ANNEX I

Source Code of Program "FishFinder, Version 3.2" in Visual C

<Original source code can be downloaded from RIHN archive database:
http://www.chikyu.ac.jp/rihn/archive_datebase/archive/index.html >

```
/*****************************************************************************
        * Program

*****************************************************************************/

using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace jPicoFishfinder1._0
{
    struct ChannelSettings
    {
        public bool DCcoupled;
        public Imports.Range range;
        public bool enabled;
    }

    class Pwq
    {
        public Imports.PwqConditions[] conditions;
        public short nConditions;
        public Imports.ThresholdDirection direction;
        public uint lower;
        public uint upper;
        public Imports.PulseWidthType type;

        public Pwq(Imports.PwqConditions[] condit,ions
          shor  t nConditions,
            Imports.ThresholdDirectio ndirection,
            uint lower, uint upper,
            Imports.PulseWidthType ty(pe
        {
            this.conditions = conditions;
            this.nConditions = nConditions;
            this.direction = direction;
            this.lower = lower;
            this.upper = upper;
            this.type = type;
        }
    }

    static class Program
    {
```

```
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);

            short handle;
            short s = tatusImports.OpenUnit(out handle);    //1. Open the oscilloscope
            if (status != 0)
            {
                MessageBox.Show("Unable to open device", "ERROR");
            }
            else
            {
                Application.Run(new Form1(handle));
//                MessageBox.Show("Close the device?");
                Imports.CloseUnit(handle);    //12. Close the oscilloscope
            }

        }
    }
}
```

```
/*****************************************************************************
       * Form1
*****************************************************************************/

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using System.IO;
using Emgu.CV;
using Emgu.CV.Structure;
using System.Net;

namespace jPicoFishfinder1._0
{

    public partial class Form1 : Form
    {
        //colorbar for GPS version
        double[] m_dLevel = { 1000, 900, 850, 800, 750, 700, 650 };



        //fishfinder
        private Color[] colorbar = new Color[8] { Color.Navy, Color.Blue, Color.Aqua,
Color.LimeGreen, Color.Yellow, Color.Orange, Color.Red, Color.DarkRed };

        Thread m_threadcvImg;
        int m_cvImgStart = 0;
        Point m_pLetfTop;
        Point m_pRightBottom;
        Bitmap m_cvBmp;
        bool m_bOdd=true;

        //1.1.1
        short[] m_sPingData;

        //AD 2 SV
        double m_dSpeed ;// Double.Parse(txtSpeed.Text);

        double m_dSR ;// Double.Parse(txtSampRate.Text);

        double m_dAbsorb;// Double.Parse(txtAbsorb.Text);

        double m_dTau;// Double.Parse(txtPulseW.Text);

        double m_dVar; // m_dVar = 10 * Math.Log10(0.5 * m_dSpeed * m_dTau * m_dPhi) -
m_dK;
```

```
        //pico below
        private readonly short _handle;
        public const int BUFFER_SIZE = 800;   //sample rage ==20k, set the range here, 1 sample
==0.0375m, 1m==26.7samples  (for 1500m/s); so (30m:800samples, 40m:1068samples)
//20150323
        public const int MAX_CHANNELS = 4;
        public const int QUAD_SCOPE = 4;
        public const int DUAL_SCOPE = 2;

        uint _timebase=1001;
        int _timeInterval;
        bool _logFlag = false;
        bool _saveFlag = false;

        short _oversample = 1;
        bool _scaleVoltages = true;

        ushort[] inputRanges = { 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000,
50000 };
        bool _ready = false;
        short _trig = 0;
        uint _trigAt = 0;
        int _sampleCount = 0;
        uint _startIndex = 0;
        bool _autoStop;
        private ChannelSettings[] _channelSettings;
        private int _channelCount;
        private Imports.Range _firstRange;
        private Imports.Range _lastRange;
        private Imports.ps4000BlockReady _callbackDelegate;
        long second = 10;

        Thread m_threadLoggin;

        public Form1(short handle)
        {
            _handle = handle;
            InitializeComponent();
            GetDeviceInfo();
            pictureBox_echograph.BackColor = Color.DarkBlue;
      //      m_cvBmp = new Bitmap(pictureBox_echograph.Width, pictureBox_echograph.Height);
        }


/****************************************************************************
        * Initialise unit' structure with Variant specific defaults

****************************************************************************/
        void GetDeviceInfo()
        {
            int variant = 0;
```

```
            string[] description = {
                        "Driver Version    ",
                        "USB Version       ",
                        "Hardware Version  ",
                        "Variant Info      ",
                        "Serial            ",
                        "Cal Date          ",
                        "Kernel Ver        "
                       };
            System.Text.StringBuilder line = new System.Text.StringBuilder(80);

            if (_handle >= 0)
            {
               string srLable2="";
               textBox1.Clear();
               for (int i = 0; i < 7; i++)
               {
                  short requiredSize;
                  Imports.GetUnitInfo(_handle, line, 80, out requiredSize, i);
                  if (i == 3)
                  {
                     variant = Convert.ToInt16(line.ToString());
                  }
                  srLable2 = String.Format("{0}: {1}", description[i], line);
                  srLable2 = srLable2 + "\r\n";
                  textBox1.AppendText(srLable2);
               }
               srLable2 = String.Format("Sampling Rate     : {0} ksps", 1000000/(50*(_timebase-
1)));
               textBox1.AppendText(srLable2);

               switch (variant)
               {
                  case (int)Imports.Model.PS4223:
                     _firstRange = Imports.Range.Range_50MV;
                     _lastRange = Imports.Range.Range_100V;
                     _channelCount = DUAL_SCOPE;
                     break;

                  case (int)Imports.Model.PS4224:
                     _firstRange = Imports.Range.Range_50MV;
                     _lastRange = Imports.Range.Range_20V;
                     _channelCount = DUAL_SCOPE;
                     break;

                  case (int)Imports.Model.PS4423:
                     _firstRange = Imports.Range.Range_50MV;
                     _lastRange = Imports.Range.Range_100V;
                     _channelCount = QUAD_SCOPE;
                     break;

                  case (int)Imports.Model.PS4424:
                     _firstRange = Imports.Range.Range_50MV;
```

```
                _lastRange = Imports.Range.Range_20V;
                _channelCount = QUAD_SCOPE;
                break;

            case (int)Imports.Model.PS4226:
                _firstRange = Imports.Range.Range_50MV;
                _lastRange = Imports.Range.Range_20V;
                _channelCount = DUAL_SCOPE;
                break;

            case (int)Imports.Model.PS4227:
                _firstRange = Imports.Range.Range_50MV;
                _lastRange = Imports.Range.Range_20V;
                _channelCount = DUAL_SCOPE;
                break;

            case (int)Imports.Model.PS4262:
                _firstRange = Imports.Range.Range_10MV;
                _lastRange = Imports.Range.Range_20V;
                _channelCount = DUAL_SCOPE;
                break;
        }
    }
}

/****************************************************************************
     * Callback
     * used by PS4000 data block collection calls, on receipt of data.
     * used to set global flags etc checked by user routines

****************************************************************************/
    void BlockCallback(short handle, short status, IntPtr pVoid)
    {
        // flag to say done reading data
        _ready = true;
    }


/****************************************************************************

     * SetDefaults - restore default settings

****************************************************************************/
    void SetDefaults()
    {
        for (int i = 0; i < _channelCount; i++) // reset channels to most recent settings
        {
            Imports.SetChannel(_handle, Imports.Channel.ChannelA + i,
                               (short)(_channelSettings[(int)(Imports.Channel.ChannelA + i)].enabled ?
1 : 0),
                               (short)(_channelSettings[(int)(Imports.Channel.ChannelA +
i)].DCcoupled ? 1 : 0),
```

```
                    _channelSettings[(int)(Imports.Channel.ChannelA + i)].range);
            }
        }


/****************************************************************************
         *
         * Select _timebase, set _oversample to on and time units as nano seconds
         *
****************************************************************************/
        void SetTimebase(uint timebase)
        {
            _timebase = timebase;
            int maxSamples;

            while (Imports.GetTimebase(_handle, _timebase, BUFFER_SIZE, out _timeInterval, 1, out
maxSamples, 0) != 0)
            {
                _timebase++;
            }
            _oversample = 1;
        }


/****************************************************************************
        * SetTrigger
        * this function sets all the required trigger parameters, and calls the
        * triggering functions

*********************************************************************************/
        short SetTrigger(Imports.TriggerChannelProperties[] channelProperties, short
nChannelProperties, Imports.TriggerConditions[] triggerConditions, short nTriggerConditions,
Imports.ThresholdDirection[] directions, Pwq pwq, uint delay, short auxOutputEnabled, int
autoTriggerMs)
        {
            short status;

            if (
              (status =
                Imports.SetTriggerChannelProperties(_handle, channelProperties, nChannelProperties,
auxOutputEnabled,
                                                    autoTriggerMs)) != 0)
            {
                return status;
            }

            if ((status = Imports.SetTriggerChannelConditions(_handle, triggerConditions,
nTriggerConditions)) != 0)
            {
                return status;
            }
```

```
        if (directions == null) directions = new Imports.ThresholdDirection[]
{ Imports.ThresholdDirection.None,
      Imports.ThresholdDirection.None, Imports.ThresholdDirection.None,
Imports.ThresholdDirection.None,
      Imports.ThresholdDirection.None, Imports.ThresholdDirection.None};

        if ((status = Imports.SetTriggerChannelDirections(_handle,
                                        directions[(int)Imports.Channel.ChannelA],
                                        directions[(int)Imports.Channel.ChannelB],
                                        directions[(int)Imports.Channel.ChannelC],
                                        directions[(int)Imports.Channel.ChannelD],
                                        directions[(int)Imports.Channel.External],
                                        directions[(int)Imports.Channel.Aux])) != 0)
        {
           return status;
        }

        if ((status = Imports.SetTriggerDelay(_handle, delay)) != 0)
        {
           return status;
        }

        if (pwq == null) pwq = new Pwq(null, 0, Imports.ThresholdDirection.None, 0, 0,
Imports.PulseWidthType.None);

        status = Imports.SetPulseWidthQualifier(_handle, pwq.conditions,
                                        pwq.nConditions, pwq.direction,
                                        pwq.lower, pwq.upper, pwq.type);

        return status;
     }



/***************************************************************************
      * adc_to_mv
      *
      * Convert an 16-bit ADC count into millivolts

***************************************************************************/
     int adc_to_mv(int raw, int ch)
     {
        return (raw * inputRanges[ch]) / Imports.MaxValue;
     }


/***************************************************************************
      * mv_to_adc
      *
      * Convert a millivolt value into a 16-bit ADC count
      *
      *  (useful for setting trigger thresholds)
```

```
*****************************************************************************/
      short mv_to_adc(short mv, short ch)
      {
         return (short)((mv * Imports.MaxValue) / inputRanges[ch]);
      }


/*****************************************************************************
      * CollectBlockTriggered
      *  this function demonstrates how to collect a single block of data from the
      *  unit, when a trigger event occurs.

*****************************************************************************/
      void SetBlockTrigger()
      {
         short triggerVoltage = mv_to_adc(1000,
(short)_channelSettings[(int)Imports.Channel.ChannelA].range); // ChannelInfo stores ADC counts

         Imports.TriggerChannelProperties[] sourceDetails = new
Imports.TriggerChannelProperties[] {
      new Imports.TriggerChannelProperties(triggerVoltage,
                                 256*10,
                                 triggerVoltage,
                                 256*10,
                                 Imports.Channel.ChannelA,
                                 Imports.ThresholdMode.Level)};

         Imports.TriggerConditions[] conditions = new Imports.TriggerConditions[] {
           new Imports.TriggerConditions(Imports.TriggerState.True,
                                 Imports.TriggerState.DontCare,
                                 Imports.TriggerState.DontCare,
                                 Imports.TriggerState.DontCare,
                                 Imports.TriggerState.DontCare,
                                 Imports.TriggerState.DontCare,
                                 Imports.TriggerState.DontCare)};

         Imports.ThresholdDirection[] directions = new Imports.ThresholdDirection[]
                                       { Imports.ThresholdDirection.Rising,
                                 Imports.ThresholdDirection.None,
                                 Imports.ThresholdDirection.None,
                                 Imports.ThresholdDirection.None,
                                 Imports.ThresholdDirection.None,
                                 Imports.ThresholdDirection.None };
         SetDefaults();
         /* Trigger enabled
          * Rising edge
          * Threshold = 1000mV */
         SetTrigger(sourceDetails, 1, conditions, 1, directions, null, 0, 0, 0);  //4. SetTrigger
      }
```

```
/*****************************************************************************
      * DataLoggin - Logging Data to file

*****************************************************************************/
      public void DataLoggin()
      {
         // setup devices

//          _timebase = 1;

         _channelSettings = new ChannelSettings[MAX_CHANNELS];


         uint sampleCount = BUFFER_SIZE;
         PinnedArray<short>[] bufPinned = new PinnedArray<short>[1];

         int timeIndisposed;

         for (int i = 0; i < MAX_CHANNELS; i++)
         {
            _channelSettings[i].enabled = true;
            _channelSettings[i].DCcoupled = true;
            _channelSettings[i].range = Imports.Range.Range_5V;
         }

         SetDefaults(); //2. Select channel ranges and AC/DC coupling

         SetTimebase(1001); //3. Select timebase until the required nanoseconds per sample is
located

         SetBlockTrigger();  //4. Use the trigger setup functions to set up the trigger


         short[] buffers = new short[sampleCount];
         bufPinned[0] = new PinnedArray<short>(buffers);
         int status = Imports.SetDataBuffer(_handle, (Imports.Channel)1, buffers,
(int)sampleCount);//7. tell the driver where your memory buffer is

         if (!Directory.Exists("C:\\FishloggerData"))
            Directory.CreateDirectory("C:\\FishloggerData");
         TextWriter writer = new StreamWriter("C:\\FishloggerData\\" +
DateTime.Now.ToString("yyyyMMdd_HHmmss") + ".csv", false); //hh 12hour; HH 24hour

         //1.1 PingData
         m_sPingData = new short[sampleCount];

         double dLogData;
         double dShowData;

         while (_logFlag == true)
         {
            _ready = false;
```

```
            _callbackDelegate = BlockCallback;
            //5. Start the oscilloscope running
            Imports.RunBlock(_handle, 0, (int)sampleCount, _timebase, _oversample, out
timeIndisposed, 0, _callbackDelegate, IntPtr.Zero);

            while (!_ready) //6. wait until the oscilloscope is ready using the ps4000BlockReady
Callback
            {
                Thread.Sleep(100);
            }

            //***********write data to file*****************
            if (_ready)   //
            {
                short overflow;
                Imports.GetValues(_handle, 0, ref sampleCount, 1,
Imports.DownSamplingMode.None, 0, out overflow);  //8. Transfer the block of data from the
ocilloscope
                writer.Write(DateTime.Now.ToString("yyyyMMdd_HH:mm:ss,"));
                for (int i = 0; i < sampleCount; i++)  //9 handle the data
                {

                    dLogData = adc_to_mv(bufPinned[0].Target[i],
(short)_channelSettings[(int)Imports.Channel.ChannelB].range); //voltage
                  //  dLogData = bufPinned[0].Target[i];      //AD

                    dShowData = dLogData;

                    writer.Write("{0},", dLogData);


                    if (dShowData > m_dLevel[0])
                        m_sPingData[i] = 7;
                    else if (dShowData > m_dLevel[1])
                        m_sPingData[i] = 6;
                    else if (dShowData > m_dLevel[2])
                        m_sPingData[i] = 5;
                    else if (dShowData > m_dLevel[3])
                        m_sPingData[i] = 4;
                    else if (dShowData > m_dLevel[4])
                        m_sPingData[i] = 3;
                    else if (dShowData > m_dLevel[5])
                        m_sPingData[i] = 2;
                    else if (dShowData > m_dLevel[6])
                        m_sPingData[i] = 1;
                    else
                        m_sPingData[i] = 0;

                }
                writer.WriteLine();

                //draw one Ping
```

```
            //************************

                //draw by Main thread using Invoker
                MethodInvoker mi = new MethodInvoker(this.drawEchograph);
                this.BeginInvoke(mi);

            //***************/

        }
        //************************************************/
    }                   //10. repeat steps 5-9

    _logFlag = false;
    Imports.Stop(_handle); //11.Stop the oscilloscope
    writer.Close();
    foreach (PinnedArray<short> p in bufPinned)
    {
        if (p != null)
            p.Dispose();
    }
    _saveFlag = true;

}

private void btnDataLoggin_Click(object sender, EventArgs e)
{
    timer1.Enabled = false;

    second=0;
    label1.Text = new DateTime(second * 10000000).ToLongTimeString();


    _logFlag = true;
    btnDataLoggin.Enabled = false;
    btnStopLoggin.Enabled = true;
    m_threadLoggin = new Thread(new ThreadStart(DataLoggin));
    m_threadLoggin.Start();
    timer2.Enabled = true;
    label3.Text = "Data is logging... ... ...";
}

private void btnStopLoggin_Click(object sender, EventArgs e)
{
    timer2.Enabled = false; //no auto logging anymore

    m_threadLoggin.Suspend();
    _logFlag = false;
    btnStopLoggin.Enabled = false;
    btnDataLoggin.Enabled = true;
    m_threadLoggin.Resume();
    label3.Text = "Data logging Over";

}
```

```
    private void timer1_Tick(object sender, EventArgs e) //one second interval
    {
        second--;
        label1.Text = new DateTime(second*10000000).ToLongTimeString();

        if (second == 0)
        {
            timer1.Enabled = false;

            _logFlag = true;
            btnDataLoggin.Enabled = false;
            btnStopLoggin.Enabled = true;
            m_threadLoggin = new Thread(new ThreadStart(DataLoggin));
            m_threadLoggin.Start();
            timer2.Enabled = true;
            label3.Text = "Data is logging... ... ...";
        }
    }

    private void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {
        if (MessageBox.Show("Sure to Stop Logging before closing the Application", "Attention",
MessageBoxButtons.OKCancel, MessageBoxIcon.Question) == DialogResult.OK)
        {

        }
        else e.Cancel=true;
    }

    private void timer2_Tick(object sender, EventArgs e) //one hour interval auto logging file
    {
        //*************Close current file first***********
        m_threadLoggin.Suspend();
        _logFlag = false;
        btnStopLoggin.Enabled = false;
        btnDataLoggin.Enabled = true;
        m_threadLoggin.Resume();
        label3.Text = "Data logging Over";

        //*************New file, restart thread**************
        while (_saveFlag == false);  //wait for the terminating of current m_threadLogging
        _saveFlag = false; //clear;
        _logFlag = true;
        btnDataLoggin.Enabled = false;
        btnStopLoggin.Enabled = true;
        m_threadLoggin = new Thread(new ThreadStart(DataLoggin));
        m_threadLoggin.Start();
        label3.Text = "Data is logging... ... ...";
    }
```

```
        //jPicoFishfinder 1.1 add functions as below
        private void drawEchograph()
        {
            uint isamples = BUFFER_SIZE;

            Bitmap bitmap = new Bitmap(this.pictureBox_echograph.Width,
this.pictureBox_echograph.Height);
            Graphics gEcho = Graphics.FromImage(bitmap);    //this is like doublebuffer, draw on the
bitmap, and show on the picTureBox


            float y = pictureBox_echograph.Height / (float)isamples;
            float x = this.pictureBox_echograph.Width - 1;// gEcho.PageScale;
            //            int x =1311; //(840*3.125/2)

            if (this.pictureBox_echograph.Image == null)
            {
                for (int i = 0; i < isamples; i++)
                    gEcho.DrawLine(new Pen(colorbar[m_sPingData[i]]), x, i * y, x, (i + 1) * y);
            }
            else
            {
                gEcho.DrawImage(this.pictureBox_echograph.Image, -1, 0); //draw image from
current picturebox to a new position DrawImage(image,x,y),==translate
                //              gEcho.DrawImage(this.pictureBox_echograph.Image, new Rectangle(0, 0,
pictureBox_echograph.Width-1, pictureBox_echograph.Height), new Rectangle(1, 0,
this.pictureBox_echograph.Image.Width-1, this.pictureBox_echograph.Image.Height),
GraphicsUnit.Pixel);
                for (int i = 0; i < isamples; i++)
                    gEcho.DrawLine(new Pen(colorbar[m_sPingData[i]]), x, i * y, x, (i + 1) * y);
            }
            //          DrawDepthLable(gEcho, irange);  // draw depth label
            this.pictureBox_echograph.Image = bitmap; //update to the picBox,

            gEcho.Dispose();

        }

    }
}
```

```
/*****************************************************************************
      * Form1_Designer
*****************************************************************************/

namespace jPicoFishfinder1._0
{
   partial class Form1
   {
      /// <summary>
      /// Required designer variable.
      /// </summary>
      private System.ComponentModel.IContainer components = null;

      /// <summary>
      /// Clean up any resources being used.
      /// </summary>
      /// <param name="disposing">true if managed resources should be disposed; otherwise,
false.</param>
      protected override void Dispose(bool disposing)
      {
         if (disposing && (components != null))
         {
            components.Dispose();
         }
         base.Dispose(disposing);
      }

      #region Windows Form Designer generated code

      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
         this.components = new System.ComponentModel.Container();
         System.ComponentModel.ComponentResourceManager resources = new
System.ComponentModel.ComponentResourceManager(typeof(Form1));
         this.btnDataLoggin = new System.Windows.Forms.Button();
         this.btnStopLoggin = new System.Windows.Forms.Button();
         this.label1 = new System.Windows.Forms.Label();
         this.timer1 = new System.Windows.Forms.Timer(this.components);
         this.label2 = new System.Windows.Forms.Label();
         this.textBox1 = new System.Windows.Forms.TextBox();
         this.timer2 = new System.Windows.Forms.Timer(this.components);
         this.label3 = new System.Windows.Forms.Label();
         this.pictureBox_echograph = new System.Windows.Forms.PictureBox();
         this.pictureBox_Colorbar = new System.Windows.Forms.PictureBox();
         ((System.ComponentModel.ISupportInitialize)(this.pictureBox_echograph)).BeginInit();
         ((System.ComponentModel.ISupportInitialize)(this.pictureBox_Colorbar)).BeginInit();
         this.SuspendLayout();
         //
         // btnDataLoggin
         //
```

```
            this.btnDataLoggin.Location = new System.Drawing.Point(940, 247);
            this.btnDataLoggin.Name = "btnDataLoggin";
            this.btnDataLoggin.Size = new System.Drawing.Size(101, 37);
            this.btnDataLoggin.TabIndex = 0;
            this.btnDataLoggin.Text = "Start Logging";
            this.btnDataLoggin.UseVisualStyleBackColor = true;
            this.btnDataLoggin.Click += new System.EventHandler(this.btnDataLoggin_Click);
            //
            // btnStopLoggin
            //
            this.btnStopLoggin.Enabled = false;
            this.btnStopLoggin.Location = new System.Drawing.Point(1055, 247);
            this.btnStopLoggin.Name = "btnStopLoggin";
            this.btnStopLoggin.Size = new System.Drawing.Size(98, 37);
            this.btnStopLoggin.TabIndex = 1;
            this.btnStopLoggin.Text = "Stop Logging";
            this.btnStopLoggin.UseVisualStyleBackColor = true;
            this.btnStopLoggin.Click += new System.EventHandler(this.btnStopLoggin_Click);
            //
            // label1
            //
            this.label1.AutoSize = true;
            this.label1.Font = new System.Drawing.Font("SimSun", 30F,
System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(134)));
            this.label1.Location = new System.Drawing.Point(954, 189);
            this.label1.Name = "label1";
            this.label1.Size = new System.Drawing.Size(185, 40);
            this.label1.TabIndex = 2;
            this.label1.Text = "00:00:00";
            //
            // timer1
            //
            this.timer1.Enabled = true;
            this.timer1.Interval = 1000;
            this.timer1.Tick += new System.EventHandler(this.timer1_Tick);
            //
            // label2
            //
            this.label2.AutoSize = true;
            this.label2.Location = new System.Drawing.Point(934, 9);
            this.label2.Name = "label2";
            this.label2.Size = new System.Drawing.Size(103, 12);
            this.label2.TabIndex = 3;
            this.label2.Text = "Device Information:\r\n";
            //
            // textBox1
            //
            this.textBox1.Location = new System.Drawing.Point(936, 33);
            this.textBox1.Multiline = true;
            this.textBox1.Name = "textBox1";
            this.textBox1.ReadOnly = true;
```

```
            this.textBox1.Size = new System.Drawing.Size(226, 106);
            this.textBox1.TabIndex = 4;
            this.textBox1.Text = "Driver Version    : \r\nUSB Version       : \r\nHardware Version  :
\r\nVariant Info   " +
                "   : \r\nSerial          : \r\nCal Date         : \r\nKernel Ver       :\r\nSampling" +
                " Rate     : ";
            //
            // timer2
            //
            this.timer2.Interval = 3600000;
            this.timer2.Tick += new System.EventHandler(this.timer2_Tick);
            //
            // label3
            //
            this.label3.AutoSize = true;
            this.label3.Font = new System.Drawing.Font("SimSun", 12F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(134)));
            this.label3.Location = new System.Drawing.Point(937, 157);
            this.label3.Name = "label3";
            this.label3.Size = new System.Drawing.Size(216, 16);
            this.label3.TabIndex = 5;
            this.label3.Text = "Data logging will start...";
            //
            // pictureBox_echograph
            //
            this.pictureBox_echograph.Location = new System.Drawing.Point(13, 13);
            this.pictureBox_echograph.Name = "pictureBox_echograph";
            this.pictureBox_echograph.Size = new System.Drawing.Size(904, 676);
            this.pictureBox_echograph.TabIndex = 6;
            this.pictureBox_echograph.TabStop = false;
            //
            // pictureBox_Colorbar
            //
            this.pictureBox_Colorbar.Anchor =
((System.Windows.Forms.AnchorStyles)(((System.Windows.Forms.AnchorStyles.Top |
System.Windows.Forms.AnchorStyles.Bottom)
                        | System.Windows.Forms.AnchorStyles.Right)));
            this.pictureBox_Colorbar.Image =
((System.Drawing.Image)(resources.GetObject("pictureBox_Colorbar.Image")));
            this.pictureBox_Colorbar.InitialImage =
((System.Drawing.Image)(resources.GetObject("pictureBox_Colorbar.InitialImage")));
            this.pictureBox_Colorbar.Location = new System.Drawing.Point(13, 172);
            this.pictureBox_Colorbar.Name = "pictureBox_Colorbar";
            this.pictureBox_Colorbar.Size = new System.Drawing.Size(24, 304);
            this.pictureBox_Colorbar.SizeMode =

System.Windows.Forms.PictureBoxSizeMode.StretchImage;
            this.pictureBox_Colorbar.TabIndex = 21;
            this.pictureBox_Colorbar.TabStop = false;
            //
            // Form1
            //
```

```
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 12F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(1165, 701);
            this.Controls.Add(this.pictureBox_Colorbar);
            this.Controls.Add(this.pictureBox_echograph);
            this.Controls.Add(this.label3);
            this.Controls.Add(this.textBox1);
            this.Controls.Add(this.label2);
            this.Controls.Add(this.label1);
            this.Controls.Add(this.btnStopLoggin);
            this.Controls.Add(this.btnDataLoggin);
            this.Icon = ((System.Drawing.Icon)(resources.GetObject("$this.Icon")));
            this.Name = "Form1";
            this.Text = "jPicoFishfinder_GPS1.0";
            this.FormClosing += new
System.Windows.Forms.FormClosingEventHandler(this.Form1_FormClosing);
            ((System.ComponentModel.ISupportInitialize)(this.pictureBox_echograph)).EndInit();
            ((System.ComponentModel.ISupportInitialize)(this.pictureBox_Colorbar)).EndInit();
            this.ResumeLayout(false);
            this.PerformLayout();

        }

        #endregion

        private System.Windows.Forms.Button btnDataLoggin;
        private System.Windows.Forms.Button btnStopLoggin;
        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.Timer timer1;
        private System.Windows.Forms.Label label2;
        private System.Windows.Forms.TextBox textBox1;
        private System.Windows.Forms.Timer timer2;
        private System.Windows.Forms.Label label3;
        private System.Windows.Forms.PictureBox pictureBox_echograph;
        private System.Windows.Forms.PictureBox pictureBox_Colorbar;
    }
}
```

```
/****************************************************************
*
* Filename:    PS4000Imports.cs
*
* Copyright:   Pico Technology Limited 2009
*
* Author:      MJL
*
* Description:
*   This file contains all the .NET wrapper calls needed to support
*   the console example. It also has the enums and structs required
*   by the (wrapped) function calls.
*
* History:
*    14Dec06 MJL     Created
*        15Oct09     RPM Modified for PS4000
*
* Revision Info: "file %n date %f revision %v"
*                                                  ""
****************************************************************/

using System;
using System.Runtime.InteropServices;
using System.Text;

namespace jPicoFishfinder1._0
{
        class Imports
        {
                #region constants
                private const string _DRIVER_FILENAME = "ps4000.dll";

                public const int MaxValue = 32764;
                #endregion

                #region Driver enums

                public enum Channel : int
                {
                        ChannelA,
                        ChannelB,
```

```
        ChannelC,
        ChannelD,
        External,
        Aux,
        None,
    }

    public enum Range : int
    {
        Range_10MV,
        Range_20MV,
        Range_50MV,
        Range_100MV,
        Range_200MV,
        Range_500MV,
        Range_1V,
        Range_2V,
        Range_5V,
        Range_10V,
        Range_20V,
        Range_50V,
Range_100V,
    }

    public enum ReportedTimeUnits : int
    {
        FemtoSeconds,
        PicoSeconds,
        NanoSeconds,
        MicroSeconds,
        MilliSeconds,
        Seconds,
    }

    public enum ThresholdMode : int
    {
        Level,
        Window
    }

    public enum ThresholdDirection : int
    {
        // Values for level threshold mode
        //
        Above,
```

```
                Below,
                Rising,
                Falling,
                RisingOrFalling,

                // Values for window threshold mode
                //
                Inside = Above,
                Outside = Below,
                Enter = Rising,
                Exit = Falling,
                EnterOrExit = RisingOrFalling,

                None = Rising,
        }

        public enum DownSamplingMode : int
        {
                None,
                Aggregate
        }

        public enum PulseWidthType : int
        {
                None,
                LessThan,
                GreaterThan,
                InRange,
                OutOfRange
        }

        public enum TriggerState : int
        {
                DontCare,
                True,
                False,
        }

public enum Model : int
{
   NONE = 0,
   PS4223 = 4223,
   PS4224 = 4224,
   PS4423 = 4423,
   PS4424 = 4424,
   PS4226 = 4226,
   PS4227 = 4227,
   PS4262 = 4262,
```

```
    }
#endregion

        [StructLayout(LayoutKind.Sequential, Pack = 1)]
        public struct TriggerChannelProperties
        {
            public short ThresholdMajor;
            public ushort HysteresisMajor;
            public short ThresholdMinor;
            public ushort HysteresisMinor;
            public Channel Channel;
            public ThresholdMode ThresholdMode;


            public TriggerChannelProperties(
                short thresholdMajor,
                ushort hysteresisMajor,
                short thresholdMinor,
                ushort hysteresisMinor,
                Channel channel,
                ThresholdMode thresholdMode)
            {
                this.ThresholdMajor = thresholdMajor;
                this.HysteresisMajor = hysteresisMajor;
                this.ThresholdMinor = thresholdMinor;
                this.HysteresisMinor = hysteresisMinor;
                this.Channel = channel;
                this.ThresholdMode = thresholdMode;
            }
        }

        [StructLayout(LayoutKind.Sequential, Pack = 1)]
        public struct TriggerConditions
        {
            public TriggerState ChannelA;
            public TriggerState ChannelB;
            public TriggerState ChannelC;
            public TriggerState ChannelD;
            public TriggerState External;
            public TriggerState Aux;
            public TriggerState Pwq;

            public TriggerConditions(
                TriggerState channelA,
                TriggerState channelB,
                TriggerState channelC,
                TriggerState channelD,
                TriggerState external,
                TriggerState aux,
                TriggerState pwq)
            {
                this.ChannelA = channelA;
```

```
            this.ChannelB = channelB;
            this.ChannelC = channelC;
            this.ChannelD = channelD;
            this.External = external;
            this.Aux = aux;
            this.Pwq = pwq;
        }
}

[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct PwqConditions
{
        public TriggerState ChannelA;
        public TriggerState ChannelB;
        public TriggerState ChannelC;
        public TriggerState ChannelD;
        public TriggerState External;
        public TriggerState Aux;

        public PwqConditions(
            TriggerState channelA,
            TriggerState channelB,
            TriggerState channelC,
            TriggerState channelD,
            TriggerState external,
            TriggerState aux)
        {
            this.ChannelA = channelA;
            this.ChannelB = channelB;
            this.ChannelC = channelC;
            this.ChannelD = channelD;
            this.External = external;
            this.Aux = aux;
        }
}

#region Driver Imports
#region Callback delegates

public delegate void ps4000BlockReady(short handle, short status, IntPtr pVoid);


public delegate void ps4000StreamingReady(
        short handle,
        int noOfSamples,
        uint startIndex,
        short ov,
        uint triggerAt,
        short triggered,
        short autoStop,
        IntPtr pVoid);


public delegate void ps4000DataReady(
        short handle,
```

```
            int noOfSamples,
            short overflow,
            uint triggerAt,
            short triggered,
            IntPtr pVoid);
        #endregion

        [DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000OpenUnit")]
        public static extern short OpenUnit(out short handle);

        [DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000CloseUnit")]
        public static extern short CloseUnit(short handle);

        [DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000RunBlock")]
        public static extern short RunBlock(
            short handle,
            int noOfPreTriggerSamples,
            int noOfPostTriggerSamples,
            uint timebase,
            short oversample,
            out int timeIndisposedMs,
            ushort segmentIndex,


ps4000BlockReady lpps4000BlockReady,
            IntPtr pVoid);

        [DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000Stop")]
        public static extern short Stop(short handle);

        [DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000SetChannel")]
        public static extern short SetChannel(
            short handle,
            Channel channel,
            short enabled,
            short dc,
            Range range);

        [DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000SetDataBuffers")]
        public static extern short SetDataBuffers(
            short handle,
            Channel channel,
            short[] bufferMax,
            short[] bufferMin,
            int bufferLth);

[DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000SetDataBuffer")]
        public static extern short SetDataBuffer(
            short handle,
            Channel channel,
```

```
                short[] buffer,
                int bufferLth);

        [DllImport(_DRIVER_FILENAME, EntryPoint =
"ps4000SetTriggerChannelDirections")]
        public static extern short SetTriggerChannelDirections(
                short handle,

    ThresholdDirection channelA,

    ThresholdDirection channelB,

    ThresholdDirection channelC,

    ThresholdDirection channelD,

    ThresholdDirection ext,

    ThresholdDirection aux);

        [DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000GetTimebase")]
        public static extern short GetTimebase(
                short handle,
                uint timebase,
                int noSamples,
                out int timeIntervalNanoseconds,
                short oversample,
                out int maxSamples,
                ushort segmentIndex);

        [DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000GetValues")]
        public static extern short GetValues(
                short handle,
                uint startIndex,
                ref uint noOfSamples,
                uint downSampleRatio,
                DownSamplingMode downSampleDownSamplingMode,
                ushort segmentIndex,
                out short overflow);

        [DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000SetPulseWidthQualifier")]
        public static extern short SetPulseWidthQualifier(
                short handle,
                PwqConditions[] conditions,
                short numConditions,
                ThresholdDirection direction,
                uint lower,
                uint upper,
```

```
                PulseWidthType type);

            [DllImport(_DRIVER_FILENAME, EntryPoint =
"ps4000SetTriggerChannelProperties")]
            public static extern short SetTriggerChannelProperties(
                short handle,
                TriggerChannelProperties[] channelProperties,
                short numChannelProperties,
                short auxOutputEnable,
                int autoTriggerMilliseconds);

            [DllImport(_DRIVER_FILENAME, EntryPoint =
"ps4000SetTriggerChannelConditions")]
            public static extern short SetTriggerChannelConditions(
                short handle,
                TriggerConditions[] conditions,
                short numConditions);

            [DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000SetTriggerDelay")]
            public static extern short SetTriggerDelay(short handle, uint delay);

            [DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000GetUnitInfo")]
            public static extern short GetUnitInfo(short handle, StringBuilder infoString, short
stringLength, out short requiredSize, int info);

            [DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000RunStreaming")]
            public static extern short RunStreaming(
                short handle,
                ref uint sampleInterval,
                ReportedTimeUnits sampleIntervalTimeUnits,
                uint maxPreTriggerSamples,
                uint maxPostPreTriggerSamples,
                bool autoStop,
                uint downSamplingRation,
                uint overviewBufferSize);

            [DllImport(_DRIVER_FILENAME, EntryPoint =
"ps4000GetStreamingLatestValues")]
            public static extern short GetStreamingLatestValues(
                short handle,
                ps4000StreamingReady lpps4000StreamingReady,
                IntPtr pVoid);

            [DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000SetNoOfCaptures")]
            public static extern short SetNoOfRapidCaptures(
                short handle,
                ushort nWaveforms);
```

```
        [DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000MemorySegments")]
        public static extern short MemorySegments(
            short handle,
            ushort nSegments,
            out int nMaxSamples);

        [DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000SetDataBufferBulk")]
        public static extern short SetDataBuffersRapid(
            short handle,
            Channel channel,
            short[] buffer,
            int bufferLth,
            ushort waveform);

        [DllImport(_DRIVER_FILENAME, EntryPoint = "ps4000GetValuesBulk")]
        public static extern short GetValuesRapid(
            short handle,
            ref uint noOfSamples,
            ushort fromSegmentIndex,
            ushort toSegmentIndex,
            short[] overflows);
        #endregion
    }
}
```

```
/*****************************************************************************
      * PS4000Pinned Array

*****************************************************************************/

using System;
using System.Runtime.InteropServices;


namespace jPicoFishfinder1
{
  public class PinnedArray<T>
  {
        GCHandle _pinnedHandle;
         private bool _disposed;

         public PinnedArray(int arraySize) : this(new T[arraySize]) { }

         public PinnedArray(T[] array)
        {
                _pinnedHandle = GCHandle.Alloc(array, GCHandleType.Pinned);
         }

        ~PinnedArray()
         {
                Dispose();
         }

         public T[] Target
        {
                 get { return (T[])_pinnedHandle.Target; }
         }

          public static implicit operator T[](PinnedArray<T> a)
         {
                 if (a == null)
                 return null;
                 else
                 return (T[])a._pinnedHandle.Target;
        }

         public void Dispose()
         {
                if (!_disposed)
                 {
                        _pinnedHandle.Free();
                        _disposed = true;

                        GC.SuppressFinalize(this);
                 }
         }
  }
}
```