Using DriverLINX® Drivers with LabWindows/CVI™

by
Dave Sherman and Mike Bayda
Keithley Instruments, Inc.

Introduction

National Instruments' LabWindows/CVI is a C programming environment for developing data acquisition (DAQ) and instrumentation applications known as Virtual Instruments (VIs). This document will show how to get LabWindows/CVI and DriverLINX communicating properly and explain the steps towards a successful linkage between both products. This will be done through a simple example program that is designed for someone who is using DriverLINX and LabWindows/CVI together for the first time. This downloadable example program accompanies this application note. The user should be familiar with LabWindows, C programming, DriverLINX, and interfacing to DAQ hardware.

In this example, LabWindows/CVI version 5 is used with the Keithley KPCI-PIO24 DAQ board and DriverLINX. The example will open the driver (DriverLINX), initialize the device (KCPI-PIO24), configure port A on a KPCI-PIO24 for digital output, and perform a digital output task on port A.

CVI Compatibility Issues

The Visual C/C++ examples provided on Keithley's DriverLINX CD take great advantage of the C++ development tools, procedures, and syntax. DriverLINX C/C++ examples are based on Microsoft® Foundation Classes (MFC), which is the standard development environment in Visual C/C++.

Although CVI claims to be backward compatible with Visual C/C++, CVI is only compatible with ANSI C, not C++. Therefore, the examples on the DriverLINX CD will not work intuitively with CVI.

DriverLINX ActiveX® controls cannot be imported into CVI because CVI is not an ActiveX container (Visual C/C++ is), so CVI must use the DriverLINX DLL API (application programming interface). Also, CVI's function panel windows are not supported by DriverLINX. Function panels help generate and test function calls within LabWindows/CVI.

Installing CVI

When installing CVI, select Visual C/C++ compatibility. If your CVI package was installed with a different compiler option, such as Borland or others, you should re-install CVI and select the Visual C/C++ option.

Building and Configuring a Project in CVI

- When you launch the "CVI IDE" (Integrated Development Environment) you should see a screen similar to the one shown in *Figure 1*. You will need to add source code (*.c), header files (*.h), and libraries (*.lib) to your project, just as you would with a Visual C/C++ project.

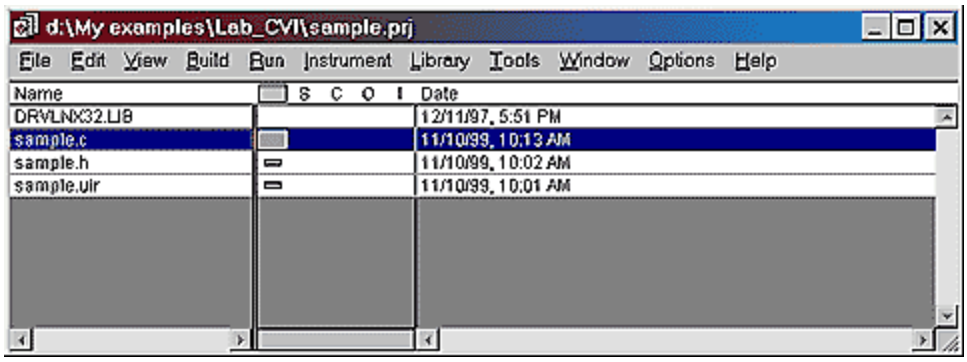

Figure 1: CVI IDE Environment

- We will use four files in this example project:

DRVLNX32.LIB	The DriverLINX library needed to link with the DLL API
sample.c	The CVI source code example, which is described below
sample.h	The CVI header file, which is created by the CVI IDE
sample.uir	The CVI graphical user interface panel displayed below

- DriverLINX is Visual C/C++ compatible, so it uses the C identifier "_MSC_VER" in its header file for preprocessor conditional compilation. This C identifier is embedded in Visual C/C++. However, it is not defined in CVI. You need to define it in CVI by adding "/D_MSC_VER=0" in the Compiler Defines window, as shown *in Figure 2*. (You can display the Compiler Defines window by selecting it from the Options menu).

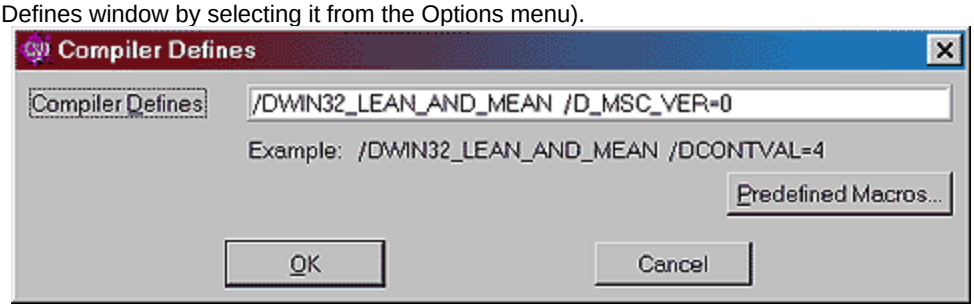

Figure 2: Compiler Defines Window

This CVI "Compiler Defines" command line is passed to the compiler that defines identifiers as macros to the preprocessor.

If "/D_MSC_VER=0" is not added to the project's "Compiler Defines," the program will compile and link properly, but it will not run. This is due to byte alignment in the Service Request (SR) structure defined by DriverLINX. When porting any C data structure from one compiler to another, the fields of the structure must have the same alignment in both compilers and they must occupy the same storage space. By adding this C identifier to CVI, you are informing DriverLINX that CVI is compatible with Microsoft Visual C/C++.

  - NOTE: The identifier "/DWIN32_MEAN_AND_LEAN," shown in *Figure 2*, is the default CVI macro that reduces the time and memory taken when compiling Windows® include files.

- In addition to the standard CVI header files and declarations, you need to include the following in the sample.c file:

```
   #include <windows.h>              // DriverLINX uses definitions from this header fi
#include "c:\drvlinx4\dlapi\drvlinx.h"  // DriverLINX header provided on your system
#include "c:\drvlinx4\dlapi\dlcodes.h"  // DriverLINX header provided on your system

HWND window;                          // Window handle needed for DriverLINX
```

```
HINSTANCE driverInstance;              // A Driver Instance needed to Open DriverLINX
DL_SERVICEREQUEST *pSR; // A Service Request pointer to communicate with DriverLINX
```

"DL_SERVICEREQUEST" is the Service Request structure type defined in "drvlinx.h."

- The user interface in *Figure 3* shows that this example program is divided into four steps. Each step is executed when its associated command button is pressed. The following sections are the commented code for each of these steps.

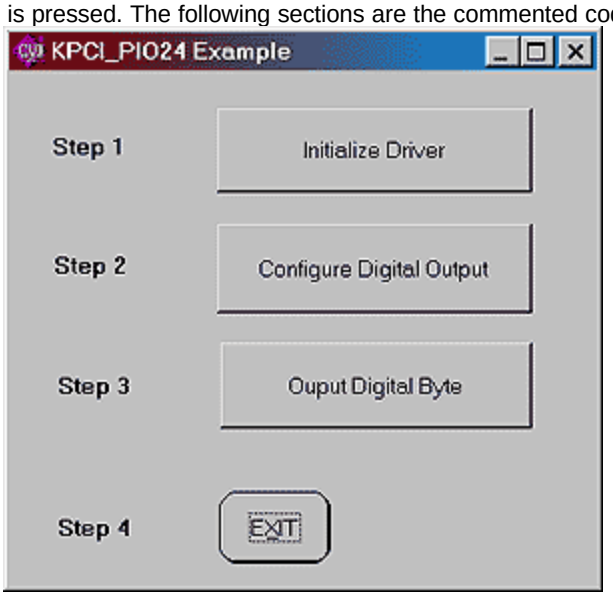

Figure 3: Sample.uir User Interface

Step 1. Initialize DriverLINX

```
int CVICALLBACK Initialize (int panel, int control, int event, void *callbackData, int eventD
{
switch (event)

{
case EVENT_COMMIT:
        window=FindWindow(NULL,"KPCI_PIO24 Example");
//Get the Window handle from the CVI panel
        driverInstance=OpenDriverLINX(window,"");
//Pass the window handle to open the driver
                pSR=(DL_SERVICEREQUEST*)malloc(sizeof(DL_SERVICEREQUEST));
//Allocate space for the Service Request
        memset(pSR,0,sizeof(DL_SERVICEREQUEST));
//Initialize the Service Request
        DL_SetServiceRequestSize(*pSR);   //Setup the Service Request for usage
        pSR->hWnd=window;                    //Set the Widows handle property
        pSR->device=0;              //Set the device property, in this case it is device 0
        pSR->subsystem=DEVICE;
//We are commnicating with the DEVICE, not AI, AO, etc.
pSR->mode=OTHER;             //The mode is OTHER, not DMA or INTerrupt
        pSR->operation=INITIALIZE;   //The operation is to INITIALIZE the driver
        DriverLINX(pSR);               //Execute the Service Request
        if(pSR->result!=NoErr)       //Check for errors
                {
                pSR->operation=MESSAGEBOX;
                DriverLINX(pSR);
                        }
```

```
                break;
                }
        return 0;
}
```

Step 2. Configure Digital Output

```
int CVICALLBACK conf_do (int panel, int control, int event, void *callbackData, int eventData
{
switch (event)
{
case EVENT_COMMIT:
        pSR->subsystem = DO;         //Talk to the Digital Output subsystem
        pSR->mode = OTHER;                    //The mode is OTHER, not DMA or INTerrupt

        pSR->operation = CONFIGURE;  //The operation is to CONFIGURE
        pSR->timing.typeEvent = DIOSETUP;    //This is a Digital I/O setup event
        pSR->timing.u.diSetup.channel = 0;           //We are setting channel 0
        pSR->timing.u.diSetup.mode = DIO_BASIC;
//The mode is Digital I/O basic operation
        DriverLINX(pSR);                        //Execute the Service Request
        pSR->timing.typeEvent = NULLEVENT;
 //Put the event back to NULL so we can reuse pSR
        if(pSR->result!=NoErr)                       //Check for errors
        {
                        pSR->operation=MESSAGEBOX;
                        DriverLINX(pSR);
        }

break;
}
return 0;
}
```

Step 3. Output Digital Byte

```
int CVICALLBACK digout (int panel, int control, int event, void *callbackData, int eventData1
{
switch (event)
{
case EVENT_COMMIT:
        pSR->mode=POLLED;                    //The mode is POLLED I/O
        pSR->channels.nChannels=1;           //The total number of channels is 1
        pSR->channels.chanGain[0].channel=0; //We are talking to channel 0
        pSR->status.typeStatus=IOVALUE;      //We are sending an I/O value
        pSR->status.u.ioValue=255;
                                //We are sending a value of 255, i.e., all 8 lines are high
        pSR->operation = START;                      //The operation is to START
        DriverLINX(pSR);                                //Execute the Service Request
        if(pSR->result!=NoErr)                       //Check for errors
        {
        pSR->operation=MESSAGEBOX;
        DriverLINX(pSR);
}
break;
}
return 0;
}
```

Step 4. Exit

```
int CVICALLBACK quit (int panel, int control, int event,
                void *callbackData, int eventData1, int eventData2)
```

```
{
        switch (event)
{
                case EVENT_COMMIT:
                        free(pSR);                              //Free memory
                        pSR = NULL;
                        CloseDriverLINX(driverInstance);        //Close the Driver
                        QuitUserInterface (0);
                break;
        }
        return 0;
}
```

Conclusion

Although LabWindows/CVI is not supported by DriverLINX, this document shows a working example that compiles and links properly. It also proves that the communication between CVI and DriverLINX is sound. The example program performs a very simple DAQ task. Other tasks such as analog input, analog output, counter/timers control, Windows messaging, buffering, and memory allocation are still to be investigated. Nothing indicates that these tasks should not function similarly to this example program.