# Performing Background Digital Input Acquisition Using the External Interrupt on a KPCI-PIO24 and DriverLINX®

**by**
**Mike Bayda**
**Keithley Instruments, Inc.**

## Introduction

Many applications require the data acquisition hardware to respond to an external digital clock or to a trigger event driven by various pieces of equipment in the system. One of the main goals of these applications is to synchronize the data acquisition with the external event providing the clock. In the application, we will be using a function generator as a source of the external interrupt, and we will read all three ports (24 lines) on the KPCI-PIO24 data acquisition board simultaneously on every interrupt.

## Setup

This application was developed to test the performance of the Interrupt handling on the KPCI-PIO24 and DriverLINX under Windows 95/98 and NT. A 24-bit counter (2X 12-bit cascaded) was designed to increment by one upon receiving a falling edge external interrupt from an HP3314A function generator (see Figure 1). All 24 digital output lines from the counter were wired to the KPCI-PIO24's ports A, B, and C, which are configured for input. The square wave out of the function generator was also connected directly to the interrupt input of the KPCI-PIO24, which is pin 1 on the DB-37 connector. The KPCI-PIO24 triggers on a rising edge. The delay between the falling and the rising edge allows the counter to settle before being read by the KPCI-PIO24 board. A 400MHz Pentium II was used and the Visual C/C++ program at the end of this document was tested using the Windows(r) NT and Windows 95 operating systems.

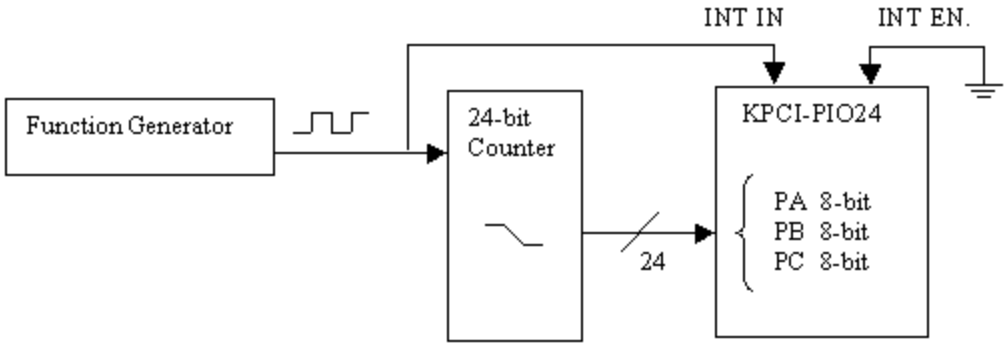


**Figure 1**

## Application

The following program sets up the KPCI-PIO24 to read all 24 lines from the 24-bit counter simultaneously upon receiving an external interrupt on pin 1 of the DB-37 connector. (The Interrupt Enable line on pin 2 can be used to start the acquisition on an external trigger. However, for all practical purposes, pin 2 is grounded since the interrupt is active low.)

DriverLINX will be configured in Background Digital Input mode, which is essentially the interrupt mode. The user will not need to poll the digital inputs; the transfer and the buffering will automatically be handled by the driver. So, based on the description above, and if the KPCI-PIO24 board is keeping up with the external interrupt, the 24-bit number read by both the board and the driver should increment by one and only one on each interrupt. If the data read does not increment by one, then gaps

occured in the data. Based on that criterion, the maximum interrupt speed is determined. It is always a good idea to use external measurement equipment to quantify the operating system's performance. With this 24-bit counter used in the system, we are able to obtain an independent and deterministic flag indicating loss in the data.

### Conclusion

After running the program below at different speeds under Windows 95/98, it was determined that 10kHz is the conservative maximum frequency that the KPCI-PIO24 can sustain without data loss. That is with the total number of samples exceeding 16 million. When the number of samples was around 30000 or less, 20-30kHz was achieved. A similar example was developed in Visual Basic and similar results were found, since the data transfer is happening in the background at the driver level and the transfer rate is essentially independent of the programming language used. Under Windows NT 4.0, the interrupt handling is more reliable and this program performed better than Windows 95/98 by a factor of four. Under Windows NT 4.0, the maximum frequency was 40kHz with 16 million gap-free samples. Since this maximum frequency is limited by how fast the operating system can respond to an interrupt, the same benchmarks should apply as well to Keithley's PIO-24 ISA board. This table summarizes the results:

| Operating System | Sustained Frequency |
|---|---|
| Windows NT4 | ~40 kHz |
| Windows 95/98 | ~10 kHz |

It is important to mention that these results were obtained running only the console application below in the foreground with no external input activity during the measurement (such as mouse, keyboard, network input, etc.) Any interrupt not generated by the KPCI-PIO24 could affect the results. Also, no two computers are identical due to different installed software and peripherals as well as patterns of usage, multitasking settings in the control panel, etc. Users must carefully test their systems to determine the highest throughput vs. transfer reliability.

Finally, it is worth mentioning that the KPCI-3160, a 96-line digital I/O board, has an internal missed interrupt detection that should agree as well with the results of this counter circuit.

### Example Program

The following console program was written in Visual C/C++ version 6. Once compiled into an executable, the user could double-click on it from Microsoft® Explorer or run it from a DOS window. While running, the program will display the number of samples acquired so far by reading the "nextElement" property of the service request. When the last sample is read, a simple "ShowData()" routine checks for missing samples in the data by comparing adjacent numbers. If a number is not an increment by one from the previous number, then an interrupt was missed.

DriverLINX is based on the Windows message-passing paradigm. However, this console application uses the sequential status polling paradigm. In the general case of using DriverLINX, we highly recommended the use of Microsoft Foundation Classes (MFC) and the application wizard when developing Visual C/C++ programs. MFC provides the message pump, which can process DriverLINX messages.

```
/* Console Example program using KPCI-PIO24 */

/* This program exercises Background Digital Inputs using an external Interrupt */
/*A ~16 Msamples(2^24) buffer is allocated, and the data is acquired in the background*/
/* in one shot. When data acquisition is complete, the data is checked for errors */


#include <afx.h>
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include "D:\drvlinx\kpci-pio\dlapi\drvlinx.h"
#include "D:\drvlinx\kpci-pio\dlapi\DLCODES.h"


//Variables
HANDLE gHandle = NULL;
HWND ghWnd = NULL;
DL_ServiceRequest *g_pSR;               //Declare the Service Request
HINSTANCE gDriverInstance;      //DriverInstance hold the driver handle
BYTE *g_pData;                  //pointer to data in buffer
```

```
BOOLEAN g_bInitialized= FALSE;  //Indicates whether the board is initialized
BOOLEAN g_bDrvOpen = FALSE;     //Indicates whether a driver has been opened

//Program Constants
const int LogicalDevice = 0;
const SINT LogicalChannel = 0;
const SINT ChannelGain = -1;    //unipolar
const UINT NumberOfSamples = 16777216;  /* 2^24 (bits) samples */
const int ChannelsSampled = 3;
const SINT NumberOfBuffers = 1;


//Function Prototypes
void AddSelectBuffers(LPServiceRequest pSR, const UINT Buffers,
                        const UINT SamplesPerChannel, const UINT ChannelsSampled,                           const U
void CleanUpDriverLINX();
void ClearBuffers();
void Execute();
void GetDriverLINXData(const UINT BufIndex);
BOOLEAN InitializeDriver();
void SetupDriverLINX();
void SetUpDriverLINXTask();
void ShowData();
void ShowDriverLINXStatus(LPServiceRequest pSR);


void main()
{
        char Title[80];
        DWORD Len= sizeof(Title);
        //Get the windows title for the console window
        GetConsoleTitle(Title, Len);
        //Find the window handle for the console window based on the Title
        ghWnd = FindWindow(NULL, Title);

        //Set up an event for termination of the program.
        gHandle = CreateEvent(NULL, TRUE, FALSE, "Termination Event");
        //Print Opening
        printf("This program will acquire digital input data from a KPCI-PIO24 board.\n");
        printf("This program will display the acquired buffer. \n");
        printf("Wait for Open DriverLINX Dialog\n");
        SetupDriverLINX();

        /*      Console applications have no way to get focus, once the program has
                lost the focus. The reason that this may or may not happen is that
                the OpenDriverLINX dialog can not pass the focus back to this
                application. If you specify the driver name in the OpenDriverLINX call,
                you may not see this behavior in your code. */

        printf("\n");
        printf("If Console Window does not have focus\n");
        printf("click on the window before running the program\n");
        printf("\n");
        char c;
        while (1){
        printf("\nPress 'Q' to quit program. \n");
        printf("Press 'G' acquire data. \n");
        c = _getch();
                switch(c){
                case 'Q':
                case 'q':
                        CleanUpDriverLINX();
                        SetEvent(gHandle);
                        goto EndProg;
                        break;
                case 'G':
                case 'g':
                        if (g_bDrvOpen && g_bInitialized) Execute();
                        if (g_bDrvOpen == FALSE) printf("Driver was not opened. Please Exit!!! \n");
                         if ((g_bDrvOpen == TRUE) && (g_bInitialized == FALSE))
                                printf("Device was not Initialized. Please Exit!!! \n");

                        break;
                }
        }
EndProg:
        //Wait for the termination event.
        WaitForSingleObject(gHandle, INFINITE);

}


/*************************************************************************
* The AddSelectBuffers function creates buffers for DriverLINX to use for
* storing data. For more information, see DrvVCLib.hlp for a description of this
* function. The function was used and created for the latest C++ example set.
* The help file is located in your \\DrvLINX4\Source\CPP4\LIB file. For more
* examples of this function, see the other examples included in the
* \\DrvLINX4\Source\CPP4 subdirectories.
*
*************************************************************************/
void AddSelectBuffers(LPServiceRequest pSR, const UINT Buffers,
                        const UINT SamplesPerChannel, const UINT ChannelsSampled,                           const U
{
```

```
        DL_BUFFERLIST *pBuffer = pSR->lpBuffers;
    //Check to see if any buffers exist in the service request data structure.
    //If they do exist, delete them and then set the DL_BUFFERLIST data structure to
    //NULL.
    if (pBuffer != NULL)  //IF data structure exists
    {
            int i;
            for (i = 0; i< pBuffer->nBuffers; i++)
            {
                        if (pBuffer->BufferAddr[i] != NULL)     //Delete old buffers
                        {
                                BufFree(pBuffer->BufferAddr[i]);
                        }
            }
                 delete [ ] pBuffer;     //Release memory
                 pBuffer = NULL;  //Set data structure to NULL
    }

    if (Buffers > 0)
    {
                 pBuffer = (DL_BUFFERLIST *) new BYTE[DL_BufferListBytes(Buffers)];
                 pBuffer->notify = NOTIFY;
                 pBuffer->nBuffers = Buffers;
                 pBuffer->bufferSize = Samples2Bytes(device, subsystem, 0,
                                                 (SamplesPerChannel * ChannelsSampled));
                 UINT i;
                 for ( i = 0; i < Buffers; i++)
                 {
                        //DriverLINX will allocate buffer memory
                        if (pSR->mode == DMA)
                        {
                                pBuffer->BufferAddr[i] = BufAlloc(GBUF_DMA16, pBuffer->bufferSize);
                        }
                        else if (pSR->mode == INTERRUPT)
                        {
                                pBuffer->BufferAddr[i] = BufAlloc(GBUF_INT, pBuffer->bufferSize);
                        }
                        else if (pSR->mode == POLLED)
                        {
                                pBuffer->BufferAddr[i] = BufAlloc(GBUF_POLLED, pBuffer->bufferSize);
                        }

                 }
    }

    pSR->lpBuffers = pBuffer;
}

/*************************************************************************
*       This function performs all of the necessary housekeeping for DriverLINX
*       when the user closes the program from the console.
*************************************************************************/

void CleanUpDriverLINX()
{
        //Free buffer memory associated with the Service Request
        ClearBuffers();

        //If driver is open, close it
        //Close Driver
        if (gDriverInstance != NULL)
        {
                CloseDriverLINX(gDriverInstance); //Close the Driver
        }

        //Free memory associated with the Service Request
        if (g_pSR != NULL)
        {
                delete g_pSR;
                g_pSR = NULL;
        }

        //free memory associated with g_pData
        if (g_pData != NULL)
        {
                delete [] g_pData;
                g_pData = NULL;
        }


}

/*************************************************************************
*       This function is used to clear the buffers associated with the
*       Service Request.
*************************************************************************/
void ClearBuffers()
{
        //Free memory associated with Service Request
        if ( g_pSR != NULL )
        {
                if (g_pSR->lpBuffers)
                {
                        for (int i=0; i<g_pSR->lpBuffers->nBuffers; i++)
```

```
                        {
                                if((g_pSR->lpBuffers->BufferAddr[i] != NULL) ||
                                        (g_pSR->lpBuffers->BufferAddr[i] != AUTO_ALLOC_BUFFER))
                                {
                                        BufFree (g_pSR->lpBuffers->BufferAddr[i]);
                                        g_pSR->lpBuffers->BufferAddr[i] = NULL;
                                }
                        }
                        delete(g_pSR->lpBuffers);
                        g_pSR->lpBuffers = NULL;
                }
        }
}



/************************************************************************
*       This function submits the Service Requests and polls to see when the
*       task is complete. After the task has completed, the function arranges for
*       the data to be loaded to the user's buffer and then arranges for the data
*       to be analyzed
*
************************************************************************/
void Execute()
{
        //Execute the Service Request
        DriverLINX(g_pSR);

        //Check for errors
        if (g_pSR->result != 0)
        {
                ShowDriverLINXStatus(g_pSR);
        }

        //Status poll the buffer to see when task is done.
        //This example has no message pump, so this is the only
       //way to know when the task is done.  MFC is more often
        //recommended when developing VC/C++ programs
        Ops OriginalOp;
        WORD wOriginalFlags;
        DWORD dwCurrent = 0;
        WORD wErrCode;
        OriginalOp = g_pSR->operation;
        g_pSR->operation = STATUS;
       wOriginalFlags = g_pSR->taskFlags;
        g_pSR->taskFlags = 0;

        printf("Current Buffer Location!\n");
        do
        {
                DriverLINX(g_pSR);
                if (dwCurrent != g_pSR->status.u.ioStatus.nextElement)
                {
                       dwCurrent = g_pSR->status.u.ioStatus.nextElement;
                        printf("%d ",dwCurrent);
                }
                else Sleep(100);
        }       while (g_pSR->result == 0) ;

        //Display the last point
        if (dwCurrent != NumberOfSamples)
                printf("%d\n",NumberOfSamples);
        else
                printf("\n");

        wErrCode = getErrCode(g_pSR->result);
        //If error other than "Service Request not found" display
        if ((wErrCode != NoErr) && (wErrCode != NotTaskErr))
        {
                ShowDriverLINXStatus(g_pSR);
                return;
        }


        //Return orginal settings
        g_pSR->operation = OriginalOp;
        g_pSR->taskFlags = wOriginalFlags;

        if (1) //(Status != done)
        {
                //Translate the DriverLINX buffer to the User Buffer
                GetDriverLINXData(0);
                //Display data on the screen
                ShowData();
        }

}

/************************************************************************
*       This function copies the data buffer and places it into
*       a buffer that is pointed to by the member variable m_pData.
*
************************************************************************/
void GetDriverLINXData(const UINT BufIndex)
```

```
{
        memcpy(g_pData, g_pSR->lpBuffers->BufferAddr[BufIndex],
                        NumberOfSamples * ChannelsSampled);
}

/*********************************************************************
*       This function sets the appropriate fields of the Service Request
*       required to initialize a logical device. This function executes
*       a Service Request. A device has to be initialized before it can
*       perform any data acquisition tasks.
*********************************************************************/
BOOLEAN InitializeDriver()
{
        g_pSR->device = LogicalDevice;
        g_pSR->subsystem = DEVICE;
        g_pSR->mode = OTHER;
        g_pSR->operation = INITIALIZE;

        //Specify Null Start, Stop, and Timing Events
        g_pSR->start.typeEvent = NULLEVENT;
        g_pSR->stop.typeEvent = NULLEVENT;
        g_pSR->timing.typeEvent = NULLEVENT;

        //Specify no channels
        g_pSR->channels.nChannels = 0;
        g_pSR->channels.chanGainList = NULL;

        //Specify no buffers
        AddSelectBuffers(g_pSR, 0, 0, 0, LogicalDevice, DEVICE);

        //Execute the Service Request
        DriverLINX(g_pSR);

        //Check for errors
        if (g_pSR->result != 0)
        {
                ShowDriverLINXStatus(g_pSR);
                return FALSE;
        }
        else
                return TRUE;
}

/*********************************************************************
*       This function creates the Service Request in Memory, and creates the data
*       buffer that holds the results. This function verifies that a driver is
*       open, that the device has been initialized, and that the task is
*       setup.
*********************************************************************/

void SetupDriverLINX()
{
        //initialize variables
        g_pSR = NULL;
        gDriverInstance = NULL;
        g_pData = NULL;

        //Create Service Request if it does not exist
        if (g_pSR == NULL)
        {
                g_pSR = (DL_ServiceRequest *) new DL_ServiceRequest;
        }

        if (g_pSR != NULL)
        {
                memset(g_pSR, 0, sizeof(DL_ServiceRequest));
        }
        g_pSR->dwSize = sizeof(DL_ServiceRequest);
        ASSERT(g_pSR != NULL);

        //Create data buffer
        if (g_pData != NULL)
                {
                        delete [] g_pData;
                        g_pData = NULL;
                }
                g_pData = new BYTE[NumberOfSamples * ChannelsSampled];


        //Open a Driver
        g_pSR->hWnd = ghWnd;
        ASSERT(IsWindow(ghWnd));
        gDriverInstance = OpenDriverLINX(g_pSR->hWnd, "");
        ASSERT(gDriverInstance != NULL);
        if (gDriverInstance == NULL)
        {
                g_bDrvOpen = FALSE;
        }
        else
        {
                g_bDrvOpen = TRUE;
        }

        //Initialize the Driver
```

```
        if (g_bDrvOpen)
        {
                g_bInitialized = InitializeDriver();
        }

        //Setup the DriverLINX task
        if (g_bInitialized)
        {
                SetUpDriverLINXTask();
        }

        return;

}
/************************************************************************
*       This function sets up a Service Request that inputs a buffer of data
*       from the DI (digital input) subsystem. The Service Requests starts when
*       submitted, and stops when the buffer is full. Data is clocked in/out
*       using external clock.
*       See DrvVCLib.CPP - SetupDriverLINXBufferedIO for more information
*       and to see how this functionality is implemented for general use.
************************************************************************/
void SetUpDriverLINXTask()
{
        /*---------------------- Service Request Group --------------------*/
        g_pSR->device = LogicalDevice;
        g_pSR->subsystem = DI;
        g_pSR->mode = INTERRUPT;
        g_pSR->operation = START;

        /*-------------------------- Event Group -------------------------*/
        g_pSR->timing.typeEvent = DIEVENT;
        g_pSR->timing.delay = 0;
        g_pSR->timing.u.diEvent.channel = DI_EXTCLK;
        g_pSR->timing.u.diEvent.mask = 1;
        g_pSR->timing.u.diEvent.match = FALSE;
        g_pSR->timing.u.diEvent.pattern = 0;

        //Specify start event
        g_pSR->start.typeEvent = COMMAND;

        //Specify stop event
        g_pSR->stop.typeEvent = TCEVENT;

        /*---------------------- Select Channel Group --------------------*/
        //Specify the channels, gain, and data format
        g_pSR->channels.nChannels = 2; /* 2 for start and stop */
        g_pSR->channels.numberFormat = tNATIVE;
        g_pSR->channels.chanGain[0].channel = 0; /* port A */
        g_pSR->channels.chanGain[1].channel = 2; /* port C */
        g_pSR->channels.chanGainList = NULL;         //zero when not used
        g_pSR->channels.chanGain[0].gainOrRange = 0; /* not used, set to zero */
        g_pSR->channels.chanGain[1].gainOrRange = 0; /* not used, set to zero */
        g_pSR->channels.simultaneousScan = TRUE;  /* Port A, B and C are read simultaneously */


        /*---------------------- Select Buffers Group --------------------*/
        //Specify buffers
        AddSelectBuffers(g_pSR, NumberOfBuffers, NumberOfSamples, ChannelsSampled,
                                        LogicalDevice, DI);

        /*-------------------------- Select Flags ----------------------*/
        g_pSR->taskFlags = CS_NONE;

        /* NOTE: you do not have to block any messages. However,
                         DriverLINX is somewhat more efficient if you do.

          NOTE: Your application must submit the service request to execute this
                        function.
          NOTE: For output Service Requests make sure to fill the buffer(s) before
                        submitting the service requests.
        */
}

/************************************************************************
*       This function displays the data obtained in the buffer on the console.
************************************************************************/
void ShowData()
{
                long counts, stored, failures=0;

               stored = (long)g_pData[0] + ((long)g_pData[1]*256) + ((long)g_pData[2]*65536);
                printf("Failure: difference in counts and location.\n");
                for (int i=1; i < NumberOfSamples; i++)
                {
                        counts = (long)g_pData[i*3] + ((long)g_pData[i*3 + 1]*256) + ((long)g_pData[i*3 + 2]*65536);
                        if ((counts - stored) != 1)
                        {
                                printf("diff: %ld,   Sample#: %ld\n", counts-stored, i);
                                failures++;
                        }
                        stored = counts;
                }
                printf("Total Failures: %ld\n", failures);
```

```
}

/************************************************************************
*       This function displays the DriverLINX error in a messagebox.
************************************************************************/
void ShowDriverLINXStatus(LPServiceRequest pSR)
{
        Ops OriginalOp;

        //If an error has occurred, DriverLINX can display a messagebox
        //that describes the status of a Service Request.
        if (pSR->result != 0)
        {
                OriginalOp = pSR->operation; // Save the current operation
                pSR->operation = MESSAGEBOX; // Change operation to display
                                                        // message box.
                DriverLINX(pSR);
                pSR->operation = OriginalOp; // Change operation back!
        }
}
```