Nano River Technologies ● www.nanorivertech.com ● support@nanorivertech.com

ViperBoard Application Examples

Nano River Technologies
October 2010

Table of Contents

ABBREVIATIONS

API	Application Programming Interface
EEPROM	Electrically Erasable Programmable Read Only Memory
G++	C++ compiler for Linux (part of GCC)
GCC	GNU Compiler Collection
GPIO	General Purpose IO
GPIOA	ViperBoard GPIO Port A (advanced GPIO interface)
GPIOB	ViperBoard GPIO Port B (digital IO interface)
I2C	Inter-Integrated Circuit
IDE	Integrated Design Environment
IIC	Inter-Integrated Circuit (same as I2C)
IO	Input / Output
Master	An interface which supplies the clock like the SPI master or I2C master on ViperBoard
NRT	Nano River Technologies
Slave	An interface which receives the clock like the SPI slave or I2C slave on ViperBoard
SPI	Serial Peripheral Interface
USB	Universal Serial Bus

1. Overview

This document provides information about the six application examples which come with the ViperBoard from Nano River Technologies. The application examples aim to show how to build your own C/C++ and Visual C++ applications and interface to the ViperBoard API.

The four example applications described in the document are:

Configuration Example (Windows):	This is a simple Command Tool (DOS) based program with GPIO, SPI, I2C and analogue IO.
Windows GPIO Tool:	This is a simple Visual C++ GUI application to demonstrate GPIO as digital IO.
Advanved Windows Analog/Digital IO Tool:	This is a more complex Visual C++ GUI application to demonstrate advanced capabilities of the GPIO and analogue inputs.
Windows SPI Tool:	This is a simple Visual C++ GUI application to demonstrate SPI master and SPI slave capabilities.
Windows I2C Tool:	This is a simple Visual C++ GUI application to demonstrate I2C master and I2C slave capabilities.
Windows EEPROM Programmer:	This is a complex Visual C++ GUI application to demonstrate how to make a more complex application with I2C master and SPI master functionality.

The Configuration Example has also been ported to Linux and MacOs. This runs using GCC.

Configuration Example (Linux and MacOS):	This is a simple Command Tool (DOS) based program with GPIO, SPI, I2C and analogue IO.

One_ example is provided to show how to use ViperBoard from Xcode Cocoa, which is the current GUI IDE for MacOS.

I2C Checker Tool:	This is a simple XCode Cocoa application for MacOS showing how to write a GUI application with I2C master capabilities.

2. Configuration Example (Windows)

This example is very simple C/C++ console application for Windows showing how to call basic functions associated with SPI, I2C, GPIO and analogue IO. The example is a good starting point if you want to try the ViperBoard for the first time and want to see a simple console application.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++ NANO RIVER TECHNOLOGIES                                      ++
++   CONFIGURATION EXAMPLE FOR VIPERBOARD (05 Nov 2009)         ++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Open Device....
   -> ViperBoard Connected!!!
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

 Initialisation
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++
 GPIO A Test
++++++++++++++++++++++
 Write AA
 Write 55
 Make bit 0 pulsed
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++
 GPIO B Test
++++++++++++++++++++++
 Write AA
 Write 55
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++
 Analogue Input Test
++++++++++++++++++++++
Analogue Channel #00 : E2
Analogue Channel #01 : 00
Analogue Channel #02 : FA
Analogue Channel #03 : CA
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++
 SPI Test
++++++++++++++++++++++
SPI Configure Channel 0...
SPI Set Frequency ...

SPI Slave Data (Written)  : AABBCCDD
SPI Master Data (To send) : 11223344

SPI Master Send Channel 0...

SPI Master Data (Read)    : AABBCCDD
SPI Slave Data (Read)     : 11223344
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++
 I2C Test
++++++++++++++++++++++
I2C Devices Connected ... 0x2B

Master I2C write...
Master Buffer    : AABBCCDDFF
Slave Buffer     : AABBCCDDFF

Master I2C read...
Slave Buffer     : 1122334455
Master Buffer    : 1122334455


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 TEST COMPLETE ....
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

<<<<< PRESS any KEY >>>>>>
```

2.1. Functionality

The example starts by testing GPIO Port A. This is the advanced GPIOs that can be programmed as PWM, pulsed, digital IO or interrupt input. In this example we write an AA pattern then a 55 pattern. Bit 0 is then pulsed continuously.

Next, GPIO Port B is tested. This port is simple digital IO. The port is tested by writing an AA then 55 pattern.

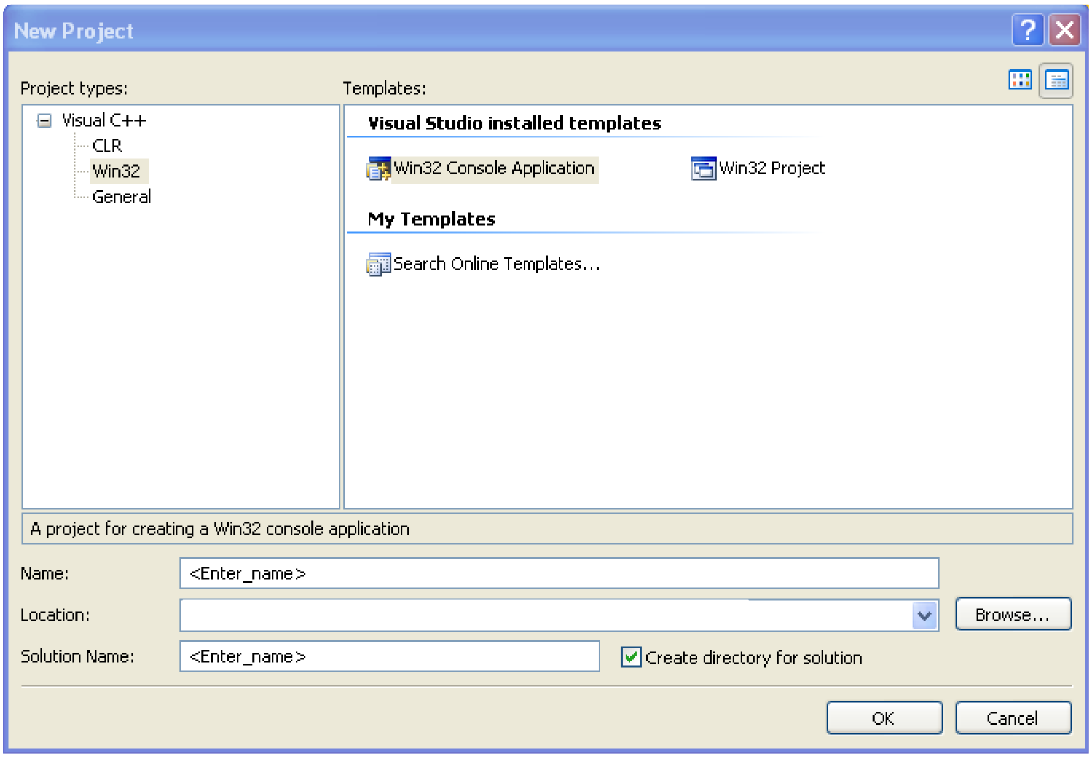
The analogue inputs are then tested. A read is made of each of the input channels.

Next to be tested is the SPI interface. This interface consists of a master and slave SPI interface. In the example we expect that master is LOOPED BACK to the slave. Since master and slave can work independently the example does a simple swap of data between master and slave and in full duplex. Data originally in the master is transferred to the slave and data originally in the slave is transferred to the master by the end of a single transaction.

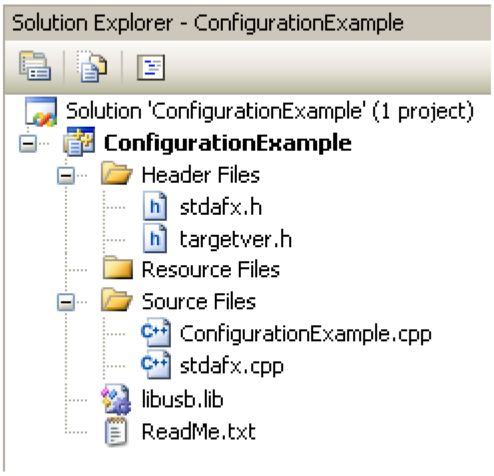
Finally the I2C interface is demonstrated. Again we have master and slave I2C interfaces to test. Both master and slave are connected to the same pins on ViperBoard so by arming the slave it is possible to do transactions between master and slave with NO external connection. The first exercise is to perform a scan of all devices connected on the I2C bus. If the I2C slave is armed then its device ID will appear in the connected device list. I2C master write and I2C master read also follow. If the I2C is armed then we can get data transfer between the master and slave.

2.2. Project Structure

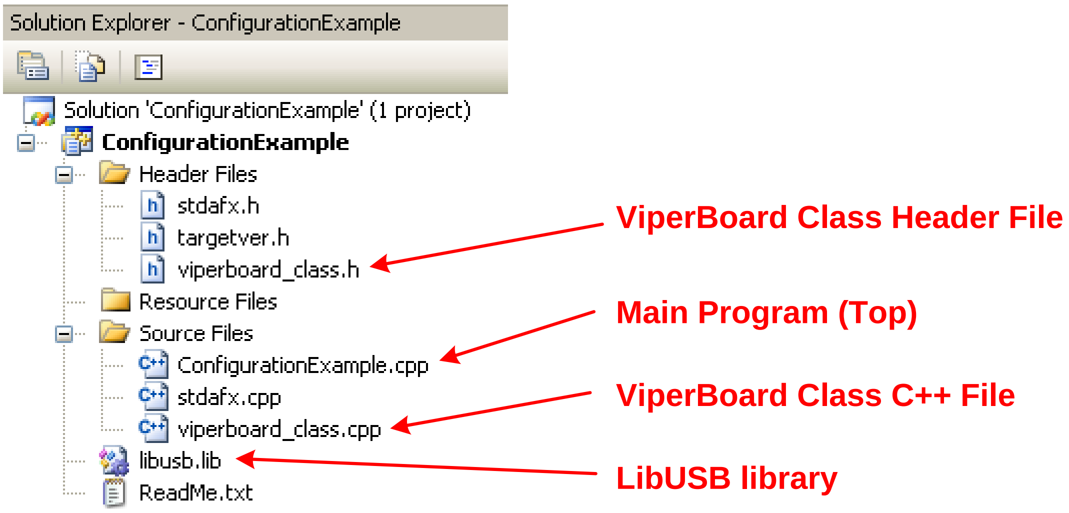
In order to create such a console application in Visual C++, first start up a new project and select "Win32 Console Application" as the project type.



This will create a standard console application. The project file structure will then look as follows.



To this, one must add the ViperBoard class header file (viperboard_class.h), the ViperBoard class C++ file (viperboard_class.cpp) and the libusb library (libusb.lib) as follows.



2.3. Design Description

The application is implemented using a single file (ConfigurationExample.cpp). The complete file and project can be downloaded from the web-site.

The file starts by linking to the ViperBoard class library header file and the standard project header file:

```
//////////////////////////////////////////////////////////////////////////
//
// Nano River Technologies
//
// File:         Configuration Example (ConfigurationExample.cpp)
//
// Desciption:   This is a console application to do basic SPI, I2C,
//               GPIO and analogue input functions with ViperBoard. The functions
//               used is just a sub-set of what is possible, but provides the way
//               to interface to ViperBoard from a console application.
//
// Revision:     Version 1.0
//
//////////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "../../common/viperboard_class.h"
using namespace std;
```

The file declares some global variables used in the application:

```
// Global Variables
//
BOOL        connected;               // True if the ViperBoard is connected
NANO_RESULT res;                     // Result of a ViperBoard function
VBc         VB;                      // Viperboard class library
BYTE        ADC_0;                   // ADC data read - channel 0
BYTE        ADC_1;                   // ADC data read - channel 1
BYTE        ADC_2;                   // ADC data read - channel 2
BYTE        ADC_3;                   // ADC data read - channel 3
BYTE        InBuffer[256];           // I2C input buffer
BYTE        OutBuffer[256];          // I2C output buffer
DEV_LIST    lst;                     // I2C device list
BYTE        in_buffer_m[512];        // SPI master buffer in
BYTE        out_buffer_m[512];       // SPI master buffer out
BYTE        in_buffer_s[512];        // SPI slave buffer in
BYTE        out_buffer_s[512];       // SPI slave buffer out
BOOL        IICMasterError;          // I2C master error
char        c;                       // Local character
int         i;                       // Loop variable
```

The main program starts by printing out a startup banner and calling the user instructions:

```
int _tmain(int argc, _TCHAR* argv[])
{

    printf("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");
    printf("++ NANO RIVER TECHNOLOGIES                                      ++\n");
    printf("++   CONFIGURATION EXAMPLE FOR VIPERBOARD (05 Nov 2009)         ++\n");
    printf("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n\n");
    printf("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");
    printf("Open Device....\n");
```

ViperBoard is then initialised by calling the *OpenDevice()* in the ViperBoard class.

```
    connected=VB.OpenDevice();
    if (connected)
        printf ("   -> ViperBoard Connected!!!\n");
    else
    printf ("   -> ViperBoard NOT Connected!!!\n");
    printf("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n\n");
    c=getch();
```

The program first initialises all GPIOs as digital outputs and writes 0 to them (*Nano_GPIOASetDigitalOutputMode(), Nano_GPIOBSetDirection() and Nano_GPIOBWrite()).* The SPI and I2C events are then flushed by disarming the SPI and I2C slaves (*Nano_SPIArm()* and *Nano_I2CArm()).*

```
    if (connected) {

        // Initialise the GPIOs and flush any events
        printf(" Initialisation\n");
        for (i=0;i<16;i++)
            res=VB.Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,i,false);
        res=VB.Nano_GPIOBSetDirection(VB.vb_hDevice,0xFFFF,0xFFFF);
        res=VB.Nano_GPIOBWrite(VB.vb_hDevice,0x0000,0xFFFF);
        res=VB.Nano_I2CSlaveArm(VB.vb_hDevice,false);
        res=VB.Nano_SPISlaveArm(VB.vb_hDevice,false);
        printf("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n\n");
        c=getch();
```

GPIO port A is tested by writing an AA then 55 pattern using *Nano_GPIOASetDigitalOutputMode()*. Bit 0 is then set into continuous pulse mode using *Nano_GPIOASetContinuousMode()*.

```
//////////////////////////////////////
// GPIO Port A Test
//////////////////////////////////////
printf("++++++++++++++++++++++\n");
printf(" GPIO A Test\n");
printf("++++++++++++++++++++++\n");
printf(" Write AA\n");
// Write AA
for (i=0;i<8;i++) {
    res=VB.Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,2*i,false);
    res=VB.Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,2*i+1,true);
}
c=getch();
// Write 55
printf(" Write 55\n");
for (i=0;i<8;i++) {
    res=VB.Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,2*i,true);
    res=VB.Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,2*i+1,false);
}
c=getch();
// Continuous Pulsed
printf(" Make bit 0 pulsed\n");
for (i=0;i<16;i++)
    res=VB.Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,i,false);
e = VB.Nano_GPIOASetContinuousMode(VB.vb_hDevice,0,1,0x40,0x80);
printf("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n\n");
c=getch();
```

GPIO port B is tested also by writing an AA then 55 pattern using *Nano_GPIOBSetDirection()* and *Nano_GPIOBWrite()*.

```
//////////////////////////////////////
// GPIO Port B Test
//////////////////////////////////////
printf("++++++++++++++++++++++\n");
printf(" GPIO B Test\n");
printf("++++++++++++++++++++++\n");
printf(" Write AA\n");
// Write AA
res=VB.Nano_GPIOBSetDirection(VB.vb_hDevice,0xFFFF,0xFFFF);
res=VB.Nano_GPIOBWrite(VB.vb_hDevice,0xAAAA,0xFFFF);
c=getch();
printf(" Write 55\n");
// Write 55
res=VB.Nano_GPIOBSetDirection(VB.vb_hDevice,0xFFFF,0xFFFF);
res=VB.Nano_GPIOBWrite(VB.vb_hDevice,0x5555,0xFFFF);
printf("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n\n");
c=getch();
```

Analogue inputs are read using *Nano_ADCRead().*

```
///////////////////////////////////
// ADC Test
///////////////////////////////////
printf("++++++++++++++++++++++\n");
printf(" Analogue Input Test\n");
printf("++++++++++++++++++++++\n");
res=VB.Nano_ADCRead(VB.vb_hDevice,0x00,&ADC_0);
res=VB.Nano_ADCRead(VB.vb_hDevice,0x01,&ADC_1);
res=VB.Nano_ADCRead(VB.vb_hDevice,0x02,&ADC_2);
res=VB.Nano_ADCRead(VB.vb_hDevice,0x03,&ADC_3);
printf("Analogue Channel #00 : %02X\n",ADC_0);
printf("Analogue Channel #01 : %02X\n",ADC_1);
printf("Analogue Channel #02 : %02X\n",ADC_2);
printf("Analogue Channel #03 : %02X\n",ADC_3);
printf("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n\n");
c=getch();
```

The SPI master and slave is tested next. First the SPI master channel 0 is configured as active low chip select, CPOL=0 and CPHA=0 using *Nano_SPIConfigure().* The SPI master frequency is set for 12MHz line rate using *Nano_SPIMasterSetFrequency().* The slave channel 0 buffer is filled with 0xAABBCCDD using *Nano_SPISlaveBufferWrite().* The slave is then armed with *Nano_SPISlaveArm().* An SPI read/write master transfer is then made with 0x11223344 as data using *Nano_SPIMasterReadWrite().*The slave buffer is then read using *Nano_SPISlaveBufferRead().* The master buffer should then contain 0x11223344 and the slave should contain 0xAABBCCDD assuming the master and slave are looped back.

```
///////////////////////////////////
// SPI Test
///////////////////////////////////
printf("++++++++++++++++++++++\n");
printf(" SPI Test\n");
printf("++++++++++++++++++++++\n");

// Configure the SPI in terms of CPOL/CPHA/CSn
printf("SPI Configure Channel 0...\n");
res=VB.Nano_SPIConfigure(VB.vb_hDevice,0,0,false,0,0);

// Set the frequency
printf("SPI Set Frequency ...\n\n");
res=VB.Nano_SPIMasterSetFrequency(VB.vb_hDevice,NANO_SPI_FREQ_12MHZ);

// Fill the slave with data
out_buffer_s[0] = 0xAA;
out_buffer_s[1] = 0xBB;
out_buffer_s[2] = 0xCC;
out_buffer_s[3] = 0xDD;
res=VB.Nano_SPISlaveBufferWrite(VB.vb_hDevice,0,4,out_buffer_s);
printf("SPI Slave Data (Written)  : %02X%02X%02X%02X\n",
    out_buffer_s[0],out_buffer_s[1],out_buffer_s[2],out_buffer_s[3]);

// Arm the slave
```

```
        res=VB.Nano_SPISlaveArm(VB.vb_hDevice,true);

        // Perform a duplex read/write on the master
        out_buffer_m[0] = 0x11;
        out_buffer_m[1] = 0x22;
        out_buffer_m[2] = 0x33;
        out_buffer_m[3] = 0x44;
        printf("SPI Master Data (To send) : %02X%02X%02X%02X\n",
            out_buffer_m[0],out_buffer_m[1],out_buffer_m[2],out_buffer_m[3]);
        printf("\nSPI Master Send Channel 0...\n\n");
        res=VB.Nano_SPIMasterReadWrite(VB.vb_hDevice,0,4,in_buffer_m,out_buffer_m);

        // Present the read data from the master
        printf("SPI Master Data (Read)     : %02X%02X%02X%02X\n",
            in_buffer_m[0],in_buffer_m[1],in_buffer_m[2],in_buffer_m[3]);

        // Present the received data at the slave
        res=VB.Nano_SPISlaveBufferRead(VB.vb_hDevice,0,4,in_buffer_s);
        printf("SPI Slave Data (Read)      : %02X%02X%02X%02X\n",
            in_buffer_s[0],in_buffer_s[1],in_buffer_s[2],in_buffer_s[3]);

        c=getch();
        printf("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n\n");
```

The I2C is then tested, firstly using an I2C scan test. The test configures the slave to have a device ID of 0x2B using *Nano_I2CSlaveConfig()*. The slave is then armed using *Nano_I2CSlaveArm()*. The line rate of the master is then set to 6MHz using *Nano_I2CMasterSetFrequency()*. Finally we call the I2C scan function to find all devices connected using *Nano_I2CMasterScanConnectedDevices()*.

```
        printf("++++++++++++++++++++++\n");
        printf(" I2C Test\n");
        printf("++++++++++++++++++++++\n");

        ///////////////////////////////////
        // I2C Scan Test
        ///////////////////////////////////

        // Arm again
        res=VB.Nano_I2CSlaveConfig(VB.vb_hDevice,0x2B);
        res=VB.Nano_I2CSlaveArm(VB.vb_hDevice,true);

        // Set the line rate
        res=VB.Nano_I2CMasterSetFrequency(VB.vb_hDevice,NANO_I2C_FREQ_6MHZ);

        // Scan all I2C devices
        res=VB.Nano_I2CMasterScanConnectedDevices(VB.vb_hDevice,&lst);
        printf("I2C Devices Connected ... ");
        for (i=0;i<128;i++) {
            if (lst.List[i]) {printf("0x%02X ",i); }
        }
        printf("\n\n");
```

The slave and master are configured as for the I2C scan testing in terms of device ID and line rate. An I2C master write is then performed with 0xAABBCCDDFF as the output data using *Nano_I2CMasterWrite()*. The data is then read from the slave using *Nano_I2CSlaveBuffer1Read()*. It should be 0xAABBCCDDFF also.

```
///////////////////////////////////
// I2C Write Test
///////////////////////////////////

// Configure and arm the slave
res=VB.Nano_I2CSlaveConfig(VB.vb_hDevice,0x2B);
res=VB.Nano_I2CSlaveArm(VB.vb_hDevice,true);

// Set the line rate
res=VB.Nano_I2CMasterSetFrequency(VB.vb_hDevice,NANO_I2C_FREQ_6MHZ);

// Perform I2C write
OutBuffer[0] = 0xAA;
OutBuffer[1] = 0xBB;
OutBuffer[2] = 0xCC;
OutBuffer[3] = 0xDD;
OutBuffer[4] = 0xFF;
res=VB.Nano_I2CMasterWrite (VB.vb_hDevice,0x2B,5,OutBuffer);
printf("Master I2C write...\n");

// Check Data
res=VB.Nano_I2CSlaveBuffer1Read(VB.vb_hDevice,5,InBuffer);
printf("Master Buffer   : %02X%02X%02X%02X%02X\n",OutBuffer[0],
    OutBuffer[1],OutBuffer[2],OutBuffer[3],OutBuffer[4]    );
printf("Slave Buffer    : %02X%02X%02X%02X%02X\n\n",InBuffer[0],
    InBuffer[1],InBuffer[2],InBuffer[3],InBuffer[4]);
```

The slave and master are configured as for the I2C scan testing in terms of device ID and line rate. The I2C slave buffer is filled with 0x1122334455 using *Nano_I2CSlaveBuffer1Write()*. An I2C read is then performed using *Nano_I2CMasterRead()*. The data should be 0x1122334455.

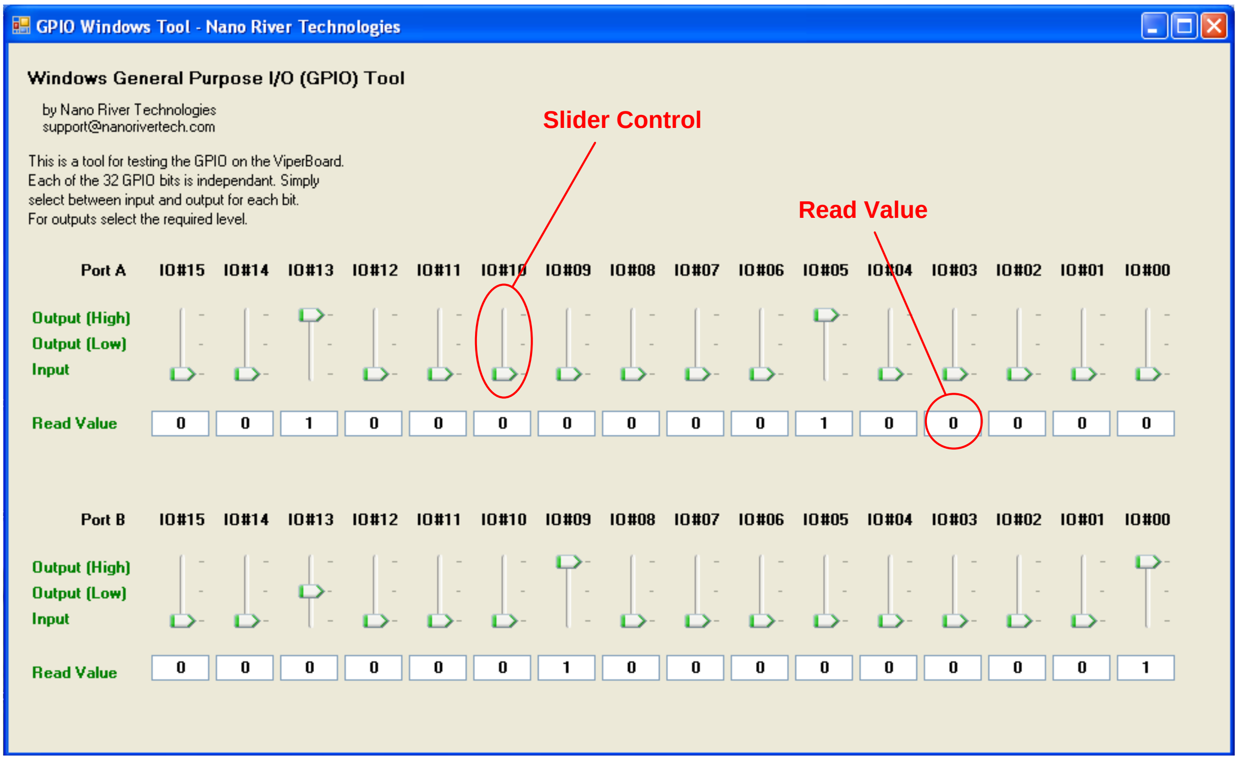
```
///////////////////////////////////
// I2C Read Test
///////////////////////////////////

// Configure and arm the slave
res=VB.Nano_I2CSlaveConfig(VB.vb_hDevice,0x2B);
res=VB.Nano_I2CSlaveArm(VB.vb_hDevice,true);

// Set the line rate
res=VB.Nano_I2CMasterSetFrequency(VB.vb_hDevice,NANO_I2C_FREQ_6MHZ);

// Put data into the slave
OutBuffer[0] = 0x11;
OutBuffer[1] = 0x22;
OutBuffer[2] = 0x33;
OutBuffer[3] = 0x44;
OutBuffer[4] = 0x55;
res=VB.Nano_I2CSlaveBuffer1Write(VB.vb_hDevice,5,OutBuffer);
```

```
        // Perform I2C read
        res=VB.Nano_I2CMasterRead (VB.vb_hDevice,0x2B,5,InBuffer);
        printf("Master I2C read...\n");
        printf("Slave Buffer    : %02X%02X%02X%02X%02X\n",OutBuffer[0],
            OutBuffer[1],OutBuffer[2],OutBuffer[3],OutBuffer[4]    );
        printf("Master Buffer   : %02X%02X%02X%02X%02X\n\n",InBuffer[0],
            InBuffer[1],InBuffer[2],InBuffer[3],InBuffer[4]);


        printf("\n");
        printf("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n\n");
```

Some important links and includes are necessary in stdafx.h. This is best seen by viewing the file in the download from the web-site.

3. Windows GPIO Tool

This example application is a very simple Visual C++ GUI application showing how to interface to the general purpose IO (GPIO) pins. This includes both port A and port B GPIOs.

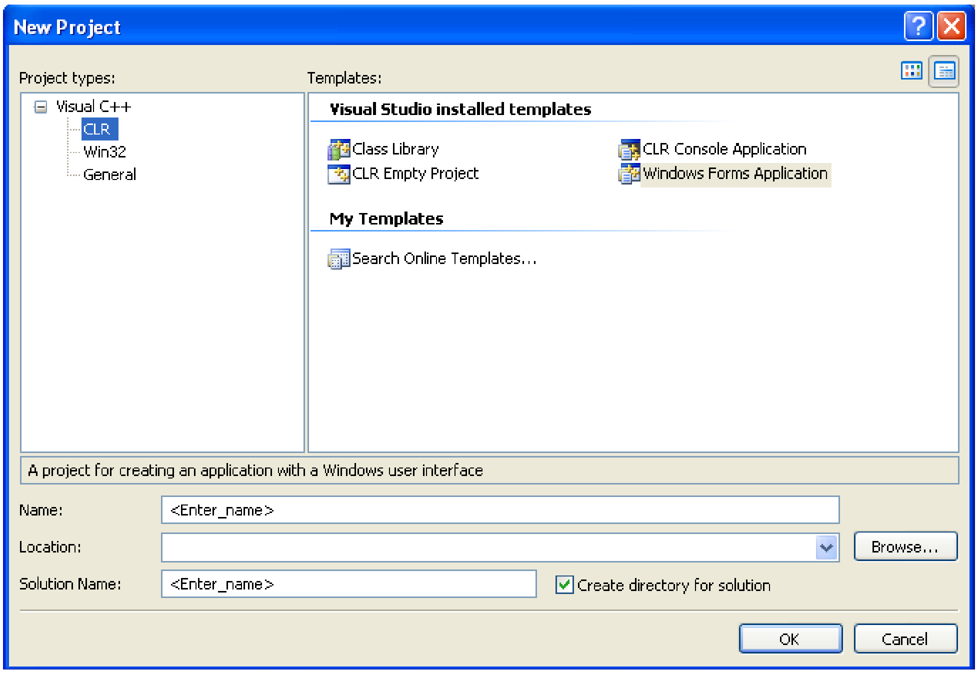


3.1. Functionality

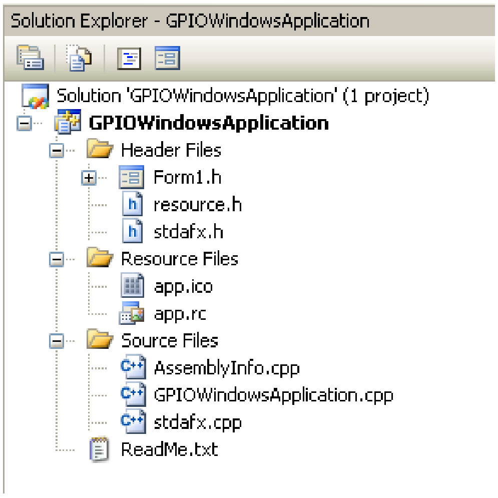
To use the application the user needs to select one of three "slider control" settings. One slider is provided for each bit. Each bit can be driven high as an output, driven low as an output or used as an input. Irrespective of whether the pin is configured as input or output the GPIO pin is then sampled and displayed as the "read value".

3.2. Project Structure

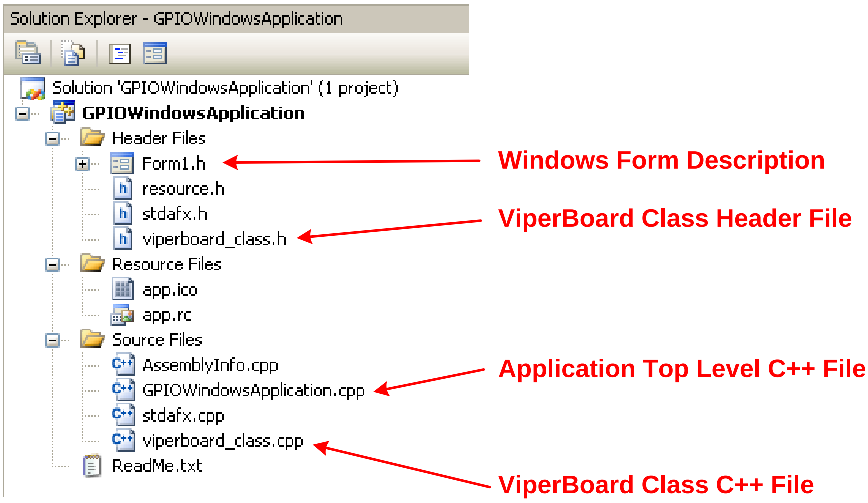
In order to create such a Windows application in Visual C++, first start up a new project and select "Window Form Application" as the project type.



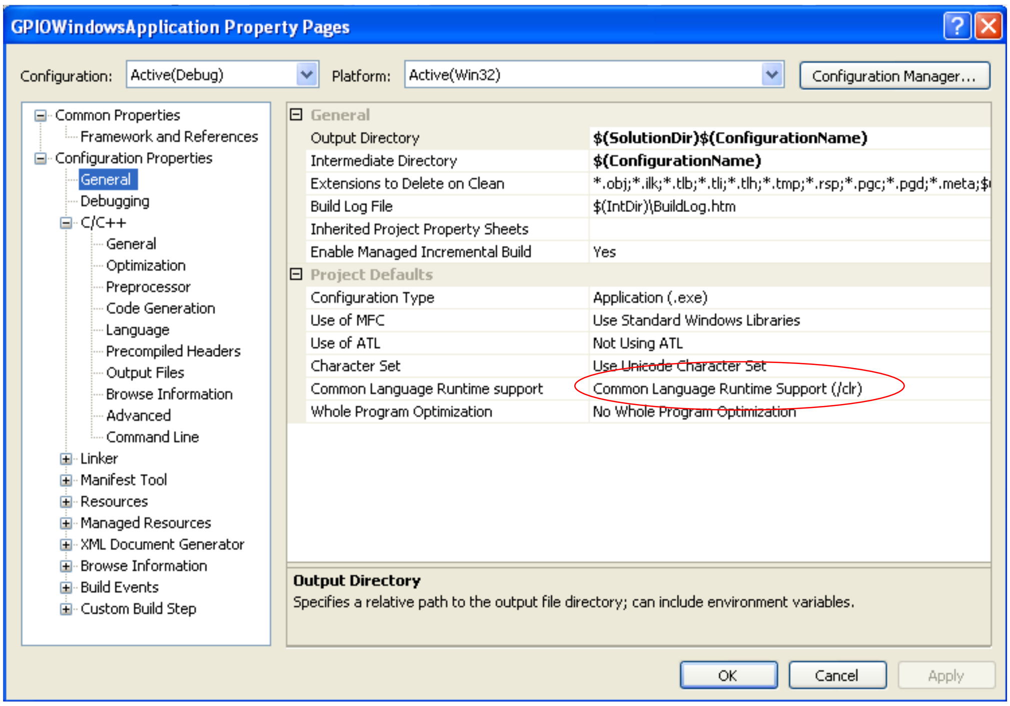
This will create a standard Windows application. The project file structure will then look as follows.



To this, one must add the ViperBoard class header file (viperboard_class.h) and ViperBoard class C++ file as follows.



Remember to change the Common Language Runtime support to /clr.



3.3. Design Description

The main file which needs to be developed in a Windows GUI application is the Windows form application file (Form1.h). This can be built up graphically by dragging in elements of the .NET development toolbox and/or by editing the Form1.h text file.

The final Form1.h file can be found in the downloaded files for the GPIO Windows application available on the web-site.

The file starts by linking to the ViperBoard class library header file:

```
//////////////////////////////////////////////////////////////////////////
//
// Nano River Technologies
//
// File:          GPIOWindowsApplication (Form.h)
//
// Desciption:    This is a simpe Windows application as a demonstration
//                to show how to build up a simple Windows application and
//                link to the GPIO functions in the Nano River Techologies
//                ViperBoard.
//
// Revision:      Version 1.0
//
//////////////////////////////////////////////////////////////////////////

#pragma once

#include "../../common/viperboard_class.h"
```

The file declares some global variables used in the application:

```
// Own Global variables here
//
NANO_RESULT  res;         // Returned result for calls to Nano River Tech functions
VBc          VB;          // The ViperBoard class library
Int32        slider;      // Slider value
Int32        bit;         // GPIO Bit number (used for both port A and port B GPIOs)
Int32        i;           // Local integer for looping
BOOL         value;       // Read value of the GPIO bit
bool         connected;   // True if the ViperBoard is connected
```

Windows objects are defined:

```
// Define all the Windows objects here
//
private: System::Windows::Forms::TrackBar^  trackBar_0;
private: System::Windows::Forms::TrackBar^  trackBar_1;
   …
private: System::Windows::Forms::TrackBar^  trackBar_30;
```

A function is defined during the form load. In this the connection to the ViperBoard is established by calling *OpenDevice()* in the ViperBoard class. For all Port A GPIOs function *Nano_GPIOASetDigitalInputMode()* mode is called in order to initialise all pins as inputs. Similarly, for Port B GPIOs functions *Nano_GPIOSetDirection()* and *Nano_GPIOWrite()* are called to initialise all pins as inputs.

```
private: System::Void Form1_Load(System::Object^  sender, System::EventArgs^  e) {
// A task called when the form is loaded. Here we
// open the USB link and initialise the ViperBoard.
// All GPIOs are initialised to all input.
//
    connected = ::VBc::OpenDevice();
    if (!connected)
        this->label_49->Visible = true;
    if (connected) {
        // Set all GPIOs to inputs
        for (i; i<16; i++) {
            res=VB.Nano_GPIOASetDigitalInputMode(VB.vb_hDevice,i,NANO_GPIO_CLK_1);
        }
        res = ::VBc::Nano_GPIOBSetDirection(VB.vb_hDevice,0x0000,0xFFFF);
        res = ::VBc::Nano_GPIOBWrite(VB.vb_hDevice,0x0000,0xFFFF);
    }
}
```

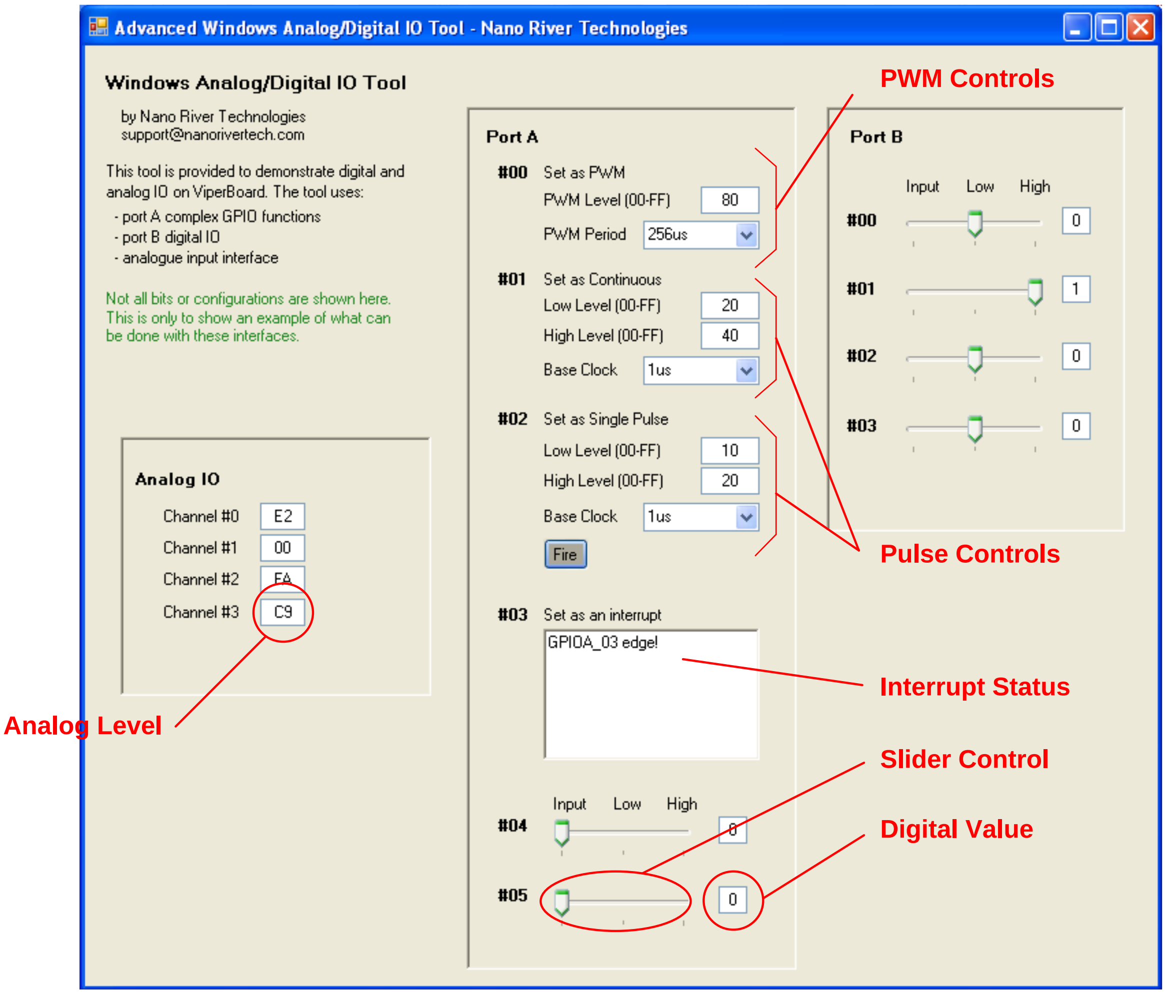
Finally, a timer service routine is defined. Each time the timer ticks the control slider is checked to see if the GPIO needs to be updated as an output or if it needs to change to an input. In addition the GPIO is read and displayed. The process is repeated for all GPIOs.

```
private: System::Void timer1_Tick(System::Object^  sender, System::EventArgs^  e) {
// Task to update all GPIOs each clock tick

    /////////////////////////
    // Port A GPIO
    /////////////////////////

    // Bit 00
    bit = 0;
    slider = this->trackBar_0->Value;
    value = (slider==2); // For demostration mode only
    if (slider==0 && connected)
        res=VB.Nano_GPIOASetDigitalInputMode(VB.vb_hDevice,bit,NANO_GPIO_CLK_1);
    else if (slider==1 && connected) {
        res=VB.Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,bit,false);
    }
    else if (slider==2 && connected) {
        res=VB.Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,bit,true);
    }
    if (connected)
        res = ::VBc::Nano_GPIOAGetDigitalInput(VB.vb_hDevice,bit,&value);
    this->textBox_0->Text = Convert::ToString(value);
   …


    /////////////////////////
    // Port B GPIO
    /////////////////////////

    // Bit 00
    bit = 00;
    slider = this->trackBar_16->Value;
    value = (slider==2); // For demostration mode only
    if (slider==0 && connected)
        res = ::VBc::Nano_GPIOBSetSingleBitDirection(VB.vb_hDevice,bit,0);    // Input
    else if (slider==1 && connected) {
        res = ::VBc::Nano_GPIOBSingleBitWrite(VB.vb_hDevice,bit,0);          // 0
        res = ::VBc::Nano_GPIOBSetSingleBitDirection(VB.vb_hDevice,bit,1);   // Output
    }
    else if (slider==2 && connected) {
        res = ::VBc::Nano_GPIOBSingleBitWrite(VB.vb_hDevice,bit,1);          // 1
        res = ::VBc::Nano_GPIOBSetSingleBitDirection(VB.vb_hDevice,bit,1);   // Output
    }
    if (connected)
        res = ::VBc::Nano_GPIOBSingleBitRead(VB.vb_hDevice,bit,&value);
    this->textBox_16->Text = Convert::ToString(value);
   …
```

Some important links and includes are necessary in stdafx.h. This is best seen by viewing the file in the download from the web-site.

4. Advanced Windows Analog/Digital IO Tool

This is a more complex Visual C++ GUI application showing some of the more complex GPIO and analog input functions. Note that not all capabilities of the GPIOs and analog inputs are demonstrated here. To get a complete understanding of what can be achieved, please refer to the ViperBoard API specification.



4.1. Functionality

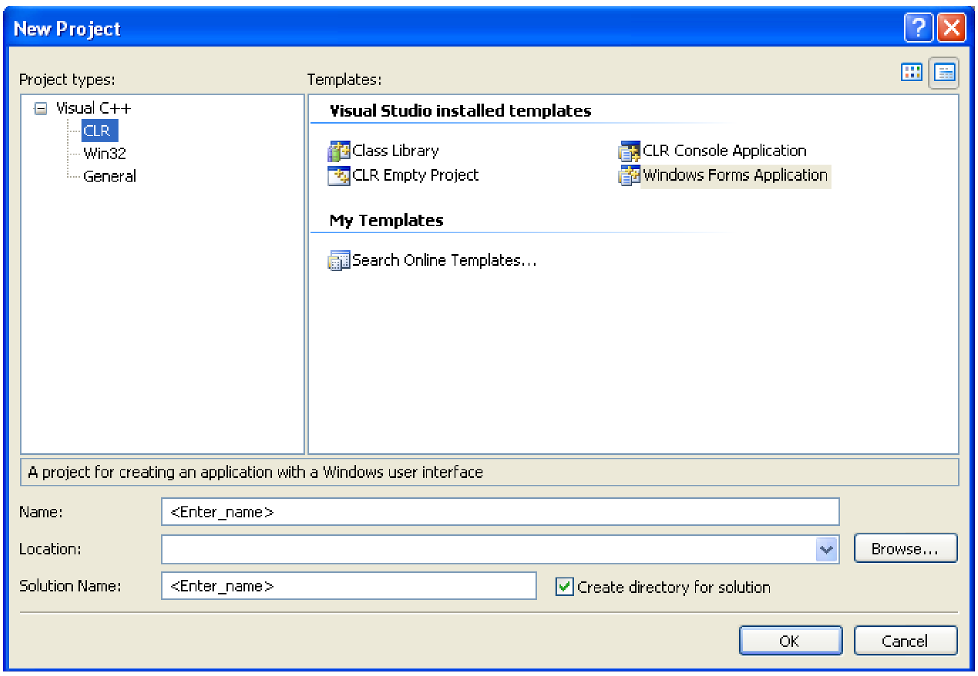
The application is divided into three panels. The first panel is for viewing the analogue inputs converted to 8 bit digital.

The second panel is for controlling the Port A advanced GPIOs. Here bit 0 is set-up as a PWM output. One can modify the PWM level and period. Bit 1 is a continuous pulsed output. Low time and high time can be controlled. Bit 2 is a one-shot active high pulsed output with controllable low and high time. The pulse is initiated by pressing the “fire” button. GPIO bit 3 is an interrupt input sensitive to rising edges. One can loop GPIO bits 2 and 3 and see interrupt events in the interrupt status window. GPIOs 4 and 5 are set as digital IO where one can select between digital input and digital output of either level.

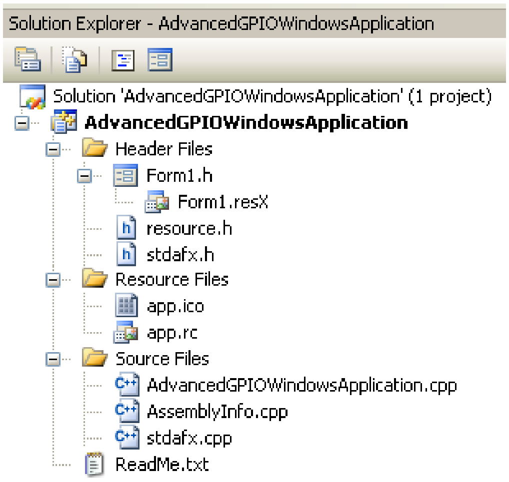
The third panel is for controlling Port B GPIOs. GPIOs 0 to 3 are digital IO where one can select between digital input and digital output of either level.

4.2. Project Structure

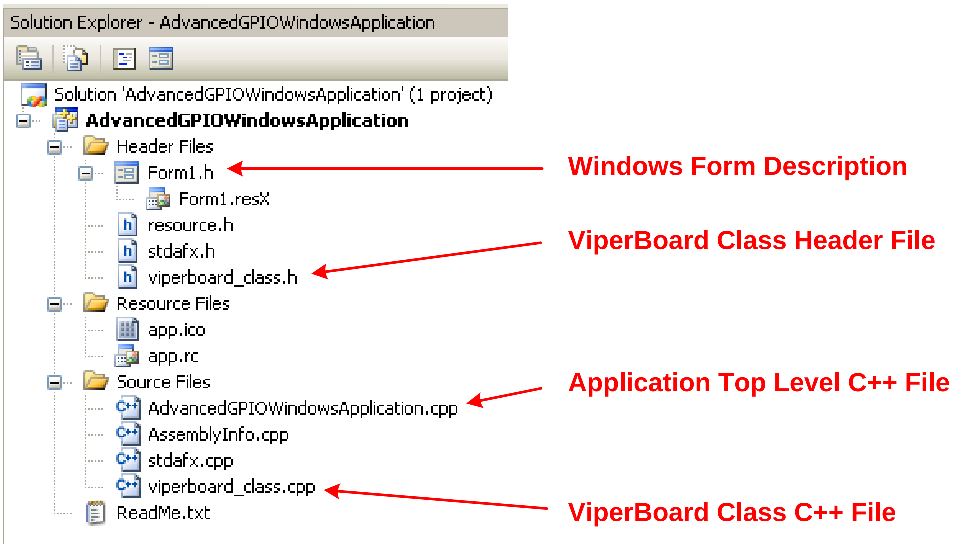
In order to create such a Windows application in Visual C++, first start up a new project and select “Window Form Application” as the project type.



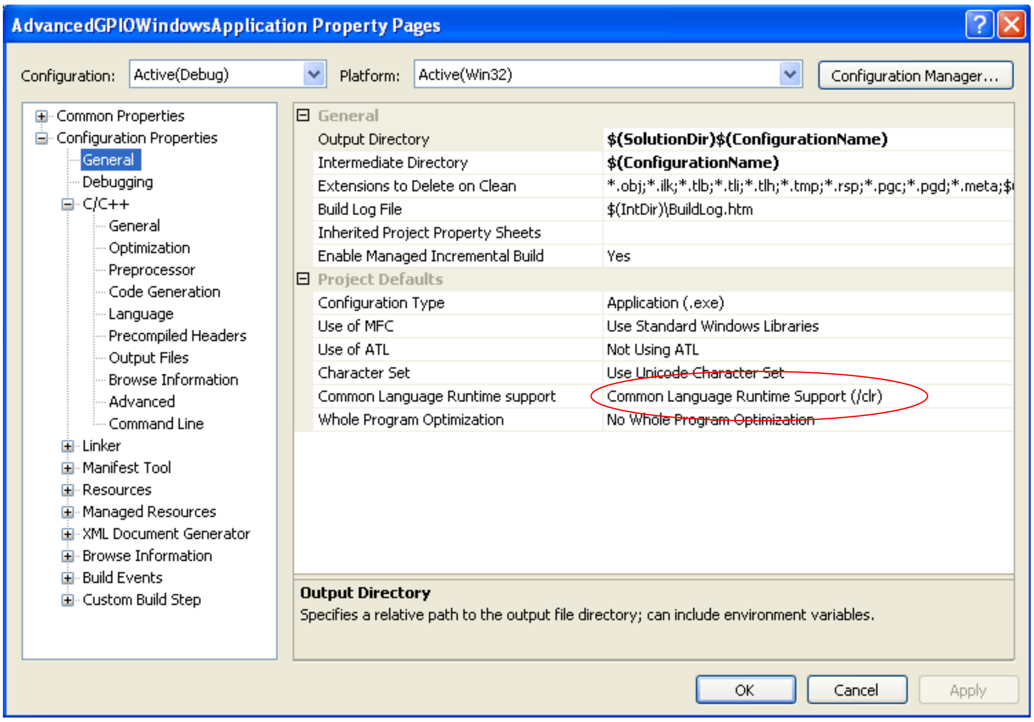
This will create a standard Windows application. The project file structure will then look as follows.



To this, one must add the ViperBoard class header file (viperboard_class.h) and ViperBoard class C++ file as follows.



Remember to change the Common Language Runtime support to /clr.



4.3. Design Description

The main file which needs to be developed in a Windows GUI application is the Windows form application file (Form1.h). This can be built up graphically by dragging in elements of the .NET development toolbox and/or by editing the Form1.h text file.

The final Form1.h file can be found in the downloaded files for the Advanced GPIO Windows Application available on the web-site.

The file starts by linking to the ViperBoard class library header file:

```
////////////////////////////////////////////////////////////////////////////
//
// Nano River Technologies
//
// File:          AdvancedGPIOWindowsApplication (Form.h)
//
// Desciption:    This is a Windows application to demonstrate some of the
//                more advanced features of the ViperBoard GPIOs. For simplicity
//                not all GPIO bits are used but the concept can obviously be
//                extended.
//
//                Port A (bit0) - Used as a PWM output
//                Port A (bit1) - Used as a continously pulsed output
//                Port A (bit2) - Used as a one-shot pulsed output
//                Port A (bit3) - Used as an interrupt input (rising edge detect)
//                Port A (bit4) - Used as a digital output
//                Port A (bit5) - Used as a digital output
//
//                Port B (bit0) - Used as a digital output
//                Port B (bit1) - Used as a digital output
//                Port B (bit2) - Used as a digital output
//                Port B (bit3) - Used as a digital output
//
//                ADC    (bit0) - This is an anlogue input
//                ADC    (bit1) - This is an anlogue input
//                ADC    (bit2) - This is an anlogue input
//                ADC    (bit3) - This is an anlogue input
//
//                Some further notes:
//                1) You can simply loop Port A bits 3&4 together to see
//                   the single pulse trigger an interrupt.
//                2) You can try the low pass filter on Port A bits 0&1
//                   using different jumper settings
//
// Revision:      Version 1.0
//
////////////////////////////////////////////////////////////////////////////

#pragma once

#include "../../common/viperboard_class.h"
```

The file declares some global variables used in the application:

```
// Own Global variables here
//
NANO_RESULT res;                      // Returned result for calls to Nano River Tech functions
VBc         VB;                       // The ViperBoard class library
bool        connected;                // True if the ViperBoard is connected
BYTE        adc_0;                    // Read ADC value (channel 0)
BYTE        adc_1;                    // Read ADC value (channel 1)
BYTE        adc_2;                    // Read ADC value (channel 2)
BYTE        adc_3;                    // Read ADC value (channel 3)
BYTE        pwm_level;                // Specified PWM level
BYTE        pwm_clock;                // Base clock for PWM
BYTE        continuous_low_time;      // Specified continuos mode low time
BYTE        continuous_high_time;     // Specified continuos mode high time
BYTE        base_clk_1;               // Base clock for continous mode
BYTE        pulsed_low_time;          // Specified pulsed mode low time (before pulse)
BYTE        pulsed_high_time;         // Specified pulsed mode high time (pulse width)
BYTE        base_clk_2;               // Base clock for the single pulse mode
BOOL        value;                    // Read value of the GPIO bit
Int32       slider;                   // Slider value
WORD        GPIOEvent;                // GPIO event (one per GPIO bit)
BOOL        SPISlaveEvent;            // SPI slave event (Not Used)
BOOL        IICSlaveEvent;            // I2C slave event (Not used)
BYTE        SPISlaveChan;             // SPI slave channel (Not Used)
WORD        SPISlaveBytes;            // SPI slave number of bytes(Not Used)
BYTE        IICSlaveTransferType;     // I2C slave tranfer type (Not used)
WORD        IICSlaveT1Bytes;          // I2C slave number of T1 bytes (Not Used)
WORD        IICSlaveT2Bytes;          // I2C slave number of T2 bytes (Not Used)
```

Windows objects are defined:

```
// Define all the Windows objects here
//
private: System::Windows::Forms::Label^  label_00;
private: System::Windows::Forms::Label^  label_01;
private: System::Windows::Forms::Label^  label_02;
   …
private: System::Windows::Forms::TextBox^  textBox_00;
private: System::Windows::Forms::TextBox^  textBox_01;
private: System::Windows::Forms::TextBox^  textBox_02;
   …
private: System::Windows::Forms::Button^  button_00;
private: System::Windows::Forms::RichTextBox^  richTextBox_00;
private: System::Windows::Forms::TrackBar^  trackBar_00;
private: System::Windows::Forms::TrackBar^  trackBar_01;
private: System::Windows::Forms::TrackBar^  trackBar_02;
   …
private: System::Windows::Forms::ComboBox^  comboBox_00;
private: System::Windows::Forms::ComboBox^  comboBox_01;
private: System::Windows::Forms::ComboBox^  comboBox_02;
private: System::Windows::Forms::Panel^  panel_00;
private: System::Windows::Forms::Panel^  panel_01;
private: System::Windows::Forms::Panel^  panel_02;
private: System::Windows::Forms::Timer^  timer1;
```

We then define some managed variables:

```
// Managed variables
String      ^s1;       // A local string variable used for RichTechBox output
String      ^s2;       // 2nd local string variable used for RichTechBox output
String      ^s3;       // 3rd local string variable used for RichTechBox output
String      ^Data;     // A string used for parsing the data input textbox
```

A function is defined during the form load. In this the connection to the ViperBoard is established by calling *OpenDevice()* in the ViperBoard class. GPIO port A bits 0, 1, 2 and 3 are initialised as PWM output, continuous pulsed output, digital output and interrupt input respectively.

```
private: System::Void Form1_Load(System::Object^  sender, System::EventArgs^  e) {
    // A task called when the form is loaded. Here we
    // open the USB link and initialise the ViperBoard.
    // We also initialise some of the GPIOs.
    //
    connected = ::VBc::OpenDevice();
    if (!connected)
        this->label_46->Visible = true;
    if (connected) {

        // Update Port A GPIO (bit 0)    <---- Configured for PWM
        if (connected) {
            pwm_level = Int32::Parse(this->
                textBox_04->Text,System::Globalization::NumberStyles::HexNumber);
                                 // Get PWM level from TextBox
            res = ::VBc::Nano_GPIOASetPWMMode(VB.vb_hDevice,0,0,pwm_level);
        }

        // Update Port A GPIO (bit 1)    <---- Configured for Continuous pulse mode
        if (connected) {
            continuous_low_time  = Int32::Parse(this->
                textBox_05->Text,System::Globalization::NumberStyles::HexNumber);
                                 // Get PWM level from TextBox
            continuous_high_time = Int32::Parse(this->
                textBox_06->Text,System::Globalization::NumberStyles::HexNumber);
                                 // Get PWM level from TextBox
            res = ::VBc::Nano_GPIOASetContinuousMode(VB.vb_hDevice,1,0,
                                                     continuous_low_time,continuous_high_time);
        }

        // Update Port A GPIO (bit 2)    <---- Configured for Single pulse mode
        if (connected) {
            // Initialise to logic low
            res = ::VBc::Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,2,false);
        }

        // Setup Port A GPIO (bit 3)
        s1="";
        res=VB.Nano_GPIOASetInterruptInputMode(VB.vb_hDevice,3,true,NANO_GPIO_CLK_1);
    }
}
```

A timer function is provided which periodically updates some IOs.

- The analog inputs are updated by reading using the *Nano_ADCRead()* function.
- GPIO port A bits 4 and 5 are polled to check the slider position and from this set the bit as either digital input or digital output using *Nano_GPIOASetDigitalInputMode()* or *Nano_GPIOASetDigitalInputMode().*
- GPIO port B bits 0, 1, 2 and 3 are updated using *NanoGPIOBSetSingleBitDirection(), NanoGPIOBSingleBitWrite()* and *NanoGPIOBSingleBitRead().*
- ViperBoard events are polled using *Nano_GetEvents()* to see if GPIO Port A bit 3 has received an interrupt. If so, then it is reported in the textbox.

```
private: System::Void timer1_Tick(System::Object^  sender, System::EventArgs^  e) {
    // This is a polling task. Here we:
    //    1) Poll and update the ADC.
    //    2) Update the digital IO bits Port A bits 4-5, Port B bits 0-3
    //    3) Poll for any interrupts (Port A bit 3)


    // Update Analogue inputs
    //
    adc_0=255;
    adc_1=255;
    adc_2=255;
    adc_3=255;
    if (connected) {
        res=VB.Nano_ADCRead(VB.vb_hDevice,0,&adc_0);
        res=VB.Nano_ADCRead(VB.vb_hDevice,1,&adc_1);
        res=VB.Nano_ADCRead(VB.vb_hDevice,2,&adc_2);
        res=VB.Nano_ADCRead(VB.vb_hDevice,3,&adc_3);
    }
    this->textBox_00->Text = adc_0.ToString("X2");
    this->textBox_01->Text = adc_1.ToString("X2");
    this->textBox_02->Text = adc_2.ToString("X2");
    this->textBox_03->Text = adc_3.ToString("X2");

    // Update Port A GPIO (bit 4)
    slider = this->trackBar_00->Value;
    value = (slider==2); // For demostration mode only
    if (slider==0 && connected)
        res=VB.Nano_GPIOASetDigitalInputMode(VB.vb_hDevice,4,NANO_GPIO_CLK_1);
    else if (slider==1 && connected) {
        res=VB.Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,4,false);
    }
    else if (slider==2 && connected) {
        res=VB.Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,4,true);
    }
    if (connected)
        res = ::VBc::Nano_GPIOAGetDigitalInput(VB.vb_hDevice,4,&value);
    this->textBox_09->Text = Convert::ToString(value);
   …

    // Update Port B GPIO (bit 0)
    slider = this->trackBar_02->Value;
    value = (slider==2); // For demostration mode only
    if (slider==0 && connected)
        res = ::VBc::Nano_GPIOBSetSingleBitDirection(VB.vb_hDevice,0,0);     // Input
```

```
    else if (slider==1 && connected) {
        res = ::VBc::Nano_GPIOBSingleBitWrite(VB.vb_hDevice,0,0);          // 0
        res = ::VBc::Nano_GPIOBSetSingleBitDirection(VB.vb_hDevice,0,1);   // Output
    }
    else if (slider==2 && connected) {
        res = ::VBc::Nano_GPIOBSingleBitWrite(VB.vb_hDevice,0,1);          // 1
        res = ::VBc::Nano_GPIOBSetSingleBitDirection(VB.vb_hDevice,0,1);   // Output
    }
    if (connected)
        res = ::VBc::Nano_GPIOBSingleBitRead(VB.vb_hDevice,0,&value);
    this->textBox_11->Text = Convert::ToString(value);
   …

    // Poll to see if we got any GPIO event
    if (connected) {
        res=VB.Nano_GetEvents(VB.vb_hDevice,
                              &GPIOEvent,
                              &SPISlaveEvent,
                              &IICSlaveEvent,
                              &SPISlaveChan,
                              &SPISlaveBytes,
                              &IICSlaveTransferType,
                              &IICSlaveT1Bytes,
                              &IICSlaveT2Bytes);
        if (GPIOEvent&0x08) {
             s1 = this->richTextBox_00->Text;
             s1 = System::String::Concat( s1,"GPIOA_03 edge!\n");
             this->richTextBox_00->Text = s1;
        }
    }
}
```

These two functions are used to detect changes to the PWM level and period. Upon detection GPIO Port A bit 0 is updated using *Nano_GPIOASetPWMMode()*.

```
private: System::Void textBox_04_TextChanged(System::Object^  sender, System::EventArgs^  e) {
    // This function updates Port A gpio bit 0 which is the PWM bit.
    //
    // Update Port A GPIO (bit 0)    <---- Configured for PWM
    if (connected) {
        // Get the PWM clock rate
        if (this->comboBox_00->Text=="256us")   pwm_clock=NANO_GPIO_CLK_1;
        if (this->comboBox_00->Text=="2.56ms")  pwm_clock=NANO_GPIO_CLK_10;
        if (this->comboBox_00->Text=="25.6ms")  pwm_clock=NANO_GPIO_CLK_100;
        if (this->comboBox_00->Text=="256ms")   pwm_clock=NANO_GPIO_CLK_1000;
        if (this->comboBox_00->Text=="2.56s")   pwm_clock=NANO_GPIO_CLK_10000;
        if (this->comboBox_00->Text=="25.6s")   pwm_clock=NANO_GPIO_CLK_100000;
        pwm_level = Int32::Parse(this->
            textBox_04->Text,System::Globalization::NumberStyles::HexNumber);
               // Get PWM level from TextBox
        res = ::VBc::Nano_GPIOASetPWMMode(VB.vb_hDevice,0,pwm_clock,pwm_level);
    }
}
```

```
private: System::Void comboBox_00_SelectedIndexChanged(System::Object^  sender,
                                                       System::EventArgs^  e) {
    // This function updates Port A gpio bit 0 which is the PWM bit.
    //
    // Update Port A GPIO (bit 0)    <---- Configured for PWM
    if (connected) {
        // Get the PWM clock rate
        if (this->comboBox_00->Text=="256us")   pwm_clock=NANO_GPIO_CLK_1;
        if (this->comboBox_00->Text=="2.56ms")  pwm_clock=NANO_GPIO_CLK_10;
        if (this->comboBox_00->Text=="25.6ms")  pwm_clock=NANO_GPIO_CLK_100;
        if (this->comboBox_00->Text=="256ms")   pwm_clock=NANO_GPIO_CLK_1000;
        if (this->comboBox_00->Text=="2.56s")   pwm_clock=NANO_GPIO_CLK_10000;
        if (this->comboBox_00->Text=="25.6s")   pwm_clock=NANO_GPIO_CLK_100000;
        pwm_level = Int32::Parse(this->
            textBox_04->Text,System::Globalization::NumberStyles::HexNumber);
                // Get PWM level from TextBox
        res = ::VBc::Nano_GPIOASetPWMMode(VB.vb_hDevice,0,pwm_clock,pwm_level);
    }
}
```

The following three functions are used to detect changes to the continuous pulsed low line, high time and clock base. Upon detection GPIO Port A bit 1 is updated using *Nano_GPIOASetContinuousMode().*

```
private: System::Void textBox_05_TextChanged(System::Object^  sender, System::EventArgs^  e) {
    // This function updates Port A gpio bit 1 which is the continous pulsed bit.
    //
    // Update Port A GPIO (bit 1)    <---- Configured for continous pulsed mode
    if (connected) {
        // Get the base clock
        if (this->comboBox_01->Text=="1us")    base_clk_1=NANO_GPIO_CLK_1;
        if (this->comboBox_01->Text=="10us")   base_clk_1=NANO_GPIO_CLK_10;
        if (this->comboBox_01->Text=="100us")  base_clk_1=NANO_GPIO_CLK_100;
        if (this->comboBox_01->Text=="1ms")    base_clk_1=NANO_GPIO_CLK_1000;
        if (this->comboBox_01->Text=="10ms")   base_clk_1=NANO_GPIO_CLK_10000;
        if (this->comboBox_01->Text=="100ms")  base_clk_1=NANO_GPIO_CLK_100000;
        continuous_low_time  = Int32::Parse(this->
            textBox_05->Text,System::Globalization::NumberStyles::HexNumber);
                // Get PWM level from TextBox
        continuous_high_time = Int32::Parse(this->
            textBox_06->Text,System::Globalization::NumberStyles::HexNumber);
                // Get PWM level from TextBox
        res = ::VBc::Nano_GPIOASetContinuousMode(VB.vb_hDevice,1,
                                   base_clk_1,continuous_low_time,continuous_high_time);
    }
}

private: System::Void textBox_06_TextChanged(System::Object^  sender, System::EventArgs^  e) {
    // This function updates Port A gpio bit 1 which is the continous pulsed bit.
    //
    // Update Port A GPIO (bit 1)    <---- Configured for continous pulsed mode
    if (connected) {
        // Get the base clock
        if (this->comboBox_01->Text=="1us")    base_clk_1=NANO_GPIO_CLK_1;
        if (this->comboBox_01->Text=="10us")   base_clk_1=NANO_GPIO_CLK_10;
        if (this->comboBox_01->Text=="100us")  base_clk_1=NANO_GPIO_CLK_100;
        if (this->comboBox_01->Text=="1ms")    base_clk_1=NANO_GPIO_CLK_1000;
```

```
        if (this->comboBox_01->Text=="10ms")  base_clk_1=NANO_GPIO_CLK_10000;
        if (this->comboBox_01->Text=="100ms") base_clk_1=NANO_GPIO_CLK_100000;
        continuous_low_time  = Int32::Parse(this->
            textBox_05->Text,System::Globalization::NumberStyles::HexNumber);
                // Get PWM level from TextBox
        continuous_high_time = Int32::Parse(this->
            textBox_06->Text,System::Globalization::NumberStyles::HexNumber);
                // Get PWM level from TextBox
        res = ::VBc::Nano_GPIOASetContinuousMode(VB.vb_hDevice,1,
                                base_clk_1,continuous_low_time,continuous_high_time);
    }
}

private: System::Void comboBox_01_SelectedIndexChanged(System::Object^  sender,
                                                       System::EventArgs^  e) {
    // This function updates Port A gpio bit 1 which is the continous pulsed bit.
    //
    // Update Port A GPIO (bit 1)    <---- Configured for continous pulsed mode
    if (connected) {
        // Get the base clock
        if (this->comboBox_01->Text=="1us")   base_clk_1=NANO_GPIO_CLK_1;
        if (this->comboBox_01->Text=="10us")  base_clk_1=NANO_GPIO_CLK_10;
        if (this->comboBox_01->Text=="100us") base_clk_1=NANO_GPIO_CLK_100;
        if (this->comboBox_01->Text=="1ms")   base_clk_1=NANO_GPIO_CLK_1000;
        if (this->comboBox_01->Text=="10ms")  base_clk_1=NANO_GPIO_CLK_10000;
        if (this->comboBox_01->Text=="100ms") base_clk_1=NANO_GPIO_CLK_100000;
        continuous_low_time  = Int32::Parse(this->
            textBox_05->Text,System::Globalization::NumberStyles::HexNumber);
                 // Get PWM level from TextBox
        continuous_high_time = Int32::Parse(this->
            textBox_06->Text,System::Globalization::NumberStyles::HexNumber);
                 // Get PWM level from TextBox
        res = ::VBc::Nano_GPIOASetContinuousMode(VB.vb_hDevice,1,
                                base_clk_1,continuous_low_time,continuous_high_time);
    }
}
```

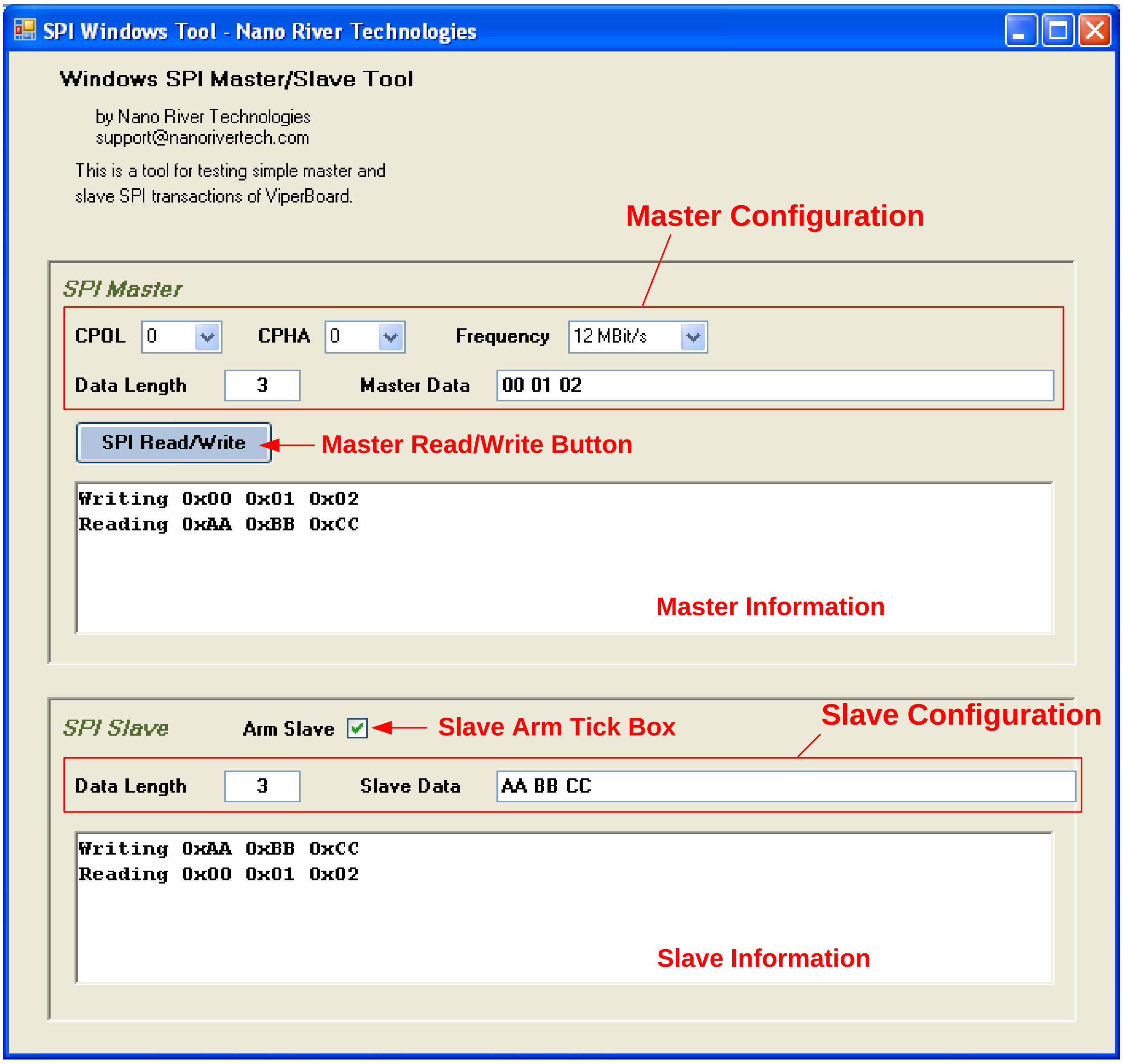
This function detects that the “fire” button has been pressed for the one-shot pulsed output for GPIO Port A bit 2. Upon detection GPIO Port A bit 2 is pulsed by calling *Nano_GPIOASetPulseMode().*

```
private: System::Void button_00_Click(System::Object^  sender, System::EventArgs^  e) {
    // This function updates Port A gpio bit 2 and send a one-shot pulse.
    //
    // Update Port A GPIO (bit 2)    <---- Configured for one-shot pulsed mode
    if (connected) {
        // Get the base clock
        if (this->comboBox_02->Text=="1us")   base_clk_2=NANO_GPIO_CLK_1;
        if (this->comboBox_02->Text=="10us")  base_clk_2=NANO_GPIO_CLK_10;
        if (this->comboBox_02->Text=="100us") base_clk_2=NANO_GPIO_CLK_100;
        if (this->comboBox_02->Text=="1ms")   base_clk_2=NANO_GPIO_CLK_1000;
        if (this->comboBox_02->Text=="10ms")  base_clk_2=NANO_GPIO_CLK_10000;
        if (this->comboBox_02->Text=="100ms") base_clk_2=NANO_GPIO_CLK_100000;
        pulsed_low_time  = Int32::Parse(this->
            textBox_07->Text,System::Globalization::NumberStyles::HexNumber);
                // Get PWM level from TextBox
        pulsed_high_time = Int32::Parse(this->
            textBox_08->Text,System::Globalization::NumberStyles::HexNumber);
                // Get PWM level from TextBox
        res = ::VBc::Nano_GPIOASetPulseMode(VB.vb_hDevice,2,
                                        base_clk_2,pulsed_low_time,pulsed_high_time,false);
    }
}
```

Some important links and includes are necessary in stdafx.h. This is best seen by viewing the file in the download from the web-site.

5. Windows SPI Tool

The Windows SPI tool is a windows GUI application in Visual C++ used to demonstrate both master and slave operation of the SPI interface for ViperBoard.



5.1. Functionality

The application consists of two panels. The top panel is for controlling the SPI master. The lower panel is for controlling the SPI slave.

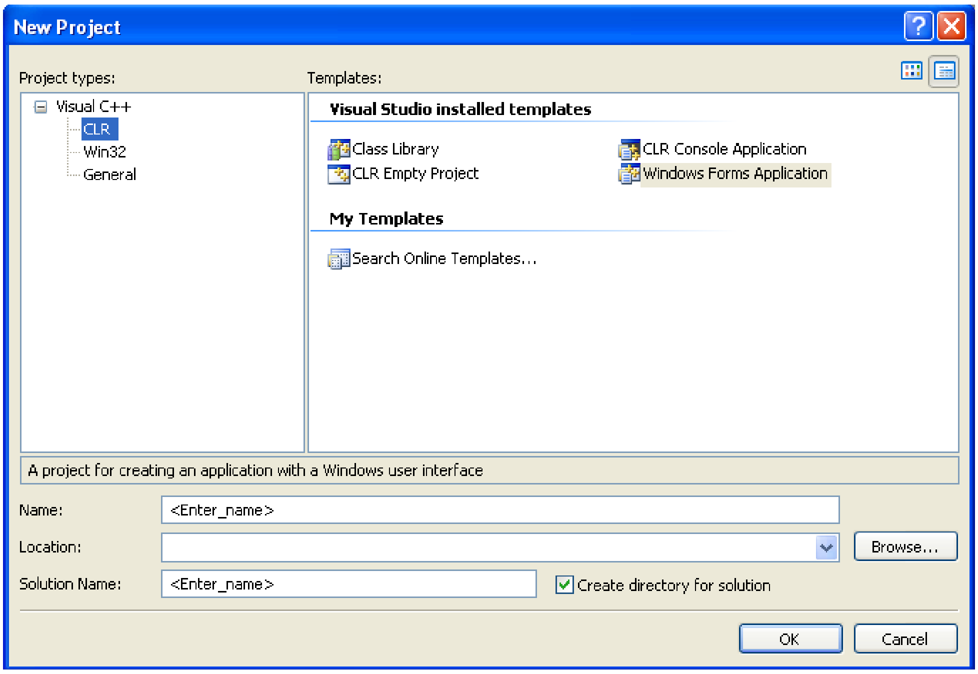
In the upper panel the user can configure the SPI clock polarity (CPOL), clock phase (CPHA) and line rate. The data to be sent is also specified in terms of the actual data and the length of the transfer. Pressing the master read/write button initiates an SPI read/write transfer of the specified length and with the specified output data. The input data received is displayed in the master information window.

In the lower panel the user specifies the data which resides in the slave buffer and the length of the data buffer. Activating the slave arm tick-box makes the slave ready to respond to a SPI master. When a transaction is performed the slave data is sent and the master data is received. The received data is displayed in the slave information window.

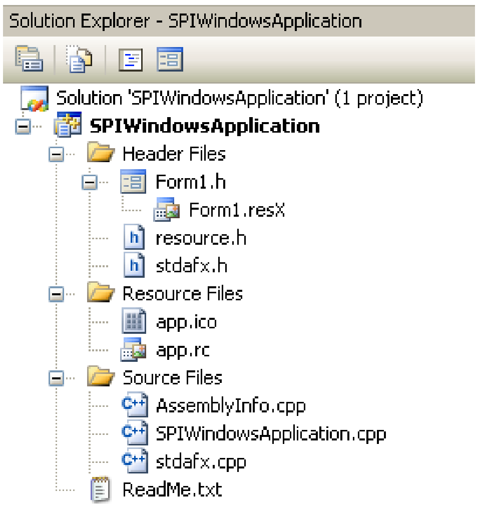
To see this application working one can simply loop the master SPI back to the slave SPI on ViperBoard using external wires. During a transfer one should see master and slave data being swapped.

5.2. Project Structure

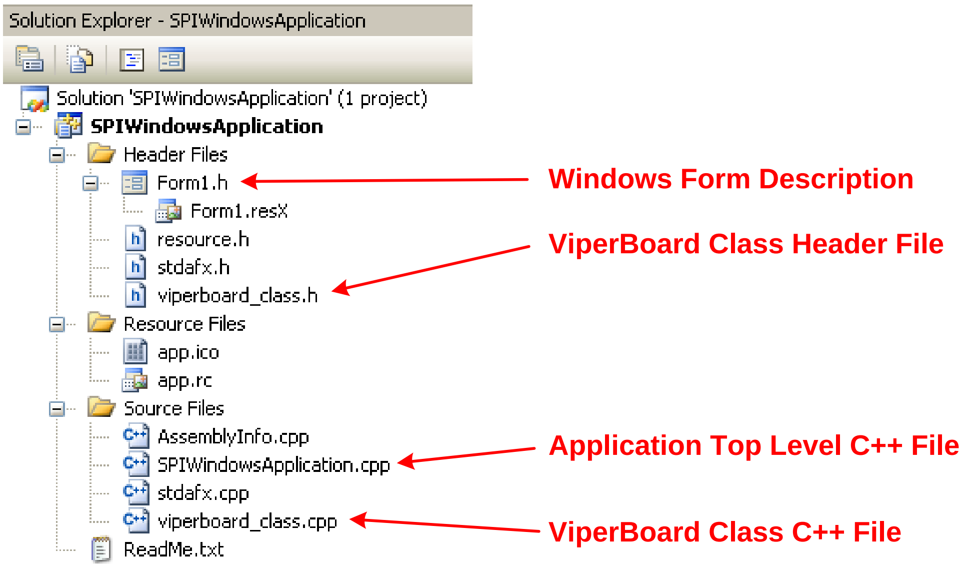
In order to create such a Windows application in Visual C++, first start up a new project and select "Window Form Application" as the project type.



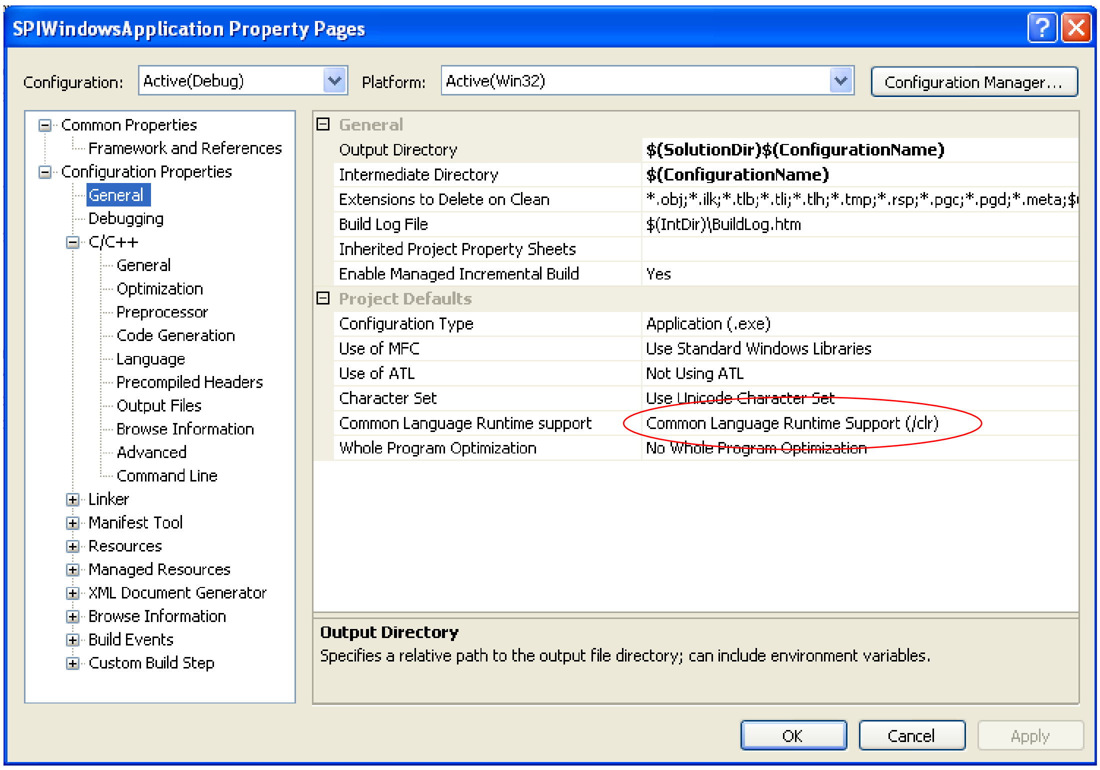
This will create a standard Windows application. The project file structure will then look as follows.



To this, one must add the ViperBoard class header file (viperboard_class.h) and ViperBoard class C++ file as follows.



Remember to change the Common Language Runtime support to /clr.



5.3. Design Description

The main file which needs to be developed in a Windows GUI application is the Windows form application file (Form1.h). This can be built up graphically by dragging in elements of the .NET development toolbox and/or by editing the Form1.h text file.

The final Form1.h file can be found in the downloaded files for the SPI Windows application available on the web-site.

The file starts by linking to the ViperBoard class library header file:

```
//////////////////////////////////////////////////////////////////////////
//
// Nano River Technologies
//
// File:         SPIWindowsApplication (Form.h)
//
// Desciption:   This is a simpe Windows application as a demonstration
//               to show how to build up a simple Windows application and
//               link to the SPI functions (both master and slave) in
//               the Nano River Techologies ViperBoard.
//
//               A good way to see this working is to simply loop back
//               the master to slave SPI interface on ViperBoard.
//
// Revision:     Version 1.0
//
//////////////////////////////////////////////////////////////////////////

#pragma once

#include "../../common/viperboard_class.h"
```

The file declares some global variables used in the application:

```
// Own Global variables here
//
NANO_RESULT res;                        // Returned result for calls to Nano River Tech functions
Int32       i;                          // General purpose loop variable
VBc         VB;                         // The ViperBoard class library
bool        connected;                  // True if the ViperBoard is connected
bool        error;                      // Flags if there was an in error
BYTE        master_spi_polarity;        // SPI polarity setting
BYTE        master_spi_phase;           // SPI phase setting
BYTE        master_spi_frequency;       // SPI phase setting
Int32       master_data_length;         // Length of the SPI transfer
BYTE        master_in_buffer[512];      // SPI input buffer (data from SPI device)
BYTE        master_out_buffer[512];     // SPI output buffer (data for SPI device)
Int32       slave_data_length;          // Length of the SPI transfer
BYTE        slave_in_buffer[512];       // SPI input buffer (data from SPI device)
BYTE        slave_out_buffer[512];      // SPI output buffer (data for SPI device)
```

```
BOOL        slave_checked;           // True if the user has armed the slave
WORD        GPIOEvent;               // GPIO event (one per GPIO bit) (Not Used)
BOOL        SPISlaveEvent;           // SPI slave event
BOOL        IICSlaveEvent;           // I2C slave event (not used)
BYTE        SPISlaveChan;            // SPI slave channel
WORD        SPISlaveBytes;           // SPI slave number of bytes
BYTE        IICSlaveTransferType;    // I2C slave tranfer type (not used)
WORD        IICSlaveT1Bytes;         // I2C slave number of T1 bytes
WORD        IICSlaveT2Bytes;         // I2C slave number of T2 bytes
```

Windows objects are defined:

```
// Define all the Windows objects here
//
private: System::Windows::Forms::Label^  label_00;
private: System::Windows::Forms::Label^  label_01;
private: System::Windows::Forms::Label^  label_02;
private: System::Windows::Forms::Label^  label_03;
   …
private: System::Windows::Forms::TextBox^  textBox_00;
private: System::Windows::Forms::TextBox^  textBox_01;
private: System::Windows::Forms::TextBox^  textBox_02;
private: System::Windows::Forms::TextBox^  textBox_03;
private: System::Windows::Forms::RichTextBox^  richTextBox_00;
private: System::Windows::Forms::RichTextBox^  richTextBox_01;
private: System::Windows::Forms::ComboBox^  comboBox_00;
private: System::Windows::Forms::ComboBox^  comboBox_01;
private: System::Windows::Forms::ComboBox^  comboBox_02;
private: System::Windows::Forms::Button^  button_00;
private: System::Windows::Forms::CheckBox^  checkBox_00;
private: System::Windows::Forms::Panel^  panel_00;
private: System::Windows::Forms::Panel^  panel_01;
private: System::Windows::Forms::Timer^  timer1;
```

We then define some managed variables:

```
// Managed variables
String      ^s1;       // Temporary string for reporting to a RichTextBox
String      ^s2;       // Temporary string for reporting to a RichTextBox
String      ^s3;       // Temporary string for reporting to a RichTextBox
String      ^s4;       // Temporary string for reporting to a RichTextBox
String      ^s5;       // Temporary string for reporting to a RichTextBox
String      ^s6;       // Temporary string for reporting to a RichTextBox
String      ^s7;       // Temporary string for reporting to a RichTextBox
String      ^s8;       // Temporary string for reporting to a RichTextBox
String      ^Data;     // A string used for parsing the master data input textbox
String      ^SlvData;  // A string used for parsing the slave data input textbox
```

A function is defined during the form load. In this the connection to the ViperBoard is established by calling *OpenDevice()* in the ViperBoard class. If connected then we also clear any pending SPI slave events and disarm the SPI slave.

```
private: System::Void Form1_Load_1(System::Object^  sender, System::EventArgs^  e) {
// A task called when the form is loaded. Here we
// open the USB link and initialise the ViperBoard.
// We also clear any pending slave events and enable
// new ones for when the slave is armed
//
    connected = ::VBc::OpenDevice();
    if (!connected)
        this->label_13->Visible = true;

    // Enable all events on SPI
    if (connected) {
        res=VB.Nano_SPISlaveArm(VB.vb_hDevice,false);
    }
}
```

A function is defined for when the user presses the master read/write button. When this occurs the state of the clock polarity and clock phase is checked and used to configure SPI master channel 0 using *Nano_SPIConfigure().* Next the line rate is read and configured using *Nano_SPIMasterSetFrequency().* The output data is then parsed. An SPI read/write command is issued using *Nano_SPIMasterReadWrite().* This function returns the read data which is then formatted and displayed in the master information window.

```
private: System::Void button_00_Click(System::Object^  sender, System::EventArgs^  e) {
// A task called when the user decides to send an SPI master command.
// The task will read the line rate, clock polarity and clock phase
// and use this in the command. The actual data sent and received is
// reported in the RichTextBox.
//

    /////////////////////////////////////////////////////////
    // SPI Configuration
    /////////////////////////////////////////////////////////

    // Get the polarity and phase
    if (this->comboBox_00->Text=="0")    master_spi_polarity=0;
    if (this->comboBox_00->Text=="1")    master_spi_polarity=1;
    if (this->comboBox_01->Text=="0")    master_spi_phase=0;
    if (this->comboBox_01->Text=="1")    master_spi_phase=1;

    // Set the clock polarity and phase
    if (connected)
        res = ::VBc::Nano_SPIConfigure(VB.vb_hDevice,0,0,false,
                                      master_spi_polarity,master_spi_phase);

    // Get the line rate
```

```
    if (this->comboBox_02->Text=="12 MBit/s")   master_spi_frequency=NANO_SPI_FREQ_12MHZ;
    if (this->comboBox_02->Text=="6 MBit/s")    master_spi_frequency=NANO_SPI_FREQ_6MHZ;
    if (this->comboBox_02->Text=="3 MBit/s")    master_spi_frequency=NANO_SPI_FREQ_3MHZ;
    if (this->comboBox_02->Text=="1 MBit/s")    master_spi_frequency=NANO_SPI_FREQ_1MHZ;
    if (this->comboBox_02->Text=="400 kBit/s")  master_spi_frequency=NANO_SPI_FREQ_400KHZ;
    if (this->comboBox_02->Text=="200 kBit/s")  master_spi_frequency=NANO_SPI_FREQ_200KHZ;
    if (this->comboBox_02->Text=="100 kBit/s")  master_spi_frequency=NANO_SPI_FREQ_100KHZ;
    if (this->comboBox_02->Text=="10 kBit/s")   master_spi_frequency=NANO_SPI_FREQ_10KHZ;

    // Set the line rate
    if (connected)
        res=VB.Nano_SPIMasterSetFrequency(VB.vb_hDevice,master_spi_frequency);

    /////////////////////////////////////////////////////////
    // SPI Master Send
    /////////////////////////////////////////////////////////

    // Get the data length
    master_data_length = Int32::Parse(this->
        textBox_00->Text,System::Globalization::NumberStyles::HexNumber);

    // Clear the text strings
    s1= "";
    s2= "";
    s3= "";

    // Parse the data to be sent and split it into bytes
    Data          = this->textBox_01->Text;
    array<String^>^ splitData = Data->Split(' ');
          // Do string splitting to get the individual data entries. Expect a space between
    for (i=0; i<splitData->Length; i++)
        if (i<master_data_length)         // Fill output buffer with data
            master_out_buffer[i]=Convert::ToInt32(splitData[i],16);
    for (i=0;i<master_data_length;i++) {  // Print Buffer
        s2 = System::String::Concat( s2," 0x");
        s2 = System::String::Concat( s2,master_out_buffer[i].ToString("X2"));
        if (((i+1)%16)==0)
            s2 = System::String::Concat( s2,"\n  ");
    }

    // Send the SPI command
    if (connected)
        res=VB.Nano_SPIMasterReadWrite(VB.vb_hDevice,0,master_data_length,
                                 master_in_buffer,master_out_buffer);
    if (res!=NANO_SUCCESS) error = true;

    for (i=0;i<master_data_length;i++) {  // Print Buffer
        s3 = System::String::Concat( s3," 0x");
        s3 = System::String::Concat( s3,master_in_buffer[i].ToString("X2"));
        if (((i+1)%16)==0)
            s3 = System::String::Concat( s3,"\n  ");
    }

    // Update the status of the read/write
    s1= "Writing";
    s1 = System::String::Concat( s1,s2);
    s1 = System::String::Concat( s1,"\nReading");
    s1 = System::String::Concat( s1,s3);
    this->richTextBox_00->Text = s1;
}
```

A function is defined for when the user ticks the SPI slave arm tick-box. When the box is "ticked" the function parses the data to be filled in the slave buffer. The data is then filled using *Nano_SPISlaveBufferWrite().* The slave is then armed using *Nano_SPISlaveArm().* The data written to the slave buffer is displayed in the slave information window.

```
private: System::Void checkBox_00_CheckedChanged(System::Object^  sender, System::EventArgs^  e)
{
// A task called when the user decides to arm the SPI slave by clicking
// on the checkbox. Essentially this is the trigger to get the slave data
// and fill the actual slave buffer memory in readiness for a slave SPI command.
//

    // Check if the box is now armed!
    slave_checked = this->checkBox_00->Checked;

    /////////////////////////////////////////////////////////
    // SPI Slave Setup Buffer
    /////////////////////////////////////////////////////////
    if (slave_checked) {
        slave_data_length = Int32::Parse(this->
            textBox_02->Text,System::Globalization::NumberStyles::HexNumber);

        s4= "";
        s5= "";
        s6= "";

        // Parse the data to be sent and split it into bytes
        SlvData         = this->textBox_03->Text;
        array<String^>^ splitSlvData = SlvData->Split(' ');
            // Do string splitting to get the individual data entries. Expect a space between
        for (i=0; i<splitSlvData->Length; i++)
            if (i<slave_data_length)         // Fill output buffer with data
                slave_out_buffer[i]=Convert::ToInt32(splitSlvData[i],16);
        for (i=0;i<slave_data_length;i++) {  // Print Buffer
            s5= System::String::Concat( s5," 0x");
            s5 = System::String::Concat( s5,slave_out_buffer[i].ToString("X2"));
            if (((i+1)%16)==0)
                s5 = System::String::Concat( s5,"\n  ");
        }

        // Write slave data into the ViperBoard SPI slave buffer
        if (connected)
            res=VB.Nano_SPISlaveBufferWrite(VB.vb_hDevice,0,slave_data_length,slave_out_buffer);

        // Arm the slave
        if (connected)
            res=VB.Nano_SPISlaveArm(VB.vb_hDevice,true);
    }
}
```

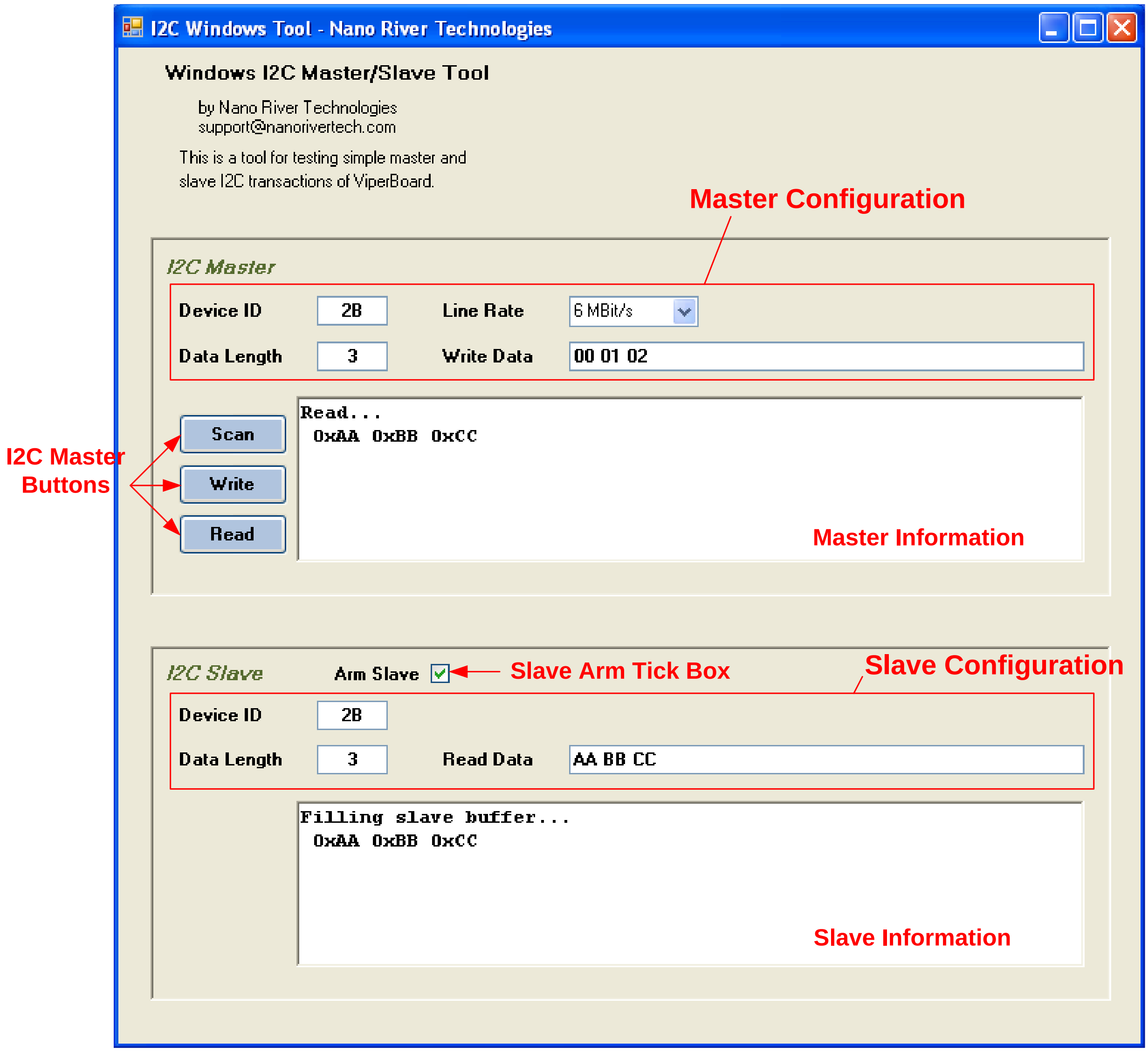
The final function is a polling function used to check if the slave has received any data. *Nano_GetEvents()* is used for this purpose. If the slave has received data then the slave buffer is read using *Nano_SPISlaveReadBuffer().* The received data is then displayed in the slave information window.

```
private: System::Void timer1_Tick(System::Object^  sender, System::EventArgs^  e) {
// This is a polling task to check if there was something on the SPI slave interface.
// If so then the actual data received is presented to the user in the RichTextBox.
//

    if (connected) {

        // Poll the event reister
        res=VB.Nano_GetEvents(VB.vb_hDevice,
                                    &GPIOEvent,
                                    &SPISlaveEvent,
                                    &IICSlaveEvent,
                                    &SPISlaveChan,
                                    &SPISlaveBytes,
                                    &IICSlaveTransferType,
                                    &IICSlaveT1Bytes,
                                    &IICSlaveT2Bytes);

        // Check if there was something on SPI channel 0
        if (SPISlaveEvent && (SPISlaveChan==0)) {

            ////////////////////////////////////////////////////////
            // SPI Slave Buffer Read
            ////////////////////////////////////////////////////////
            s7= "";
            s8= "";

            res=VB.Nano_SPISlaveBufferRead(VB.vb_hDevice,0,slave_data_length,slave_in_buffer);
            for (i=0;i<slave_data_length;i++) {  // Print Buffer
                s7 = System::String::Concat( s7," 0x");
                s7 = System::String::Concat( s7,slave_in_buffer[i].ToString("X2"));
                if (((i+1)%16)==0)
                    s7 = System::String::Concat( s7,"\n  ");
            }

            // Update the status of the read/write
            s8= "Writing";
            s8 = System::String::Concat( s8,s5);
            s8 = System::String::Concat( s8,"\nReading");
            s8 = System::String::Concat( s8,s7);
            this->richTextBox_01->Text = s8;
            this->checkBox_00->Checked = false;
        }
    }
}
```

Some important links and includes are necessary in stdafx.h. This is best seen by viewing the file in the download from the web-site.

6. Windows I2C Tool

The Windows I2C tool is a windows GUI application in Visual C++ used to demonstrate both master and slave operation of the I2C interface for ViperBoard.



6.1. Functionality

The application consists of two panels. The top panel is for controlling the I2C master. The lower panel is for controlling the I2C slave.

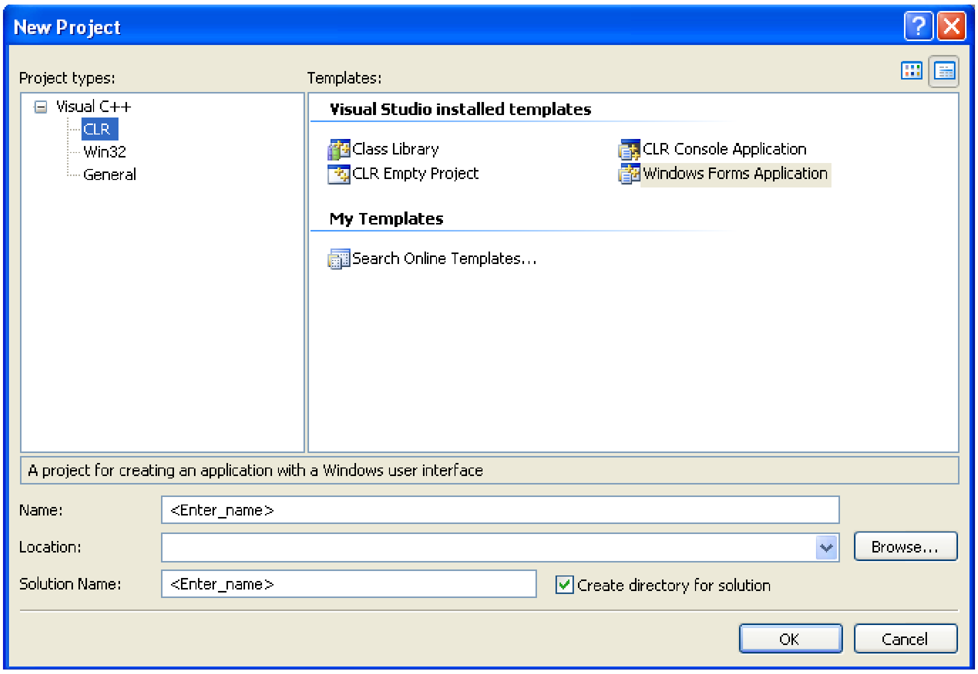
In the upper panel the user can configure the master device ID and line rate. The data to be sent is also specified in terms of the actual data and the length of the transfer. The user may press one of three buttons. The scan button scans checks to see which I2C devices are connected to the master and displays this in the master information window. Alternatively the user can perform an I2C write or an I2C read, the results of which are also displayed in the master information window.

In the lower panel the user specifies the slave device ID and the data which resides in the slave buffer. Activating the slave arm tick-box makes the slave ready to respond to an I2C master. When a transaction is performed the result of the transaction is indicated in the slave information window.

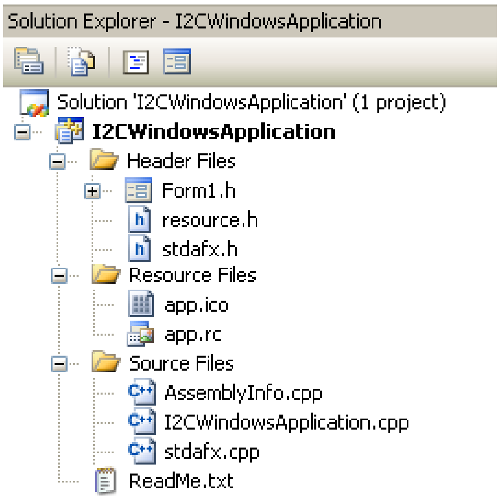
To see this application working one can simply set the master device ID and slave device ID equal. Because the I2C master and slave sit on the same ViperBoard I2C bus, then the slave will respond to the master. No external connection is required. Remember to arm the slave! The master and slave can then transfer data.

6.2. Project Structure

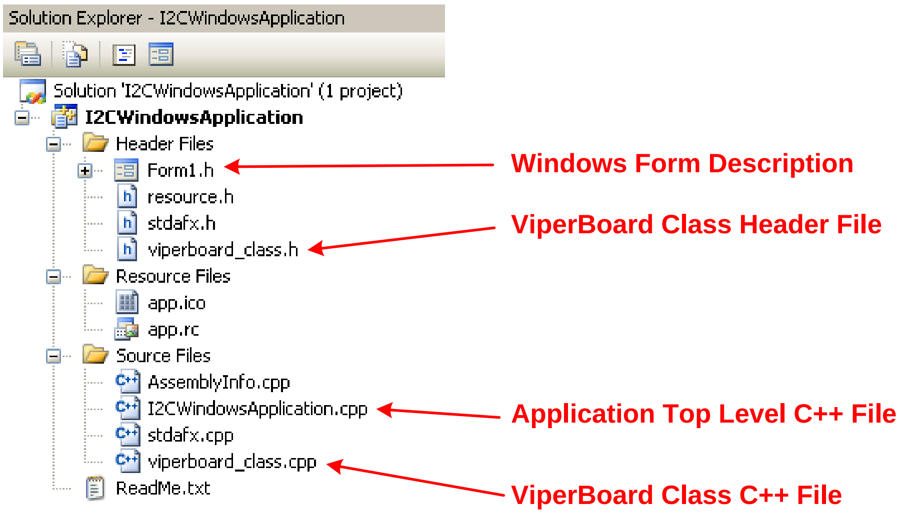
In order to create such a Windows application in Visual C++, first start up a new project and select "Window Form Application" as the project type.



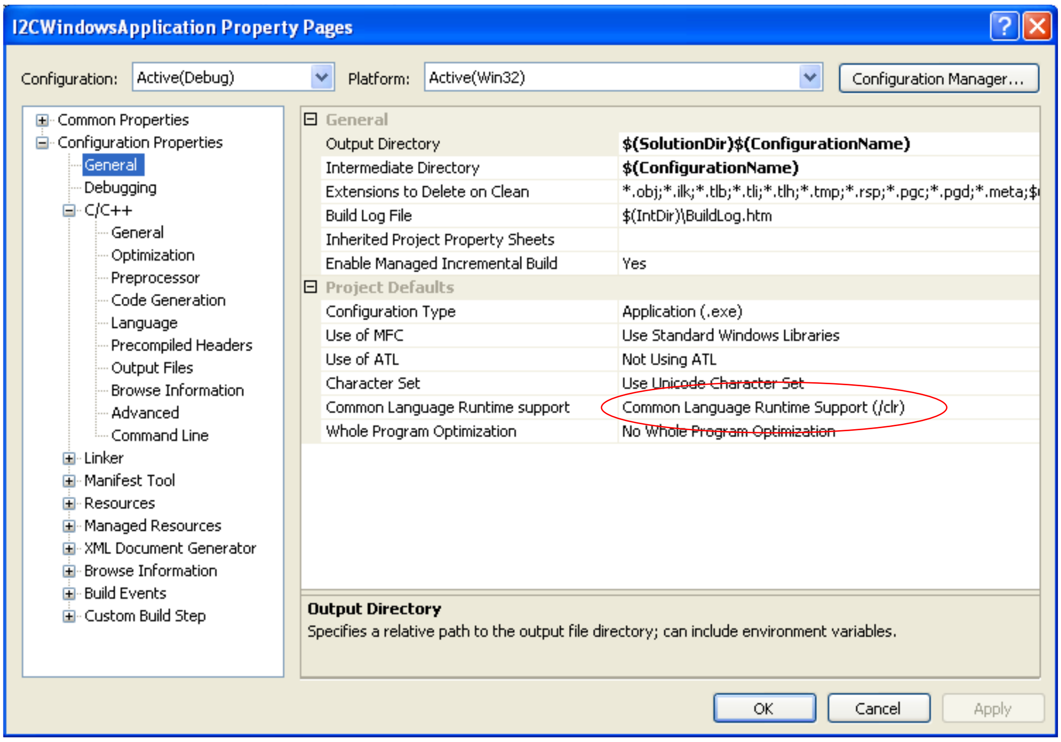
This will create a standard Windows application. The project file structure will then look as follows.



To this, one must add the ViperBoard class header file (viperboard_class.h) and ViperBoard class C++ file as follows.



Remember to change the Common Language Runtime support to /clr.



6.3. Design Description

The main file which needs to be developed in a Windows GUI application is the Windows form application file (Form1.h). This can be built up graphically by dragging in elements of the .NET development toolbox and/or by editing the Form1.h text file.

The final Form1.h file can be found in the downloaded files for the I2C Windows application available on the web-site.

The file starts by linking to the ViperBoard class library header file:

```
//////////////////////////////////////////////////////////////////////////
//
// Nano River Technologies
//
// File:          I2CWindowsApplication (Form.h)
//
// Desciption:    This is a simpe Windows application as a demonstration
//                to show how to build up a simple Windows application and
//                link to the I2C functions (both master and slave) in
//                the Nano River Techologies ViperBoard.
//
//                The ViperBoard can have master and slave working together
//                so the master and slave is by default looped-back since
//                they are connected to the same connector pins. In this way
//                you can test this application without needing to connect
//                to any real I2C devices.
//
// Revision:      Version 1.0
//
//////////////////////////////////////////////////////////////////////////

#pragma once

#include "../../common/viperboard_class.h"
```

The file declares some global variables used in the application:

```
// Own Global variables here
//
NANO_RESULT   res;                    // Returned result for Nano River Tech functions
Int32        i;                      // General purpose loop variable
VBc          VB;                     // The ViperBoard class library
bool         connected;              // True if the ViperBoard is connected
bool         error;                  // Flags if there was an in error
BYTE         master_frequency;       // Master I2C line rate
BYTE         master_device_id;       // Master I2C Device ID
Int32        master_data_length;     // Length of the Master I2C transfer
BYTE         master_in_buffer[512];  // Master I2C input buffer (data from I2C device)
BYTE         master_out_buffer[512];// Master I2C output buffer (data for I2C device)
BYTE         slave_device_id;        // Slave I2C Device ID
```

```
Int32       slave_data_length;     // Length of the Slave I2C transfer
BYTE        slave_in_buffer[512];  // Slave I2C input buffer (data from I2C device)
BYTE        slave_out_buffer[512]; // Slave I2C output buffer (data for I2C device)
BOOL        slave_checked;         // Flag indicating the user has armed the I2C slave
BOOL        IICMasterError;        // I2C master error
WORD        GPIOEvent;             // GPIO event (one per GPIO bit) (Not Used)
BOOL        SPISlaveEvent;         // SPI slave event (not used)
BOOL        IICSlaveEvent;         // I2C slave event
BYTE        SPISlaveChan;          // SPI slave channel (not used)
WORD        SPISlaveBytes;         // SPI slave number of bytes (not used)
BYTE        IICSlaveTransferType;  // I2C slave tranfer type
WORD        IICSlaveT1Bytes;       // I2C slave number of T1 bytes
WORD        IICSlaveT2Bytes;       // I2C slave number of T2 bytes
DEV_LIST    lst;                   // Returned list of connected I2C devices
```

Windows objects are defined:

```
// Define all the Windows objects here
//
private: System::Windows::Forms::Label^  label_00;
private: System::Windows::Forms::Label^  label_01;
private: System::Windows::Forms::Label^  label_02;
private: System::Windows::Forms::Label^  label_03;
   …
private: System::Windows::Forms::TextBox^  textBox_00;
private: System::Windows::Forms::TextBox^  textBox_01;
private: System::Windows::Forms::TextBox^  textBox_02;
private: System::Windows::Forms::TextBox^  textBox_03;
   …
private: System::Windows::Forms::RichTextBox^  richTextBox_00;
private: System::Windows::Forms::RichTextBox^  richTextBox_01;
private: System::Windows::Forms::ComboBox^  comboBox_00;
private: System::Windows::Forms::Button^  button_00;
private: System::Windows::Forms::Button^  button_01;
private: System::Windows::Forms::Button^  button_02;
private: System::Windows::Forms::CheckBox^  checkBox_00;
private: System::ComponentModel::IContainer^  components;
private: System::Windows::Forms::Panel^  panel_00;
private: System::Windows::Forms::Panel^  panel_01;
private: System::Windows::Forms::Timer^  timer1;
```

We then define some managed variables:

```
// Managed variables
String      ^s1;       // Temporary string for reporting to a RichTextBox
String      ^s2;       // Temporary string for reporting to a RichTextBox
String      ^s3;       // Temporary string for reporting to a RichTextBox
String      ^s4;       // Temporary string for reporting to a RichTextBox
String      ^Data;     // A string used for parsing the master data input textbox
String      ^SlvData;  // A string used for parsing the slave data input textbox
```

A function is defined during the form load. In this the connection to the ViperBoard is established by calling *OpenDevice()* in the ViperBoard class. If connected then we also clear any pending I2C slave events and disarm the I2C slave.

```
private: System::Void Form1_Load(System::Object^  sender, System::EventArgs^  e) {
// A task called when the form is loaded. Here we
// open the USB link and initialise the ViperBoard.
// We also clear any pending slave events and enable
// new ones for when the slave is armed
//
    connected = ::VBc::OpenDevice();
    if (!connected)
        this->label_13->Visible = true;

    if (connected) {
        // Clear any pending I2C slave events
        res=VB.Nano_I2CSlaveArm(VB.vb_hDevice,false);
    }

}
```

A function is defined for when the user presses the master scan button. When this occurs the master line rate is checked and set using *Nano_I2CMasterSetFrequency()*. A scan is then performed using *Nano_I2CMasterScanConnectedDevices()*. Connected devices are then displayed in the master information window.

```
private: System::Void button_00_Click(System::Object^  sender, System::EventArgs^  e) {
// A task called when the user decides to send an I2C scan command.
// The task will read the line rate and use this in the I2C scan
// command. Devices connected are reported in the RichTextBox.
//

    // Get the line rate
    if (this->comboBox_00->Text=="6 MBit/s")    master_frequency=NANO_I2C_FREQ_6MHZ;
    if (this->comboBox_00->Text=="3 MBit/s")    master_frequency=NANO_I2C_FREQ_3MHZ;
    if (this->comboBox_00->Text=="1 MBit/s")    master_frequency=NANO_I2C_FREQ_1MHZ;
    if (this->comboBox_00->Text=="400 kBit/s")  master_frequency=NANO_I2C_FREQ_FAST;
    if (this->comboBox_00->Text=="200 kBit/s")  master_frequency=NANO_I2C_FREQ_200KHz;
    if (this->comboBox_00->Text=="100 kBit/s")  master_frequency=NANO_I2C_FREQ_STD;
    if (this->comboBox_00->Text=="10 kBit/s")   master_frequency=NANO_I2C_FREQ_10KHZ;

    s1= "Scanning.....\n";
    s2= "  Device\(s\) found:";
    s3= "  No Device Connected";
    bool found;
    found = false;

    // Setup the line rate and then perform scan
    if (connected) {
        res = VB.Nano_I2CMasterSetFrequency(VB.vb_hDevice,master_frequency);
```

```
            // Setup the clock speed
        res = VB.Nano_I2CMasterScanConnectedDevices(VB.vb_hDevice,&lst);
    }

    // Report to the screen
    for (i=0;i<128;i++) {
        if (lst.List[i]) {
            found=true;
            s2 = System::String::Concat( s2," 0x");
            s2 = System::String::Concat( s2,i.ToString("X2"));
        }
    }
    if (found) {
        s1 = System::String::Concat( s1,s2);
    } else {
        s1 = System::String::Concat( s1,s3);
    }
    this->richTextBox_00->Text = s1;
}
```

A function is defined for when the user presses the master write button. When this occurs the master line rate is checked and set using *Nano_I2CMasterSetFrequency()*. The master device ID and data is then parsed. An I2C write is performed using *Nano_I2CMasterWrite()*. A summary of the transfer appears in the master information window.

```
private: System::Void button_01_Click(System::Object^  sender, System::EventArgs^  e) {
// A task called when the user decides to send an I2C write command.
// The task will read the line rate, device ID, data length and data
// and use this in the I2C write command. The data written is reported
// in the RichTextBox.
//

    // Get the line rate
    if (this->comboBox_00->Text=="6 MBit/s")    master_frequency=NANO_I2C_FREQ_6MHZ;
    if (this->comboBox_00->Text=="3 MBit/s")    master_frequency=NANO_I2C_FREQ_3MHZ;
    if (this->comboBox_00->Text=="1 MBit/s")    master_frequency=NANO_I2C_FREQ_1MHZ;
    if (this->comboBox_00->Text=="400 kBit/s")  master_frequency=NANO_I2C_FREQ_FAST;
    if (this->comboBox_00->Text=="200 kBit/s")  master_frequency=NANO_I2C_FREQ_200KHz;
    if (this->comboBox_00->Text=="100 kBit/s")  master_frequency=NANO_I2C_FREQ_STD;
    if (this->comboBox_00->Text=="10 kBit/s")   master_frequency=NANO_I2C_FREQ_10KHZ;

    // Get the Device ID
    master_device_id = Int32::Parse(this->
       textBox_00->Text,System::Globalization::NumberStyles::HexNumber);

    // Get the data length
    master_data_length = Int32::Parse(this->
       textBox_01->Text,System::Globalization::NumberStyles::HexNumber);

    // Parse the data to be sent and split it into bytes
    s0= "";
    Data          = this->textBox_02->Text;
    array<String^>^ splitData = Data->Split(' ');
      // Do string splitting to get the individual data entries. Expect a space between
    for (i=0; i<splitData->Length; i++)
        if (i<master_data_length)         // Fill output buffer with data
```

```
            master_out_buffer[i]=Convert::ToInt32(splitData[i],16);
    for (i=0;i<master_data_length;i++) {  // Print Buffer
        s0 = System::String::Concat( s0," 0x");
        s0 = System::String::Concat( s0,master_out_buffer[i].ToString("X2"));
        if (((i+1)%16)==0)
            s0 = System::String::Concat( s0,"\n  ");
        }

    // Update the status of the read/write
    s1= "Writing...\n";
    s1 = System::String::Concat( s1,s0);

    // Set the line rate and perform the I2C write
    IICMasterError = false;
    if (connected) {
        res=VB.Nano_I2CMasterSetFrequency(VB.vb_hDevice,master_frequency);
        res=VB.Nano_I2CMasterWrite(VB.vb_hDevice,master_device_id,
                                   master_data_length,master_out_buffer);
    IICMasterError = (res!=NANO_SUCCESS);
    }

    s2="";
    if (IICMasterError)
        s2="\nError during write!\n";
    s2 = System::String::Concat( s1,s2);
    this->richTextBox_00->Text = s2;
}
```

A function is defined for when the user presses the master read button. When this occurs the master line rate is checked and set using *Nano_I2CMasterSetFrequency()*. An I2C read is performed using *Nano_I2CMasterRead()*. A summary of the transfer including the received data appears in the master information window.

```
private: System::Void button_02_Click(System::Object^  sender, System::EventArgs^  e) {
// A task called when the user decides to send an I2C read command.
// The task will read the line rate, device ID, data length and data
// and use this in the I2C read command. The data read is reported
// in the RichTextBox.
//

    // Get the line rate
    if (this->comboBox_00->Text=="6 MBit/s")    master_frequency=NANO_I2C_FREQ_6MHZ;
    if (this->comboBox_00->Text=="3 MBit/s")    master_frequency=NANO_I2C_FREQ_3MHZ;
    if (this->comboBox_00->Text=="1 MBit/s")    master_frequency=NANO_I2C_FREQ_1MHZ;
    if (this->comboBox_00->Text=="400 kBit/s")  master_frequency=NANO_I2C_FREQ_FAST;
    if (this->comboBox_00->Text=="200 kBit/s")  master_frequency=NANO_I2C_FREQ_200KHz;
    if (this->comboBox_00->Text=="100 kBit/s")  master_frequency=NANO_I2C_FREQ_STD;
    if (this->comboBox_00->Text=="10 kBit/s")   master_frequency=NANO_I2C_FREQ_10KHZ;

    // Get the Device ID
    master_device_id = Int32::Parse(this->
       textBox_00->Text,System::Globalization::NumberStyles::HexNumber);

    // Get the data length
    master_data_length = Int32::Parse(this->
       textBox_01->Text,System::Globalization::NumberStyles::HexNumber);

    // Initialise the buffer
    for (i=0;i<master_data_length;i++) {
```

```
        master_in_buffer[i]=0;
    }

    // Perform the read
    IICMasterError = false;
    if (connected) {
        res=VB.Nano_I2CMasterSetFrequency(VB.vb_hDevice,master_frequency);
        res=VB.Nano_I2CMasterRead(VB.vb_hDevice,master_device_id,
                                  master_data_length,master_in_buffer);
        IICMasterError = (res!=NANO_SUCCESS);
    }

    // Report the read data
    s0= "";
    for (i=0;i<master_data_length;i++) {  // Print Buffer
        s0 = System::String::Concat( s0," 0x");
        s0 = System::String::Concat( s0,master_in_buffer[i].ToString("X2"));
        if (((i+1)%16)==0)
            s0 = System::String::Concat( s0,"\n  ");
    }

    if (IICMasterError) {
        s1= "Error during read!\n";
    } else {
        s1= "Read...\n";
        s1 = System::String::Concat( s1,s0);
    }
    this->richTextBox_00->Text = s1;
}
```

A function is defined for when the user ticks the I2C slave arm tick-box. When the box is "ticked" the function parses the data to be filled in the slave buffer. The data is then filled using *Nano_I2CSlaveBuffer1Write*(). The data written to the slave buffer is displayed in the slave information window. The function also parses the slave device ID and sets this using *Nano_I2CSlaveConfig().* The slave is armed using *NanoI2CSlaveArm().*

```
private: System::Void checkBox_00_CheckedChanged(System::Object^  sender, System::EventArgs^  e)
{
// A task called when the user decides to arm the I2C slave by clicking
// on the checkbox. Essentially this is the trigger to get the slave data
// and fill the actual slave buffer memory in readiness for a slave I2C command.
//

    // Check if the box is now armed!
    slave_checked = this->checkBox_00->Checked;

    if (slave_checked) {

        // Get the Device ID
        slave_device_id = Int32::Parse(this->
            textBox_03->Text,System::Globalization::NumberStyles::HexNumber);
        if (connected)
            res=VB.Nano_I2CSlaveConfig(VB.vb_hDevice,slave_device_id);

        // Get the data length
        slave_data_length = Int32::Parse(this->
            textBox_04->Text,System::Globalization::NumberStyles::HexNumber);

        // Parse the data to be sent and split it into bytes
        s0= "";
        Data          = this->textBox_05->Text;
        array<String^>^ splitData = Data->Split(' ');
             // Do string splitting to get the individual data entries.
             // Expect a space between
        for (i=0; i<splitData->Length; i++)
            if (i<slave_data_length)          // Fill output buffer with data
                slave_out_buffer[i]=Convert::ToInt32(splitData[i],16);
        for (i=0;i<slave_data_length;i++) {  // Print Buffer
            s0 = System::String::Concat( s0," 0x");
            s0 = System::String::Concat( s0,slave_out_buffer[i].ToString("X2"));
            if (((i+1)%16)==0)
                s0 = System::String::Concat( s0,"\n  ");
        }

        s1= "Filling slave buffer...\n";
        s1 = System::String::Concat( s1,s0);
        this->richTextBox_01->Text = s1;

        // Write to the slave buffer
        if (connected)
            res=VB.Nano_I2CSlaveBuffer1Write(VB.vb_hDevice,
                                             slave_data_length,slave_out_buffer);
        if (connected)
            res=VB.Nano_I2CSlaveArm(VB.vb_hDevice,true);
    }
}
```

The final function is a polling function used to check if the slave has received any data. *Nano_GetEvents()* is used for this purpose. If the slave has received data then the slave buffer is read using *Nano_I2CSlaveBuffer1Read()*. The received data is then displayed in the slave information window. The event is cleared by writing to *Nano_I2CSlaveArm()*.

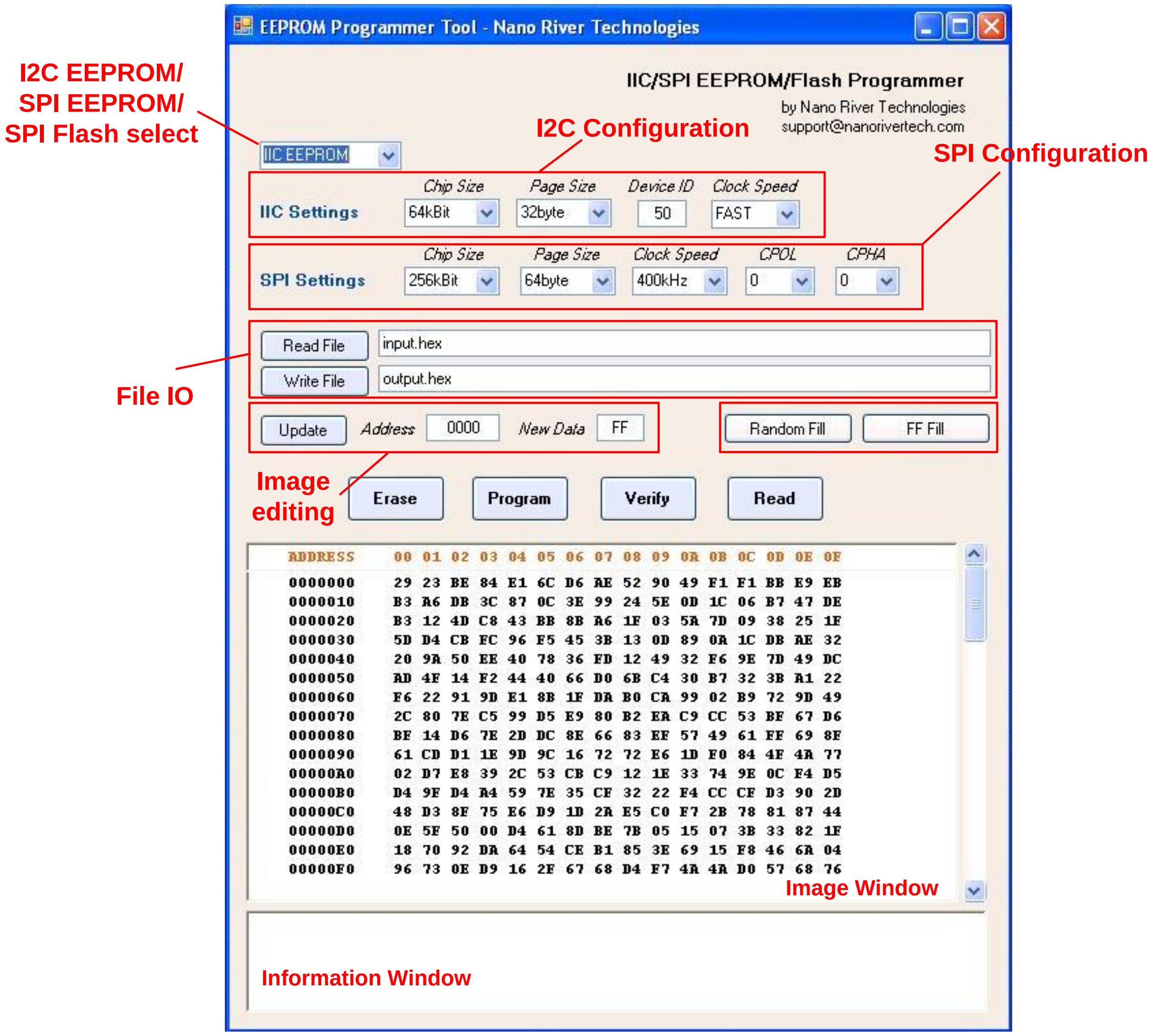
```
private: System::Void timer1_Tick(System::Object^  sender, System::EventArgs^  e) {
// This is a polling task to check if there was something on the I2C slave interface.
// If so then the actual data received is presented to the user in the RichTextBox.
//

    if (connected) {

        s0="";

        // Poll the event register
        res=VB.Nano_GetEvents(VB.vb_hDevice,
                              &GPIOEvent,
                              &SPISlaveEvent,
                              &IICSlaveEvent,
                              &SPISlaveChan,
                              &SPISlaveBytes,
                              &IICSlaveTransferType,
                              &IICSlaveT1Bytes,
                              &IICSlaveT2Bytes);

        // Check if there was something on I2C
        if (IICSlaveEvent) {
            if (IICSlaveTransferType==0) { //A write)
                res=VB.Nano_I2CSlaveBuffer1Read(VB.vb_hDevice,
                                                IICSlaveT1Bytes,slave_in_buffer);

                for (i=0;i<IICSlaveT1Bytes;i++) {  // Print Buffer
                    s0 = System::String::Concat( s0," 0x");
                    s0 = System::String::Concat( s0,slave_in_buffer[i].ToString("X2"));
                    if (((i+1)%16)==0)
                    s0 = System::String::Concat( s0,"\n  ");
                }

                // Update the status of the read/write
                s4= "Received...\n";
                s4 = System::String::Concat( s4,s0);

            } else if (IICSlaveTransferType==1) { //A read)
                res=VB.Nano_I2CSlaveBuffer1Read(VB.vb_hDevice,
                                                IICSlaveT1Bytes,slave_in_buffer);

                for (i=0;i<IICSlaveT1Bytes;i++) {  // Print Buffer
                    s0 = System::String::Concat( s0," 0x");
                    s0 = System::String::Concat( s0,slave_in_buffer[i].ToString("X2"));
                    if (((i+1)%16)==0)
                        s0 = System::String::Concat( s0,"\n  ");
                }

                // Update the status of the read/write
                s4= "Sent...\n";
                s4 = System::String::Concat( s4,s0);

            } else {
                s4= "Unknown I2C transfer!";
            }

            this->richTextBox_01->Text = s4;
```

```
            // service the I2C
            res=VB.Nano_I2CSlaveArm(VB.vb_hDevice,false);
            this->checkBox_00->Checked = false;
        }
    }
}
```

Some important links and includes are necessary in stdafx.h. This is best seen by viewing the file in the download from the web-site.

7. Windows I2C/SPI EEPROM / SPI Flash Programmer

This example application example is a simple programmer for I2C EEPROMs, SPI EEPROMs or SPI flash memories. The application is intended as a slightly more complex example of how to interface to both I2C and SPI interfaces in a real-life example.



7.1. Functionality

To use this application the user needs to select between I2C EEPROMs, SPI EEPROMs and SPI flash memories. This is achieved by using the I2C EEPROM / SPI EEPROM / SPI Flash pull-down menu.

For I2C EEPROMs, the user must then continue to configure the chip size, page size, device ID, clock speed by the various pull-down menus and text boxes in the I2C configuration. Similarly for SPI EEPROMs and SPI flash the user needs to configure the chip size, page size, clock rate, clock polarity, clock phase from the SPI configuration.

The current image in the programmer memory is displayed in the image window. One can scroll through this image to see all locations. To read from file one should specify the file name and press the “Read File” button. To save an image to file one should specify the file name and press the “Write File” button.

To edit a single location of the programmer image then specify the address and new data then press the “Update” button in the image editing. For random image import or to have 0xFF in all locations then press the “Random Fill” or “FF Fill” buttons.

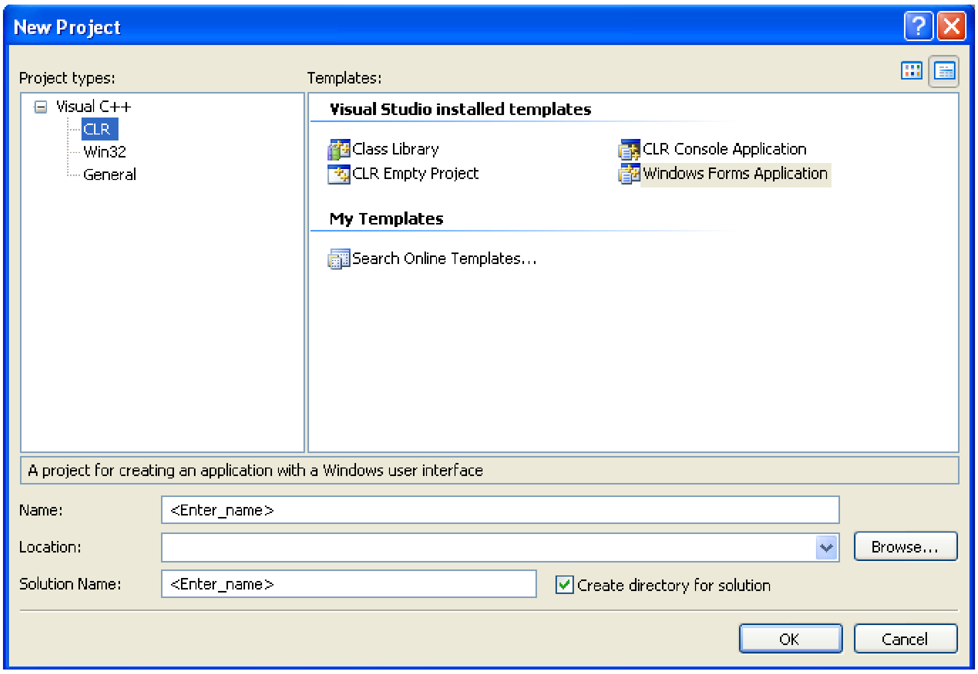
In the case of SPI flash a button is provided to erase the chip. This will erase all locations and make them 0xFF.

When the image is ready for programming one should press the “Program” button. The programming should immediately be checked by pressing the “Verify” button. An image can be read into the programmer’s memory from chip using the “Read” button.

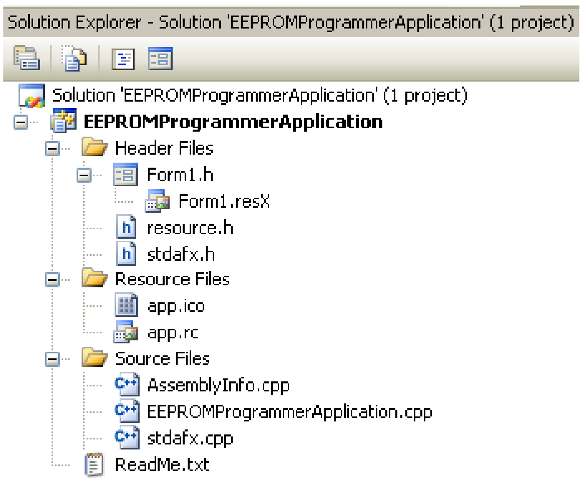
Success or otherwise of any of the operations is displayed in the information window.

7.2. Project Structure

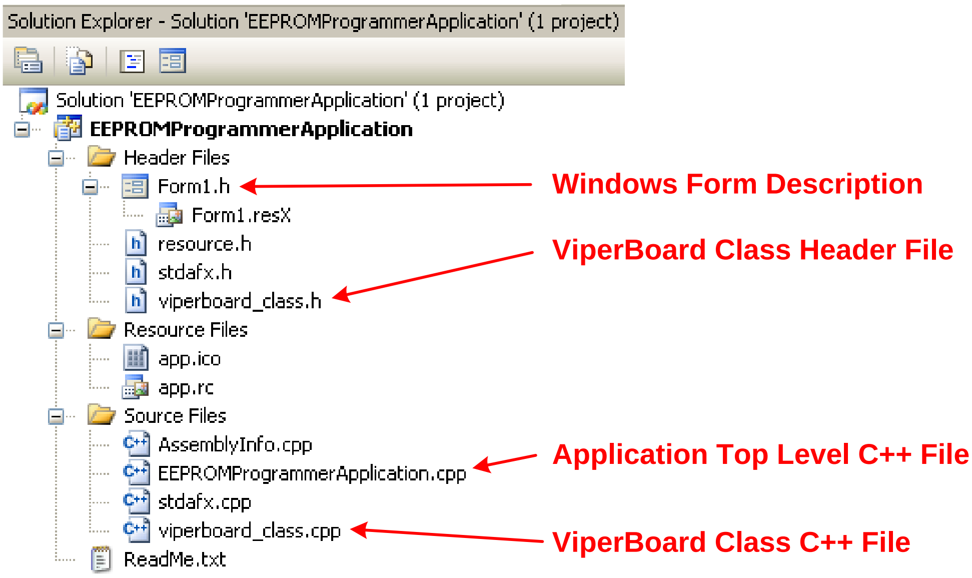
In order to create such a Windows application in Visual C++, first start up a new project and select "Window Form Application" as the project type.



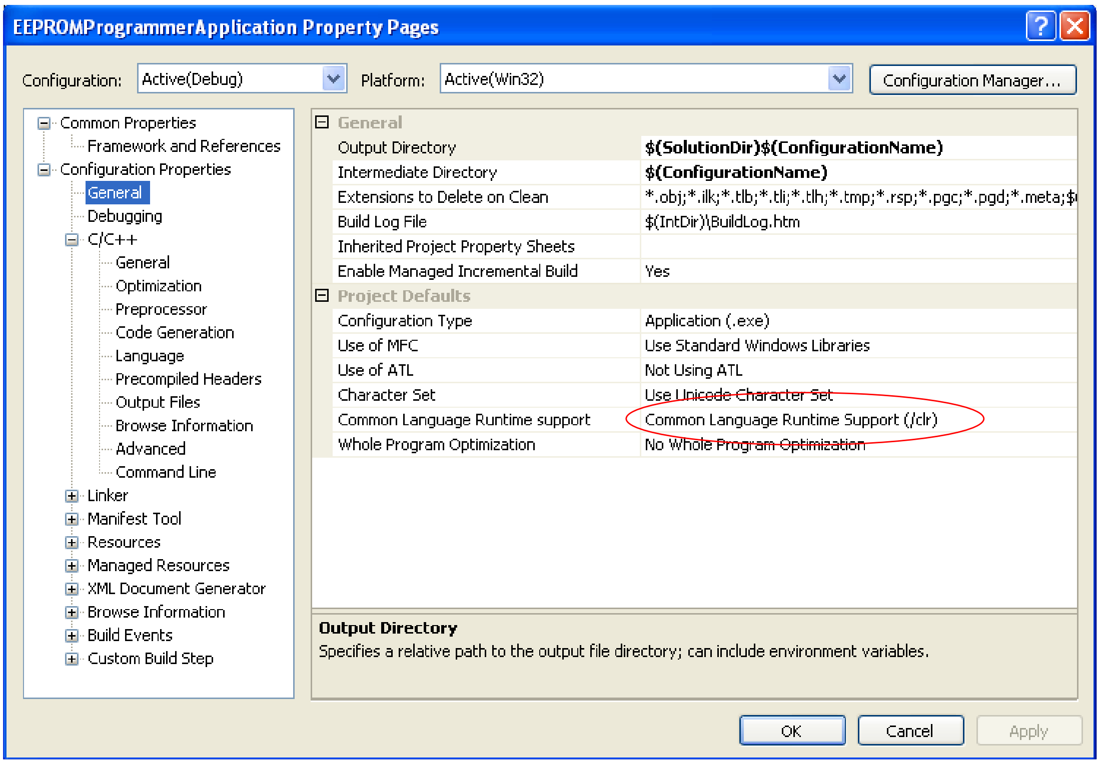
This will create a standard Windows application. The project file structure will then look as follows.



To this, one must add the ViperBoard class header file (viperboard_class.h) and ViperBoard class C++ file as follows.



Remember to change the Common Language Runtime support to /clr.



7.3. Design Description

The main file which needs to be developed in a Windows GUI application is the Windows form application file (Form1.h). This can be built up graphically by dragging in elements of the .NET development toolbox and/or by editing the Form1.h text file.

The final Form1.h file can be found in the downloaded files for the EEPROM Programmer application available on the web-site.

The file starts by linking to the ViperBoard class library header file:

```
//////////////////////////////////////////////////////////////////////////
//
// Nano River Technologies
//
// File:          EEPROMProgrammerApplication (Form.h)
//
// Desciption:    This is a simple EEPROM programmer working for IIC or SPI
//                style EEPROMs and flash. The programmer is made as an example of how
//                to build a Windows style application using IIC and SPI
//                commands and the Nano River Technologies ViperBoard.
//
// Revision:      Version 1.0
//
//////////////////////////////////////////////////////////////////////////


#pragma once

#include "../../common/viperboard_class.h"
```

The file declares some global variables used in the application:

```
   // Own Global variables here
   //
   const       Int32  MEMORYMAX = 131072*128;    // Maximum size of the memory array (128MBit)
   Int32       no_screens;                       // Number of screens of data to be displayed
   Int32       chip_size;                        // The actual chip size
   Int32       page_size;                        // The actial chip page size
   Int32       page_no;                          // Page number
   Int32       iic_write_time;                   // Chip write time (for IIC devices)
   Int32       spi_write_time;                   // Chip write time (for SPI devices)
   NANO_RESULT res;                              // Returned result for calls to
                                                 // Nano River Tech Functions

   Int32       i,j;                              // Loop variables
   BYTE        SlaveAddress;                     // IIC slave address (Device ID)
   Int32       StartAddress;                     // IIC transaction start address
   WORD        BufferLength;                     // IIC buffer length
   BYTE        iic_frequency;                    // IIC serial bit rate
   BYTE        InBuffer[4096];                   // IIC input buffer (data from IIC device)
   BYTE        OutBuffer[4096];                  // IIC output buffer (data for IIC device)
```

```
BYTE        spi_frequency;                  // SPI serial clock rate
BYTE        spi_polarity;                   // SPI polarity setting
BYTE        spi_phase;                      // SPI phase setting
BYTE        in_buffer[4096];                // SPI input buffer (data from SPI device)
BYTE        out_buffer[4096];               // SPI output buffer (data for SPI device)
BYTE        memory[MEMORYMAX];              // Storage to hold data to be programmed
BYTE        verify_memory[MEMORYMAX];       // Storage to hold data that was read
VBc         VB;                             // The ViperBoard class library
bool        connected;                      // True if the ViperBoard is connected
```

Windows objects are defined:

```
private: System::Windows::Forms::Label^  label_000;
private: System::Windows::Forms::Label^  label_001;
private: System::Windows::Forms::Label^  label_002;
…
private: System::Windows::Forms::ComboBox^  comboBox_000;
private: System::Windows::Forms::ComboBox^  comboBox_001;
private: System::Windows::Forms::ComboBox^  comboBox_002;
…
private: System::Windows::Forms::TextBox^  textBox_000;
private: System::Windows::Forms::TextBox^  textBox_001;
private: System::Windows::Forms::TextBox^  textBox_002;
…
private: System::Windows::Forms::Button^  button_000;
private: System::Windows::Forms::Button^  button_001;
private: System::Windows::Forms::Button^  button_002;
…
private: System::Windows::Forms::RichTextBox^  richTextBox_000;
private: System::Windows::Forms::RichTextBox^  richTextBox_001;
private: System::Windows::Forms::RichTextBox^  richTextBox_002;
private: System::Windows::Forms::VScrollBar^  vScrollBar_000;
```

Some managed variables are then declared:

```
// Managed variables
String      ^s1;      // A local string variable used for RichTechBox output
String      ^s2;      // 2nd local string variable used for RichTechBox output
String      ^s3;      // 3rd local string variable used for RichTechBox output
String      ^Data;    // A string used for parsing the data input textbox
```

A function is defined to display the current programmer contents. This is displayed in the image window. The EEPROM image is broken into screens and displayed one screen at a time. The scroll bar changes to the next screen of image.

```
private: System::Void ScreenUpdate() {
// A function to display the current screen of EEPROM data.
// Manages to show only the current screen based on the virtical slider.
//
   Int32 i,j;
   j = this->vScrollBar_000->Value;
   s2="";
   s2 = System::String::Concat( s2,j.ToString("X6"));
   this->richTextBox_001->Text = s2;
   s2 = "";
   for (i=0; i<256; i++) {
      if ((i%16)==0) {
         s2= System::String::Concat( s2,"    ");
         s2 = System::String::Concat( s2,(256*j+i).ToString("X7"));
         s2= System::String::Concat( s2,"   ");
      }
      s2 = System::String::Concat( s2," ");
      s2 = System::String::Concat( s2,memory[256*j+i].ToString("X2"));
      if (((i+1)%16)==0) {
         s2= System::String::Concat( s2,"\n");
      }
   }
   this->vScrollBar_000->Maximum = no_screens+8;
   this->richTextBox_001->Text = s2;
}
```

A function is defined during the form load. In this connection to the ViperBoard is established by calling the *OpenDevice()* in the ViperBoard class. The function also initialises the programmer contents and also the global variables. Finally a call is made to display the first screen of the image.

```
private: System::Void Form1_Load(System::Object^  sender, System::EventArgs^  e) {
// A task called when the form is loaded. Here we
// open the USB link and initialise the ViperBoard.
// Also initialise all the global varaibles.
//
   connected = ::VBc::OpenDevice();     // Open the device
   if (!connected)
      this->label_300->Visible = true;
   for (i=0; i<MEMORYMAX; i++)
      memory[i] = rand()&0xFF;         // Randomise the image
   no_screens = 32;                     // Default values are set below...
   page_size = 32;
   page_no = 0;
   chip_size = no_screens * 256;
   iic_frequency=NANO_I2C_FREQ_FAST;
   spi_frequency=NANO_SPI_FREQ_400KHZ;
   iic_write_time = 5;
   spi_write_time = 5;
   ScreenUpdate();                      // Screen update
}
```

This function updates the image window when the user clicks on the vertical scroll bar.

```
private: System::Void vScrollBar_000_Scroll(System::Object^  sender,
System::Windows::Forms::ScrollEventArgs^  e) {
// On a virtical slider movement update the screen
//
   ScreenUpdate();
}
```

These two functions re-calculate chip size and number of screens to display of the image in the case there has been an update to the chip size pull-down menu. One function is provided for I2C and one for the SPI case.

```
private: System::Void comboBox_001_SelectedIndexChanged(System::Object^  sender,
System::EventArgs^  e) {
// Update the chip size when IIC EEPROM is selected
//
   if (this->comboBox_000->Text=="IIC EEPROM") {
      page_no = 0;
      if (this->comboBox_001->Text=="1MBit")   no_screens = 512;
      if (this->comboBox_001->Text=="512kBit") no_screens = 256;
      if (this->comboBox_001->Text=="256kBit") no_screens = 128;
      if (this->comboBox_001->Text=="128kBit") no_screens = 64;
      if (this->comboBox_001->Text=="64kBit")  no_screens = 32;
      if (this->comboBox_001->Text=="32kBit")  no_screens = 16;
      if (this->comboBox_001->Text=="16kBit")  no_screens = 8;
      if (this->comboBox_001->Text=="8kBit")   no_screens = 4;
      if (this->comboBox_001->Text=="4kBit")   no_screens = 2;
      if (this->comboBox_001->Text=="2kBit")   no_screens = 1;
      this->vScrollBar_000->Value=0;
      chip_size = no_screens * 256;
   }
}

private: System::Void comboBox_005_SelectedIndexChanged(System::Object^  sender,
System::EventArgs^  e) {
// Update the chip size when SPI EEPROM is selected
//
   if ((this->comboBox_000->Text=="SPI EEPROM")||(this->comboBox_000->Text=="SPI Flash 25xx")) {
      page_no = 0;
      if (this->comboBox_005->Text=="128MBit") no_screens = 65536;
      if (this->comboBox_005->Text=="64MBit")  no_screens = 32768;
      if (this->comboBox_005->Text=="32MBit")  no_screens = 16384;
      if (this->comboBox_005->Text=="16MBit")  no_screens = 8192;
      if (this->comboBox_005->Text=="8MBit")   no_screens = 4096;
      if (this->comboBox_005->Text=="4MBit")   no_screens = 2048;
      if (this->comboBox_005->Text=="2MBit")   no_screens = 1024;
      if (this->comboBox_005->Text=="1MBit")   no_screens = 512;
      if (this->comboBox_005->Text=="512kBit") no_screens = 256;
      if (this->comboBox_005->Text=="256kBit") no_screens = 128;
      if (this->comboBox_005->Text=="128kBit") no_screens = 64;
      if (this->comboBox_005->Text=="64kBit")  no_screens = 32;
      if (this->comboBox_005->Text=="32kBit")  no_screens = 16;
      if (this->comboBox_005->Text=="16kBit")  no_screens = 8;
      if (this->comboBox_005->Text=="8kBit")   no_screens = 4;
      if (this->comboBox_005->Text=="4kBit")   no_screens = 2;
```

```
      if (this->comboBox_005->Text=="2kBit")   no_screens = 1;
      this->vScrollBar_000->Value=0;
      chip_size = no_screens * 256;
   }
}
```

These two functions re-calculate page size in the case there has been an update to the page size pull-down menu. One function is provided for I2C and one for the SPI case.

```
private: System::Void comboBox_002_SelectedIndexChanged(System::Object^  sender,
System::EventArgs^  e) {
// Update the page size variable when IIC is selected
//
   if (this->comboBox_000->Text=="IIC") {
      if (this->comboBox_002->Text=="4byte")   page_size = 4;
      if (this->comboBox_002->Text=="8byte")   page_size = 8;
      if (this->comboBox_002->Text=="16byte")  page_size = 16;
      if (this->comboBox_002->Text=="32byte")  page_size = 32;
      if (this->comboBox_002->Text=="64byte")  page_size = 64;
      if (this->comboBox_002->Text=="128byte") page_size = 128;
      if (this->comboBox_002->Text=="256kBit") page_size = 256;
   }
}
private: System::Void comboBox_006_SelectedIndexChanged(System::Object^  sender,
System::EventArgs^  e) {
// Update the page size variable when SPI is selected
//
   if ((this->comboBox_000->Text=="SPI EEPROM")||(this->comboBox_000->Text=="SPI Flash 25xx")) {
      if (this->comboBox_006->Text=="4byte")   page_size = 4;
      if (this->comboBox_006->Text=="8byte")   page_size = 8;
      if (this->comboBox_006->Text=="16byte")  page_size = 16;
      if (this->comboBox_006->Text=="32byte")  page_size = 32;
      if (this->comboBox_006->Text=="64byte")  page_size = 64;
      if (this->comboBox_006->Text=="128byte") page_size = 128;
      if (this->comboBox_006->Text=="256kBit") page_size = 256;
   }
}
```

These two functions re-calculate clock frequency in the case there has been an update to the clock frequency pull-down menu. One function is provided for I2C and one for the SPI case.

```
private: System::Void comboBox_003_SelectedIndexChanged(System::Object^  sender,
System::EventArgs^  e) {
// Update the clock frequency for IIC EEPROMs
//
   if (this->comboBox_003->Text=="10kHz")   iic_frequency=NANO_I2C_FREQ_10KHZ;
   if (this->comboBox_003->Text=="STD")     iic_frequency=NANO_I2C_FREQ_STD;
   if (this->comboBox_003->Text=="200kHz")  iic_frequency=NANO_I2C_FREQ_200KHz;
   if (this->comboBox_003->Text=="FAST")    iic_frequency=NANO_I2C_FREQ_FAST;
   if (this->comboBox_003->Text=="1MHz")    iic_frequency=NANO_I2C_FREQ_1MHZ;
   if (this->comboBox_003->Text=="3MHz")    iic_frequency=NANO_I2C_FREQ_3MHZ;
   if (this->comboBox_003->Text=="6MHz")    iic_frequency=NANO_I2C_FREQ_6MHZ;
}

private: System::Void comboBox_007_SelectedIndexChanged(System::Object^  sender,
System::EventArgs^  e) {
// Update the clock frequency for SPI EEPROMs
//
   if (this->comboBox_007->Text=="10kHz")    spi_frequency=NANO_SPI_FREQ_10KHZ;
   if (this->comboBox_007->Text=="100kHz")   spi_frequency=NANO_SPI_FREQ_100KHZ;
   if (this->comboBox_007->Text=="200kHz")   spi_frequency=NANO_SPI_FREQ_200KHZ;
   if (this->comboBox_007->Text=="400kHz")   spi_frequency=NANO_SPI_FREQ_400KHZ;
   if (this->comboBox_007->Text=="1MHz")     spi_frequency=NANO_SPI_FREQ_1MHZ;
   if (this->comboBox_007->Text=="3MHz")     spi_frequency=NANO_SPI_FREQ_3MHZ;
   if (this->comboBox_007->Text=="6MHz")     spi_frequency=NANO_SPI_FREQ_6MHZ;
   if (this->comboBox_007->Text=="12MHz")    spi_frequency=NANO_SPI_FREQ_12MHZ;
}
```

These two functions re-calculate the SPI clock polarity and phase in the case there was a change to the polarity or phase pull-down menu respectively.

```
private: System::Void comboBox_008_SelectedIndexChanged(System::Object^  sender,
System::EventArgs^  e) {
// Update the polarity setting for SPI EEPROMs/Flash
//
   if (this->comboBox_008->Text=="0")    spi_polarity=0;
   if (this->comboBox_008->Text=="1")    spi_polarity=1;
}

private: System::Void comboBox_009_SelectedIndexChanged(System::Object^  sender,
System::EventArgs^  e) {
// Update the phase setting for SPI EEPROMs/Flash
//
   if (this->comboBox_009->Text=="0")    spi_phase=0;
   if (this->comboBox_009->Text=="1")    spi_phase=1;
}
```

This function is provided for when there is a click on the "Update" button. In this case the address and new data is used to update one locate in the programmer memory. A screen update is made to make sure the user sees this too.

```
private: System::Void button_002_Click(System::Object^  sender, System::EventArgs^  e) {
// This function is for the Update button click detection.
// When the button is clicked then the corresponding memory location is updated.
//
   Int32 updateAddress;
   Int32 updateData;
   updateAddress = 0xFFFFFF & Int32::Parse(this->
      textBox_003->Text,System::Globalization::NumberStyles::HexNumber);
   updateData = 0xFF & Int32::Parse(this->
      textBox_004->Text,System::Globalization::NumberStyles::HexNumber);
   memory[updateAddress] = updateData;
   ScreenUpdate();
   s2="Updated location ";
   s2 = System::String::Concat( s2,updateAddress.ToString("X4"));
   s2 = System::String::Concat( s2," with ");
   s2 = System::String::Concat( s2,updateData.ToString("X2"));
   s2 = System::String::Concat( s2,".");
   this->richTextBox_002->Text = s2;
}
```

This function is provided for erasing SPI flash devices it also writes 0xFF to EEPROMs. For full details see the source code. For speed improvements the functions use the function Nano_SPIMasterReadWrite4() function since not all contents of the in_buffer need to be controlled; at most 4 bytes.

```
private: System::Void button_008_Click(System::Object^  sender, System::EventArgs^  e) {
// Inplements the click activity for the ERASE button.
// Implements both IIC and SPI behaviour.
//
…
```

This function is provided for writing the image to the physical device. The function handles I2C EEPROMs, SPI EEPROMs and SPI flash devices. For full details see the source code.

```
private: System::Void button_003_Click(System::Object^  sender, System::EventArgs^  e) {
// Inplements the click activity for the WRITE button.
// Implements both IIC and SPI behaviour.
//
…
```

This function is provided for reading the physical device into the programmer image. For full details see the source code.

```
private: System::Void button_005_Click(System::Object^  sender, System::EventArgs^  e) {
// Inplements the click activity for the READ button.
// Implements both IIC and SPI behaviour.
//
…
```

This function is provided for verifying the image that got written is the same as the programmer image. For full details see the source code.

```
private: System::Void button_004_Click(System::Object^  sender, System::EventArgs^  e) {
// Inplements the click activity for the VERIFY button.
// Implements both IIC and SPI behaviour.
//
…
```

To follow is a function to read in an intel hex format file, parse it and then use this as the programmer image. For full details see the source code.

```
private: System::Void button_000_Click(System::Object^  sender, System::EventArgs^  e) {
// This is a simple parser for Intel HEX files.
// At this point the parser is limited to record types 00 and 01.
// Also the file must include a RETURN at the on the last line.
```

To follow is a function to take the programmer image, convert this to an intel hex file and then write the file to disk. For full details see the source code.

```
private: System::Void button_001_Click(System::Object^  sender, System::EventArgs^  e) {
// Function to handle writing out the memory to file.
//
```

Two functions are provided to fill the programmer image 0xFF or with pseudo random numbers respectively. Upon completion the image window is updated.

```
private: System::Void button_006_Click(System::Object^  sender, System::EventArgs^  e) {
// Function used to update all memory locations with random values
//
    srand(0);
    for (j=0; j<(chip_size); j++) {
       memory[j] = rand() & 0xFF;
    }
    ScreenUpdate();
}


private: System::Void button_007_Click(System::Object^  sender, System::EventArgs^  e) {
// Function used to update all memory locations with FFs
//
    for (j=0; j<(chip_size); j++) {
       memory[j] = 0xFF;
    }
    ScreenUpdate();
}
```

Some important links and includes are necessary in stdafx.h. This is best seen by viewing the file in the download from the web-site.

8. Configuration Example (Linux and MacOS Versions)

This is a very simple example showing how to call basic functions associated with SPI, I2C, GPIO and analogue IO. The example is a good starting point if you want to try the ViperBoard for the first time and want to see a simple console application. The example does not show all possible configuration options. It is left to the user to read the API specification to see how to make more advanced applications. This example is the same as the one described in section 2 but ported for Linux and MacOS.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++ NANO RIVER TECHNOLOGIES                                      ++
++   CONFIGURATION EXAMPLE FOR VIPERBOARD (05 Nov 2009)         ++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Open Device....
   -> ViperBoard Connected!!!
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++
 GPIO A Test
++++++++++++++++++++++
 Write AA

 Write 55

 Make bit 0 pulsed
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++
 GPIO B Test
++++++++++++++++++++++
 Write AA

 Write 55
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++
 Analogue Input Test
++++++++++++++++++++++
Analogue Channel #00 : FF
Analogue Channel #01 : FF
Analogue Channel #02 : FF
Analogue Channel #03 : FF
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++
 SPI Test
++++++++++++++++++++++
SPI Configure Channel 0...
SPI Set Frequency ...

SPI Slave Data (Written)  : AABBCCDD
SPI Master Data (To send) : 11223344

SPI Master Send Channel 0...

SPI Master Data (Read)    : AABBCCDD
SPI Slave Data (Read)     : 11223344

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++
 I2C Test
++++++++++++++++++++++
I2C Devices Connected ... 0x2B
Master I2C write...
Master Buffer   : AABBCCDDFF
Slave Buffer    : AABBCCDDFF

Master I2C read...
Slave Buffer    : 1122334455
Master Buffer   : 1122334455


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

8.1. Functionality

The example starts by testing GPIO Port A. This is the advanced GPIOs that can be programmed as PWM, pulsed, digital IO or interrupt input. In this example we write an AA pattern then a 55 pattern. Bit 0 is then pulsed continuously.

Next, GPIO Port B is tested. This port is simple digital IO. The port is tested by writing an AA then 55 pattern.

The analogue inputs are then tested. A read is made of each of the input channels.

Next to be tested is the SPI interface. This interface consists of a master and slave SPI interface. In the example we expect that master is LOOPED BACK to the slave. Since master and slave can work independently the example does a simple swap of data between master and slave and in full duplex. Data originally in the master is transferred to the slave and data originally in the slave is transferred to the master by the end of a single transaction.

Finally the I2C interface is demonstrated. Again we have master and slave I2C interfaces to test. Both master and slave are connected to the same pins on ViperBoard so by arming the slave it is possible to do transactions between master and slave with NO external connection. The first exercise is to perform a scan of all devices connected on the I2C bus. If the I2C slave is armed then its device ID will appear in the connected device list. I2C master write and I2C master read also follow. If the I2C is armed then we can get data transfer between the master and slave.

8.2. Project Structure

This application is located in the Examples/ directory of the Linux and MacOS installation. The project file structure is implemented as follows.

```
Examples/
├── common/
│   ├── viperboard.h         <-- ViperBoard Header File
│   ├── viperboard_class.h   <-- ViperBoard Class Header File
│   └── viperboard_class.c   <-- ViperBoard Class C++ File
├── src/
│   └── ConfigurationExample.c   <-- Main Program
└── scripts/
    └── go_ConfigurationExample  <-- Compile Script for Application
```

The ViperBoard user interface is a simple class library of high level functions to control the board as I2C/SPI/GPIO/AnalogueIO or some combination. The class library is provided through `viperboard_class.h` and `viperboard_class.c`. The functions make simple calls to LibUsb OpenSource USB driver already assumed to have been installed as part of the installation process.

In this example there is one main program `ConfigurationExample.c`. This program then makes calls to the class library to control in this case the I2C/SPI/GPIO/AnalogueIO interfaces.

A simple compile script has been provided to show exactly how to compile the application. This is `go_ConfigurationExample`.

To use this application, first navigate to the Examples/ directory.

```
shell%>  cd <installation_directory>/Examples
```

Next compile the application using the provided compile script. The compile script will use G++ for the compile.

```
shell%>  ./scripts/go_ConfigurationExample
```

Finally run the application.

```
shell%>  ./ConfigurationExample
```

You should then see data written to and read from the devices as shown earlier in this chapter.

8.3. Design Description

The application is implemented using a single file `ConfigurationExample.c`.

The file starts by linking to the ViperBoard class library header file:

```
/////////////////////////////////////////////////////////////////////////////
//
// Nano River Technologies
//
// File:         Configuration Example (ConfigurationExample.c)
//
// Desciption:   This is a console application to do basic SPI, I2C,
//               GPIO and analogue input functions with ViperBoard. The functions
//               used is just a sub-set of what is possible, but provides the way
//               to interface to ViperBoard from a console application.
//
// Revision:     Version 1.0
//
/////////////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include "../common/viperboard.h"
#include "../common/viperboard_class.h"
```

The file declares some global variables used in the application:

```
// Global Variables
//
BOOL        connected;              // True if the ViperBoard is connected
NANO_RESULT res;                    // Result of a ViperBoard function
VBc         VB;                     // Viperboard class library
BYTE        ADC_0;                  // ADC data read - channel 0
BYTE        ADC_1;                  // ADC data read - channel 1
BYTE        ADC_2;                  // ADC data read - channel 2
BYTE        ADC_3;                  // ADC data read - channel 3
BYTE        InBuffer[256];          // I2C input buffer
BYTE        OutBuffer[256];         // I2C output buffer
DEV_LIST    lst;                    // I2C device list
BYTE        in_buffer_m[512];       // SPI master buffer in
BYTE        out_buffer_m[512];      // SPI master buffer out
BYTE        in_buffer_s[512];       // SPI slave buffer in
BYTE        out_buffer_s[512];      // SPI slave buffer out
BOOL        IICMasterError;         // I2C master error
char        c;                      // Local character
int         i;                      // Loop variable
```

The main program starts by printing out a startup banner and calling the user instructions:

```
int main(void)
{

    printf("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");
    printf("++ NANO RIVER TECHNOLOGIES                                     ++\n");
    printf("++   CONFIGURATION EXAMPLE FOR VIPERBOARD (05 Nov 2009)         ++\n");
    printf("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");
    printf("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");
    printf("Open Device....\n");
```

ViperBoard is then initialised by calling the *OpenDevice()* in the ViperBoard class.

```
    connected=VB.OpenDevice();
    if (connected)
        printf ("   -> ViperBoard Connected!!!\n");
    else
    printf ("   -> ViperBoard NOT Connected!!!\n");
    printf("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");
    c=getchar();
```

The program first initialises all GPIOs as digital outputs and writes 0 to them (*Nano_GPIOASetDigitalOutputMode(), Nano_GPIOBSetDirection() and Nano_GPIOBWrite()).* The SPI and I2C events are then flushed by disarming the SPI and I2C slaves (*Nano_SPIArm()* and *Nano_I2CArm()).*

```
    if (connected) {

        // Initialise the GPIOs and flush any events
        printf(" Initialisation\n");
        for (i=0;i<16;i++)
            res=VB.Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,i,false);
        res=VB.Nano_GPIOBSetDirection(VB.vb_hDevice,0xFFFF,0xFFFF);
        res=VB.Nano_GPIOBWrite(VB.vb_hDevice,0x0000,0xFFFF);
        res=VB.Nano_I2CSlaveArm(VB.vb_hDevice,false);
        res=VB.Nano_SPISlaveArm(VB.vb_hDevice,false);
        printf("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");
        c=getchar();
```

GPIO port A is tested by writing an AA then 55 pattern using *Nano_GPIOASetDigitalOutputMode().* Bit 0 is then set into continuous pulse mode using *Nano_GPIOASetContinuousMode().*

```
//////////////////////////////////
// GPIO Port A Test
//////////////////////////////////
printf("+++++++++++++++++++++\n");
printf(" GPIO A Test\n");
printf("+++++++++++++++++++++\n");
printf(" Write AA\n");
// Write AA
for (i=0;i<8;i++) {
    res=VB.Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,2*i,false);
    res=VB.Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,2*i+1,true);
}
c=getchar();
// Write 55
printf(" Write 55\n");
for (i=0;i<8;i++) {
    res=VB.Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,2*i,true);
    res=VB.Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,2*i+1,false);
}
c=getchar();
// Continuous Pulsed
printf(" Make bit 0 pulsed\n");
for (i=0;i<16;i++)
    res=VB.Nano_GPIOASetDigitalOutputMode(VB.vb_hDevice,i,false);
e = VB.Nano_GPIOASetContinuousMode(VB.vb_hDevice,0,1,0x40,0x80);
printf("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");
c=getchar();
```

GPIO port B is tested also by writing an AA then 55 pattern using *Nano_GPIOBSetDirection()* and *Nano_GPIOBWrite().*

```
//////////////////////////////////
// GPIO Port B Test
//////////////////////////////////
printf("+++++++++++++++++++++\n");
printf(" GPIO B Test\n");
printf("+++++++++++++++++++++\n");
printf(" Write AA\n");
// Write AA
res=VB.Nano_GPIOBSetDirection(VB.vb_hDevice,0xFFFF,0xFFFF);
res=VB.Nano_GPIOBWrite(VB.vb_hDevice,0xAAAA,0xFFFF);
c=getchar();
printf(" Write 55\n");
// Write 55
res=VB.Nano_GPIOBSetDirection(VB.vb_hDevice,0xFFFF,0xFFFF);
res=VB.Nano_GPIOBWrite(VB.vb_hDevice,0x5555,0xFFFF);
printf("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");
c=getchar();
```

Analogue inputs are read using *Nano_ADCRead()*.

```
    ///////////////////////////////////
    // ADC Test
    ///////////////////////////////////
    printf("++++++++++++++++++++++\n");
    printf(" Analogue Input Test\n");
    printf("++++++++++++++++++++++\n");
    res=VB.Nano_ADCRead(VB.vb_hDevice,0x00,&ADC_0);
    res=VB.Nano_ADCRead(VB.vb_hDevice,0x01,&ADC_1);
    res=VB.Nano_ADCRead(VB.vb_hDevice,0x02,&ADC_2);
    res=VB.Nano_ADCRead(VB.vb_hDevice,0x03,&ADC_3);
    printf("Analogue Channel #00 : %02X\n",ADC_0);
    printf("Analogue Channel #01 : %02X\n",ADC_1);
    printf("Analogue Channel #02 : %02X\n",ADC_2);
    printf("Analogue Channel #03 : %02X\n",ADC_3);
    printf("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");
    c=getchar();
```

The SPI master and slave is tested next. First the SPI master channel 0 is configured as active low chip select, CPOL=0 and CPHA=0 using *Nano_SPIConfigure()*. The SPI master frequency is set for 12MHz line rate using *Nano_SPIMasterSetFrequency()*. The slave channel 0 buffer is filled with 0xAABBCCDD using *Nano_SPISlaveBufferWrite()*. The slave is then armed with *Nano_SPISlaveArm()*. An SPI read/write master transfer is then made with 0x11223344 as data using *Nano_SPIMasterReadWrite()*.The slave buffer is then read using *Nano_SPISlaveBufferRead()*. The master buffer should then contain 0x11223344 and the slave should contain 0xAABBCCDD assuming the master and slave are looped back.

```
    ///////////////////////////////////
    // SPI Test
    ///////////////////////////////////
    printf("++++++++++++++++++++++\n");
    printf(" SPI Test\n");
    printf("++++++++++++++++++++++\n");

    // Configure the SPI in terms of CPOL/CPHA/CSn
    printf("SPI Configure Channel 0...\n");
    res=VB.Nano_SPIConfigure(VB.vb_hDevice,0,0,false,0,0);

    // Set the frequency
    printf("SPI Set Frequency ...\n\n");
    res=VB.Nano_SPIMasterSetFrequency(VB.vb_hDevice,NANO_SPI_FREQ_12MHZ);

    // Fill the slave with data
    out_buffer_s[0] = 0xAA;
    out_buffer_s[1] = 0xBB;
    out_buffer_s[2] = 0xCC;
    out_buffer_s[3] = 0xDD;
    res=VB.Nano_SPISlaveBufferWrite(VB.vb_hDevice,0,4,out_buffer_s);
    printf("SPI Slave Data (Written)  : %02X%02X%02X%02X\n",
        out_buffer_s[0],out_buffer_s[1],out_buffer_s[2],out_buffer_s[3]);
```

```
// Arm the slave
res=VB.Nano_SPISlaveArm(VB.vb_hDevice,true);

// Perform a duplex read/write on the master
out_buffer_m[0] = 0x11;
out_buffer_m[1] = 0x22;
out_buffer_m[2] = 0x33;
out_buffer_m[3] = 0x44;
printf("SPI Master Data (To send) : %02X%02X%02X%02X\n",
    out_buffer_m[0],out_buffer_m[1],out_buffer_m[2],out_buffer_m[3]);
printf("\nSPI Master Send Channel 0...\n\n");
res=VB.Nano_SPIMasterReadWrite(VB.vb_hDevice,0,4,in_buffer_m,out_buffer_m);

// Present the read data from the master
printf("SPI Master Data (Read)     : %02X%02X%02X%02X\n",
    in_buffer_m[0],in_buffer_m[1],in_buffer_m[2],in_buffer_m[3]);

// Present the received data at the slave
res=VB.Nano_SPISlaveBufferRead(VB.vb_hDevice,0,4,in_buffer_s);
printf("SPI Slave Data (Read)      : %02X%02X%02X%02X\n",
    in_buffer_s[0],in_buffer_s[1],in_buffer_s[2],in_buffer_s[3]);

c=getchar();
printf("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");
```

The I2C is then tested, firstly using an I2C scan test. The test configures the slave to have a device ID of 0x2B using *Nano_I2CSlaveConfig().* The slave is then armed using *Nano_I2CSlaveArm().* The line rate of the master is then set to 6MHz using *Nano_I2CMasterSetFrequency().* Finally we call the I2C scan function to find all devices connected using *Nano_I2CMasterScanConnectedDevices().*

```
printf("++++++++++++++++++++++\n");
printf(" I2C Test\n");
printf("++++++++++++++++++++++\n");

///////////////////////////////////
// I2C Scan Test
///////////////////////////////////

// Arm again
res=VB.Nano_I2CSlaveConfig(VB.vb_hDevice,0x2B);
res=VB.Nano_I2CSlaveArm(VB.vb_hDevice,true);

// Set the line rate
res=VB.Nano_I2CMasterSetFrequency(VB.vb_hDevice,NANO_I2C_FREQ_6MHZ);

// Scan all I2C devices
res=VB.Nano_I2CMasterScanConnectedDevices(VB.vb_hDevice,&lst);
printf("I2C Devices Connected ... ");
for (i=0;i<128;i++) {
    if (lst.List[i]) {printf("0x%02X ",i); }
}
printf("\n");
```

The slave and master are configured as for the I2C scan testing in terms of device ID and line rate. An I2C master write is then performed with 0xAABBCCDDFF as the output data using *Nano_I2CMasterWrite()*. The data is then read from the slave using *Nano_I2CSlaveBuffer1Read()*. It should be 0xAABBCCDDFF also.

```
///////////////////////////////////
// I2C Write Test
///////////////////////////////////

// Configure and arm the slave
res=VB.Nano_I2CSlaveConfig(VB.vb_hDevice,0x2B);
res=VB.Nano_I2CSlaveArm(VB.vb_hDevice,true);

// Set the line rate
res=VB.Nano_I2CMasterSetFrequency(VB.vb_hDevice,NANO_I2C_FREQ_6MHZ);

// Perform I2C write
OutBuffer[0] = 0xAA;
OutBuffer[1] = 0xBB;
OutBuffer[2] = 0xCC;
OutBuffer[3] = 0xDD;
OutBuffer[4] = 0xFF;
res=VB.Nano_I2CMasterWrite (VB.vb_hDevice,0x2B,5,OutBuffer);
printf("Master I2C write...\n");

// Check Data
res=VB.Nano_I2CSlaveBuffer1Read(VB.vb_hDevice,5,InBuffer);
printf("Master Buffer   : %02X%02X%02X%02X%02X\n",OutBuffer[0],
    OutBuffer[1],OutBuffer[2],OutBuffer[3],OutBuffer[4]    );
printf("Slave Buffer    : %02X%02X%02X%02X%02X\n\n",InBuffer[0],
    InBuffer[1],InBuffer[2],InBuffer[3],InBuffer[4]);
```

The slave and master are configured as for the I2C scan testing in terms of device ID and line rate. The I2C slave buffer is filled with 0x1122334455 using *Nano_I2CSlaveBuffer1Write()*. An I2C read is then performed using *Nano_I2CMasterRead()*. The data should be 0x1122334455.

```
///////////////////////////////////
// I2C Read Test
///////////////////////////////////

// Configure and arm the slave
res=VB.Nano_I2CSlaveConfig(VB.vb_hDevice,0x2B);
res=VB.Nano_I2CSlaveArm(VB.vb_hDevice,true);

// Set the line rate
res=VB.Nano_I2CMasterSetFrequency(VB.vb_hDevice,NANO_I2C_FREQ_6MHZ);

// Put data into the slave
OutBuffer[0] = 0x11;
OutBuffer[1] = 0x22;
OutBuffer[2] = 0x33;
OutBuffer[3] = 0x44;
OutBuffer[4] = 0x55;
res=VB.Nano_I2CSlaveBuffer1Write(VB.vb_hDevice,5,OutBuffer);
```

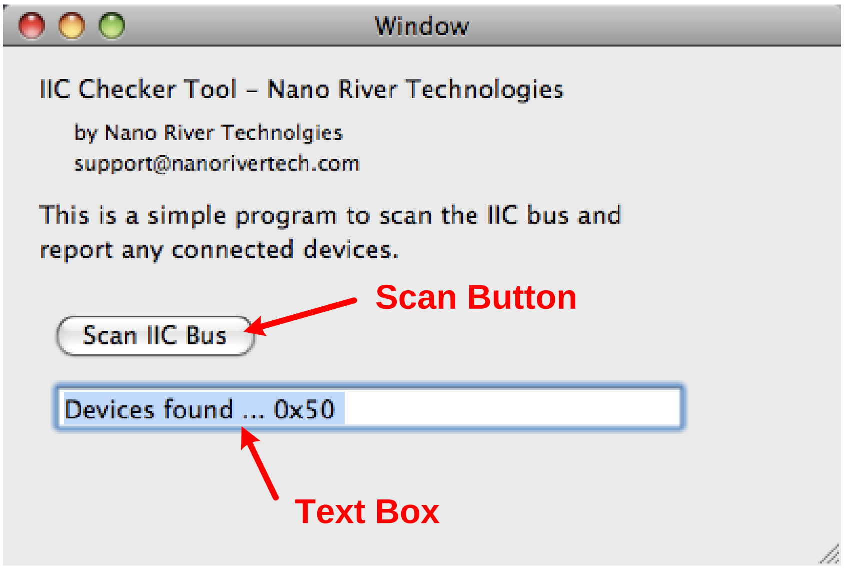
```
    // Perform I2C read
    res=VB.Nano_I2CMasterRead (VB.vb_hDevice,0x2B,5,InBuffer);
    printf("Master I2C read...\n");
    printf("Slave Buffer    : %02X%02X%02X%02X%02X\n",OutBuffer[0],
        OutBuffer[1],OutBuffer[2],OutBuffer[3],OutBuffer[4]    );
    printf("Master Buffer   : %02X%02X%02X%02X%02X\n\n",InBuffer[0],
        InBuffer[1],InBuffer[2],InBuffer[3],InBuffer[4]);


    printf("\n");
    printf("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");
```

Finally the main program completes.

```
    ///////////////////////////////////////////////////////////////////////
    // Finished ....
    ///////////////////////////////////////////////////////////////////////
    printf("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");
    printf(" TEST COMPLETE ....\n");
    printf("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");

    printf("\n");
    printf ("<<<<< PRESS RETURN KEY >>>>>>\n");
    c=getchar();

    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////

    return 0;
```

9. I2C Checker Tool (MacOS Xcode Cocoa)

This example application is a GUI application for MacOS using Xcode and Cocoa. The example shows how to use the ViperBoard from Cocoa applications to call I2C scan commands. Whilst the example is developed using MacOS 10.5 and Xcode 3.1 other versions should link in the same way.

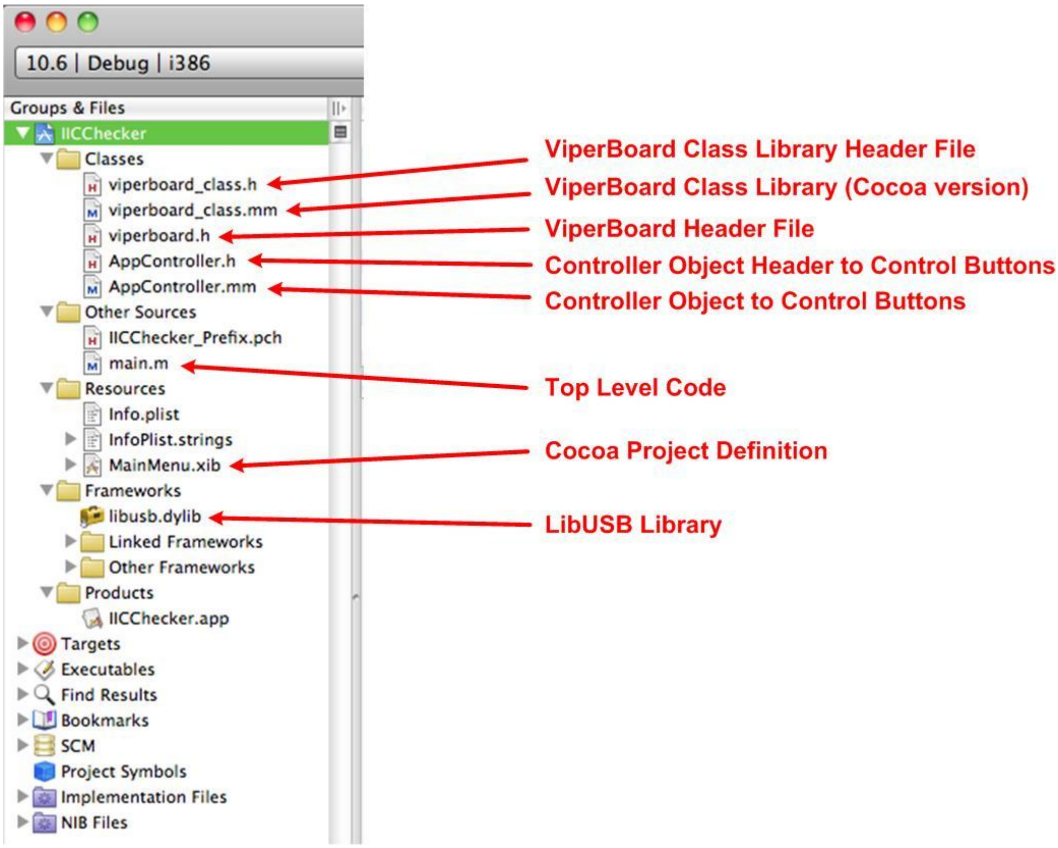


9.1. Functionality

In this application the user should just press the scan button. If devices are present then they will be listed in the textbox output. Missing ViperBoard connection and missing I2C devices are also messaged.

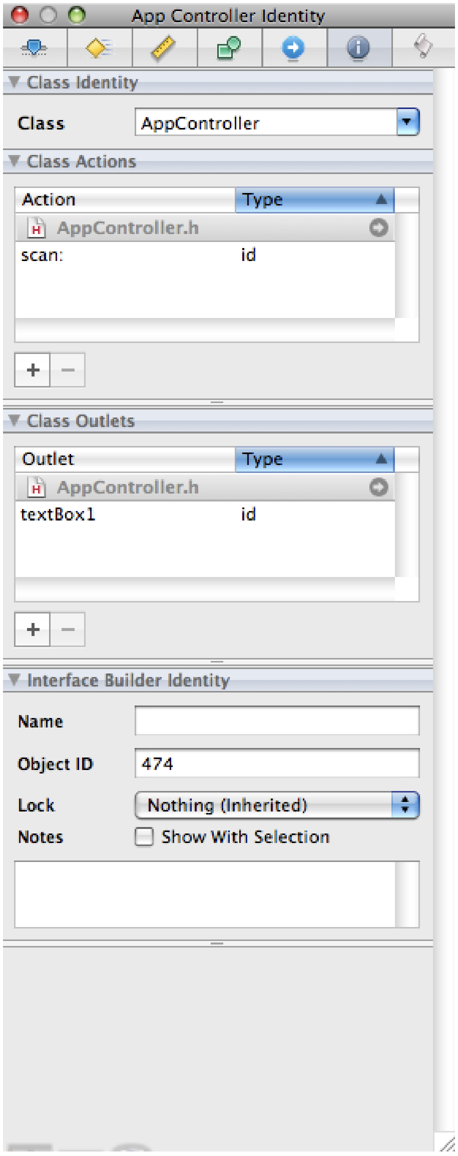
9.2. Project Structure

This project was created as a standard Cocoa application in Xcode. The project structure is shown below.

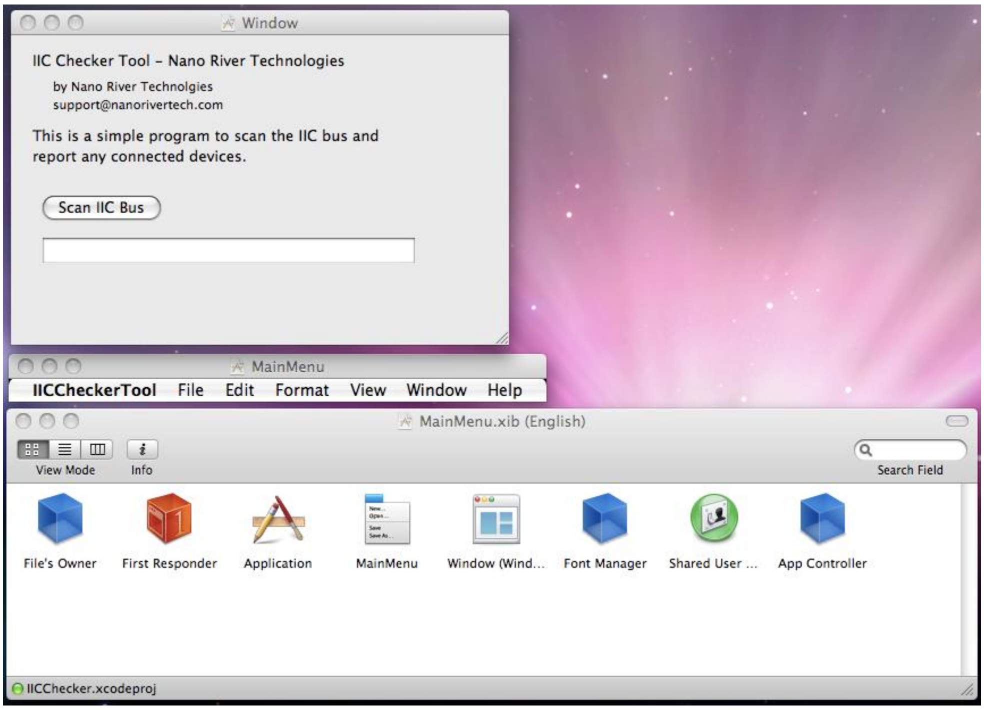


In the classes folder Cocoa versions of the ViperBoard Class Library and ViperBoard header file are placed. **Some modification was needed from the one used for GCC applications owing to the linking between standard C and Object C++ so please make sure you use these for Cocoa applications.**

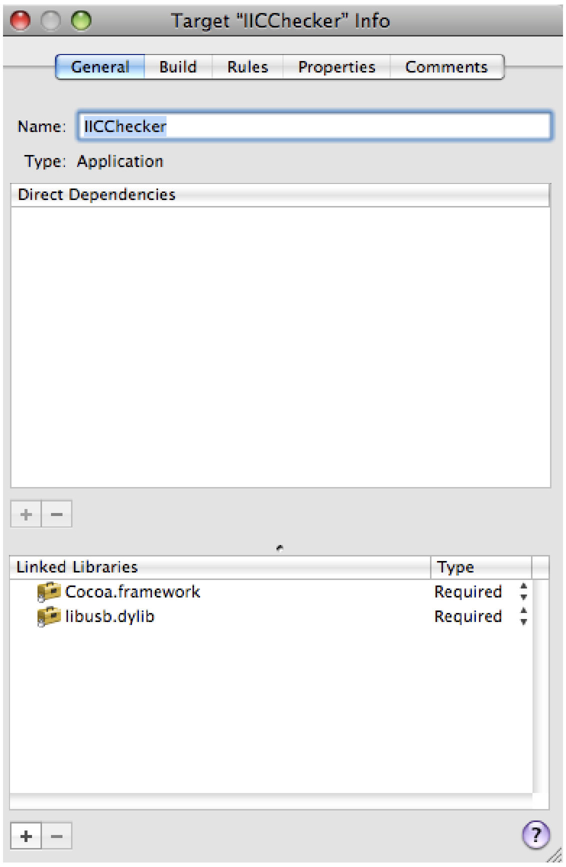
Also in the classes folder is an application controller object which has been built up and linked to button and textbox. It has an action called scan which is linked to the button and an outlet called textBox1 connected to the textbox. Coding for this will be discussed further on.



The user window has been built up graphically by dragging and dropping objects from the Cocoa library – for example labels, a button and the textbox.



Linking to the LibUSB dynamic library is achieved by highlighting IICChecker under targets and then calling the information browser. By selecting + then one can find libusb.dynlib to add – assuming that LibUSB has been installed before-hand.



9.3. Design Description

The main file which needs to be developed for this Cocoa application is the application controller `AppController.mm`. The skeleton of the file was written out by Xcode after the actions and outlets were defined. The rest of the file was written by hand.

The file starts by linking to the ViperBoard class library header file and the application controller header file:

```
///////////////////////////////////////////////////////////////////////////
//
// Nano River Technologies
//
// File:          AppController (AppController.mm)
//
// Desciption:    This is a controller object which detects SCAN button presses.
//                When the button press is detected then a scan command is sent
//                to ViperBoard. The list of IIC devices found is then printed
//                to a textbox.
//
// Revision:      Version 1.0
//
///////////////////////////////////////////////////////////////////////////

#import "AppController.h"
#import "viperboard_class.h"
```

Next various variables are defined.

```
NSString *displayOutput;            // A text string for the output to the screen
NSString *deviceNo;                 // An IIC device found to be connected
NANO_RESULT e;                      // Result of ViperBoard Commands (here a scan)
DEV_LIST    lst;                    // Returned list of connected IIC devices
BOOL d;                             // True if ViperBoard is connected
int i;                              // Integer for looping all IIC numbers
BOOL iic_devices_found;             // Flag which is true if ANY IIC device was connected
```

Finally the implementation of the application controller is made. In the action for the scan key press an instance for the class library is made. We then make a call to ViperBoard to open the device. If the ViperBoard was successfully connected then we make a call to the ViperBoard scan function. The results from the scan are used to create a string containing the addresses of the connected I2C devices. The string is written into the textbox.

```
@implementation AppController

//////////////////////////////////////////////////
// Action for when the SCAN button is pressed
//////////////////////////////////////////////////
- (IBAction)scan:(id)sender {

    iic_devices_found = false;
    VBc         VB;
    deviceNo = @"";
    displayOutput = @"ViperBoard not connected ...";
    d=VB.OpenDevice();

    // Branch in case the ViperBoard is connected
    if (d) {
        displayOutput = @"Devices found ... ";
        e=MB.Nano_I2CMasterScanConnectedDevices(VB.vb_hDevice,&lst);
                                    // The Nano River Tech IIC Scan command
        for (i=0;i<128;i++) {
            if (lst.List[i]) {
                deviceNo = [NSString stringWithFormat:@"0x%02x ", i];
                displayOutput = [displayOutput stringByAppendingString:deviceNo];
                iic_devices_found = true;
            }
        }
        // Branch in case there were no IIC devices found
        if (iic_devices_found==false)
            displayOutput = @"No devices found!";
    }

    // Write out the results
    [textBox1 setStringValue:displayOutput];

}
@end
```