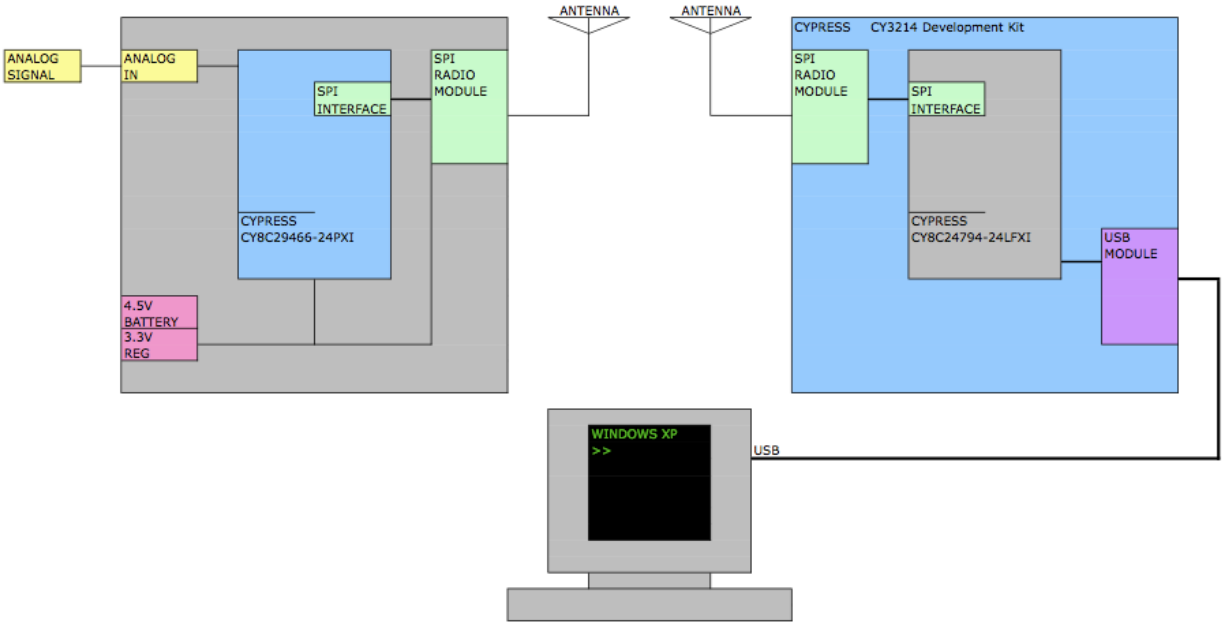
Wireless USB Oscilloscope

CMPE 121 Microprocessor System Design, Spring 2011
Steven Paul Lewis and Christopher Upton



Operating Parameters

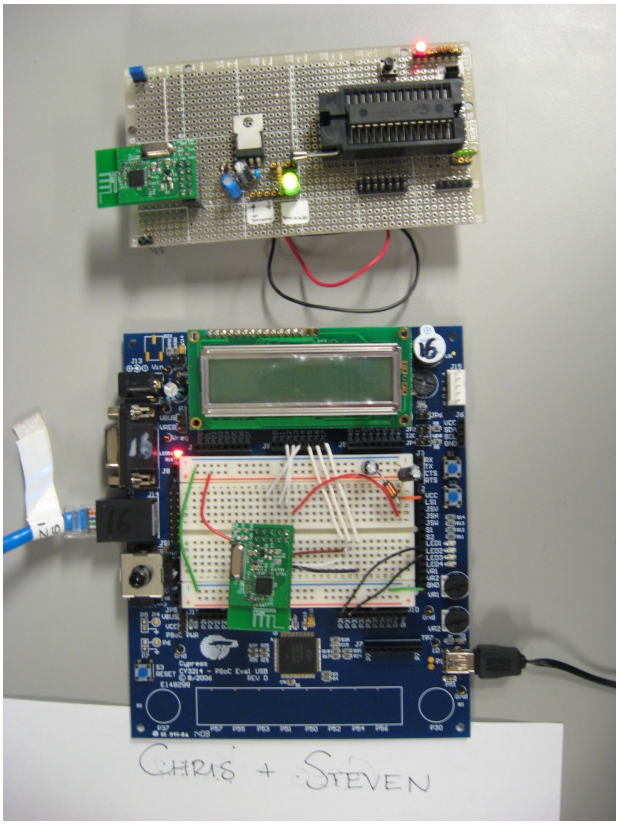
TRANSMITTER

V_{CC}/V_{BAT}	3.3 V
Max Ripple V_{CC}	12 mVAC
Max Ripple V_{BAT}	2.6 mVAC
CPU Clock	12 MHz
SPI Clock	2 MHz
SPI Mode	0*
Payload Length	16 Bytes
Radio Effective Distance	~50 ft
Broadcast Frequency	2.412 GHz
PN Code (8 Byte)	x B9 8E 19 74 6F 65 18 74
ADC Sample Rate	240 sps
Analog Input Range	-1.65 — 1.65 V
Input Frequency Range	0 — 80 Hz

*Data captured on rising clock edge, propagated on falling clock edge.

RECEIVER

V_{CC}/V_{BAT}	3.3 V
Max Ripple V_{CC}	16 mVAC
Max Ripple V_{BAT}	12 mVAC
CPU Clock	12 MHz
USB Transfer Mode	Bulk
USB Packet Length	128 Bytes



Notes on Operation

Wi-Fi Technology

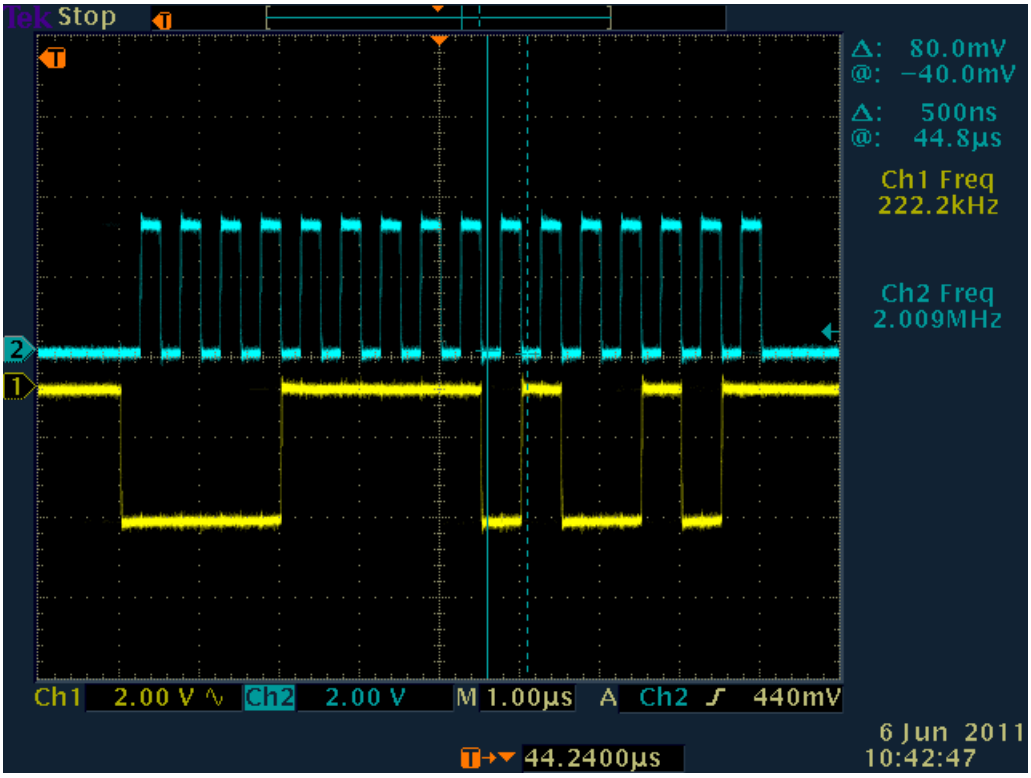
The Cypress SPI Radio is classified as a wireless network standard 802.15, “Wireless Personal Area Networks.” Specifically, the device classifies as a short range, high bandwidth wireless network link, broadcasting over a spread spectrum using PN code modulation. The SPI radio is a Direct Sequence Spread Spectrum (DSSS) device.

V_{BAT} Ripple

The SPI radio module must be powered from a very pure DC voltage source at 3.3 V. Any significant ripple at the supply pin will disrupt the transceiver and cause the program to crash. Thus, there is a necessity for a large coupling capacitor near the V_{BAT} supply pin in conjunction with an inductor, creating a LC “smoothing” circuit to keep the device running at a steady 3.28 V, with a ripple attenuated to 2.6 mVAC. Without the coupling capacitor, this ripple is too large: approximately 12 mVAC.

SPI Interaction

Below is the captured waveform of the signals SClk (blue) and MOSI (yellow) that was sent to the radio module.

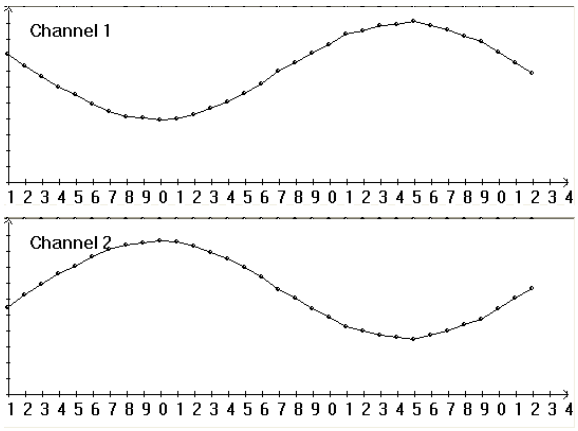


Oscilloscope Inverting Channel

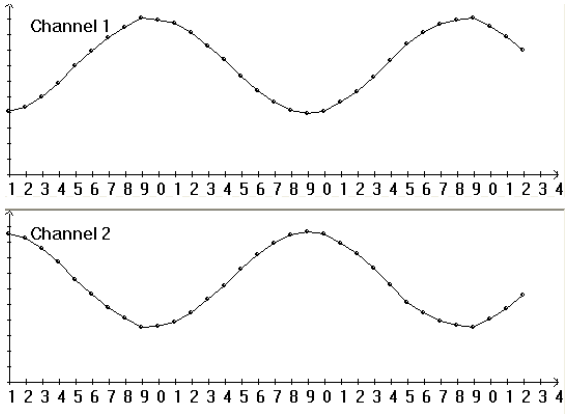
Since the voltage range on the scope display is limited, Channel 2 displays the inverse of the analog input. This aids in finding a signal that does not have a proper DC offset.

Performance of Oscilloscope

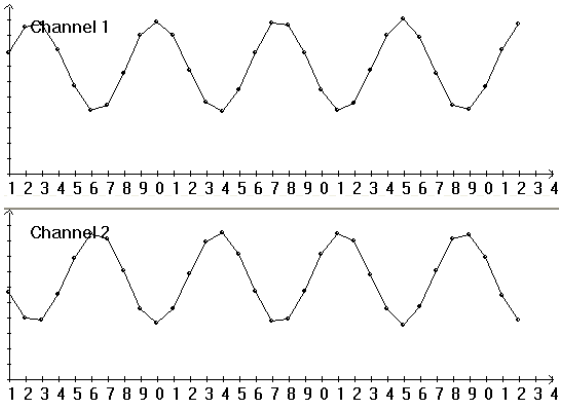
Below is included a number of screen captures that illustrate the ability of the oscilloscope to reproduce an analog waveform input to the transmitter.



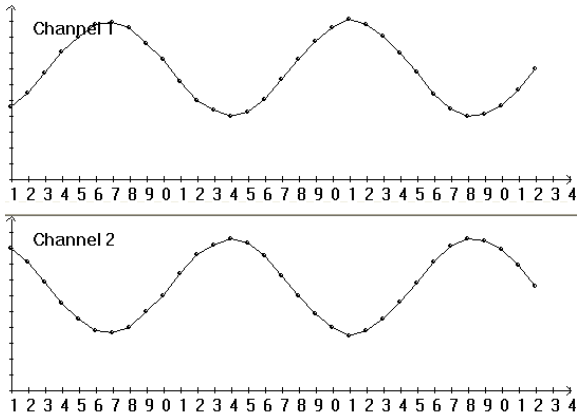
~8 Hz Sinusoidal



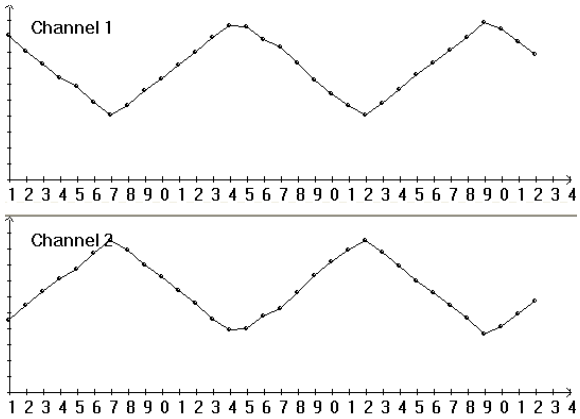
~15 Hz Sinusoidal



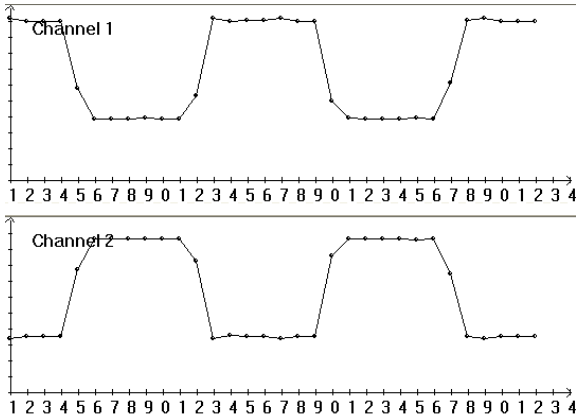
~60 Hz Sinusoidal



~30 Hz Sinusoidal



~30 Hz Triangle



~30 Hz Square

```
//-------------------------------------------------------------
//      Radio Transmitter Firmware
//      Wireless USB Oscilloscope
//      CMPE 121 – Spring 2011
//      Steven Paul Lewis and Christopher Upton
//-------------------------------------------------------------
#include <m8c.h>        // PSoC Part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules
#include "PSoCgpioint.h" // Contains pin specific defines
#include "lpregs.h"
#include "lpradio.h"

// Local Definitions and Types
#define PAYLOAD_LENGTH  16
#define CHANNEL         10
#define SOP_PN_CODE     7
#define DATA_RATE       DATMODE_8DR

#define mmSPI_ADDRESS    0x3F
#define bbSPI_WRITE      0x80

#define DELAY_COUNT       0x1eeF // ~100ms
#define SWITCH_DEBOUCE    0x0300 // ~7.5ms


// Variables and arrays
unsigned char outBuf[PAYLOAD_LENGTH];
unsigned int dataBuf1, dataBuf2;
int i, j;

void main()
{
        RADIO_STATE txState;

        LED_1_Start();
        PGA_1_Start(3);
        ADCINC12_1_Start(3);

        M8C_EnableGInt;

        LED_1_On();

    SPIM_Radio_Start(SPIM_Radio_SPIM_MODE_0 | SPIM_Radio_SPIM_MSB_FIRST);
    RadioInit(XACT_CFG_RST | ACK_TO_8X, (TX_CFG_RST & ~TX_DATMODE_MSK) | DATA_RATE | PA_0_DBM);
    RadioSetFrequency(CHANNEL);
    RadioSetSopPnCode(SOP_PN_CODE);
    RadioSetLength(PAYLOAD_LENGTH);

        LED_1_Off();

        ADCINC12_1_GetSamples(0);

        while(1)
        {
                LED_1_On();
                for(i = 0; i < PAYLOAD_LENGTH; i += 4)
        {
                while(ADCINC12_1_fIsDataAvailable() == 0){} // loop until value is ready
                ADCINC12_1_ClearFlag();
                dataBuf1 = ADCINC12_1_iGetData(); // Get result
                dataBuf2 = 2048 - dataBuf1; // convert to unsigned (inverted)
                dataBuf1 += 2048; // convert to unsigned (non-inverted)
                outBuf[i] = (dataBuf1 >> 8) & 0x00FF;
                outBuf[i + 1] = dataBuf1 & 0x00FF;
                outBuf[i + 2] = (dataBuf2 >> 8) & 0x00FF;
                outBuf[i + 3] = dataBuf2 & 0x00FF;
        }

        RadioSetPtr(outBuf);
        txState = RadioBlockingTransmit(4, sizeof(outBuf));
                LED_1_Off();
        }
}
```

```
//-------------------------------------------------------------
//      Radio Receiver Firmware
//      Wireless USB Oscilloscope
//      CMPE 121 – Spring 2011
//      Steven Paul Lewis and Christopher Upton
//-------------------------------------------------------------
#include <m8c.h>
#include "PSoCAPI.h"
#include "PSoCgpioint.h"
#include "lpregs.h"
#include "lpradio.h"

// Local definitions and types
#define PAYLOAD_LENGTH  16
#define CHANNEL         10
#define SOP_PN_CODE     7
#define DATA_RATE       DATMODE_8DR

#define mmSPI_ADDRESS   0x3F
#define bbSPI_WRITE     0x80

#define DELAY_COUNT     0x0F00
#define SWITCH_DEBOUCE  0x0300
#define PACKET_SIZE    128

// Variables and arrays
unsigned char PayloadBuffer[PAYLOAD_LENGTH];
unsigned char outBuf[PACKET_SIZE];
int i, j;


unsigned char ReceivedCount = 0;

RADIO_STATE rxState;
RADIO_RX_STATUS rxStatus;

// Function declarations
void RadioInit(XACT_CONFIG defaultXactState, TX_CONFIG defaultTxState);
void    RadioSetFrequency(BYTE);
void    RadioSetSopPnCode(BYTE);

void          HardwareInit(void);
void          RadioStartReceive(void);
unsigned char RadioGetReceiveState(void);
unsigned char RadioRead(unsigned char Address);
void          RadioWrite(unsigned char Address, unsigned char Data);
void          Transmit(unsigned char* Data, unsigned char Length);
void          Delay(unsigned Count);

void Delay(unsigned Count)
{
    for (; Count; --Count)
    {
        asm("nop");
    }
}
```

```
// Main
void main()
{
        unsigned char ReceivedPayloadSize;



        LED_1_Start();
        LED_2_Start();

        USBFS_1_Start(0,3);    // initialize usb module

        M8C_EnableGInt;


        LED_1_On();
    while(!USBFS_1_bGetConfiguration());     // wait here until usb is configured
    USBFS_1_INT_REG |= USBFS_1_INT_SOF_MASK;
    USBFS_1_LoadInEP(1, outBuf, PACKET_SIZE, 0);
    LED_1_Off();

    SPIM_Radio_Start(SPIM_Radio_SPIM_MODE_0 | SPIM_Radio_SPIM_MSB_FIRST);
    RadioInit(XACT_CFG_RST | ACK_TO_8X, (TX_CFG_RST & ~TX_DATMODE_MSK) | DATA_RATE | PA_0_DBM);
    RadioSetFrequency(CHANNEL);
    RadioSetSopPnCode(SOP_PN_CODE);
    RadioSetLength(PAYLOAD_LENGTH);
    RadioSetPtr(PayloadBuffer);
    RadioStartReceive();

        j = 0;

    while(1)
    {
        while(j < PACKET_SIZE)
        {
                rxState = RadioGetReceiveState();

                if (rxState & RADIO_COMPLETE)
                {
                                ReceivedPayloadSize = RadioEndReceive();
                                Delay(DELAY_COUNT);
                                if(!(rxState & RADIO_ERROR))
                                {
                                        LED_2_Invert();
                                        for(i = 0; i < PAYLOAD_LENGTH; i++)
                                        {
                                                outBuf[j] = PayloadBuffer[i];
                                                j++;
                                        }
                                }
                                else
                                {
                                        rxStatus = RadioGetReceiveStatus();
                                        RadioInit(XACT_CFG_RST | ACK_TO_8X, (TX_CFG_RST & ~TX_DATMODE_MSK)
| DATA_RATE | PA_0_DBM);
                                RadioSetFrequency(CHANNEL);
                                RadioSetSopPnCode(SOP_PN_CODE);
                                RadioSetLength(PAYLOAD_LENGTH);
                                RadioSetPtr(PayloadBuffer);
                                }
                                RadioStartReceive();
                        }
                }
                j = 0;

                while(!USBFS_1_bGetEPAckState(1));
        USBFS_1_LoadInEP(1, outBuf, PACKET_SIZE, USB_TOGGLE);
        }
}
```

```
//-------------------------------------------------------------
//      Windows Application
//      Wireless USB Oscilloscope
//      CMPE 121 – Spring 2011
//      Steven Paul Lewis and Christopher Upton
//-------------------------------------------------------------

// headers for project
#include "stdafx.h"
#include "resource.h"

// headers for the usb guid
#include <initguid.h>
#include <setupapi.h>
#include <basetyps.h>
#include "usbdi.h"

// headers from rwbulk
#include <windows.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#include "devioctl.h"
#include "bulkusr.h"

#define TIMER_MICRO_SECOND 1
#define PACKET_SIZE 128

static HANDLE hRead = NULL;
static char my[80] = {"UCSC CMPE121 USB Scope\n"};
char inPipe[32] = "PIPE00";

UINT success;
int ReadLen = PACKET_SIZE;
char pinBuf[PACKET_SIZE];
ULONG nBytesRead;
int i, j;

char completeDeviceName[256] = "";

void changedata();
void myplot(HDC hdc, int myid, int * mydata, LPTSTR legend, int xstart, int xend, int ystart, int yend);

const char g_szClassName[] = "myWindowClass";

const int ID_TIMER = 1;
const int width = 540;        // big window
const int height = 380;       // big window
int wh = (height / 2) - 25;   // little window

HWND win_chan1 = NULL, win_chan2 = NULL;  // two channel scope
int ydata_array1[32], ydata_array2[32];

//-------------------------------------------------------------
//      Functions for Handling USB Device
//-------------------------------------------------------------
HANDLE
OpenOneDevice (
    IN       HDEVINFO                    HardwareDeviceInfo,
    IN       PSP_DEVICE_INTERFACE_DATA   DeviceInfoData,
        IN               char *devName
    )
/*++
Routine Description:

    Given the HardwareDeviceInfo, representing a handle to the plug and
    play information, and deviceInfoData, representing a specific usb device,

    open that device and fill in all the relevant information in the given
    USB_DEVICE_DESCRIPTOR structure.
```

```
Arguments:

    HardwareDeviceInfo:  handle to info obtained from Pnp mgr via SetupDiGetClassDevs()
    DeviceInfoData:      ptr to info obtained via SetupDiEnumDeviceInterfaces()

Return Value:

    return HANDLE if the open and initialization was successfull,
        else INVLAID_HANDLE_VALUE.

--*/
{
    PSP_DEVICE_INTERFACE_DETAIL_DATA     functionClassDeviceData = NULL;
    ULONG                                predictedLength = 0;
    ULONG                                requiredLength = 0;
        HANDLE                                                          hOut = INVALID_HANDLE_VALUE;

    //
    // allocate a function class device data structure to receive the
    // goods about this particular device.
    //
    SetupDiGetDeviceInterfaceDetail (
            HardwareDeviceInfo,
            DeviceInfoData,
            NULL, // probing so no output buffer yet
            0, // probing so output buffer length of zero
            &requiredLength,
            NULL); // not interested in the specific dev-node


    predictedLength = requiredLength;
    // sizeof (SP_FNCLASS_DEVICE_DATA) + 512;

    functionClassDeviceData =  (PSP_DEVICE_INTERFACE_DETAIL_DATA ) malloc(predictedLength);
    if(NULL == functionClassDeviceData) {
        return INVALID_HANDLE_VALUE;
    }
    functionClassDeviceData->cbSize = sizeof (SP_DEVICE_INTERFACE_DETAIL_DATA);

    //
    // Retrieve the information from Plug and Play.
    //
    if (! SetupDiGetDeviceInterfaceDetail (
               HardwareDeviceInfo,
               DeviceInfoData,
               functionClassDeviceData,
               predictedLength,
               &requiredLength,
               NULL)) {
                free( functionClassDeviceData );
        return INVALID_HANDLE_VALUE;
    }

        strcpy( devName,functionClassDeviceData->DevicePath) ;
//        printf( "Attempting to open %s\n", devName );

    hOut = CreateFile (
                  functionClassDeviceData->DevicePath,
                  GENERIC_READ | GENERIC_WRITE,
                  FILE_SHARE_READ | FILE_SHARE_WRITE,
                  NULL, // no SECURITY_ATTRIBUTES structure
                  OPEN_EXISTING, // No special create flags
                  0, // No special attributes
                  NULL); // No template file

    if (INVALID_HANDLE_VALUE == hOut) {
//                printf( "FAILED to open %s\n", devName );
    }
        free( functionClassDeviceData );
        return hOut;
}

HANDLE OpenUsbDevice( LPGUID  pGuid, char *outNameBuf)
```

```
/*++
Routine Description:

   Do the required PnP things in order to find
   the next available proper device in the system at this time.

Arguments:

    pGuid:      ptr to GUID registered by the driver itself
    outNameBuf: the generated name for this device

Return Value:

    return HANDLE if the open and initialization was successful,
        else INVLAID_HANDLE_VALUE.
--*/
{
   ULONG NumberDevices;
   HANDLE hOut = INVALID_HANDLE_VALUE;
   HDEVINFO                 hardwareDeviceInfo;
   SP_DEVICE_INTERFACE_DATA deviceInfoData;
   ULONG                    i;
   BOOLEAN                  done;
   PUSB_DEVICE_DESCRIPTOR   usbDeviceInst;
   PUSB_DEVICE_DESCRIPTOR       *UsbDevices = &usbDeviceInst;
   PUSB_DEVICE_DESCRIPTOR   tempDevDesc;

   *UsbDevices = NULL;
   tempDevDesc = NULL;
   NumberDevices = 0;

   //
   // Open a handle to the plug and play dev node.
   // SetupDiGetClassDevs() returns a device information set that contains info on all
   // installed devices of a specified class.
   //
   hardwareDeviceInfo = SetupDiGetClassDevs (
                           pGuid,
                           NULL, // Define no enumerator (global)
                           NULL, // Define no
                           (DIGCF_PRESENT | // Only Devices present
                            DIGCF_DEVICEINTERFACE)); // Function class devices.

   //
   // Take a wild guess at the number of devices we have;
   // Be prepared to realloc and retry if there are more than we guessed
   //
   NumberDevices = 4;
   done = FALSE;
   deviceInfoData.cbSize = sizeof (SP_DEVICE_INTERFACE_DATA);

   i=0;
   while (!done) {
      NumberDevices *= 2;

      if (*UsbDevices) {
            tempDevDesc = (PUSB_DEVICE_DESCRIPTOR) realloc (*UsbDevices, (NumberDevices * sizeof
(USB_DEVICE_DESCRIPTOR)));
            if(tempDevDesc) {
                *UsbDevices = tempDevDesc;
                tempDevDesc = NULL;
            }
            else {
                free(*UsbDevices);
                *UsbDevices = NULL;
            }
      } else {
         *UsbDevices = (PUSB_DEVICE_DESCRIPTOR) calloc (NumberDevices, sizeof (USB_DEVICE_DESCRIPTOR));
      }

      if (NULL == *UsbDevices) {

         // SetupDiDestroyDeviceInfoList destroys a device information set
```

```
         // and frees all associated memory.

         SetupDiDestroyDeviceInfoList (hardwareDeviceInfo);
         return INVALID_HANDLE_VALUE;
      }

      usbDeviceInst = *UsbDevices + i;

      for (; i < NumberDevices; i++) {

         // SetupDiEnumDeviceInterfaces() returns information about device interfaces
         // exposed by one or more devices. Each call returns information about one interface;
         // the routine can be called repeatedly to get information about several interfaces
         // exposed by one or more devices.

         if (SetupDiEnumDeviceInterfaces (hardwareDeviceInfo,
                                     0, // We don't care about specific PDOs
                                                                  pGuid,
                                     i,
                                     &deviceInfoData)) {

            hOut = OpenOneDevice (hardwareDeviceInfo, &deviceInfoData, outNameBuf);
                        if ( hOut != INVALID_HANDLE_VALUE ) {
               done = TRUE;
               break;
                        }
         } else {
            if (ERROR_NO_MORE_ITEMS == GetLastError()) {
               done = TRUE;
               break;
            }
         }
      }
   }

   NumberDevices = i;

   // SetupDiDestroyDeviceInfoList() destroys a device information set
   // and frees all associated memory.

   SetupDiDestroyDeviceInfoList (hardwareDeviceInfo);
   free ( *UsbDevices );
   return hOut;
}



BOOL GetUsbDeviceFileName( LPGUID  pGuid, char *outNameBuf)
/*++
Routine Description:

    Given a ptr to a driver-registered GUID, give us a string with the device name
    that can be used in a CreateFile() call.
    Actually briefly opens and closes the device and sets outBuf if successfull;
    returns FALSE if not

Arguments:

    pGuid:      ptr to GUID registered by the driver itself
    outNameBuf: the generated zero-terminated name for this device

Return Value:

    TRUE on success else FALSE

--*/
{
        HANDLE hDev = OpenUsbDevice( pGuid, outNameBuf );
        if ( hDev != INVALID_HANDLE_VALUE )
        {
                CloseHandle( hDev );
                return TRUE;
        }
```

```
        return FALSE;

}


HANDLE open_file( char *filename)
/*++
Routine Description:

    Called by main() to open an instance of our device after obtaining its name

Arguments:

    None

Return Value:

    Device handle on success else NULL

--*/
{
        int success = 1;
    HANDLE h;

    if (!GetUsbDeviceFileName((LPGUID) &GUID_CLASS_I82930_BULK, completeDeviceName))
        {
                return  INVALID_HANDLE_VALUE;
        }
    strcat (completeDeviceName, "\\");

    if((strlen(completeDeviceName) + strlen(filename)) > 255)
        {
        return INVALID_HANDLE_VALUE;
    }

    strcat (completeDeviceName, filename);

        h = CreateFile(completeDeviceName,
                GENERIC_WRITE | GENERIC_READ,
                FILE_SHARE_WRITE | FILE_SHARE_READ,
                NULL,
                OPEN_EXISTING,
                0,
                NULL);
        if (h == INVALID_HANDLE_VALUE)
                {
                success = 0;
        }
                return h;
}

//---------------------------------------------------------------
//      User Application Specific Functions
//---------------------------------------------------------------
void changedata()
{
        if(hRead == NULL)
        {
                hRead = INVALID_HANDLE_VALUE;
                hRead = open_file(inPipe);
                if(hRead == INVALID_HANDLE_VALUE)
                {
                        CloseHandle(hRead);
                        exit(-1);
                }
        }

        j = 0;
        success = ReadFile(hRead, &pinBuf[0], ReadLen, &nBytesRead, NULL);
        if(success)
        {
                for(i = 0; i < PACKET_SIZE; i += 4)
                {
```

```
                        unsigned int d1, d2;
                        d1 = (pinBuf[i] << 8) & 0xFF00; // shifts in top byte
                        d1 |= (pinBuf[i + 1] & 0x00FF); // shifts in bottom byte
                        ydata_array1[j] = d1;

                        d2 = (pinBuf[i + 2] << 8) & 0xFF00;
                        d2 |= (pinBuf[i + 3] & 0x00FF);
                        ydata_array2[j] = d2;

                        j++;
                }
        }
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
        static int myi = 0;

        switch(msg)
        {
                case WM_CREATE:
                {
                        UINT ret;
                        changedata(); // initialize

                 // make sub window

                        win_chan1 = CreateWindowEx(WS_EX_CLIENTEDGE, "EDIT", "",
                                WS_CHILD | WS_VISIBLE | ES_MULTILINE | ES_AUTOVSCROLL |
                 ES_AUTOHSCROLL, 0, 0, width+5, height/2, hwnd, NULL,
                 GetModuleHandle(NULL), NULL);

                        win_chan2 = CreateWindowEx(WS_EX_CLIENTEDGE, "EDIT", "",
                         ES_AUTOHSCROLL | WS_CHILD | WS_VISIBLE | ES_MULTILINE,
                 0, height/2+10, width+5, height/2+10, hwnd, NULL, GetModuleHandle(NULL), NULL);


                        //ret = SetTimer(hwnd, ID_TIMER, 1, NULL);
                        //ret = SetTimer(hwnd, WM_TIMER, 1, NULL);
                        ret = SetTimer(hwnd, TIMER_MICRO_SECOND, 2, NULL);
                        if(ret == 0)
                                MessageBox(hwnd, "Could not SetTimer()!", "Error", MB_OK |
MB_ICONEXCLAMATION);
                }
                break;
                case WM_TIMER:
                {
                        if( win_chan1 !=NULL)
                        {
                                HDC hdc = GetDC(win_chan1);
                                RECT rcClient;
                                GetClientRect(win_chan1, &rcClient);

                 FillRect(hdc, &rcClient, NULL);
                 myplot(hdc, 0, ydata_array1, "Channel 1",6,width-5,  wh, 3);
                 UpdateWindow(win_chan1);
                                ReleaseDC(hwnd, hdc);
                        }
                        // end of chan1

                        if(win_chan2 !=NULL ){
                                HDC hdc = GetDC(win_chan2);
                                RECT rcClient;
                                GetClientRect(win_chan2, &rcClient);
                                FillRect(hdc, &rcClient, NULL);

                                myplot(hdc, 1, ydata_array2, "Channel 2",6,width-5, wh, 3);
                                UpdateWindow(win_chan2);
                                ReleaseDC(win_chan2, hdc);
                        }
                        // end of chan2
```

```
                        changedata();
                }
                break;

                case WM_DESTROY:
                        KillTimer(hwnd, ID_TIMER);
                        KillTimer(hwnd, WM_TIMER);
                        PostQuitMessage(0);
                break;
                case WM_CLOSE:
                        DestroyWindow(hwnd);
                break;
                default:
                        return DefWindowProc(hwnd, msg, wParam, lParam);
        }
        return 0;
}

void myplot(HDC hdc, int myid, int * mydata, LPTSTR legend, int xstart, int xend, int ystart, int yend)
{

        int i, x = 0, y = 0;
        char xaxis[5];
        int vadjust = 5;

        // calculate the x,y starting and ending points here

                // draw the two axis
                  MoveToEx(hdc,xstart,ystart,0);
                  LineTo(hdc,xend,ystart);

                  MoveToEx(hdc,xstart,yend,0);
                  LineTo(hdc,xstart,ystart);

                // draw a little arrow
                  MoveToEx(hdc,xend,ystart,0);
                  LineTo(hdc,xend-5,ystart-5);
                  MoveToEx(hdc,xend,ystart,0);
                  LineTo(hdc,xend-5,ystart+5);

                // draw a little arrow
                  MoveToEx(hdc,xstart,0,0);
                  LineTo(hdc,0,5);
                  MoveToEx(hdc,xstart,0,0);
                  LineTo(hdc,10,5);

                  TextOut(hdc, 25, 12,legend,9);
// horiz ticks
                xaxis[1]='\0';
                      int j=0;
                   for(i=0;i<xend;i=i+16){

                     xaxis[0]='0'+ (++j)%10;

                      MoveToEx(hdc,xstart+i,ystart-3,0);
                      LineTo(hdc,xstart+i,ystart+3);
                      TextOut(hdc, xstart+i-2, ystart+4,xaxis,1);
                   }
// veritical ticks
                    for(i=ystart;i>yend;i=i-15){
                      MoveToEx(hdc,xstart-2,i,0);
                      LineTo  (hdc,xstart+2,i);
                    }

                    MoveToEx(hdc,xstart,ystart,0);

                    x= 5;

// plot 36 data points

                  for(i=0; i < 32; ++i){   // scan x direction
```

```
                    if(x > xend) break;

                       y= -(mydata[i] / 20) + wh;   // rescale, a hack

                    if(y < 0) y=vadjust;
                    else if(y> wh+10) y= wh -vadjust; // cap it

                   if(i>0)LineTo(hdc,x,y);
                   MoveToEx(hdc,x,y,0);

                   Ellipse(hdc,x-2,y-2,x+2,y+2); // draw a small circle
                   Ellipse(hdc,x-1,y-1,x+1,y+1); // draw a small circle
                   x+= 16;
                  }

}

//-------------------------------------------------------------
//      Main
//-------------------------------------------------------------
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
{
        WNDCLASSEX wc;
        HWND hwnd;
        MSG Msg;

        wc.cbSize       = sizeof(WNDCLASSEX);
        wc.style        = 0;
        wc.lpfnWndProc  = WndProc;
        wc.cbClsExtra   = 0;
        wc.cbWndExtra   = 0;
        wc.hInstance    = hInstance;
        wc.hIcon                = LoadIcon(NULL, IDI_APPLICATION);
        wc.hCursor              = LoadCursor(NULL, IDC_ARROW);
        wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
        wc.lpszMenuName  = NULL;
        wc.lpszClassName = g_szClassName;
        wc.hIconSm              = LoadIcon(NULL, IDI_APPLICATION);

        if(!RegisterClassEx(&wc))
        {
                MessageBox(NULL, "Window Registration Failed!", "Error!",
                        MB_ICONEXCLAMATION | MB_OK);
                return 0;
        }

        hwnd = CreateWindowEx(
                WS_EX_CLIENTEDGE,
                g_szClassName,
                "USB PSOC Scope",
                WS_OVERLAPPEDWINDOW,
                CW_USEDEFAULT, CW_USEDEFAULT, 680, 660,
                NULL, NULL, hInstance, NULL);

        if(hwnd == NULL)
        {
                MessageBox(NULL, "Window Creation Failed!", "Error!",
                        MB_ICONEXCLAMATION | MB_OK);
                return 0;
        }

        ShowWindow(hwnd, nCmdShow);
        UpdateWindow(hwnd);

        while(GetMessage(&Msg, NULL, 0, 0) > 0)
        {
                TranslateMessage(&Msg);
                DispatchMessage(&Msg);
        }
         if(hRead != INVALID_HANDLE_VALUE) { CloseHandle(hRead); }
        return Msg.wParam;
}
```