DL_NOTIFY
.Res_Sta_typeStatus = DL_IOSTATUS ' status about the operation
.Refresh
End With
Debug.Print DriverLINXSR1.Message

' must stop the operation!!!
Private Sub cmdStop_Click() ' a button on form called cmdStop
DriverLINXSR1.Req_op = DL_STOP
DriverLINXSR1.Refresh
Debug.Print DriverLINXSR1.Message
End Sub

```

C/C++

C/C++ The following sample code demonstrates a continuously running background task.

```

clearBuffers(); //Clear existing buffers
// Register DriverLINX as message source
WORD m_DLmsg;
m_DLmsg = RegisterWindowMessage(DL_MESSAGE);
m_pSR->operation=START;
m_pSR->subsystem=AI;
m_pSR->mode=INTERRUPT; //or DMA;
m_pSR->start.typeEvent=COMMAND;

```

```

m_pSR->timing.typeEvent=RATEEVENT;
m_pSR->stop.typeEvent=COMMAND;
m_pSR->channels.nChannels=1;
m_pSR->channels.chanGain[0].channel=0;
m_pSR->channels.chanGain[0].gainOrRange=
    Gain2Code(m_device,AI,-1.0);
m_pSR->channels.numberFormat=tNATIVE;
m_pSR->lpBuffers=(DL_BUFFERLIST*) new BYTE[DL_BufferListBytes(m_buffers)];
//make a buffer list for three buffers
m_pSR->lpBuffers->notify=NOTIFY;
m_pSR->lpBuffers->nBuffers=m_buffers; //Set number of buffers = 3
m_pSR->lpBuffers->bufferSize=
    Samples2Bytes(m_device,AI,0,m_samples); //m_samples=100
//Set size of buffers in bytes
m_pSR->lpBuffers->BufferAddr[0]=BufAlloc(GBUF_INT,m_pSR->lpBuffers-
    >bufferSize); //allocate buffer 0
m_pSR->lpBuffers->BufferAddr[1]=BufAlloc(GBUF_INT,m_pSR->lpBuffers-
    >bufferSize); //allocate buffer 1
m_pSR->lpBuffers->BufferAddr[2]=BufAlloc(GBUF_INT,m_pSR->lpBuffers-
    >bufferSize); //allocate buffer 2
m_pSR->timing.u.rateEvent.channel=DEFAULTTIMER;
m_pSR->timing.u.rateEvent.mode=RATEGEN;
m_pSR->timing.u.rateEvent.clock=INTERNAL1;
m_pSR->timing.u.rateEvent.gate=DISABLED;
m_pSR->timing.u.rateEvent.period=
    Sec2Tics(m_device,AI,INTERNAL1,0.01f); // 100Hz
m_pSR->timing.u.rateEvent.pulses=0;
DriverLINX(m_pSR);
showMessage(m_pSR);
m_task=m_pSR->taskId; //save the task ID so we can stop it later

LRESULT CMultbufDlg::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
// TODO: Add your specialized code here and/or call the base class
if(message == m_DLmsg) //Check for DriverLINX message
{
    switch(wParam)
    {
        case DL_BUFFERFILLED: //Check for buffer filled message
            m_bufnum=getBufIndex(lParam); //Get index of filled buffer
            done(m_bufnum);
            //process readings...use CONVERT operations from before

```

```
        break;
    }
}
return CDialog::WindowProc(message, wParam, lParam);
}

void CMultbufDlg::OnStop()
{
    m_pSR->taskId=m_task; // restore taskID
    m_pSR->operation=STOP;
    m_pSR->mode=INTERRUPT;
    DriverLINX(m_pSR);
    showMessage(m_pSR);
}
```

Using an external digital trigger

DriverLINX supports the capability to trigger a task based on an input from an external digital trigger channel.

In this example, an external digital trigger is used to start a background task using analog input channel 0. The task will continue to take readings on analog input channel 0 at a rate of 100 Hz until another Service Request is received with a stop event. This task uses three buffers to store the analog input readings. In addition, this example uses the DriverLINX StartEvent notification, and status polling.

When using a digital start trigger, DriverLINX requires that three properties be specified. First, a mask must be specified for the input as `Evt_Str_diMask`. The mask tells DriverLINX what bits of the input to monitor. Second, you must specify a pattern for the input to match as the `Evt_Str_diPattern`. This pattern will be compared to the input. Third, you must specify whether the input must match or not match the pattern. This is set by specifying the `Evt_Str_diMatch` property as `DL_Equals` or `DL_NotEquals`.

This example illustrates the following key concepts:

- Start on external digital trigger — This example uses an external digital trigger to begin the task.

- StartEvent notification — When DriverLINX detects that the external digital trigger conditions match those specified in the Service Request, the task will begin and DriverLINX will send a StartEvent message. This example uses the StartEvent message to notify us that the trigger occurred and to start a timer.
- Status polling — Every time the specified timer interval has elapsed, this example submits a status Service Request to DriverLINX. DriverLINX returns the number of the current buffer being filled and the number of the next buffer element. Applications can use this information to check on the progress of a task.
- Continuous buffer processing — This example uses the BufferFilled event sent by DriverLINX to notify the application that a buffer has been filled. This allows the application to process each buffer as it becomes available.

Visual Basic

VB

The following sample code demonstrates using an external digital trigger to start a task.

```
With DriverLINXSR1
.Req_subsystem = DL_AI
.Req_mode = DL_DMA
.Req_op = DL_START

' start on Digital Trigger
.Evt_Str_type = DL_DIEVENT
.Evt_Str_diChannel = DL_DI_EXTTRG
.Evt_Str_diMask = 1
.Evt_Str_diMatch = DL_NotEquals
.Evt_Str_diPattern = 0

.Evt_Stp_type = DL_COMMAND

.Evt_Tim_type = DL_RATEEVENT
.Evt_Tim_rateChannel = DL_DEFAULTTIMER
.Evt_Tim_rateMode = DL_RATEGEN
.Evt_Tim_rateClock = DL_INTERNAL1
.Evt_Tim_rateGate = DL_DISABLED
.Evt_Tim_ratePeriod = .DLSecs2Tics(DL_DEFAULTTIMER, 1 / 100)
.Sel_chan_format = DL_tNATIVE
.Sel_chan_N = 1
.Sel_chan_start = 0
.Sel_chan_startGainCode = .DLGain2Code(-1)
```

```

.Sel_buf_N = 3
.Sel_buf_samples = 100
.Sel_buf_notify = DL_NOTIFY + DL_NOTIFY_START
.Res_Sta_typeStatus = DL_IOSTATUS      ' status about the operation
.Refresh
End With
Debug.Print DriverLINXSR1.Message
Debug.Print "Awaiting Digital Trigger...."

' From the Service Start Event Procedure:
Private Sub DriverLINXSR1_StartEvent(task As Integer, device As Integer,
subsystem As Integer, mode As Integer)
Debug.Print "Trigger has occurred...."
' start timer
Timer1.Interval = 500
Timer1.Enabled = True
End Sub

' From the Timer object's event procedure:
Private Sub Timer1_Timer()
With DriverLINXSR1
.Res_op = DL_STATUS
.Refresh
Debug.Print "Current Buffer is: " + Str(.Res_IO_currentBuffer)
Debug.Print "The next element in this buffer is: " +
    Str(.Res_IO_nextElement)
End With
End Sub

' Buffer Filled messages still sent and can be responded to:
Private Sub DriverLINXSR1_BufferFilled(task As Integer, device As Integer,
subsystem As Integer, mode As Integer, bufIndex As Integer)
Dim convert As Single
Dim dataArray(16) As Single
Beep
convert = DriverLINXSR1.VBArrayBufferConvert(bufIndex, 0, 16, dataArray,
    DL_tSINGLE, 0, 0)
Debug.Print dataArray(0)
End Sub

```

C/C++

C/C++ The following sample code demonstrates using an external digital trigger to start a task.

```
clearBuffers();
//de-allocate any exisiting buffers in the service request

m_pSR->operation=START; //Start the acquisition
m_pSR->subsystem=AI; //using the AI subsystem
m_pSR->mode=INTERRUPT; //use interrupt mode

m_pSR->start.typeEvent= DIEVENT; // start on Digital Trigger
m_pSR->start.u.diEvent.channel=DI_EXTTRG;
m_pSR->start.u.diEvent.mask=1;
m_pSR->start.u.diEvent.match=0;
m_pSR->start.u.diEvent.pattern=0;

m_pSR->timing.typeEvent=RATEEVENT;
//timing will be determined by the rate generator
m_pSR->stop.typeEvent=COMMAND; //Stop on command
m_pSR->channels.nChannels=1; //Acquire on 1 channel
m_pSR->channels.chanGain[0].channel=m_logicalChannel;
//use the channel defined by m_logicalChannel
m_pSR->channels.chanGain[0].gainOrRange=
    Gain2Code(m_logicalDevice,AI,-1.0);
//Use bipolar unity gain
m_pSR->channels.numberFormat=tNATIVE; //use the native format (integer
    counts)
m_pSR->lpBuffers=(DL_BUFFERLIST*) new BYTE[DL_BufferListBytes(3)];
m_pSR->lpBuffers->notify=NOTIFY; //enable the buffer filled message
m_pSR->lpBuffers->nBuffers=3; //use 3 buffers
m_pSR->lpBuffers->bufferSize=Samples2Bytes
    (m_logicalDevice,AI,m_logicalChannel,m_samples);
//set the size of the buffer (in bytes) to hold the number of samples
// allocate the 3 buffers
m_pSR->lpBuffers->BufferAddr[0]=BufAlloc(GBUF_INT,m_pSR->
    lpBuffers->bufferSize);
m_pSR->lpBuffers->BufferAddr[1]=BufAlloc(GBUF_INT,m_pSR->
    lpBuffers->bufferSize);
m_pSR->lpBuffers->BufferAddr[2]=BufAlloc(GBUF_INT,m_pSR->
    lpBuffers->bufferSize);
```

```

m_pSR->timing.u.rateEvent.channel=DEFAULTTIMER;
m_pSR->timing.u.rateEvent.mode=RATEGEN;
m_pSR->timing.u.rateEvent.clock=INTERNAL1; //Internal1 time base
m_pSR->timing.u.rateEvent.gate=DISABLED; //no gating will be used
m_pSR->timing.u.rateEvent.period=
    Sec2Tics(m_logicalDevice,AI,INTERNAL1,0.01f);
DriverLINX(m_pSR);
showMessage(m_pSR); //show any errors
m_task=m_pSR->taskId; //Need to save the task ID of the AI task
                        // so it can be stopped later
/*See the WindowProc function to see what happens next*/

/* NOTE: A Resource has been added (View|Resource Symbols), called
IDC_TIMER, to intercept WM_TIMER messages. Use the MFC function call below
to set the timer for 1000 msec intervals. Use Class Wizard to create and
assign a call back for this message. */

SetTimer(IDC_TIMER, (int) (1000), NULL);

// the call back function for WM_TIMER messages:
void CAIbufferDlg::OnTimer(UINT nIDEvent)
{

    // use case statement to test if it was our timer, IDC_TIMER,
    // that sent the WM_TIMER message
    switch(nIDEvent)
    case IDC_TIMER:
    {
        // yes, it was us. Let's do STATUS operation
        m_pSR->mode=INTERRUPT; //restore interrupt mode selection
        m_pSR->operation=STATUS;
        //Use the status operation to get the timer count
        m_pSR->status.typeStatus=IOSTATUS;
        m_pSR->taskId=m_task;
        DriverLINX(m_pSR); //Get the current timer count
        // assign currentBuffer and nextElement to variable names
        // for text box controls on the form
        m_bufNum = m_pSR->status.u.ioStatus.currentBuffer;
        m_bufElement = m_pSR->status.u.ioStatus.nextElement;
        UpdateData(FALSE); //Update the controls on the form
        showMessage(m_pSR); //show any errors from the STATUS operation
        break;
    }
}

```

```
    }
    CDialog::OnTimer(nIDEvent);
} // OnTimer

/* NOTE: Meanwhile, the WindowProc function is processing the BufferFilled
Messages and calling the done() function to do the data conversion as
before. Since the START operation, CONVERT operation and this new STATUS
operation are all sharing the same Service Request structure, the foreground
mode gets changed back and forth between OTHER and INTERRUPT. The convert op
needs it to be OTHER, .the Status op needs it to be the same as the back
ground task, eg, Interrupt or DMA. */

// stop the status polling and the IRQ mode task
void CAIbufferDlg::OnStop()
{
    // TODO: Add your control notification handler code here
    KillTimer(IDC_TIMER); //stop the timer
    m_pSR->operation=STOP; //Use a stop operation to halt the AI
    m_pSR->mode=INTERRUPT;
    m_pSR->start.typeEvent=NULLEVENT;
    m_pSR->taskId=m_task; //Restore the task so it can be stopped
    DriverLINX(m_pSR);
    showMessage(m_pSR);
    m_startButton.EnableWindow(TRUE); //re-enable the start button
}
```

Analog Input Events

DriverLINX provides the capability to use an analog input to provide a start or stop event as introduced on [page 6-9](#). Depending on the capabilities of the data acquisition board, analog events can be set to trigger on level, or on a rising or falling edge. This example uses an analog input channel to check for the falling edge of a 2 VDC signal.

Another property of the Service Request used in this example is the delay property. As discussed on [page 6-12](#), a delay can be used to postpone the processing of an event until the specified number of samples have been discarded or taken. If a start event is provided with a delay, the specified number of samples are discarded after the event occurs before data is placed in the buffer. If a stop event is provided with a delay, the specified number of samples are buffered before the task is stopped.

This example illustrates the following key concepts:

- Stop on an analog event — This example monitors analog input channel 0. When the falling edge of a 2 VDC signal is detected, a stop event is generated. This is only one of the many ways an analog event can be set up. Refer to Chapter 6 of the *DriverLINX Technical Reference Manual* or the *DriverLINX/VB Technical Reference Manual* for a complete description of the ways to configure an analog event.
- Delay an event — The example code used the delay property of an event. The analog stop event uses a 100 sample delay. This causes the task to buffer an additional 100 samples after the stop event is detected. This techniques can be used to record any transients that may result form the triggering event. If the delay were used on the start event, the task would discard 100 samples before data was buffered. This can be used to avoid the transient associated with start events.
- Channel/gain list — Rather than specifying a single channel or a range of channels, this example uses a channel/gain list to specify the channels used. A channel/gain list can be used to specify any number of channels. The channels do not need to be consecutive, and they do not need to use the same gain. Channel/gain lists may not be supported by all data acquisition boards. Refer to the *Using DriverLINX With Your Hardware* manual to see if your data acquisition board supports a channel/gain list.

VB

VB

When programming using Visual Basic, unlike C/C++, the StopEvent message contains the values for bufIndex and bufElement. These contain the buffer number and buffer element that was being processed when the stop event occurred. It is not necessary to use a separate function to retrieve this information.

The following sample code demonstrates using an analog event with a delay to stop a task.

```
Dim i As Integer
Dim threshold As Long

With DriverLINXSR1
    .Req_subsystem = DL_AI
    .Req_mode = DL_DMA
    .Req_op = DL_START
    .Evt_Str_type = DL_COMMAND
```

```

.Evt_Stp_type = DL_AIEVENT
.Evt_Stp_aiChannel = 0
.Evt_Stp_aiGainCode = .DLGain2Code(-1)

' NOTE: The ActiveX control has a little bug in it: the
' properties for threshold are defined as integers; the
' DLVolts2Code method returns a LONG. Integers in VB are
' ALWAYS signed (-32768 to +32767). If the value returned
' by DLVolts2Code is greater than max value of a VB integer
' then we shift it down.

threshold = .DLVolts2Code(2.0)
' the method returns a LONG...2 volts passed in

If threshold > 32767 Then ' need to cast as integer
threshold = threshold - 65535
End If

.Evt_Stp_aiLowerThreshold = threshold    ' property is integer
.Evt_Stp_aiUpperThreshold = threshold
.Evt_Stp_aiSlope = AnaTrgNegInside    ' falling edge
.Evt_Stp_delay = 100    ' take 100 samples after the trigger occurs

.Evt_Tim_type = DL_RATEEVENT
.Evt_Tim_rateChannel = DL_DEFAULTTIMER
.Evt_Tim_rateMode = DL_RATEGEN
.Evt_Tim_rateClock = DL_INTERNAL1
.Evt_Tim_rateGate = DL_DISABLED
.Evt_Tim_ratePeriod = .DLSecs2Tics(DL_DEFAULTTIMER, 1 / 100)
.Sel_chan_format = DL_tNATIVE
' use channel gain list
.Sel_chan_N = 4
For i = 0 To 3
.Sel_chan_list(i) = i
.Sel_chan_gainCodeList(i) = .DLGain2Code(-1)
Next i
.Sel_buf_N = 3
.Sel_buf_samples = 100
.Sel_buf_notify = DL_NOTIFY
.Refresh
End With

```

```
Debug.Print DriverLINXSR1.Message
```

```
Private Sub DriverLINXSR1_StopEvent(task As Integer, device As Integer, sub
    system As Integer, mode As Integer, bufIndex As Long, bufElement As Long)
Debug.Print "buffer in which Trigger Occured: " + Str(bufIndex)
Debug.Print "Element in that buffer: " + Str(bufElement)
Beep
End Sub
```

C/C++

C/C++ When programming using C/C++, unlike Visual Basic, the StopEvent message does not contain the buffer number and buffer element that was being processed when the stop event occurred. A separate function call must be made to retrieve this information.

The following sample code demonstrates using an analog event with a delay to stop a task.

```
clearBuffers(); //de-allocate existing buffers and
    Channel/Gain Lists
m_pSR->operation= START ; //Start the acquisition
m_pSR->subsystem=AI; //using the AI subsystem
m_pSR->mode=DMA;
m_pSR->start.typeEvent= COMMAND;
m_pSR->timing.typeEvent=RATEEVENT;
//timing will be determined by the rate generator

m_pSR->stop.typeEvent=AIEVENT; //Stop on Analog Trigger
m_pSR->stop.delay = 100;
// acquire 100 samples after the Trigger occurs
m_pSR->stop.u.aiEvent.channel = 0;
m_pSR->stop.u.aiEvent.gain = Gain2Code(0,AI,-1);
m_pSR->stop.u.aiEvent.flags = AnaTrgNegInside; // negative slope
m_pSR->stop.u.aiEvent.upperThreshold = Volts2Code(m_logicalDevice,AI,2.0f);
m_pSR->stop.u.aiEvent.lowerThreshold = Volts2Code(m_logicalDevice,AI,2.0f);
// channel gain list
m_pSR->channels.nChannels=4; //Acquire on 4 channels
m_pSR->channels.chanGainList=new CHANGAIN[4];
//allocate channel gain list of 4
for(i=0;i<4;i++)
{
```

```

    m_pSR->channels.chanGainList[i].channel=i;
    m_pSR->channels.chanGainList[i].gainOrRange=
    Gain2Code(m_logicalDevice,AI,-1.0);
}

m_pSR->channels.numberFormat=tNATIVE;
//use the native format (integer counts)
m_pSR->lpBuffers=(DL_BUFFERLIST*) new BYTE[DL_BufferListBytes(3)];
m_pSR->lpBuffers->notify=NOTIFY; //enable the buffer filled message
m_pSR->lpBuffers->nBuffers=3; //use 3 buffers
m_pSR->lpBuffers->bufferSize=Samples2Bytes
    (m_logicalDevice,AI,m_logicalChannel,m_samples);
m_pSR->lpBuffers->BufferAddr[0]=BufAlloc
    (GBUF_INT,m_pSR->lpBuffers->bufferSize);

m_pSR->lpBuffers->BufferAddr[1]=BufAlloc
    (GBUF_INT,m_pSR->lpBuffers->bufferSize);
m_pSR->lpBuffers->BufferAddr[2]=BufAlloc
    (GBUF_INT,m_pSR->lpBuffers->bufferSize);

m_pSR->timing.u.rateEvent.channel=DEFAULTTIMER;
m_pSR->timing.u.rateEvent.mode=RATEGEN;
m_pSR->timing.u.rateEvent.clock=INTERNAL1;
m_pSR->timing.u.rateEvent.gate=DISABLED;
m_pSR->timing.u.rateEvent.period=Sec2Tics
    (m_logicalDevice,AI,INTERNAL1,0.01f);

DriverLINX(m_pSR);
//Execute the service request to start the acquisition
showMessage(m_pSR); //show any errors
m_task=m_pSR->taskId; //Need to save the task ID of the AI task
// so it can be stopped later
// See the WindowProc function to see what happens next

LRESULT CAIbufferDlg::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
    // TODO: Add specialized code here and/or call the base class
    if(message==m_DLmsg) //Did DriverLINX post a message?
    //We only want to act on the DriverLINX buffer filled message
    {
        switch(wParam)
        {

```

```

case DL_BUFFERFILLED: // Was the DriverLINX message
                    // the buffer filled message?
    m_whichbuf=getBufIndex(lParam);
                    //Get index of filled buffer
    done(m_whichbuf);
    //if so, call the done() function to process the results
    break;

```

/* NOTE: Use the info passed with the Windows message in lParam. When Dlinx is the message source and this is a STOP EVENT message, extra stuff is packed in there. DriverLINX documents say EvtMsg.prm1 is total number of buffers, but this does not seem to be true. Seems instead to be the buffer in which the trigger occurred.

Additional variables declared for this are:

```

HDLEVENT      m_EvtHandle;
DLEVENTMSG    m_EvtMsg;  */

```

```

case DL_STOPEVENT:
    m_EvtHandle = getEventHandle(lParam);
    // get handle to info from THIS event
    m_EvtMsg.size = DLEVENTMSGSIZE;
    // set size according to THIS event
    GetEvent (&m_EvtMsg,m_pSR->hWnd,m_EvtHandle);
    // get Event info
    OnStopEvent(m_EvtMsg.prm1,m_EvtMsg.prm2);
    // pass info along
    break;

```

```

    }
}
return CDialog::WindowProc(message, wParam, lParam);
}

```

```

void CAIbufferDlg::OnStopEvent(int bufNum,int bufElement)
{
    m_bufNum = bufNum;
    m_bufElement = bufElement;
    UpdateData(FALSE); //Update the display
}

```

/* NOTE: m_pSR->stop.u.aiEvent.flags = AnaTrgNegInside; // negative slope. DriverLINX defines edge, band, limit, and level type analog triggering. Not

all boards support all types. The area at about line # 900 in drvlinx.h file is very helpful about the valid entries for this member of the SR structure:

An analog input channel can be monitored as a start or stop event.

Eight types of analog triggers are defined:

- 1) signal is above threshold (signal > threshold)
- 2) signal is below threshold (signal < threshold)
- 3) signal is between thresholds (lo <= signal <= hi)
- 4) signal is outside thresholds (signal < lo or signal > hi)
- 5) signal is crossing threshold with positive slope
(last <= hi and signal > hi)
- 6) signal is crossing threshold with negative slope
(last >= hi and signal < hi)
- 7) signal is crossing both thresholds with positive slope
(last <= lo and signal > lo then
last <= hi and signal > hi)
- 8) signal is crossing both thresholds with negative slope
(last >= hi and signal < hi then
last >= lo and signal < lo)

Setup required:

Type	Upper	Lower	flags
1	threshold	min	AnaTrgPosOutside
2	max	threshold	AnaTrgPosOutside
3	high	low	AnaTrgNegInside
4	high	low	AnaTrgPosOutside
5	threshold	threshold	AnaTrgPosOutside
6	threshold	threshold	AnaTrgNegInside
7	high	low	AnaTrgPosOutside AnaTrgDUALCROSSING
8	high	low	AnaTrgNegInside AnaTrgDUALCROSSING

*/

```
#ifndef DL_REDECLARE
```

```
#define AnaTrgNegInside 0 // negative slope or inside limits
#define AnaTrgPosOutside 1 // positive slope or outside limits
#define AnaTrgDUALCROSSING 2 // dual threshold crossing flag
#define AnaTrgACCOUPLING 4 // AC trigger coupling
```

```
#define OutsideOrPosSlope flags // for compatibility with old name
```

```
#define AnaDefTrgChn ((SINT)-1) // default analog trigger channel
```

```
#endif
```

7 Alternate API for Digital I/O Boards

Introduction

When programming the purely digital I/O boards, an alternate API is available. The API allows simple read/write command control based on mode 0 of the Intel 8255 chip. This alternate API is based on the Direct I/O COM Object. This is an ActiveX control which can be used with Visual Basic and Visual C++. This API can be used with any of the purely digital I/O boards such as the KPCI-PIO24, KPCI-PIO96, KPCI-3160, or any of the ISA cards such as the PIO-12, PIO-24, PIO-32 Series, PDISO-8, REL-16, etc.

Code developed using this API cannot be used to control the digital I/O channels of a multifunction card. Any applications developed using the alternate API may not be portable to other boards. If your application uses a multi-function board, uses the interrupt or DMA modes, or must run on several different boards, you should use the full DriverLINX API.

The direct I/O API is 32-bit only, and does not allow coordination with the standard DriverLINX interface. In addition, the direct I/O interface supports only 8-bit hardware access. The direct I/O API supports only the polled mode of operation. Interrupt transfers must use the Service Request API. Direct I/O transfers also do not support any form of buffering. If buffering is desired, the Service Request API must be used.

The direct I/O API is very fast interface, providing the fastest access directly to the hardware registers. The standard Service Request can take anywhere from a few milliseconds to 10's of milliseconds. The direct I/O API can yield successive reads or writes on the order of a few microseconds. This can provide throughput up to three orders of magnitude faster than with a Service Request.

An additional advantage of the direct I/O API is that it permits Port C to be split in half using code. This allows for half the port to be configured for input, and the other half for output. When using the standard Service Request API, this type of configuration must be done from the DriverLINX Configuration Panel. Splitting Port C from code is not supported by the Service Request API.

Configuration

The direct I/O API emulates the I/O address map and protocols of mode 0 of the Intel 8255 chip. The 8255 standard provides three 8-bit ports of digital I/O, or 24 lines of I/O. Only 4 I/O addresses are required to control the 24 lines of digital I/O. This includes one control register and one 8-bit I/O address for each of the three groups of 8 DIO lines.

On a bus reset, or during equipment power up, all ports are configured as inputs. Before a port can be used for output, it must be reconfigured for output. This is done by writing a zero to the proper bit of the control register (base +3). The control register of each group of 24 lines must be written individually.

The following is the control register of the emulated 8255 mode 0.

Bit	Values
0	Port C lower: 1 = input, 0 = output
1	Port B: 1 = input, 0 = output
2	Not used, set to 0
3	Port C upper: 1 = input, 0 = output
4	Port A: 1 = input, 0 = output
5	Not used, set to 0
6	Not used, set to 0
7	Not used, set to 0

For example:

- 80 — All ports set as outputs
- 82 — Ports A & C set as output, Port B set as input
- 9B — All ports set as inputs

When using the direct I/O API, you need to know the base address of the card. Although we are not using the Service Request, the DriverLINX driver still handles the base address of the data acquisition card. Using the direct I/O API, you only need to specify the read/write operations at an offset. For example, the control register is

typically at an offset of 3 from the base. Using this API, you indicate the offset (three) and the underlying driver will know where the base address resides.

When using a card with more than 24 DIO lines, additional registers are used. The offset necessary will change to accommodate the additional lines. The following table shows the offset map for a 96 line card.

Offset	Function
0	Port A Group 0
1	Port B Group 0
2	Port C Group 0
3	Control register, Group 0
4	Port A Group 1
5	Port B Group 1
6	Port C Group 1
7	Control register, Group 1
8	Port A Group 2
9	Port B Group 2
10	Port C Group 2
11	Control register, Group 2
12	Port A Group 3
13	Port B Group 3
14	Port C Group 3
15	Control register, Group 3

NOTE *If you are using boards that have a different number of digital I/O lines, such as the PIO-32 series, PDSIO-8, or REL-16, please refer to the corresponding user's manual for I/O register maps.*

Direct I/O using Visual Basic

If your data acquisition card is capable of using the direct I/O API, it will be explained in the manual set provided with your card. Refer to the *Using DriverLINX With Your Hardware* manual for your data acquisition card.

Direct I/O using C/C++

When creating a C/C++ application using the direct I/O API, you must use the #import compiler directive to incorporate information from the object's type library as follows:

```
#import "c:\drvlinx4\common\kisapio.dll"  
using namespace KISAPIOLib;
```

Call `CoInitialize(NULL);` // to initialize the COM library

Next, use a COM Smart Pointer to create an instance of the object. Pass the GUID of the object to be created:

```
IISAPIOPtr pdio(_uuidof(KISAPIO));
```

After the object has been created, its methods and properties can be accessed as follows:

```
// pdio->Write(offset, value to write);  
pdio->Write(3,0x80); //Write to control register to configure ports  
pdio->Write(0,0xF0); //Write Port A value of F0
```

Sample console application

Here is a sample of a simple console (DOS box) application using C.

```
#include <iostream.h>  
#include <conio.h>  
#import "c:\drvlinx4\common\kisapio.dll"
```

NOTE *This program is written for an ISA board (kmbpio driver). If you are using a different type of digital I/O board you need to import the appropriate Type*

Library Name COM object. Refer to the table on [page 7-8](#). If you are using a Win32 Console Application in MS C++, the above #include lines must appear in the header file of your application.

```
using namespace KISAPIOLib;
```

NOTE *Use the appropriate namespace if your digital I/O board is not an ISA board (kmbpio driver). Refer to the table on [page 7-8](#).*

```
int main()
{
    unsigned char data;
    if (FAILED(CoInitialize(NULL)))
    {
        cout << "Unable to initialize COM" << endl;
        return 1;
    }

    try
    {
        // smart pointer declaration will create an
        // instance of the object.

        IISAPIOPtr pdio(__uuidof(KISAPIO));

        NOTE Use the appropriate interface name + “Ptr” (IISAPIOPtr etc.) if your digital I/O board is not an ISA board (kmbpio driver). Refer to the table on page 7-8.

        pdio->OpenDevice(0); //Open KMBPIO device
            // device number assigned in DLinux Config Panel

        pdio->Write(3,0x80); //write to control register;
            // make all outputs
        pdio->Write(0,0x0); //Write port A value of 00
        cout << " Port A all bits are logic low..." << endl;
        while( !kbhit() )
        {
        }
        getch();
    }
}
```

```

    pdio->Write(0,0x1); //Write port A value of 01
    cout << " Port A LSB is logic 1....." << endl;
    while( !kbhit() )
    {
    }
    data=pdio->Read(2); //read back port C
    cout << "Port C value = 0x" << hex <<
    static_cast<unsigned>(data) << endl;
    pdio->CloseDevice();
}
catch (const _com_error& Err)
{
    // Use a separate try/catch blocks around statements to make
    // error handling statement specific.
    cout << "Error using KMBPIO" << endl;
    cout << "Error number is 0x" << hex << Err.Error() << endl;
    cout << "Error message is " << Err.ErrorMessage() << endl;
    // Display error description if rich error info supported
    if (Err.ErrorInfo())
        cout << Err.Description() << endl;
    return 1;
}

// Closes the COM library on the current thread,
// unloads all DLLs loaded by the thread,
// frees any other resources that the thread maintains,
// and forces all RPC connections on the thread to close.
cout << "Closing COM" << endl;
CoUninitialize();
char    ch[8];
cout << "Hit <Enter> to exit..." << endl;
cin.get(ch, 8);
return 0;
}

```

GUI applications

When creating GUI based applications using C/C++, be sure to include support for Automation in the App Wizard. This will ensure the COM library will be properly initialized and uninitialized.

There are three separate COM objects, one for each of the three board types:

Board Type	Type Library Name	Interface Name	Namespace	Visual Basic Object Name
KPCI-PIO24, KPCI-PIO96	KPCIODIO.DLL	IKPCIPIO	KPCIDIOLib	KPCIPIO
ISA products covered by kmbpio driver	KISAPIO.DLL	IISAPIO	KISAPIOLib	KISAPIO
KPCI-3160	KDIGIO.DLL	IKDigitalIo	KDIGIOLib	KdigitalIo

8 Counter/Timer Programming

Introduction

Counter/Timer programming with DriverLINX can be a challenging task. The Service Request interface and dialog used for analog and digital I/O is reused for counter/timer operations. This provides a consistent interface with DriverLINX for all modes of programming. However, the way the Service Request properties were used for analog and digital I/O do not always match the way they are used in counter/timer operations. When programming counter/timer tasks, the *Counter/Timer User's Guide* is an invaluable reference for how the properties are used.

Task vs. group mode

DriverLINX supports two special modes for counter/timer operations: task-oriented and hardware-oriented. For common operations such as event counting or frequency measurement, task mode provides a quick and easy way to set up counter/timer functions. For more complex or flexible implementations, the counter/timer hardware can be programmed with a Group mode operation.

Task mode

Task mode provides DriverLINX with the ability to perform many common counter/timer tasks easily with generic procedures. This provides a way to program portable counter/timer tasks. Tasks supported by task mode include event counting, frequency and interval measurement, and frequency, pulse, and strobe generation.

When using task mode, and depending on the type of task requested, more than one channel of the counter/timer subsystem may be put into service. For example, 32 bit event counting with channel n will automatically also make use of channel $n+1$; the connection to cascade the two channels will be automatic and internal to the hardware.

Group mode

Group mode allows for the near simultaneous start or stop of multiple counter/timer channels. Due to system overhead, the channels may not be started or stopped at the exact same time. However, DriverLINX will perform the operation as rapidly as possible.

Step one of a group mode task would be to carry out a configure operation on each of the channels of interest. Refer to Hardware Reference in the Counter/Timer Programming Guide for how the properties of the Service Request are used to accomplish selection of a particular hardware mode of a C/T chip. Once all the channels are configured, step two is to use a start operation to allow each channel to begin to carry out their tasks.

Group mode tasks must be performed as either polled or interrupt mode tasks. When a group mode task is performed in polled mode, you must request the status of the task to retrieve the counter values. When status polling the task, DriverLINX uses a buffer to store the value of each channel in the group.

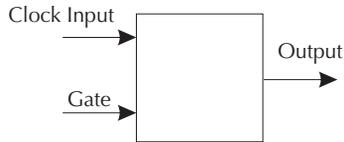
When operating in interrupt mode, DriverLINX will store each channel's current value in a buffer at the rate prescribed by the event timing parameters (`.Evt_Tim_rateXXX` or `m_pSR->timing.u.rateEvent.XXXX`). Knowing this 'sampling rate', software that analyzes the data can draw conclusions from the values. An application of this is time interval measurement. The accumulated counts on the first counter before the second counter starts to accumulate is a measure of the time interval.

When polling the status of an interrupt mode task, DriverLINX will return the current buffer number and the position of the next sample in that buffer.

Brief Hardware Review

Keithley data acquisition boards make use of three different counter/timer chips: the 82C54, the AM9513 or a proprietary chip that provides four 16-bit general-purpose counters and two 24-bit interval counters. There can be significant operational differences between these various chips. However, they all provide the same building block as shown in [Figure 8-1](#): a 16-bit counter with input, gate and output connections.

Figure 8-1
Basic counter elements



The input terminal can be connected to external pulse sources, or to the board's crystal timebase. The various boards provide widely different values of timebase. Some models permit the timebase to be further divided to a slower value without use of one of the available 16-bit counters to accomplish the division. Refer to [Table 8-1](#) for a list of some timebase values. Refer to the Using DriverLINX With Your Hardware manual specific to your data acquisition board for more information.

Table 8-1
Timebase values

Model	Timebase	Dividable?
KPCI-3108/KPCI-3107	10MHz	Yes
KPCI-3101/2/3/4/16/10	20MHz	No
KPCI-3140	40MHz	No
CTM-05/A & CTM-10	1 or 5MHz	Yes

In contrast to the analog or digital subsystems, counter/timer operations do not give notifications, or messages, about the measurement status. This poses the challenge to the overall application of coordinating starting the task with knowing when to read the result or stop the task. For example, with a gated event counting task, you can repeatedly read the value from the counter. When the value is no longer zero, the application can conclude that the measurement has started. When the value from the counter is no longer increasing in value, the application can conclude that the measurement is complete.

NOTE *The KPCI-3140 has 8 special purpose digital input lines. These lines can generate an interrupt on a change of state. Use of one of these lines in conjunction with other signals associated with the counter/timer task can greatly help in program flow coordination. These 8 special purpose lines can act together to detect an 8-bit pattern of interest, or can operate independently to monitor 8 individual signals.*

Event counting

Event counting is one of the simplest counter/timer tasks. The task counts the number of edges received at the clock input of the channel. DriverLINX can use 1, 2, or 4 channels for event counting. This provides a 16-, 32-, or 64-bit wide counter. Counters can be set up as repetitive or non-repetitive. Repetitive counters will wrap around to a count of zero and continue counting without an indication of overflow. Non-repetitive counters will wrap around to a count of 1 and stop.

This example demonstrates the following key concepts:

- Software command start and stop event — This task will start immediately when the Service Request is submitted, and continue until a stop event is received. This task will not stop automatically.
- Polled mode operation — This polled mode task operates in the foreground.
- No buffers used — The task returns its results by using the Res_Tim_count property of the Service Request.
- Repetitive counting — The Evt_Tim_ratePulses property has been set to 0. This results in a counter that wraps around when it hits the maximum count and keeps counting. A non-repetitive counter will stop when it hits the maximum count and go to a value of 1. Setting Evt_Tim_ratePulses to 1 will create a non-repetitive counter.
- 16-bit counting — The use of a single counter channel results in a 16-bit counter with a maximum value of 65,535. This is set by the Evt_Time_rateChannel property of the Service Request. Make sure that your counter uses enough channels that it does not wrap around unexpectedly.

Visual Basic

VB The following sample code demonstrates event counting.

```
With DriverLINXSR1
.Req_op = DL_START
.Req_mode = DL_POLLED
.Req_subsystem = DL_CT
.Evt_Str_type = DL_COMMAND
.Evt_Tim_type = DL_RATEEVENT
.Evt_Stp_type = DL_COMMAND
.Evt_Tim_rateClock = DL_EXTERNAL
.Evt_Tim_rateChannel = 1
.Evt_Tim_rateGate = DL_NOCONNECT
.Evt_Tim_rateMode = DL_COUNT
.Evt_Tim_rateOnCount = 0
.Evt_Tim_ratePeriod = 0
.Evt_Tim_ratePulses = 0
.Refresh
End With
```

```
With DriverLINXSR1
.Req_op = DL_STATUS
.Refresh
Text1.Text = .Res_Tim_count
End With
```

C/C++

C/C++ The following sample code demonstrates event counting.

```
m_pSR->operation=START;
m_pSR->subsystem=CT;
m_pSR->mode=POLLED;
m_pSR->start.typeEvent=COMMAND;
m_pSR->timing.typeEvent=RATEEVENT;
m_pSR->timing.u.rateEvent.channel=0;
m_pSR->timing.u.rateEvent.mode=COUNT;
m_pSR->timing.u.rateEvent.clock=EXTERNAL;
m_pSR->timing.u.rateEvent.gate=NOCONNECT;
m_pSR->timing.u.rateEvent.period=0; /*The period and onCount fields
    should be 0
```

```
m_pSR->timing.u.rateEvent.onCount=0;
m_pSR->timing.u.rateEvent.pulses=0; /* a 1 here would mean that the
    counter will not roll over at 65535*/
DriverLINX(m_pSR);
showMessage(m_pSR);

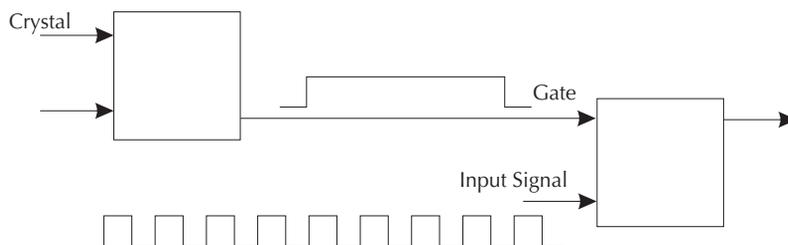
taskID=m_pSR->taskId;
m_pSR->operation=STATUS;
m_pSR->status.typeStatus=TIMERSTATUS;
DriverLINX(m_pSR);
showMessage(m_pSR);
m_ctvalue.Format("%d",m_pSR->status.u.timerStatus.count);
UpdateData(FALSE);
```

Frequency measurement

Frequency counting is very similar to event counting. However, rather than just count events, the counting is performed over a known period of time. The number of events that occur over that known period of time provide the frequency of the input signal.

This example uses an internal clock (timebase) and one or two channels to generate a pulse of known duration. This pulse is used as a gate signal for the event counting channel(s). See [Figure 8-2](#).

Figure 8-2
16-bit frequency measurement



NOTE *Task based frequency measurement is not implemented for boards that use the 82C54 chip (KPCI-3108 Series). This chip cannot generate a one-shot pulse that is used to control gating over a known time interval. Alternate hardware or an alternate software approach is required.*

Frequency measurement is accomplished using the frequency mode of the task based operations. This mode uses two or more channels to measure frequency. In order to perform frequency measurements, the output of the pulse generation channel must be externally connected to the gate of the event counting channel.

When using hardware that has a fast timebase and no ability to internally divide the timebase, more than one channel will be used to generate a gating pulse of sufficiently long duration for the frequency measurement. Refer to the Counter/Timer Programming Guide for more information on channel allocation for task mode operations.

For example, the KPCI-3140 has a 40MHz timebase. With this timebase, one 16-bit counter can generate a pulse no longer than 15 micro seconds in duration. Therefore, when a task mode frequency operation is requested of this board, DriverLINX will make use of two channels to generate the pulses. This impacts the physical connections required.

The channel specified by the `Evt_Time_rateChannel` property of the Service Request, indicates the first channel in the series of channels that will be used by the task. When using 16-bit event counting, only one channel is used for event counting. For 32-bit frequency counting, two channels are used to accomplish 32-bit event counting.

As with event counting, frequency measurements can be set up as repetitive or non-repetitive using the `Evt_Tim_ratePulses` property of the Service Request. This is only supported by the CTM series boards.

This example demonstrates the following key concepts:

- 16-bit frequency measurement — The `FREQ` mode sets the Service Request to perform 16-bit counting. For 32-bit counting, set the mode to `FREQ32`. 16-bit counting uses one channel, and 32-bit counting uses two channels. Depending on the hardware in use and the requested duration of the gating pulse, more than one channel may be used to generate the pulse.
- Frequency display — DriverLINX returns the result of the operation in the `Res_Tim_count` property of the Service Request. This value is the count, and not the frequency. The value must be divided by the gated period to determine the frequency of the input signal.

Visual Basic

VB The following sample code demonstrates measuring frequency.

```

With DriverLINXSR1
    .Req_op = DL_START
    .Req_mode = DL_POLLED
    .Req_subsystem = DL_CT
    .Evt_Str_type = DL_COMMAND
    .Evt_Tim_type = DL_RATEEVENT
    .Evt_Stp_type = DL_COMMAND
    'use this line for KPCI-3100
    '.Evt_Tim_rateClock = DL_INTERNAL1
    'use this line for CTM
    .Evt_Tim_rateClock = DL_SOURCE1
    .Evt_Tim_rateChannel = 0
    .Evt_Tim_rateGate = DL_NOCONNECT
    .Evt_Tim_rateMode = DL_FREQ
    .Evt_Tim_rateOnCount = .DLSecs2Tics(DL_INTERNAL1, 1#)
    .Evt_Tim_ratePeriod = 0
    .Evt_Tim_ratePulses = 1
    .Refresh
End With
showMessage DriverLINXSR1

With DriverLINXSR1
    .Req_op = DL_STATUS
    .Refresh
    Text1.Text = .Res_Tim_count
End With

```

C/C++

C/C++ The following sample code demonstrates measuring frequency.

```

m_pSR->operation=START;
m_pSR->mode=POLLED;
m_pSR->subsystem=CT;
m_pSR->start.typeEvent=COMMAND;
m_pSR->timing.typeEvent=RATEEVENT;
m_pSR->stop.typeEvent=TCEVENT;
//m_pSR->timing.u.rateEvent.clock=INTERNAL1; /*This is for KPCI-3100*/
m_pSR->timing.u.rateEvent.clock=SOURCE1; /*This is for CTM*/

```

```
m_pSR->timing.u.rateEvent.channel=0;
m_pSR->timing.u.rateEvent.gate=DISABLED;
m_pSR->timing.u.rateEvent.mode=FREQ;
m_pSR->timing.u.rateEvent.onCount=Sec2Tics(0,CT,INTERNAL1,1.0f);
m_pSR->timing.u.rateEvent.pulses=1;
DriverLINX(m_pSR);
showMessage(m_pSR);

m_pSR->operation=STATUS;
m_pSR->status.typeStatus=TIMERSTATUS;
DriverLINX(m_pSR);
showMessage(m_pSR);
m_read.Format("%d",m_pSR->status.u.timerStatus.count);
UpdateData(FALSE);
```

Pulse width measurement

Pulse width measurement is performed by connecting the unknown signal to the gate of a channel that is counting a known frequency. An internal clock is often used as the known frequency. The input signal is used as a level triggered gate signal. When the signal exceeds the gate threshold, counting begins. Counting stops when the signal falls below the threshold. The resulting count provides the pulse width of the signal in terms of the frequency applied to the counter's input.

This example demonstrates the following key concepts:

- Pulse width and interval measurement — This example demonstrates a simple method of measuring both a pulse width and an interval. This is controlled by the statement `If (Check1.Value = 1) Or if(m_Interval)`. If this condition evaluates as true, an interval measurement will be performed. Otherwise, a pulse width measurement will be performed.
- Single channel measurement — Pulse width is always measured using a single channel. Interval measurements can be taken on a single channel or between two channels, depending on the hardware features. This is controlled by the Pulses property of the Service Request. Setting this to a value of 0 causes both the starting and ending event to be taken on the same channel. Setting Pulses to 1 causes the measurement to be taken across two channels. The first channel, as specified by the Channel property, will be used to start the measurement. The next channel will be monitored for the stop pulse.

NOTE *When performing pulse width measurements, care should be taken to ensure the counter does not reach maximum value before the pulse is complete. For example, the KPCI-3101 uses an internal timebase of 20MHz. A single 16-bit counter will overflow in 3.27 msec (65535 / 20MHz). If the pulse of interest is longer than 3.27 msec, then the counter will roll over and begin counting up again from zero. No error message will occur. The implementation should take care to invoke the appropriate combination of timebase and number of counters to ensure a proper measurement. For example, use a gated 32-bit event counting task and interpret the data appropriately.*

Visual Basic

VB The following sample code demonstrates pulse width measurement.

```

With DriverLINXSRL
.Req_op = DL_START
.Req_mode = DL_POLLED
.Req_subsystem = DL_CT
.Evt_Str_type = DL_COMMAND
.Evt_Tim_type = DL_RATEEVENT
.Evt_Stp_type = DL_COMMAND
.Evt_Tim_rateClock = DL_INTERNAL1
.Evt_Tim_rateChannel = 0 'Change to CHAN1 for KPCI-3108=CTO
.Evt_Tim_rateGate = DL_ENABLED
If (Check1.Value = 1) Then
.Evt_Tim_rateMode = DL_INTERVAL
.Evt_Tim_ratePeriod = 0
.Evt_Tim_rateGate = HIEDGEGATEN
Else
.Evt_Tim_rateMode = DL_PULSEWD
.Evt_Tim_ratePeriod = 0
.Evt_Tim_rateGate = DL_ENABLED
End If
.Evt_Tim_rateOnCount = 0
.Evt_Tim_ratePulses = 0
.Refresh
End With
showMessage DriverLINXSRL 'Error Checking
' add the code below to a command button _
' or timer object to read the counter's value

```

```

With DriverLINXSr1
.Req_op = DL_STATUS
.Refresh
Text1.Text = .DLTicks2Secs(0,.Res_Tim_count) ' convert count to seconds
End With

```

C/C++

C/C++ The following sample code demonstrates pulse width measurement.

```

UpdateData(TRUE);
m_pSR->operation=START;
m_pSR->mode=POLLED;
m_pSR->subsystem=CT;
m_pSR->start.typeEvent=COMMAND;
m_pSR->timing.typeEvent=RATEEVENT;
m_pSR->stop.typeEvent=COMMAND;
m_pSR->timing.u.rateEvent.clock=INTERNAL1;
if(m_Interval)
{
    m_pSR->timing.u.rateEvent.mode=INTERVAL;
    m_pSR->timing.u.rateEvent.period=0;
    m_pSR->timing.u.rateEvent.gate=HIEDGEGATEN;
}
else
{
    m_pSR->timing.u.rateEvent.mode=PULSEWD;
    m_pSR->timing.u.rateEvent.period=0;
    m_pSR->timing.u.rateEvent.gate=ENABLED;
}
m_pSR->timing.u.rateEvent.pulses=0;
m_pSR->timing.u.rateEvent.onCount=0;
DriverLINX(m_pSR);
showMessage(m_pSR);
// add the code below to a command button
// or timer object to read the counter's value

m_pSR->operation=STATUS;
m_pSR->status.typeStatus=TIMERSTATUS;
DriverLINX(m_pSR);
showMessage(m_pSR);
m_reading.Format("%d",m_pSR->status.u.timerStatus.count);
UpdateData(FALSE);

```

Square wave generator

Setting up a square wave generator using DriverLINX is a simple task requiring only a few properties of the Service Request. The frequency of the square wave is determined by the `Evt_Time_ratePeriod` property of the Service Request. Note that the period must be converted to a hardware code. The example program performs this conversion using the DriverLINX `Secs2Tics` method. As in all cases, the unused properties of the Service Request should be set to zero. If desired, the square wave output may be edge or level gated. This sample does not use gating.

This example demonstrates the following key concepts:

- Software command start and stop event — This task will start immediately when the Service Request is submitted, and continue until a stop event is received. This task will not stop automatically. A stop routine is provided.
- Square wave output — Setting the Mode to SQWAVE tells DriverLINX to produce a square wave output on the channel specified by the Channel property. The longest period that can be generated by a square wave generator is equal to 65535 times the internal clock period.

Visual Basic

VB

The following sample code demonstrates a square wave generator. Place the code below in the click event of a command button.

```
With DriverLINXSR1
.Req_op = DL_START
.Req_mode = DL_POLLED
.Req_subsystem = DL_CT
.Evt_Str_type = DL_COMMAND
.Evt_Tim_type = DL_RATEEVENT
.Evt_Stp_type = DL_COMMAND
.Evt_Tim_rateClock = DL_INTERNAL1
.Evt_Tim_rateChannel = 0
.Evt_Tim_rateGate = DL_NOCONNECT
.Evt_Tim_rateMode = DL_SQWAVE
.Evt_Tim_rateOnCount = 0
.Evt_Tim_ratePeriod = .DLSecs2Tics(DL_INTERNAL1, 1 / 1000)
.Evt_Tim_ratePulses = 0
.Refresh
```

```

End With
showMessage DriverLINXSR1
' code below will stop the task
' place in second command button
With DriverLINXSR1
.Req_op = DL_STOP
.Refresh
End With

```

C/C++

C/C++ The following sample code demonstrates a square wave generator.

```

m_pSR->operation=START;
m_pSR->subsystem=CT;
m_pSR->mode=POLLED;
m_pSR->start.typeEvent=COMMAND;
m_pSR->timing.typeEvent=RATEEVENT;
m_pSR->stop.typeEvent=COMMAND;
m_pSR->timing.u.rateEvent.channel=0;
m_pSR->timing.u.rateEvent.mode=SQWAVE;
m_pSR->timing.u.rateEvent.pulses=0;
m_pSR->timing.u.rateEvent.clock=INTERNAL1;
m_pSR->timing.u.rateEvent.period=Sec2Tics(0,CT,INTERNAL1,0.001f);
m_pSR->timing.u.rateEvent.onCount=0;
m_pSR->timing.u.rateEvent.gate=NOCONNECT;
DriverLINX(m_pSR);
showMessage(m_pSR);
//Use the code below to stop the task
m_pSR->operation=STOP
DriverLINX(m_pSR);

```

One shot pulse generator

One shot pulse generators use two properties of the Service Request to specify the pulse properties. The application can specify the delay before the pulse occurs, and the duration of the pulse. The delay before the pulse occurs is set using the Period property of the Service Request. The OnCount property specifies the duration of the pulse.

This example demonstrates the following key concepts:

- Software command start and stop event — This task will start immediately when the Service Request is submitted, and continue until a stop event is received. This task will not stop automatically.
- Output — This task sets the output to start as high, and toggle to low when the pulse is sent. This is set using the Evt_Tim_rateOutput property of the Service Request.

Visual Basic

VB

The following sample code demonstrates a one shot pulse generator.

```
With DriverLINXSR1
    .Req_op = DL_START
    .Req_mode = DL_POLLED
    .Req_subsystem = DL_CT
    .Evt_Str_type = DL_COMMAND
    .Evt_Tim_type = DL_RATEEVENT
    .Evt_Stp_type = DL_COMMAND
    .Evt_Tim_rateClock = DL_INTERNAL1
    .Evt_Tim_rateChannel = 0
    .Evt_Tim_rateGate = DL_NOCONNECT
    .Evt_Tim_rateMode = DL_PULSEGEN
    .Evt_Tim_rateOnCount = .DLSecs2Tics(DL_INTERNAL1, 0.002)
    .Evt_Tim_ratePeriod = .DLSecs2Tics(DL_INTERNAL1, 0.001)
    .Evt_Tim_rateOutput = CT_Output_HiToggled
    .Evt_Tim_ratePulses = 1
    .Refresh
End With
showMessage DriverLINXSR1
```

C/C++

C/C++ The following sample code demonstrates a one shot pulse generator.

```
m_pSR->operation=START;
m_pSR->mode=POLLED;
m_pSR->subsystem=CT;
m_pSR->start.typeEvent=COMMAND;
m_pSR->timing.typeEvent=RATEEVENT;
m_pSR->stop.typeEvent=TCEVENT;
m_pSR->timing.u.rateEvent.clock=INTERNAL1;
m_pSR->timing.u.rateEvent.mode=PULSEGEN;
m_pSR->timing.u.rateEvent.gate=NOCONNECT;
m_pSR->timing.u.rateEvent.period=Sec2Tics(0,CT,INTERNAL1,0.001f);
m_pSR->timing.u.rateEvent.onCount=Sec2Tics(0,CT,INTERNAL1,0.002f);
m_pSR->timing.u.rateEvent.pulses=1;
m_pSR->timing.u.rateEvent.pulses |= OUTPUTHITOGGLED;
DriverLINX(m_pSR);
showMessage(m_pSR);
```

Time interval measurement

Time interval measurement can be a very tricky procedure to program. At the heart, measuring a time interval is similar to measuring pulse width. The unknown input signal is used to gate the counting of a known frequency generated by an internal clock. The input signal is gated using an edge sensitivity trigger for both the start and stop events. The count between the two edges is the time interval.

Time interval measurements can use one or two channels. Care must be taken when using two channels. If the first channel rolls over before the second channel detects the edge signifying a stop, the measurement will be invalid.

Time interval measurement can be very hardware dependent. The method used to program the Service Request will depend on the timer chip installed on the card (AM9515 or 82C54), and the specific model of the card. For specific information related to programming a specific chip, refer to the Am9513 or Intel 8254 Reference sections of the *Counter/Timer User's Guide*. For information regarding the specific features supported by an individual card, refer to the *Using DriverLINX With Your Hardware* manual for that specific card.

Listed below are some of the restrictions for time interval measurement:

- Under task-based operation, only the KPCI-3108 supports two channel time interval measurements.
- Under task-based operation, only the ISA CTM cards support single channel time interval measurements.
- The KPCI-3140 and KPCI-3100 series cards must always use group mode for time interval measurement.
- The KPCI-3140 and KPCI-3108 cards can use IRQ mode group tasks.
- The KPCI-3101/2/3/4 cards have no hardware IRQ. These cards must use polled mode group tasks read counter/timer channels.

For a discussion of group and task mode operation, refer to “[Task vs. group mode](#)” earlier in this section and see the following examples.

Visual Basic

VB

The following samples demonstrate a two channel task based time interval measurement written specifically for the KPCI-3108. This example demonstrates the following key concepts:

- Two channel measurement — The Pulses property of the Service Request is set to 1, specifying a two channel measurement. The first channel is specified by the Channel property, which is set to 1. The start pulse will be sensed on channel 1, and the stop pulse will be sensed on channel 2.
- 100kHz clock — The Clock property of the Service Request specifies that the interval should be timed using the Internal5 clock source. For the KPCI-3108 card, this clock is a 100kHz clock. On other cards, this clock may not be available, or may have a different frequency. If adapting this example for use on a different card, check for the availability and specifications of clock channels. This information is available in the appropriate *Using DriverLINX With Your Hardware* manual.

```
With DriverLINXSr1
.Req_device = 0
.Req_subsystem = DL_CT
.Req_op = DL_START
.Req_mode = DL_POLLED
.Evt_Tim_type = DL_RATEEVENT
.Evt_Tim_rateChannel = 1 ' start pulse to gate of this channel
```

```
.Evt_Tim_rateMode = DL_INTERVAL
.Evt_Tim_rateClock = DL_INTERNAL5
.Evt_Tim_rateGate = DL_ENABLED
.Evt_Tim_ratePeriod = 2 'stop pulse to gate of this channel
.Evt_Tim_ratePulses = 1 ' for separate start/stop pulses
.Evt_Tim_rateOutput = CT_Output_Default
.Evt_Str_type = DL_NULLEVENT
.Evt_Stp_type = DL_NULLEVENT
.Sel_buf_N = 0
.Sel_chan_N = 0
.Refresh
End With
' Add the code below to a command button to read the interval
DriverLINXSR1.Req_op = DL_STATUS
DriverLINXSR1.Refresh
Debug.Print DriverLINXSR1.Res_Tim_count
```

The following samples demonstrate a two channel group based time interval measurement written specifically for the KPCI-3140. This example demonstrates the following key concepts:

- Two channel measurement — This example uses two separate channels to measure the time interval. The time interval is measured by using the input signals to start two separate timer channels. The values of the two channels are monitored to determine the status of the interval measurement.
- Channel configuration — To set the channels for group mode, they are configured using `Req_op = DL_CONFIGURE`. This allows us to configure each individual channel as desired, but not start the task when the Service Request is submitted. The first two Service Requests submitted by this example configure the two channels that will be used. The third Service Request will start both channels at the same time from a start/stop list.
- Initial timer counts — Both timer channels are set to an initial value of 2 using the `Evt_Time_ratePeriod` property of the Service Request. This sets both timers to a known value when the Service Request is submitted. This will later be used to evaluate the data returned by the task.
- Group mode operation — Group mode operation is selected by configuring both channels, and then starting them from a start/stop list. All channels in the list will be started at as close to the same time as DriverLINX can start them.

The delay between the channels depends on the hardware being used. Some hardware configurations support simultaneous starts of multiple channels.

- Buffers and buffer filled notification — The group mode task uses interrupt mode to read the timer values at a rate of 1000Hz into a buffer. Two buffers are used to allow the filling of one buffer while one buffer is processed by the application. When a buffer is filled, a buffer filled message will be generated. When the buffer filled message is received, the buffer results are transferred into a local array using the VBArryBufferXfer method.
- Data processing — The data in the array is processed into three separate lists. List1 contains the data in the start channel. List2 contains the data in the stop channel. List3 contains the difference between the two channels. Data in the lists is evaluated as follows:
 - If the data in List3 is 0, then the timers have not yet started.
 - If the data in List3 is greater than 0, then a start event was received and the first timer channel has started.
 - If the data in List3 is negative, then a stop event was received, starting the second timer channel. However, no start event was received to start the first timer, or it was received after the stop event. This would indicate an error condition.
 - If the last data item in List2 is not greater than 2, then the stop event has not yet been received. The value of the second timer channel is still the value that was set by the configuration Service Request.
 - If the last data item in List2 is greater than 2, then both timers have started. Both the start and stop events were received.
- Stopping the task — When the buffer filled routine determines that both the start and the stop event have been received, the application displays a message that the time interval message is complete, and displays the time interval. A fourth Service Request is then submitted to stop the timers.

```
' step 1: configure
With DriverLINXSRI
.Req_op = DL_CONFIGURE 'configure counter 0 for group operation
.Req_mode = DL_OTHER
.Req_subsystem = DL_CT
.Evt_Tim_type = DL_RATEEVENT
.Evt_Tim_rateChannel = 0 'Use counter 0
.Evt_Tim_rateClock = DL_EXTERNAL ' apply TTL clock signal to input of
                                counter0
.Evt_Tim_rateGate = HIEDGEGATEN
```

```

.Evt_Tim_rateMode = DL_PULSEGEN    ' mode 0
.Evt_Tim_rateOnCount = 2    ' minimum value
.Evt_Tim_rateOutput = CT_Output_Default
.Evt_Tim_ratePeriod = 65533 'initial value will be 2 when counter is read
.Evt_Tim_ratePulses = 0
.Refresh
End With
showmessage DriverLINXSR1

With DriverLINXSR1
.Req_op = DL_CONFIGURE 'configure counter 1 for group operation
.Req_mode = DL_OTHER
.Req_subsystem = DL_CT
.Evt_Tim_type = DL_RATEEVENT
.Evt_Tim_rateChannel = 1
.Evt_Tim_rateClock = DL_EXTERNAL    ' apply same TTL clock signal to input of
                                     counter1

.Evt_Tim_rateGate = HIEDGEGATEN
.Evt_Tim_rateMode = DL_PULSEGEN
.Evt_Tim_rateOnCount = 2
.Evt_Tim_rateOutput = CT_Output_Default
.Evt_Tim_ratePeriod = 65533
'initial value will be 2 when counter is read
.Evt_Tim_ratePulses = 0
.Refresh
End With
showmessage DriverLINXSR1

' the oneshot mode requires both a period and an onCount value.
' the minimum onCount value the driver will accept is 2
' this leaves no more than 65533 for the Period
' when the gate edge is detected, the counter starts from value of
' onCount and will count up to max value specified by Period

' step 2: set up IRQ mode task to read the C/T
With DriverLINXSR1
.Req_op = DL_START
.Req_mode = DL_INTERRUPT
'this will be an interrupt-driven application
.Req_subsystem = DL_CT
.Evt_Tim_type = DL_RATEEVENT
.Evt_Str_type = DL_COMMAND

```

```

.Evt_Stp_type = DL_COMMAND
.Evt_Tim_rateChannel = 8
' one of the 24bit interval timers that can be used for pacing
.Evt_Tim_rateClock = DL_INTERNAL1 '
.Evt_Tim_rateGate = DL_DISABLED
.Evt_Tim_rateMode = DL_RATEGEN      'Use the rate generator mode

' pick a rate at which to read the counters
' this one set up for 1000Hz rate
.Evt_Tim_ratePeriod = .DLSecs2Tics(.Evt_Tim_rateChannel, 1 / 1000!)
.Evt_Tim_ratePulses = 0
.Evt_Tim_rateOnCount = 0
.Sel_chan_N = 2 'Read both counters on interrupt
.Sel_chan_start = 0
.Sel_chan_stop = 1
.Sel_chan_format = DL_tNATIVE
.Sel_chan_simultaneousScan = True
'read both counters at the same time
.Sel_buf_N = 2 'Using two buffers so one can be read if the other
' is being written to
.Sel_buf_samples = 1000 'take 500 readings from each counter
' take care not to have the combination of ratePeriod
' and buf_samples such that the buffer_filled messages come
' too fast to process. Windows' messaging is slow.

.Sel_buf_notify = DL_NOTIFY 'Enable buffer filled message
.Refresh
End With
showmessage DriverLINKSR1

' step 3: Buffer Filled messages arrive. Analyze the data
Private Sub DriverLINKSR1_BufferFilled(task As Integer, device As Integer,
    subsystem As Integer, mode As Integer, bufIndex As Integer)
    Dim i As Integer
    Dim timeDiff As Integer
    Dim localbuf(1000) As Integer 'Local buffer to hold readings
    Dim result As Long

    List1.Clear    'List box controls to display the data
    List2.Clear
    List3.Clear
    result = DriverLINKSR1.VBArrayBufferXfer(bufIndex, localbuf,

```

```
DL_BufferToVBAArray)
' When buffer fills, transfer to array

For i = 1 To 40 Step 2
List1.AddItem localbuf(i - 1)
'even number index is the start channel
List2.AddItem localbuf(i)
' odd number index is the stop channel
timeDiff = localbuf(i - 1) - localbuf(i)
List3.AddItem timeDiff
Next i

' test the last values in the buffer
If timeDiff = 0 Then lblStatus.Caption = "timers not yet started"
If timeDiff > 0 Then lblStatus.Caption = "timer has started"
' if stop channel is no longer at initial value of 2, then assume
' the stop pulse has occurred and the reading is valid

If localbuf(39) > 2 Then
lblStatus.Caption = " time interval measurement is complete"
Text1.Text = timeDiff 'Display buffer contents.
Command4_Click ' call stop the counter
End If

' if list3 contains negative numbers then the stop channel got it's
' pulse before the start channel did.
' when both the list1 and list2 no longer contain the initial value
' (2 in this case) then the difference between them is indicative
' of the time interval.
' the time interval is the difference in count multiplied by the
' time base.
' for my testing, I used external time base of 1Hz and manually
' applied signal to the gates of the counter channels to start them
' stop channel must get it's pulse before start channel reaches
' maximum count

' step 4: Stop the IRQ mode task
Private Sub Command4_Click()
With DriverLINXSR1
.Req_op = DL_STOP
.Refresh
End With
```

```
lblStatus.Caption = " Timers are stopped"  
End Sub
```

C/C++

C/C++ Due to the complex nature of performing interval measurement under C/C++, no sample programs are provided. If you need assistance with interval measurement, please contact Keithley Instrument's customer support.

9 Troubleshooting

Introduction

There are several ways to troubleshoot DriverLINX applications. The LearnDL application can be used to test the set up of a Service Request. The Edit Service Request dialog can be used to display the DriverLINX interpretation of a Service Request that has been created by an application. DriverLINX also provides an extensive error detection and reporting scheme.

LearnDL application

About LearnDL

The LearnDL application is a very useful tool for troubleshooting. This application can be used to troubleshoot hardware problems, driver problems, and Service Request property setup. When DriverLINX is installed, a shortcut to LearnDL is installed in the Utilities folder under the DriverLINX program group.

LearnDL implements a simple digital storage oscilloscope. The application was created using C++ and DriverLINX. In this way, LearnDL serves as a real example of what can be accomplished with DriverLINX, a powerful tool for diagnosing problems, a way to experiment with the different portions of the Service Request, and a tool for exploring the capabilities of the data acquisition hardware.

LearnDL is set up to utilize the full capabilities of the installed hardware. Anything that can be done with LearnDL can be done using the DriverLINX driver through the supported programming languages. If an application is generating errors when executing a Service Request, that same Service Request can be set up and executed with LearnDL. This helps determine if the error is in the Service Request setup or the application. If LearnDL returns an error, the problem is most likely in the way the Service Request is set up. LearnDL can then be used to try different Service Request properties to determine a correct setup for the Service Request. If LearnDL can execute the Service Request with no errors, then the problem is most likely located elsewhere in the application.

Using LearnDL

LearnDL uses a process very similar to that used by a data acquisition program. LearnDL forces you to perform the same steps in the same order as you would when programming. This can be useful when programming a data acquisition task. Walk through the task with LearnDL, recording the steps you use to complete the task. You then have a record of what needs to be done in your application to perform that very same task.

When setting up Service Requests, LearnDL uses the Edit Service Request dialog described in [Section 4, Learning DriverLINX](#). The dialog is an interactive method of filling out a Service Request. As choices are made, the dialog changes to show the appropriate controls available for the current selections. Controls are added and removed in response to selections. The Edit Service Request dialog will only show controls and choices valid for the selected device.

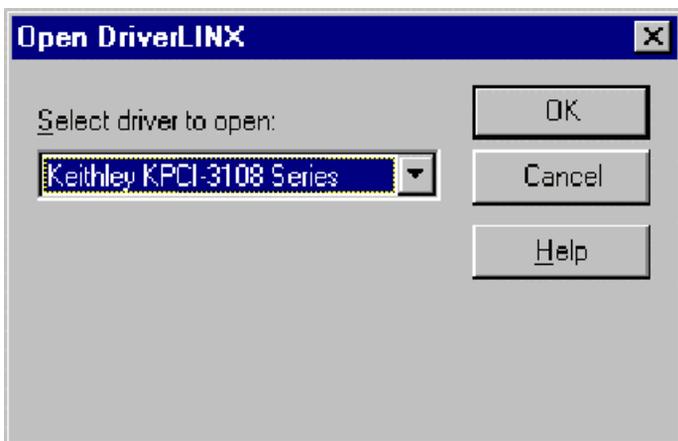
Preparing LearnDL

The following procedure provides the necessary steps to prepare LearnDL to perform a data acquisition task.

1. Start the LearnDL application by selecting the LearnDL icon from the utilities folder in the DriverLINX program group.
2. When DriverLINX starts, it will display the Open DriverLINX dialog to select a driver, as shown in [Figure 9-1](#). Pick the driver from the drop down box that corresponds to the appropriate hardware. If only one data acquisition board is installed, the appropriate driver should be displayed when the dialog appears.

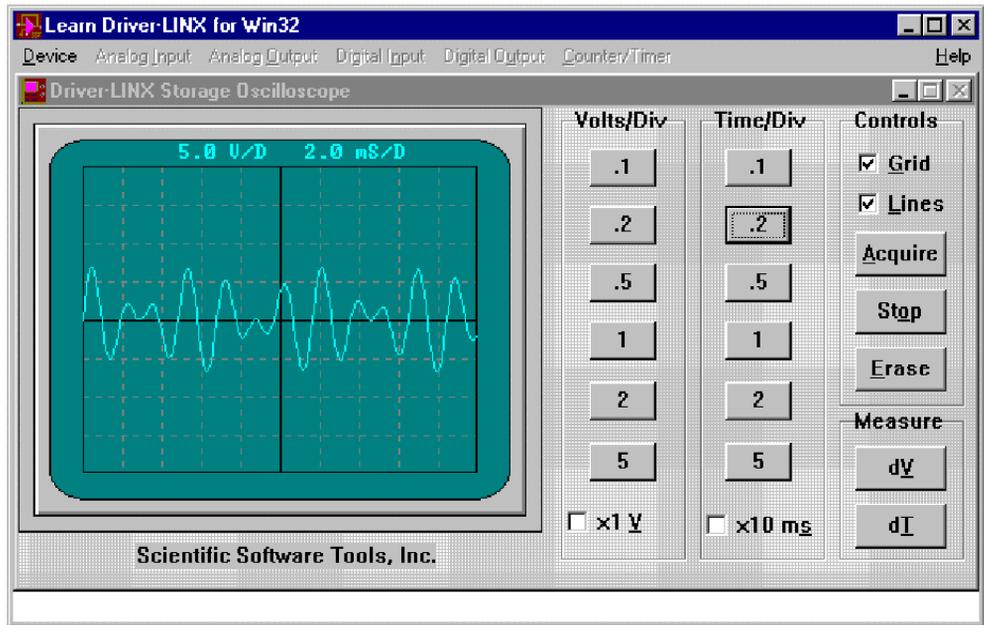
Figure 9-1

OpenDriverLINX dialog



3. After the driver had been selected, the LearnDL main window will appear, as shown in Figure 9-2. The window displays the digital oscilloscope on the left side, and the oscilloscope controls on the right. The menu bar along the top lists all the available subsystems. Note that all menu items except the Device menu are disabled. This is because the device has not yet been selected or initialized.

Figure 9-2
LearnDL main window



4. After the main window appears, the first thing to do is to select the data acquisition board to use with LearnDL. To select the board, click on the Select item under the Device menu. Note that this is the only selection on the device menu that is enabled. The other items will be enabled after a device has been selected. The Select Device dialog shown in [Figure 9-3](#) will appear.

Figure 9-3
Select device dialog



5. From the Logical device drop down box, select the logical device number to be used. If the system has only one data acquisition board installed, the device will normally be device 0. Click the OK button to return to the main window.
6. After the device has been selected, it must be initialized. Click the Device menu to open it. Note that the other selections, Configure and Initialize, are now enabled. Click on Initialize to initialize the selected device.
7. When the initialization process is complete, the menu items at the top of the LearnDL window will be enabled.

LearnDL is now ready to perform a data acquisition task. The next topic will perform a data acquisition task.

Acquiring data with LearnDL

After LearnDL has been prepared for a data acquisition task by selecting and initializing the device, it is ready to acquire data. The following procedure will read a single analog value from analog channel 0.

1. The first step in performing data acquisition with LearnDL is to initialize the subsystem. To initialize the analog input subsystem, select Initialize from the Analog Input menu. Note that Initialize is the only option available from the Analog Input menu. The other options will become available after the subsystem is initialized.
2. After the subsystem has been initialized, the remaining items on the analog input menu will be available. To set up the Service Request to perform the task, select the Edit/execute item from the Analog Input menu. the Edit Service Request dialog, shown in [Figure 9-4](#) will appear.

Figure 9-4
Edit Service Request dialog



3. Set the Service Request properties to the following values:
 - Device: Device 0
 - Subsystem: Analog input
 - Mode: Polled
 - Operation: Start
 - Timing: None
 - Start: Command
 - Stop: Terminal count
 - Channel: Single, 0
 - Gain: 1B
 - Format: Native
 - Buffers: 1
 - Samples: 1
4. When all properties are entered, click the OK button. This will submit the Service Request to DriverLINX for execution. When the task is complete, control will be returned to LearnDL, and the display will be updated with the results.
5. To run the task again, select Start from the Analog Input menu.

Refer to [Section 4](#), Displaying the Edit Service Request Dialog from an Application, for information on interactively using the LearnDL tool from your own code.

Error messages

DriverLINX contains a comprehensive set of error detection and reporting features. Any errors will be reported with an error code, message, and severity level. The majority of error codes are standard across all DriverLINX versions. All standard error codes, along with explanations, are listed in the *DriverLINX Technical Reference Manual* in Appendix A: Error Messages, and in the *DriverLINX/VB Technical Reference Manual* in Appendix A: Error Reporting. A list of all the error codes for the specific installed version of DriverLINX is contained in the files DLCODES.TXT and DLCODES.H, located in the \DrvLINX4\DLApi directory.

Displaying error messages

DriverLINX provides a built-in utility to display errors. This utility will present a modal dialog box, shown in [Figure 9-5](#). The dialog box is displayed with VB by setting the Req_op property of the Service Request to DL_MESSAGEBOX and using the Refresh method on the Service Request. For C/C++, the dialog is displayed by setting the operation property of the Service Request to MESSAGEBOX and calling the Service Request. If there are currently no errors, then the message box will not be displayed. If the dialog is displayed, it must be acknowledged by the user before the application will continue.

Figure 9-5
DriverLINX modal dialog



Visual Basic

VB The following sample code can be used to display the DriverLINX message box.

```
With DriverLINXSR1
    .Req_Op = DL_MESSAGEBOX
    .Refresh
End With
```

C/C++

C/C++ The following sample code can be used to display the DriverLINX message box.

```
void CmyDlgClassDlg::showMessage(DL_ServiceRequest *SR)
{
    SR->operation=MESSAGEBOX;
    DriverLINX(SR);
}
```

Common error messages

There are several error messages that are commonly seen by users when programming with DriverLINX. The table below lists these error messages, and some possible corrective actions.

Error Number	Error Name	Corrective action
25	Invalid Logical Device	The device number used to initiate the Service Request is invalid. Check the device number assigned to the data acquisition board using the DriverLINX Configuration Panel.
30	Invalid Subsystem	The subsystem specified by the Service Request is not valid for the specified hardware. This error means that the subsystem being used, for example an analog output channel, does not exist on the data acquisition board.
120	Device Busy	This error means that the subsystem requested by the Service Request is already in use by another task. The task may have been started by the current application or another application.
165	Channel/Gain list has too few/many entries	When programming a channel/gain list, you must specify the number of channels that will be in the list. If the required number of channels are not provided in the list, this error will be generated.
170	Invalid channel in channel/gain list	The channel/gain list contained an invalid channel. This often happens with digital output tasks where the channel has not yet been configured for digital output. Ensure that all the required channels have been configured as output channels before specifying the channel/gain list.
210	Buffer does not hold integral # of scans	Buffer sizes must be an even multiple of the number of channels involved in the task. For example, if you have three channels, the number of samples a buffer can hold must be a multiple of three.
515	Pulse on-count > specified period	When using a burst mode task, the burst duration is specified by the onCount property. This duration must be less than the overall period, or repeat rate, for the task. The burst duration must be short enough that it can be completed before it has to be started again.
565	Data Lost	This error indicates that some data from the task has been lost. This can occur due to a FIFO error or a memory buffer overflow. Ensure that the task is not generating messages at a high rate, and that the buffers are large enough to allow the application time to process the buffers before they need to be reused by the task.

Interpreting error messages

Creating an application that intelligently handles error messages can greatly increase the versatility of an application. Rather than simply pop up a cryptic error message, the application can detect the error, determine what the error means, and then take appropriate action.

For example, consider a task that acquires data over a long period of time, such as a few days. This task may be more interested in acquiring as much data as possible over that time, rather than acquiring an unbroken string of data. If the task were to experience a buffer overrun during the night, the DriverLINX message box function would pause data acquisition until an operator acknowledged the error. If the application was programmed to detect the error, it could take corrective action. The application could restart the task, e-mail an error message to an operator, or take any other corrective action. Intelligent error handling in this case maximizes the amount of data captured.

Decoding error messages

Intelligently handling errors requires that the application have a way of identifying what error occurred. This is easily accomplished with VB. Decoding the errors with C/C++ takes a little more work.

Visual Basic

VB When programming with Visual Basic, DriverLINX returns an error code with every Service Request. The `.Res_result` property of the Service Request returns an error code that corresponds to the error codes listed in Appendix A of the *DriverLINX/VB Technical Reference Manual*. If the error string is desired, the `DriverLINXSR1.Message` property will return the error message string.

C/C++

C/C++ Decoding an error message using C/C++ is a little more complicated than with Visual Basic. The following sample code can be used to take action if an error condition is present.

```
if( DriverLINX( m_pSR ) != NoErr )
{
    // the corrective action here based on value of m_pSR->result
}
```

The result property of the Service Request contains both the error code and the error string. The following sample code can be used to separate the error code and the error string. The error codes and strings can be found in Appendix A of the *DriverLINX Technical Reference Manual*.

```
char errString[100];
DWORD size=50;
int errNumber;

errNumber = getErrCode(SR->result);
// errNumber corresponds to DriverLINX documentation
ReturnMessageString(SR->hWnd, SR->result, errString, size);
// errString corresponds to DriverLINX documentation
```

Responding to error messages

There are several ways that an application can respond to a message from DriverLINX. The following examples illustrate some of the ways this can be accomplished.

Visual Basic

VB Visual Basic provides an easy method to create the framework code necessary to react to DriverLINX messages.

1. From the top of the code window, use the Object drop down list. Select the DriverLINXSR1 object.
2. Use the Procedure drop down list to select the BufferFilled procedure. The following framework code will be added to your application.

```
Private Sub DriverLINXSR1_BufferFilled(task As Integer, device As integer,
    subsystem As integer, mode As Integer, bufIndex As Integer)
    ' add code here
End Sub
```

Using this same procedure, code can be added to any of the DriverLINX messages (data lost, etc.). Code can be added to restart the task, provide user feedback, or call for help.

C/C++

C/C++ The following code can be attached to the message pump to detect and respond to DriverLINX messages.

```
LRESULT CAIbufferDlg::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
    if(message==m_DLmsg) // Is this a DriverLINX message
    {
        switch(wParam)
        {
            case DL_BUFFERFILLED:
                // Place code to process the data here
                break;
            case DL_STOPEVENT:
                // Place the code to process the STOPEVENT here
                break;
            case DL_DATALOST:
                // Place code to restart the task or call for help here
                break;
        }
    }
    return CDialog::WindowProc(message, wParam, lParam);
}
```

Errors with Windows messaging

In a multi-tasking environment, the way that Windows handles messages can cause problems if they are not handled properly. Improper handling can cause data loss, or even system instability and crashes.

One situation that can cause problems is the amount of time that it takes for Windows to give control to an application so it can process messages. Certain system functions, such as accessing a floppy drive that does not have a disk in it, have very high time-outs. This can easily delay giving control to an application long enough for FIFO overflows to occur. There is little that a programmer can do to prevent this from occurring. Intelligent error handling can be used to recover from situations like this by logging the event and restarting the task.

A common situation encountered by new programmers is flooding the message queue. When the buffers used for a task are too small, they are filled at a very rapid rate. The

resulting stream of BufferFilled message can rapidly overwhelm the system. The latencies inherent in the Windows messaging system, which normally handles such events as mouse clicks, can cause the messages to not be delivered reliably enough to process the data from very small buffers. As the messages stack up, the buffers are not processed. Eventually there are no more buffers available to the task and a DataLost message will result.

If messages are being generated at a very high rate, the GUI will become unresponsive to mouse clicks. This is due to the fact that the BufferFilled messages have a higher priority than mouse events. The rapid flow of BufferFilled messages prevents the mouse clicks from being processed. The GUI screen will not update, as it has the lowest priority of all messages. Eventually, the system will crash.

To avoid flooding the Windows message queue, ensure that enough buffers have been allocated, and that they are of sufficient size. Buffers should be large enough to hold at least one second of data. Message rates in excess of 10 msec, or 100 Hz, cannot be sustained for more than a few minutes. For long term sustained data processing, a message rate of 100 msec, or 10 Hz, should be used. Note that this is not the data acquisition rate, but the rate at which the task generates messages to be processed by the application.

Specifications are subject to change without notice.

All Keithley trademarks and trade names are the property of Keithley Instruments, Inc. All other trademarks and trade names are the property of their respective companies.



Keithley Instruments, Inc.

28775 Aurora Road • Cleveland, Ohio 44139 • 440-248-0400 • Fax: 440-248-6168
1-888-KEITHLEY (534-8453) www.keithley.com

BELGIUM: Keithley Instruments B.V.
CHINA: Keithley Instruments China
FRANCE: Keithley Instruments Sarl
GERMANY: Keithley Instruments GmbH
GREAT BRITAIN: Keithley Instruments Ltd.
INDIA: Keithley Instruments GmbH
ITALY: Keithley Instruments s.r.l.
KOREA: Keithley Instruments Korea
NETHERLANDS: Keithley Instruments B.V.
SWITZERLAND: Keithley Instruments SA
TAIWAN: Keithley Instruments Taiwan

Bergensesteenweg 709 • B-1600 Sint-Pieters-Leeuw • 02/363 00 40 • Fax: 02/363 00 64
Yuan Chen Xin Building, Room 705 • 12 Yumin Road, Dewai, Madian • Beijing 100029 • 8610-6202-2886 • Fax: 8610-6202-2892
3, allée des Garays • 91127 Palaiseau Cédex • 01-64 53 20 20 • Fax: 01-60 11 77 26
Landsberger Strasse 65 • D-82110 Germering • 089/84 93 07-40 • Fax: 089/84 93 07-34
Unit 2 Commerce Park, Brunel Road • Theale • Reading • Berkshire RG7 4AB • 0118 929 7500 • Fax: 0118 929 7519
Flat 2B, WILLOCRISSE • 14, Rest House Crescent • Bangalore 560 001 • 91-80-509-1320/21 • Fax: 91-80-509-1322
Viale San Gimignano, 38 • 20146 Milano • 02-48 39 16 01 • Fax: 02-48 30 22 74
2FL., URI Building • 2-14 Yangjae-Dong • Seocho-Gu, Seoul 137-130 • 82-2-574-7778 • Fax: 82-2-574-7838
Postbus 559 • NL-4200 AN Gorinchem • 0183-635333 • Fax: 0183-630821
Kriesbachstrasse 4 • 8600 Dübendorf • 01-821 94 44 • Fax: 01-820 30 81
1 FL., 85 Po Ai Street • Hsinchu, Taiwan, R.O.C. • 886-3-572-9077 • Fax: 886-3-572-9031