Mechatronic Systems Design 431

GUI Development Reader

Written By: Tom Stewart

Date: 24th February, 2011

\*\*Before looking through this document; read through the milestones document make sure you understand the basic structure and outcomes of the unit’s laboratory component.

# Background

Given the prerequisite above, you should be aware that an important part of the laboratory component of this unit is the development of an application to interact with both a User, via a Graphical User Interface and some hardware via the I/O card.

You are free to accomplish this through any means you choose, this being a 4th unit we encourage you to use your engineering judgement to identify the best “how” to meet the objectives of the project. We do however, have recommendations for what we consider to be the best way to carry out the project based on both the tools available to you and our experience with the unit.

The recommended development language has previously been C in conjunction with the Allegro Library. Allegro is a game programming library that allows (among other things) drawing basic 2D graphics to the screen as well as some mouse and keyboard interactions.

The advantage of this approach was that C is a language that we are all very familiar with and after a little investigation of Allegro’s API we could quickly begin developing a GUI. The drawback of this was that Allegro is a very low-level library and to do even a simple task such as positioning a textbox required a substantial amount of time and code.

Instead, we now recommend C# in conjunction with the .NET framework. If you have come straight from Mechatronic Project 331 you will have had some experience with C# in a Console Application setting. By all accounts C# is a very developer friendly language, it takes care of its own memory management and its syntax is very similar to C. Its access to the .NET framework (Windows’ collection of libraries) makes an absolute enormity of tasks possible in only a few lines of code. It is well supported and there are an abundance of references with sample code if ever you get stuck.

There is a drawback with C# in that it adopts a number of programming conventions that you may not be familiar with. As such there is a somewhat steep learning curve before your application can get up and running.

So to address this learning curve, this guide will cover the basic concepts and fundamentals that you are likely to come into contact with during the implementation of your program. Again, this being a 4th year unit we will expect you to exercise your self-directed learning skills in order to investigate these concepts further. The emphasis here will be to give you a conceptual understanding and to point you in the direction of more in-depth material.

# Object Oriented Programming

Object Oriented Programming (OOP) is a very well developed concept and learning it down to the last detail will probably require more time than is available to you in this unit. That said, C# is an object oriented language and this paradigm underpins the .NET framework so understanding the terminology of classes, objects, methods and members etc will be vital to working with the language and its resources.

To best approach OOP, let’s first consider the structure of a non-OOP or ‘procedural’ approach (written in C). In general, one of our first moves is to declare a number of variables to store the data we need to work with later in the program. From there, we use a long list of commands to operate on those variables to produce whatever functionality we need from the program.

To improve the readability of the code we might group data pertaining to a common item into a struct. For example:

float height;

float width;

float depth;

float density;

Becomes:

typedef struct //declare the Box type

{

float height;

float width;

float depth;

float density;

} Box;

Box aBox; //declare a variable of type Box to work with later on

We might also group commands that perform a specific task into functions, for example:

float volume = aBox.width \* aBox.height \* aBox.depth;

float mass = volume \* aBox.density;

Becomes:

float calculateMass(Box aBox)

{

float volume = aBox.width \* aBox.height \* aBox.depth;

float mass = volume \* aBox.density;

return mass;

}

The central theme here is abstraction; the type Box is an abstract representation of a group variables that describe a box. Similarly, calculateMass(Box aBox) allows us to calculate the mass of a box without having to worry about the exact details of how this is carried out.

The Object Oriented paradigm simply takes this abstraction a step further by encapsulating related data and functions into a single unit, an object. You can think of an object in a similar manner that you would any other variable, but with a few extra features and semantics.

Consider the following; a declaration of a type of object, Box implemented in C#:

class Box

{

private float height;

private float width; //private members

private float depth;

public float density; //a public member

public float calculateMass() //a method

{

float volume = width \* height \* depth;

float mass = volume \* density;

return mass;

}

public Box(float dimension, float \_density)// the constructor

{

height = dimension;

width = dimension; //boxes are cubes by default

depth = dimension;

density = \_density; //with the specified density

}

}

Some definitions then:

* An object’s type is called its class
* An object is said to be an instance of a class
* Accordingly, the creation of an object is referred to as its instantiation
* A function inside an object ifs called a method
* Calling a method is often referred to as messaging it
* A variable inside an object is called a member (you might also see them referred to as fields)

All classes in C# will contain a special method known as a constuctor. The constructor is called when the object is instantiated and is used to do any custom initialisation that may be required before it is used.

To put this in perspective, if the services of a Box were required in code then we could declare an object of that type using:

//declare an object of type Box to work with later on

Box aBox = new Box(1.2,7800);

The code before the = sign should be familiar, we are telling the compiler that aBox will be an object of the Box class. The new keyword acts the same as malloc in C and serves to allocate memory for the instance of the class. The Box(1.2,7800) part is a call to the Box object’s special method, it’s constructor.

Having instantiated a Box object, aBox we can access its methods and members using the dot operator, for example:

float howDense = aBox.density;

float howHeavy = aBox.calculateMass();

Be aware though, methods and members can have varying levels of accessibility which are determined by the keyword used before their declaration. A private method or member can only be accessed within its own class (by its methods). A public method or member can be accessed anywhere.

The convention is that all members be private and all methods be public, with a few exceptions. Internal methods that provide specific functionality to other methods in the class have no need to be advertised to external elements and should be private. Conversely, circumstances are likely to arise where certain members need to be accessed outside the scope of their own class. In this case we recommend the use of get & set methods shown below rather than widening their visibility to public. Getters and setters, also called accessor methods are particularly useful for when additional code needs to be executed to check or display the newly modified variable.

//you actually don’t have to specify private, it’s the default for members

private double aDouble; //your private member

public double ANameForYourGetterOrSetter

{

get

{

// <--some additional code to record the request of aDouble?

return aDouble;

}

set

{

//<--some additional code to validate or display the new value?

aDouble = value;

}

}

Accessibility is intended to prevent variables from being accidentally modified and inadvertently introducing bugs into your code. This notion, referred to as encapsulation is one of the main benefits of object oriented programming over the procedural paradigm.

If you’ve understood this section then you should have a working knowledge of Object Oriented Programming. There are a number of other points worth investigating such as inheritance and polymorphism so for more in depth explanations refer to the following:

<http://msdn.microsoft.com/en-us/library/dd460654.aspx> - Microsoft Developers Network, if you’ve not come across it already, bookmark it. It’s a very reliable source for all things C#

We encourage you to observe how this has been used in almost every facet of the .NET framework and try to incorporate it into your code. It’s a conceptual leap at first but once you become familiar with the concept you’ll notice how it enhances the readability and stability of your application.

As a final note, identifying what objects you are likely to need in your application is not an easy task. In this case they might be particularly useful for representing aspects of your physical hardware in code. Some examples might be:

* A HCTL
* A Motor
* An I/O Card (you might already be familiar with a class of this name)

In a more general sense they are useful in areas where a specific functionality is required, such as:

* A Timer
* A PID Controller
* A Graph
* A Low-Pass Filter

If you refer to Appendix A: Clock.cs you will find the complete code for a clock class that might be useful for timing the dynamics of your system.

# C#

With a basic understanding of OOP and given your past experience with either C# itself or C you likely already have a working understanding of the syntax and constructs at play. As such we’re not going to give a step by step description of the language here, if you feel uncomfortable or lost with C# you’d benefit from taking the time to revisit some fundamentals. The following are good places to do so:

<http://msdn.microsoft.com/en-us/library/618ayhy6%28v=VS.71%29.aspx>

<http://www.programmersheaven.com/2/CSharpBook>

Instead, let’s investigate a few of C#’s higher level features that you might find useful in the development of your application.

Any statements preceded by the # symbol denote preprocessor directives. Preprocessor directives relate not to your application but to its code. More specifically, these statements issue instructions to the preprocessor about how to modify your code before it is converted to instructions by the compiler.

The #if directive is particularly useful for separating code that you would like to execute during debugging versus when you are online, the structure of this is:

#if DEBUG

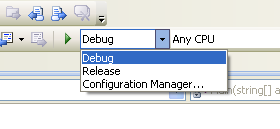
//some code

#else

//some alternate code

#endif

DEBUG is defined or not defined automatically by Visual Studio by selecting your Solution Configurations from the standard toolbar as such:



To provide a less than subtle hint, an example of this would be when you interface with your I/O card. If you plan to work on your code outside of the labs you’ll want to direct the compiler away from the import of "inpout32.dll" which is unlikely to exist on that computer and more than likely to generate a runtime error. You can try to remember to comment and uncomment the appropriate code or you can let the preprocessor do it for you via the #if directive. You’ll note by reading through the “iocard.cs” source code that this measure is already in place; nevertheless it is an important point for you to be aware of.

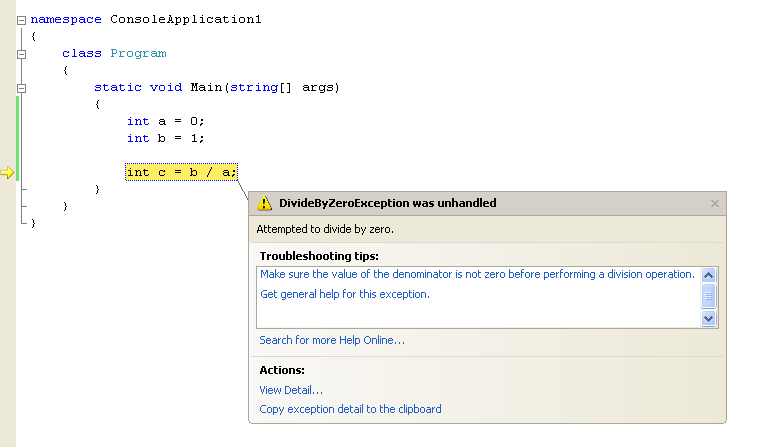
On a more cosmetic level, the #region directive is an elegant alternative to the following for separating regions of code.

/////////////////////////NAME OF SECTION OF CODE///////////////////////////

By entering #region NAME OF SECTION OF CODE and an associated #endregion you can minimise that region into an innocuous  in the text editor.

C# provides a means of handling exceptions (runtime errors). The notion of a runtime error should be largely self explanatory but put simply, they are errors that occur during the operation of your code. An example you may already be familiar with is a segmentation fault brought about by trying to access elements outside the bounds of an array.

Any uncaught exceptions brought about by your code will halt its execution and bring up a notification similar to the one below:



To avoid this you can use the try catch construct and attempt to handle these errors in code. An example of its usage is as follows:

try

{

//try this code

int a = 0;

int b = 1;

int c = b / a;

}

catch (DivideByZeroException ex)

{

//if it throws a DivideByZeroException exception, then handle it here

MessageBox.Show("Invalid arguments, please re-specify");

}

This construct is particularly useful for protecting your program from potentially invalid or unexpected inputs from the user. Before attempting to parse a string from a textbox (double.Parse(aTextBox.Text)) ensure that you give your program the means to recover from a FormatException in the event that the user specifies for example, a nonsense position setpoint: “fdfte”.

Exception handling is a powerful tool, but has the tendency to be misused. For an in-depth guide of best practices in exception handling refer to the following:

<http://www.codeproject.com/KB/architecture/exceptionbestpractices.aspx>

As a final note, arrays are supported in C# the syntax for instantiating one (of type int) is:

int[] newArray = new int[LENGTH];

Where LENGTH is an int whose value indicates the number of elements in the array. In situations where the size of the array isn’t known at compile time, consider using the List<> class instead. A list (of type int) can be instantiated as such:

List<int> newList = new List<int>();

A list can be used in much the same way an array would but offers much more flexibility and includes methods to operate on its elements to sort, append, remove or search them, as well as all the methods for averaging, finding the maximum etc provided by the array class. You may at some point in the development of your application encounter the need for a moving-average filter, the use of a List<> data structure might prove useful at this point.

# Using Visual Studio 2008

With the fundamentals of C# covered we can now look at a C# application with a more informed outlook. Since your end goal is an application that incorporates a GUI you’ll want to create a project based on the forms application template.

To do that, in Visual Studio select New->Project and select “Windows Forms Application” under “Visual C#”. Doing this will result in a screen similar to that shown below.

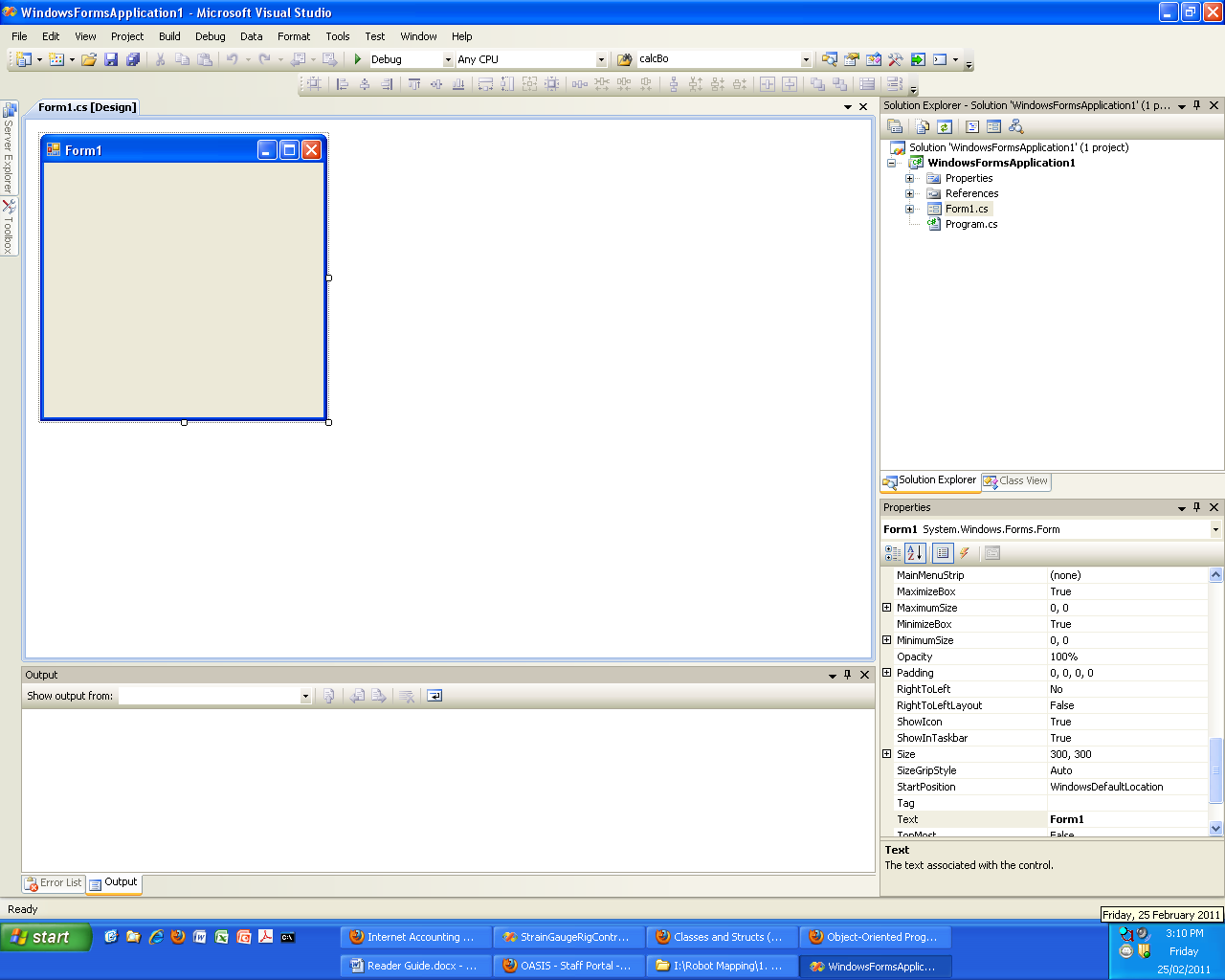


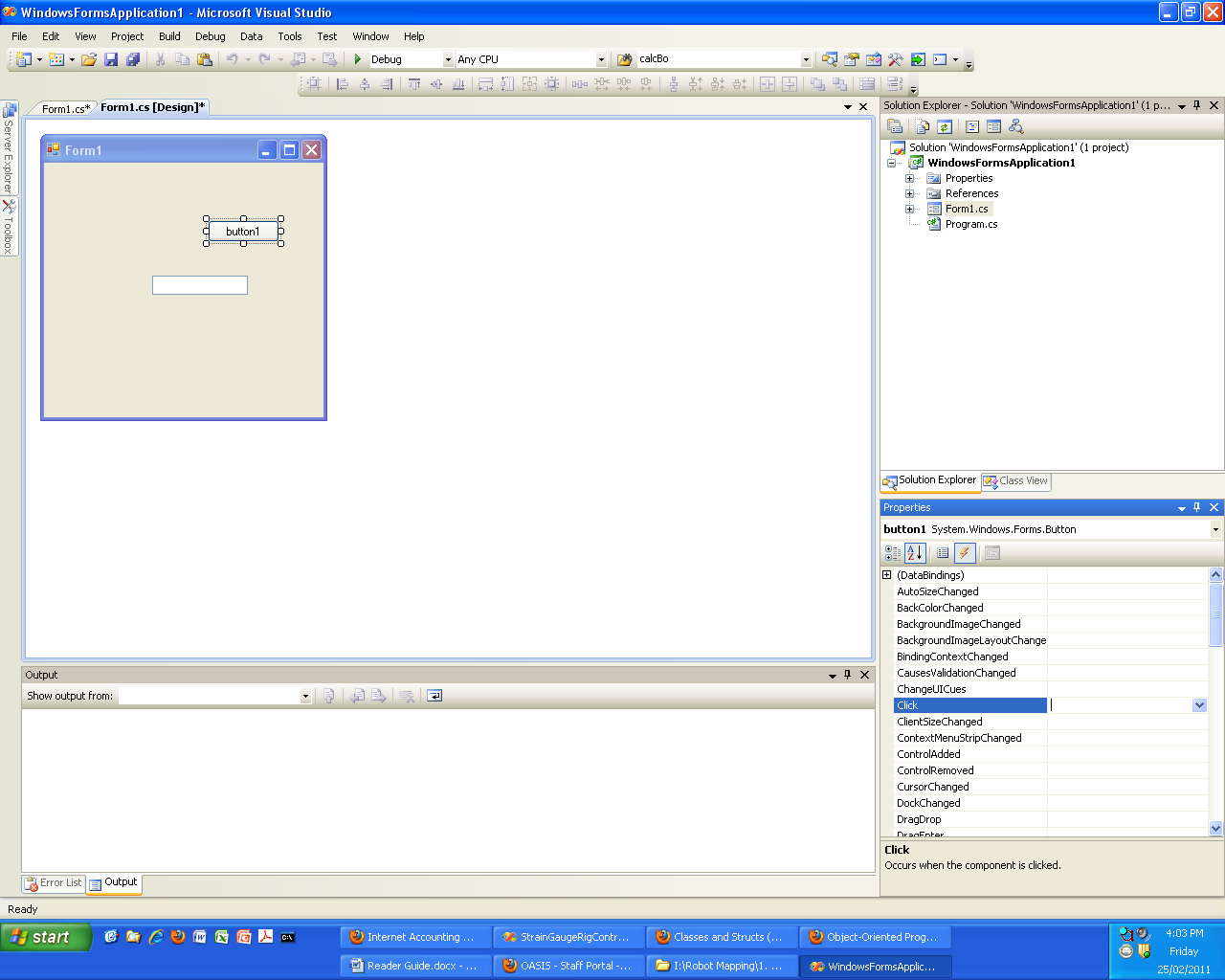
Figure : Windows Forms Application

Referring to Figure 1 the important points to note are:

* On the top right of is the Solution Explorer, which lists all the files associated with your project, most notably the class files (.cs) and in particular the Form class.
* On the top left is a blank form. Using this designer you can drag a variety of common windows components into your form to populate your GUI.

To get started try dragging a textbox and a button from the toolbox, place them anywhere on your form. Select the button and refer to the properties window (shown on the bottom right in Figure 1). Listed in the Properties Window are all the modifiable members of the Button class that will define the behaviour of this instance during runtime. It might also be useful at this point to change the (Name) field of the textbox and the button to something more intuitive or memorable. (Name) is the name of the textbox/button object and is how you will refer to it in code.

Click the lightning bolt and the Properties Window will display a list of events that can be responded to by the Button class.



Double clicking in the drop down next to ‘Click’ will open the editor for the main form class and generate the following method:

private void button1\_Click(object sender, EventArgs e)

{

}

button1\_Click will now execute every time the user clicks the corresponding button on the GUI. To demonstrate this, add the code:

private void button1\_Click(object sender, EventArgs e)

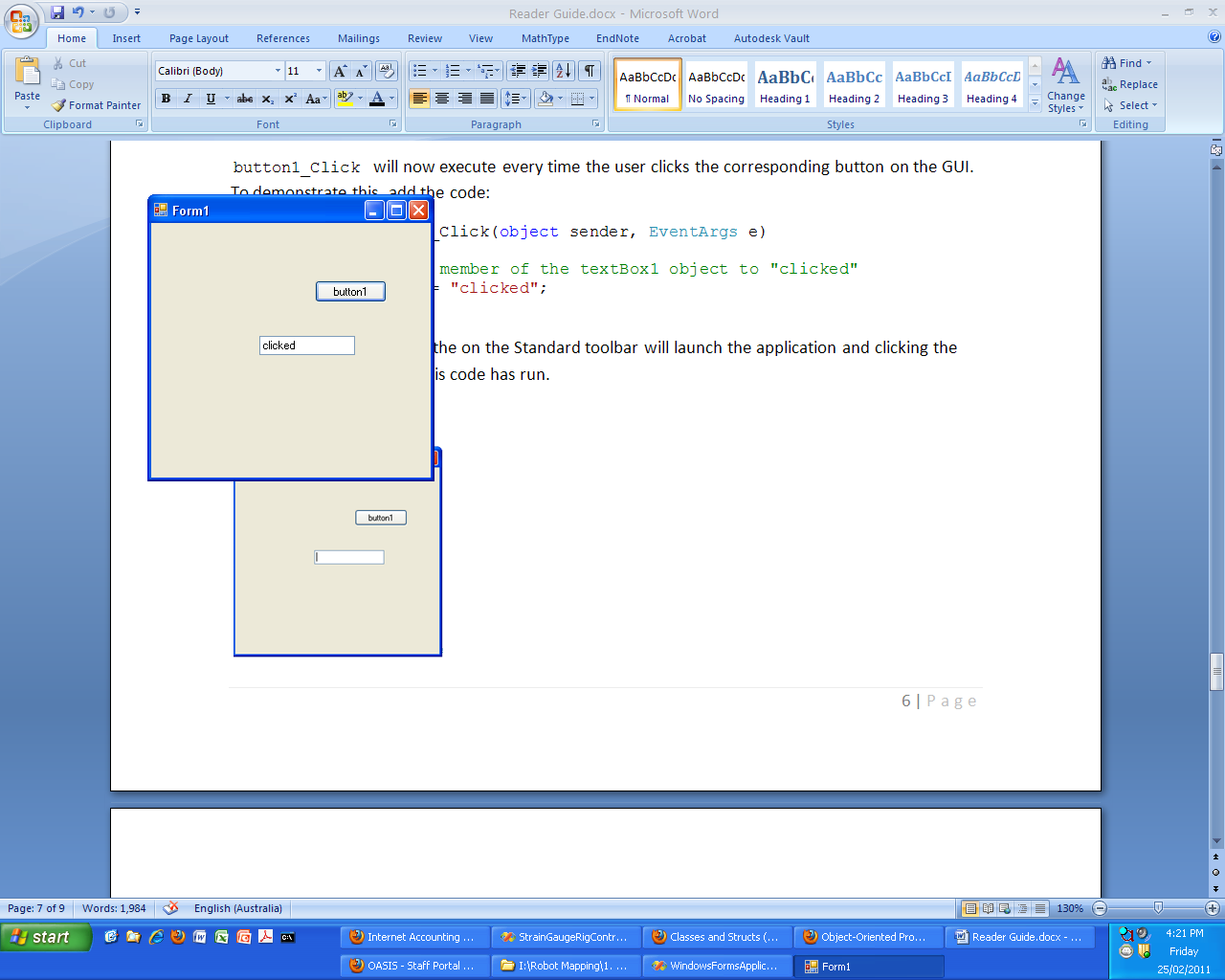
{

//set the Text member of the textBox1 object to "clicked"

textBox1.Text = "clicked";

}

Clicking the play button on the on the Standard toolbar will launch the application and clicking the button will illustrate that this code has executed.



In Visual C# your code is not centred around one main() function. Instead, it is distributed between a number the methods of your Form class corresponding to these events. Make sure that you familiarise yourself with the types of events a component can respond to and ensure that your GUI responds effectively to the events generated both externally via the user or internally by timers, backgroundworkers etc.

It’s helpful to note that Microsoft hasn’t done away with main(), it can be found the program class (Program.cs) and contains the following commands:

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new Form1());

}

Most important in the above is the use of the Run() method of the Application class. Run() starts what is known as an event loop which polls for external and internal events (a mouse click, a timer expiring etc). Upon detecting said events it dispatches them to an instance your Form class (seen above as new Form1()) which calls the appropriate method.

At this point it might also be helpful to demystify the link between the designer and your code. Without advertising it, Visual Studio generates a substantial amount of code on your behalf.

Whenever a component is dragged onto a form from the toolbox Visual Studio declares and instantiates an object responsible for it. If any properties outside of the defaults were specified for that component these are set via the appropriate methods. Finally if an event is to be managed the code to inform the EventManager of this, and to set the delegate is created.

This code generation will occur every time the project is successfully built and will appear in the InitializeComponent() method located in the Constructor of your Form class.

There are two take home messages from this:

1. Don’t delete the InitializeComponent() call in the constructor and don’t add code relating to the GUI elements before it. At this point, as far as the compiler is concerned, they don’t exist.
2. Don’t rename the methods handling events without specifying this in the designer or modifying the code in the InitializeComponent()

Visual Studio is an excellent development tool and we encourage you to investigate it further for yourself. If you are not already familiar with the debugger set a few break points by left clicking in the space to the left of the text editor and try stepping through your code. The intellisense (the auto-complete) is a very handy way of browsing the classes, methods and objects available to you at any given point in your code. Finally before writing any of your own code, make sure it hasn’t already been implemented in the framework. On more than one occasion I have spent a considerable amount of time implementing an algorithm or feature only to find that it exists already, in a better form as part of the .NET Framework.

# Using Custom Components

If you browse through your toolbox you’ll note that the .NET framework provides you with a very rich set of components that you can add to your GUI. You’ll probably find that once you’ve designed it, actually creating your GUI can be accomplished in a fairly short amount of time (the code that drives it will be a different matter).

Having said that, the toolbox is intended to be used by a great number of GUI designs each with its own set of requirements so unsurprisingly, in some areas it falls short. To make up for this deficiency, Visual Studio includes a means by which third party vendors can create and package their own components such that you can drag onto your form as you would a textbox or a button.

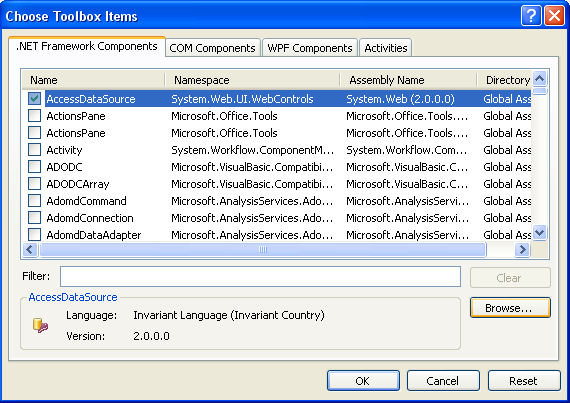
ZedGraph is one such third party component and is a very flexible graphing tool. Assuming you’ve read through the milestones document you’ll have no doubt noticed a need for such a component. The details for ZedGraph, a number of tutorials, sample code and so on can be found here:

<http://zedgraph.org/wiki/index.php?title=Main_Page>

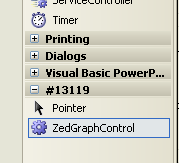
To add any custom component to your project you’ll need to download it in the form of a .dll. ZedGraph’s .dll can be downloaded here:

<http://sourceforge.net/projects/zedgraph/files/>

To add this it to your toolbox right click in it and select “Choose Items…”. Click Browse as per below:



Navigate to the .dll and click “Open”. The Component should then appear in your toolbox as such:



Depending on how your requirements analysis pans out, you may find yourself in need of an X-Y graph, a gauge or any number of other indicators to represent your system. The following are links to other components that may prove useful and we encourage you to investigate them and any others that may enhance the intuitiveness of your GUI.

* <http://www.mitov.com/html/plotlab.html> - Some plotting components
* <http://www.ucancode.net/CSharp_Tutorial_GDI+_Gauge_Source_Code.htm> – Some gauge components

# Threading

Threading is probably one of the more difficult concepts that you’ll encounter in the development of your controller and a very useful piece of knowledge to be gained from these laboratories.

Put simply, a thread is an independent execution path that can exist within your application. By creating multiple threads you can instruct your program to do multiple tasks at the same time. If, for example you wanted to execute two while(true) loops simultaneously within your program you’d create two threads and execute one within each.

As an interesting aside, threads are also the preferred tool for taking advantage of the multiple cores prevalent in modern CPU’s. On a dual, quad core etc threads can be allocated to their own core and executed simultaneously. On a single core processor, threads are still useful but are serviced through time multiplexing (switching back and forth between threads over time).

For a more in-depth explanation of threading read through:

<http://www.albahari.com/threading/#_Introduction>

Assuming you’ve read the link above you’ll likely have realised why threading is necessary in your application. Threading allows you to maintain a responsive user interface. If any time-consuming computations (such as recalculating a PID controller output or reading from a HCTL) are included in the user interface thread they will block the upkeep of your GUI. Depending on how hefty the computations, the GUI may appear jerky or even as if it has crashed.

By relegating any time-consuming computations to a worker thread you can leave your GUI thread free to process the user’s requests and to redraw itself in a timely fashion. This might also be a good time to point out that you should never include a Thread.Sleep() call on your GUI thread.

Thanks to the .NET and more specifically the BackgroundWorker class, your application will be able to incorporate threading in just a few lines of code. All the background information you’ll need to use the BackgroundWorker can be found here:

<http://www.albahari.com/threading/part3.aspx#_BackgroundWorker>

You can find the BackgroundWorker class in the toolbox in your designer as a component that you can drag onto your form. Make sure you give it a meaningful name and also set the WorkerSupportsCancellation and WorkerReportsProgress fields to true. Double click in the DoWork and ProgressChanged events to add them to your code.

The DoWork method contains the code that will actually execute on the worker thread. As a skeleton your DoWork method might resemble the following:

static void bw\_DoWork(object sender, DoWorkEventArgs e)

{

// you’ll need to define this struct

MSDServoMotorDynamics dynamics = new MSDServoMotorDynamics();

// so long as the thread is active, execute indefinitely

while (true)

{

// unless for whatever reason it is cancelled

if (bw.CancellationPending)

{

e.Cancel = true;

return;

}

// read the hardware, do the appropriate control, record

// the details in “dynamics”

// report progress to the main (GUI) thread

bw.ReportProgress(0, dynamics);

}

// there’s no point in executing as quickly as possible

Thread.Sleep(workerSleepTime);

}

}

Some points to note on the previous:

workerSleepTime is a variable of type int that should reflect the number of milliseconds you want the worker thread to sleep before reiterating. The BackgroundWorker will run fairly quickly so checking the hardware at every available opportunity isn’t really necessary. Inserting this delay will allow the processor to allocate more resources to the GUI thread.

The BackgroundWorker class includes a method called ReportProgress() which lets it inform the main thread of the state of its work and allows the main thread to update its GUI accordingly. ReportProgress() takes two parameters: int percentProgress (which you can ignore) and object userState. Calling this method will invoke bw\_ProgressChanged on the main thread which will receive a parameter e of type ProgressChangedEventArgs, e.UserState will be set to whatever you specified as userState in the worker thread.

Assuming you want to return more than one variable from your BackgroundWorker (position, velocity and acceleration for example) you’ll want to specify userState as an instance of a struct that contains all the data you want to be displayed on the GUI. The instance of MSDServoMotorDynamics, dynamics has been used in the above, since you’ll have to define this struct for yourself, you are free to name it however you’d like.

The actual update of the GUI can then be carried out in the bw\_ProgressChanged which resides in the main thread. Cast e.UserState into a more useful form and update the GUI. This will probably resemble the following:

//Runs when the background worker reports its progress

void bw\_ProgressChanged(object sender, ProgressChangedEventArgs e)

{

if (bw.CancellationPending == false)

{

updateGUI((MSDServoMotorDynamics)e.UserState);

}

}

A final note on threading is thread safety. When two or more threads have access to a common resource (a variable etc) special considerations need to be made to prevent threads from interfering with one and other. For the specifics, refer to:

<http://www.albahari.com/threading/part2.aspx>

As a general rule though, in situations where two or more threads have access to code that modifies a variable that code should be surrounded with a lock construct. The lock construct requires an object to lock on and that object needs to be visible to each thread. The best option is to declare a static object of the base type; object to carry this out. An example of this can be found in the startTicking() method in the Clock in .

# Appendix A: Clock.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Runtime.InteropServices;

using System.IO;

namespace StrainGaugeRigController

{

//A class that acts as a stop watch, it observes the clock time in microseconds when //setInitialTime() is called then returns it

public class Clock

{

[DllImport("KERNEL32")]

private static extern bool QueryPerformanceCounter(out long lpPerformanceCount);

[DllImport("KERNEL32")]

private static extern bool QueryPerformanceFrequency(out long lpFrequency);

private double initialTime = 0;

static readonly object locker = new object(); //thread safety

public Clock()

{

//nothin

}

//Records the PC time in seconds

public void startTicking()

{

lock(locker)

{

initialTime = getAbsoluteTime();

}

}

//Returns the PC time in seconds

private double getAbsoluteTime()

{

long PCfreq;

long PCcount;

QueryPerformanceFrequency(out PCfreq);

QueryPerformanceCounter(out PCcount);

return (double)PCcount \* 1000000 / (double)PCfreq;

}

//getter method for the time since the last setInitialTime() was called in seconds

public double Time

{

get

{

return (getAbsoluteTime() - initialTime) / 1000000;

}

}

}

}