MECH 423 – Lab 1: Data Acquisition using Visual C#
Visual C# Tutorial

By Richard Ang, Vahid Bazargan, and Hongshen Ma
Department of Mechanical Engineering, University of British Columbia

Introduction to C# and Lab #1

C# is a high-level computer language created by Microsoft. C# is extremely popular among programmers. Visual Studio is an integrated development environment (IDE). C# combines the syntax from C with graphical access to standard Windows building blocks (e.g. buttons, menus, options), otherwise known as Windows Forms controls. The programming environment provides context-sensitive code-completion and a comprehensive online help system (MSDN) that will enable you to quickly get started coding in C#. In fact, a key objective of this lab is to teach you *how to learn to program*. Therefore, there are three things to remember:

1. Do not rely on this document as your sole source of information on C#.
2. Seeking out additional resources (e.g. MSDN and Google) is required to complete the lab.
3. Your prof and TAs will not be able to solve all your problems.

In MECH 423, our primary interest is to use C# to communicate with peripheral devices to use PCs as part of a measurement and instrumentation system. Specifically, our objective in this lab is to use C# to create a data acquisition program that interfaces with a microprocessor connected to your computer via a USB port. The microprocessor samples data continuously from a 3-axis accelerometer. This guide begins by presenting some of the basics of the Visual C# language and programming environment. This guide includes tutorials for three mini-projects aimed at developing skills required to complete Lab 1. These mini-projects can be modified and incorporated into a final program.

Table of Contents

1 Getting Started with C#

We are programming using Visual Studio Community 2019. The software can be downloaded for free from Microsoft. Students will need to search for Visual Studio 2019 and download the installer for the free "Community" version. Run the installer and select only the ".NET Desktop Development" option. After installing, start learning about how to create apps using C# from here, as well as learn about Windows Forms from here.

1.1 Practices to Improve Program Readability

(Excerpt from http://msdn.microsoft.com/en-us/library/ms229042)

Writing code in a consistent style is critical in successful software development. It is amazing how quickly you will forget your own thought process; having consistent and readable code will allow you to quickly remember.

1.1.1 Word Choice

Use easily readable identifier names. For example, a property named HorizontalAlignment is more readable in English than AlignmentHorizontal.

Favor readability over brevity. The property name CanScrollHorizontally is better than ScrollableX.

Do not use underscores, hyphens, or any other non-alphanumeric characters.

Avoid Hungarian notation, which is the practice of including a prefix in identifiers to encode some metadata about the parameter, such as the data type of the identifier.

Avoid using identifiers that conflict with keywords of widely used programming languages.

1.1.2 Naming Conventions

Methods should contain verbs to describe its action and should use Pascal Case "RetriveCat()"

Parameters (Variables) should use Camel Case "unknownBoxOfMagic"

1.1.3 Capitalization Styles

Pascal Case: All first letters have a capital letter. Primary style used for objects, methods, events, etc… Example: "HelloWorld"

Camel Case: First word is lower case and each following word's first character is capitalized. Example: "helloCamelCase"

Upper Case: All letters are capitalized, only used for identifiers consisting of one or two letters

Hungarian notation: (do not use for variables, only for controls) Adding pre-fixes to objects to identify its type. Example: (frmMainWindow)

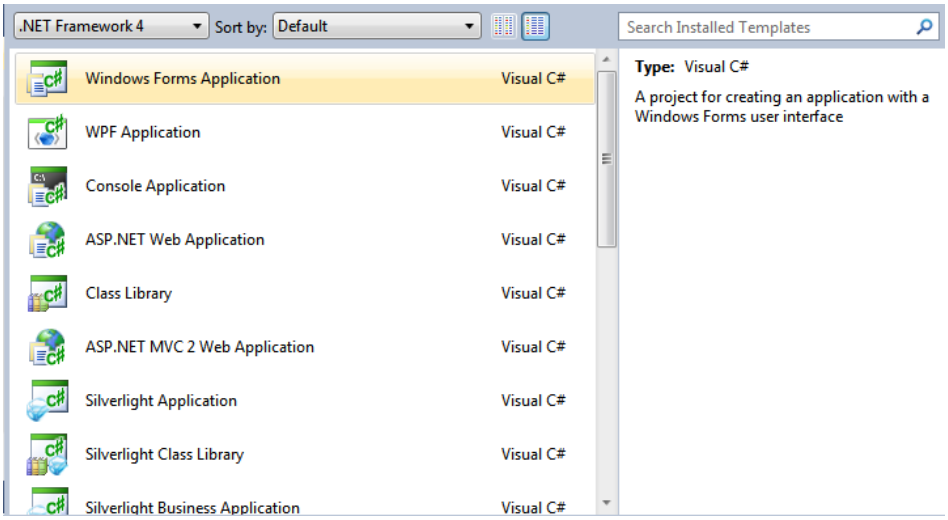
Examples follow,

Identifier	Case	Example
Class	Pascal	**AppDomain**
Event	Pascal	**ValueChange**
Exception class	Pascal	**WebException,** Always ends with the suffix **Exception**.
Method	Pascal	**ToString**
Namespace	Pascal	**System.Drawing**

Parameter	Camel	**typeName**
Property	Pascal	**BackColor**

1.2 Starting or Opening a Project

A **project** in C# is a place to store and organize components of your program. From the **File** menu click **New Project**, to start a new project where there will be several types of projects to choose from.



Select **Windows Forms Application** and click **OK**. You will be prompted to give the project a name; please give a meaningful one! On clicking **OK** again, Visual Studio will create a new form along with the necessary files for your project, all of which are shown in the **Solution Explorer** window. Initially, the IDE will show the form designer view containing an empty form. This form is the window that will be displayed when the program is run. Many programs display more than one window, so a project can contain multiple forms.

1.3 The Visual Form Designer

A fundamental component of Visual C# is the **form**, which can be viewed in the **Form1.cs [Design]** tab. A form is a window where other Windows components, such as buttons, menus, and text boxes can exist. These Windows components are known collectively as **controls**. Programming in C# is a combination of visually arranging controls on a **form**, specifying properties and actions of those controls, and writing code that direct how these controls respond to events. Controls are added to **forms** by drag-and-drop. Associated with each control are **events** and **properties**. An event is triggered during run-time when a user performs an operation on the control, such as a mouse click on a button. Properties are characteristics of the control (e.g. colour, font, enabled/disabled). Default attribute values are provided when the control is created, but may be changed by the programmer. Many attribute values can also be modified during run-time based on the action of user and other controls to create a dynamic application.

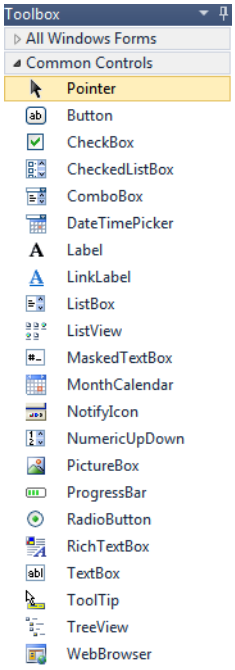
The **Form1.cs** tab is where the code for your application can be found. Any event handling functions are implemented by directly writing code here.

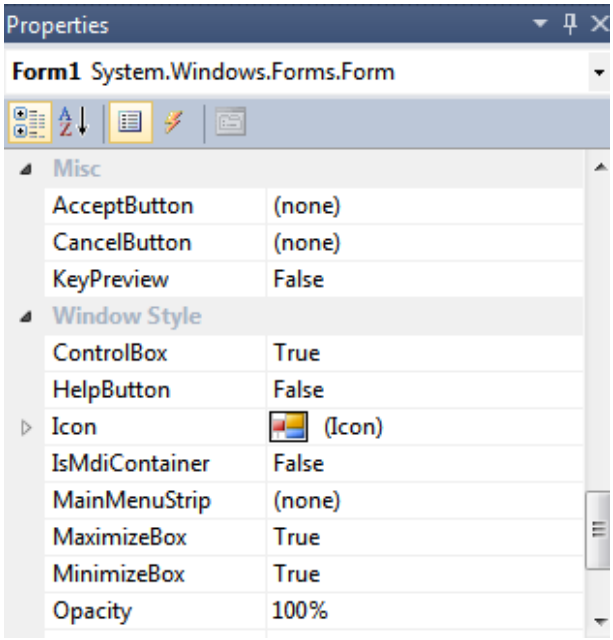
1.4 Creating a User Interface

The **Form1.cs [Design]** tab displays forms and various program modules in C#. The form designer shows what your program will look like. Everything you will place on this form will appear in your program. Initially, your form is empty. The **Toolbox** is on the left side of Visual Studio and consists of several tabs such as **Data**, **Components**, and **All Windows Forms**. Each tab contains a set of **controls**, which are components that can be added to your form. For example, the

Common Controls tab has entries named **TextBox**, **Button**, and **CheckBox** that represent controls that you can add to your application by dragging them onto the form.

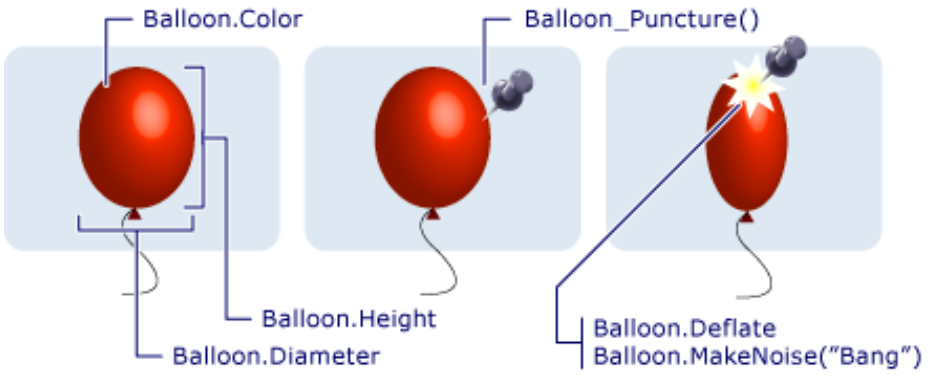
Button controls are typically used to perform tasks when the user clicks them using the mouse. **TextBox** controls are used to enter text on a screen through the keyboard. You may have noticed that when you add different controls to your form and click on them, the **Properties** window located at the right bottom are updated because different controls have different functions. The **Properties** window can be used to set the properties that will control the appearance and behavior of your application. All visual components in C#, including forms and controls, have their own **properties**, **methods**, and **events**. Properties can be thought of as the properties of an object, methods as its actions, and events as ways to trigger specific actions.





1.5 Controls and Review of the Framework of Object-Oriented Programming

Each control is an object in C# (In fact, *everything* is an object in C#!). In order to understand the elements associated with controls, we will review the concepts associated with object-oriented programming. An everyday object such as a helium balloon also has properties, methods, and events. A balloon's properties include visible properties such as its height, diameter, and color. Other properties describe its state (inflated or deflated), or properties that are not visible, such as its age. All balloons have these properties, although the values of these properties may differ from one balloon to another. A balloon also has methods or actions that can be performed. It has an inflate method (filling it with helium), a deflate method (expelling its contents), and a rise method (letting go of it). Again, all balloons can perform these methods. Balloons also have responses to certain external events. For example, a balloon responds to the event of being punctured by deflating, or to the event of being released by rising.



A balloon has **properties** (Color, Height, and Diameter), responds to **events** (Puncture), and can perform **methods** (Deflate, MakeNoise). As in this simple analogy, we program with objects (forms and controls) in C# by deciding which properties should be changed, which methods should be invoked, and which events should be responded to and how they should be responded to in order to achieve the desired appearance and behavior.

1.5.1 Properties

If you construct a balloon object in C#, the code that sets a balloon's properties might look something like this:

```
Balloon.Color = System.Drawing.Color.Red;
Balloon.Diameter = 10;
Balloon.Inflated = True;
```

Notice the syntax to set object properties — the object (Balloon), followed by the property (Diameter), followed by the assignment of the value (= 10). You could change the balloon's diameter by substituting a different value.

1.5.2 Methods

A balloon's methods are called as follows.

```
Balloon.Inflate();
Balloon.Deflate();
Balloon.Rise(5);
```

The order resembles that of a property—the object (a noun), followed by the method (a verb). In the third method, there is an additional item, called an argument, which is a parameter of this method. In this case, the argument specifies the distance the balloon will rise. Methods can have zero or several arguments to further describe the action to be performed.

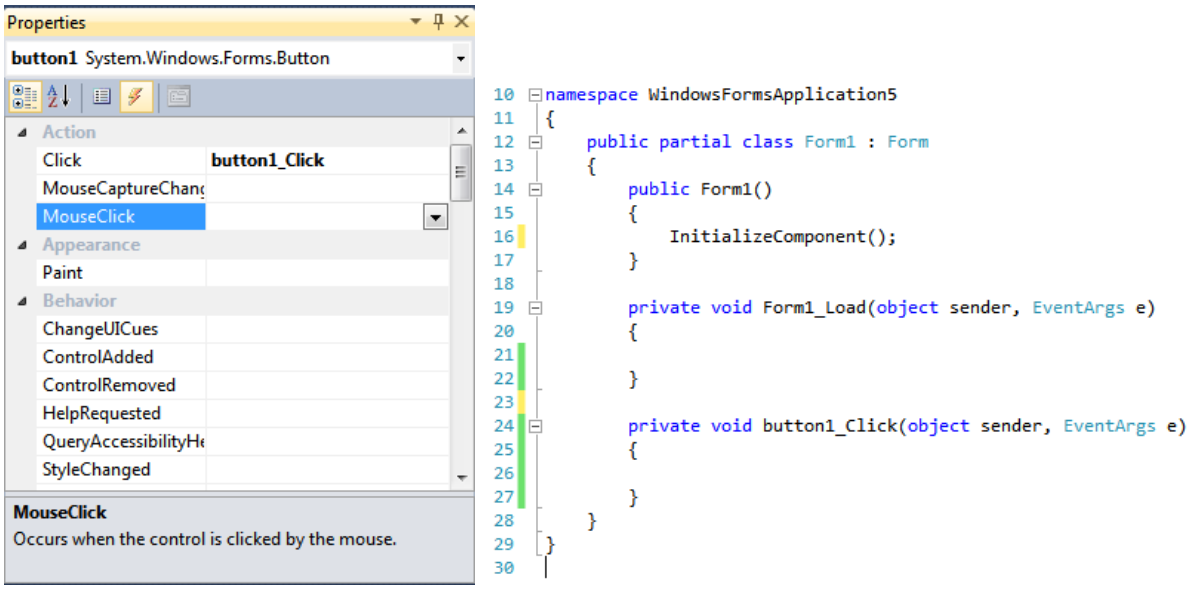
1.5.3 Events

Puncture is an event that may occur and here is some possible code to direct how the object may respond to that event. In this case, the code describes the balloon's behavior when a Puncture event occurs. When this event occurs, call the MakeNoise method with an argument of "Bang" (the type of noise to make), then call the Deflate method. Since the balloon is no longer inflated, the Inflated property is set to False.

```
public void Balloon_Puncture()
{
    Balloon.MakeNoise("Bang");
    Balloon.Deflate();
    Balloon.Inflated = False;
}
```

1.6 Events and Event-Handlers

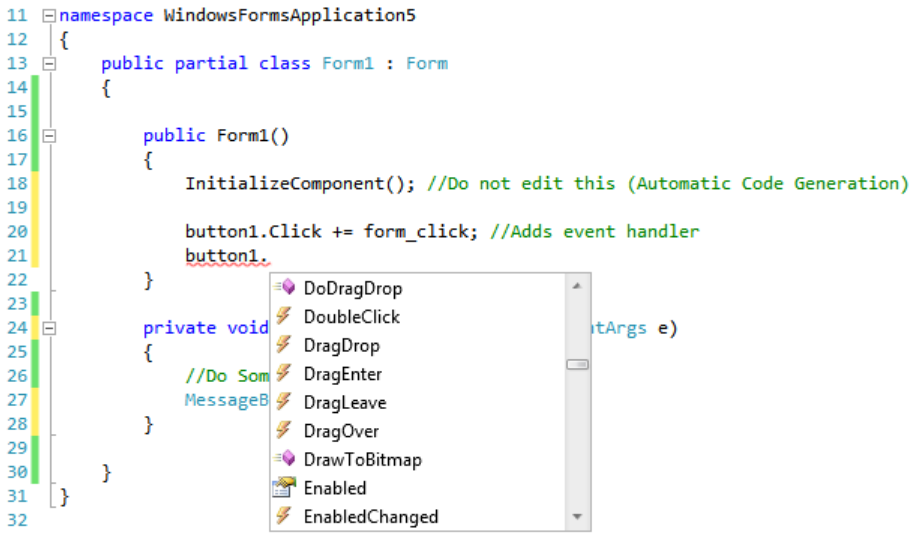
Much of the behaviour of a Visual C# application is determined by its response to events. Coding for each control in C# is typically done in three stages: 1) insert the control, 2) set the properties, and 3) write the event handling code. Most controls have one most common event. For example, single mouse clicks are most common for buttons, text changes are most common for textboxes. If you double-click on a control in the IDE, C# will create an empty event-handling routine for the most common event. If you want to create an event-handler for another event, click on the lightning bolt button to get to the event menu in the properties pane for a control in the IDE. A code window should now be displayed like the one shown below.



Note that you should only delete events from the properties window in the **Form1.cs [Design]** tab, as the C# IDE is automatically generating code to link the event handler to the UI element. Removing the event handler will cause an exception for a missing handler. All code should go between the curly braces for each event handler.

1.6.1 Manual Event Handler Definition

If you want to create an event handler without the IDE interface, you can also do so in the **Form1.cs** tab by directly accessing the object event handles and assigning your own method using the following procedure.



As shown above you can see all the events an object has in the auto-complete dialog (Any lightning bolt symbol is an event that can be subscribed to)

To attach a method to an event, type the following code before the form is loaded. Since the events are programmable you can subscribe and unsubscribe at any point in the program.

```
PublisherObject.EventName += EventHandlerMethodName;
```

This line will subscribe the method with the same name to handle the event for the object. As shown below by subscribing before the start of the program you can have UI events manually setup for events for the lifetime of the form.

```
public Form1()
{
    InitializeComponent(); //Do not edit this (Automatically Generated Code)

    button1.Click += form_click; //Adds event handler
}

//Event Handler
private void form_click(object sender, EventArgs e)
{
    //Do Something Useful
    MessageBox.Show("Test");
    button1.Click -= form_click; //Removes event handler, won't execute in subsequent events
}
```

The line below will unsubscribe the method with the same name to handle the event for the object. As shown below by unsubscribing you can have events stop listening when required.

```
PublisherObject.EventName -= EventHandlerMethodName;
```

If you want to assign the same handler to multiple controls, you can use the following code to iterate through controls.

```
//Adds event handler for every button in the forms controls list
//foreach (TYPE TEMP_NAME in CLASS.GROUP){DO_STUFF();}
foreach (Button target in this.Controls)
{
    target.Click += form_click;
}
```

This code above will attach the form_click event handler method to all buttons in this form's controls. (You can also unsubscribe using the same method).

Remember not to use the same name and different case for variables between types as this can become confusing and error-prone. (Ex: Don't use, `foreach (Button button in this.Controls))`

1.7 Namespaces in C#

One of the important advantages of C# is its ability to access libraries of classes in the Windows OS. Namespace is a naming structure used to catalog these class libraries. A namespace begins with a root and is followed by branches and sub-branches that are delimited using a dot (.). For example, the **System.IO** namespace include classes for data access with files. The following code specifically copies one file into another:

```
System.IO.File.Copy("source_file.txt", "destination_file.txt");
```

Every project in C# is also a root **namespace**, which is set in the Property page of the project. When we create a project, by default, the name of the root namespace for the project is set to the name of the new project. For example, the root namespace for a project named MyProject is MyProject. Usually, when we create a project, we would like to give it an identifying name. Therefore, we will rename MyProject as **Project1**.

Namespace can be used explicitly through direct addressing or implicitly through the **Using** statement. Direct addressing involves directly accessing any class in the namespace by providing the fully qualified name. Example of using fully qualified name is given below:

```
System.Media.SoundPlayer myplayer1 = new System.Media.SoundPlayer;
```

If we want to make all the classes in a given namespace available without the need to type the entire namespace each time, you can use the **Using** statement. An example of using the Using statement is given below:

```
Using System.Media
...
SoundPlayer myplayer1 = new SoundPlayer;
```

1.8 Version Control

Version control is now a required part of every software development project. In fact, you may have noticed that Visual Studio does not have a "Save As" function, which tend to create conflicting versions. Visual Studio has a built local repository (see the "Team Explore" pane), where you can commit changes, create branches, etc. To set up a remote repository for online backups and collaborations, set up an account on Atlassian Bitbucket Git. In Visual Studio, add the Bitbucket Extension under Tools>"Extensions and Updates".

2 Programming in C#

Up to this point, the interface that you need in your project has been created and you want to build a program statement that include any combination of **C# Keyword, properties, object names, variable, numbers** and other values that collectively create a valid instruction recognized by C# compiler.

2.1 Object-oriented Programming Features

2.1.1 Scope in C#

C# is an object orientated language and various level of separation exist to allow interoperability between objects.

Nested Scope: If you initialize a variable inside a conditional statement it will only be accessible in that same conditional branch as there is a possibility that the variable might not exist outside of the conditional statement.

```
 if (test == 50) { string message = "Match"; }
Console.WriteLine(message);   //message does not exist in the current context
```

Method Scope: If you initialize a variable inside a method then it is accessible and persistent for the duration of the method. Once the method finishes the variable will not be available.

```
public int testData(string input)
{
    int result = 0;         //Only exists within method

    if (input == "magic")
    {
```

```
            result = 10;        //Can be used within method
        }

        return result;          //Passed by value (copied) to caller
    }
```

Object Scope: In class instances the data will be accessible and persist in the parent object as long as the object is not disposed.

```
//Container For State Information
public class StateMachine
{
    private int nextState = 0;      //Internal Variables (Not Accessible)
    private int prevState = 0;
    private int intendedState = 0;
    ...
```

2.1.2 Access Control

Controlling the access and separation of variables, classes, and methods is critical in programming to create a properly functioning and logical program.

By default C# uses private as the default access level which restricts the access to the body of code it currently resides in this is the least open state and does not allow access from other objects or derived classes.

Public: No restrictions on access

Private: Limited to class definition (Within class or method where it is declared)

2.1.3 Data Persistence

Global variables can cause a great deal of problems for software programs because they break encapsulation, which can cause unintentional modification of variable when pieces of code are combined. Furthermore, in multithreaded applications, global variables can cause issues with contention, deadlock, and synchronization. High-level programming languages, including C#, deals with this problem by allowing variables to persist only within a narrowly defined scope. The scope of a variable could be within a statement, a routine, or an object. However, in a hardware system, global variables storing system state often need to persist if such systems remain active. The recommended way to store a state in C# is to use class variables to store the state of a particular system and as long as the object is not reset or disposed the state will remain.

The following example shows how one can store and retrieve a state from a class.

```
//Container For State Information
public class StateMachine
{
    private int nextState = 0;      //Internal Variables
    private int prevState = 0;
    private int intendedState = 0;

    //Setters Control State Information (Can enforce rules or additional logic)
    public void SetNext(int targetState)
    {
        prevState = nextState;
```

```
            nextState = targetState;
        }

        public void SetIntent(int targetState)  { intendedState = targetState; }

        //Getters Retrive State Information
        public int Next()   { return nextState; }
        public int Intent() { return intendedState; }
        public int Prev()   { return prevState; }
    }
```

In this example, state information is protected using methods (Data is private, methods used to get and change variables). If you just set the variables to public there will be no protection on the data but as long as the object that created this class instance exists, the data will persist. The following code in your form class is needed to use the state machine information.

```
        StateMachine ControlDat = new StateMachine();
```

The following example shows how a class can store a variable in a globally accessible manner.

```
    static class GlobalState
    {
        private static int stateGlobal = 0;

        public static void setGlobal(int targetState) { stateGlobal = targetState; }
        public static int getGlobal() { return stateGlobal; }
    }
```

Note: in this example you do not need to initialize the GlobalState class and its methods are globally accessible.

Note: In multithreaded applications global variables can cause contention, deadlock, and synchronization issues, without careful design. If possible only one thread should write to a globally accessible variable and many can read from it. Adding synchronization to a variable can cause performance issues and if not properly designed can cause intermittent crashes that can be hard to trace. (Note: most variables, objects in .NET are NOT thread safe)

2.2 Variables in C#

Variables are containers for data that are fundamental to any computer programming language. The key properties of variables are type (i.e. the size of the container) and scope (i.e. where the container could be accessed from). Variables are declared in C# using the **VariableType Name** format. This declaration reserves a location in memory for the variable at run time so that C# knows what type of data to expect. For example, the statement

```
        string Lastname = "MyLastName"; //Declares and Intializes String
        Lastname = "NewName"; //Accessing Variable
```

will hold space for a string of characters for a variable named **Lastname**. Below some data types that can be used for variables are listed in the following figure.

IMPORTANT NOTE: C# is case sensitive and NEVER use the same variable name with different case.

Data Type	Size [Bytes]	Range	Sample Use	Note

bool	**1**	**null,true,false**	**bool test; test = false;**	**No implicit conversions given (x=0) (if(x) equivalent to if(x!=0), if(x==0) must occur)**
byte	**1**	**0 to 255**	**byte mybyte; mybyte = 13**	**Unsigned (No Negative Numbers)**
sbyte	1	-128 to 127	sbyte mysbyte; mybyte = 13	Signed
char	**2**	**U+0000 to U+ffff**	**char mychar; mychar = 'Z'; mychar = (char)88**	**Unicode symbol within 16bit range can be converted to numerical formats implicitly but must be explicitly converted from other formats using casting**

decimal	16	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$	decimal myDec;
			myDec = 300.5m;
			myDec = 300;
				Decimals are large numbers that can store decimal points (needs suffice m or M as compiler normally treats floating point numbers as doubles). Integers are implicitly cast. (28-29 Digits of Precision)

double	**8**	**$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$**	**double myDbl; myDbl = 30D; myDbl = 30.32;**	**Doubles are standard floating point numbers (Need suffix d or D if you want an integer to be treated as a double) (15-16 Digits of Precision)**
float	4	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	float myFloat; myFloat = 3.5F;	Floats are low precision floating point numbers (Need suffix F for numbers as double is the default numerical representation)
int	**4**	**-2,147,483,648 to 2,147,483,647**	**int myint; myint = 123;**	**Integers only require explicit conversions were information might be lost (float to int, long to int)**
long	16	– 9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	long long1 = 4294967296;	64 bit integer (ulong is the unsiged version)
short	2	-32,768 to 32,767	short x = 32767;	Can be implicitly cast to any larger format
string	16/char	Unicode characters	string a = "hello";	Note strings are immutable and any changes results in a new string being created and replacing the old one. (This can cause large overheads if you are using long strings with repeated operations)

2.2.1 Parameters

Parameters are variables that are passed to a subroutine (known as a **method** in C#). For example, if you click on the form and choose the **MouseMove** from the event pane, the subroutine below is generated:

```
private void Form1_MouseMove(object sender, MouseEventArgs e)
{

}
```

In this subroutine, *e* has defined as an **Object** variable type that contains mouse event arguments; for instance, *e.X* is the horizontal and *e.Y* is the vertical location of the mouse on the screen.

2.2.2 Type Casting

Type casting is the conversion of one type of variable to another. The **Object.ToString()** method returns the value of a variable or object to a human-readable string. This method is inherited by (*i.e.* is a part of) all variable types and therefore can be accessed as follows:

```
int dataPoint = 5;
string outputString = dataPoint.ToString();
```

For general conversion between different variable types, the **Convert** class can be used. For example:

```
int dataPoint = Convert.ToInt16(outputString);
```

2.3 C# Operators

The following table lists the C# operators in the order that they are evaluated. The first line is evaluated before the lines below and assignment operations occur last. Click on the sections to see the MSDN documentation for more information.

Section	Category	Operators	Note	Usage
Section 7.5	**Primary**	`x.y f(x) a[x] x++ x-- new typeof checked unchecked`	Object Access, Method/Precedence, Array Index, Post-inc/dec, Object Creation, Type/Overflow	Most basic operations
Section 7.6	**Unary**	`+ - ! ~ ++x --x (T)x`	Postive, Negative, Logical Not, Logical complement, pre-inc, pre-dec, type cast	Sign operators, pre-inc/dec, type cast
Section 7.7	**Multiplicative**	`* / %`	Multiplication, Division, Remainder	basic math
Section 7.7	**Additive**	`+ -`	Add/Subtraction	basic math
Section 7.8	**Shift**	`<< >>`	Bitwise shifts	bit-operations
Section 7.9	**Relational and type testing**	`< > <= >= is as`	Less than, Greater than, Less than or equal, Greater than or equal, is of type, or treat as type	comparison
Section 7.9	**Equality**	`== !=`	equal to, not equal to	comparison
Section 7.10	**Logical Ops**	`& ^ |`	Logical AND, Logical XOR, Logical OR	bit-operations (Precedence in order listed)
Section 7.11	**Conditional Ops**	`&& || ?:`	Conditional AND, Conditional OR, Conditional (Condition ? Expression1 : Experssion2)	conditional-operations (Precedence in order listed)
Section 7.13	**Assignment**	`= *= /= %= += -= <<= >>= &= ^= |=`	assign, multiply assign, divide assign, modulo assign, add assign, subtract assign, shift left, shift right, logical and, logical xor, logical or	assignment operations

The following table is a partial list of math methods in the **System.Math** class that you can use within your program. The argument *n* in the table represents the number, variable, or expression you want the method to evaluate. If you use any of these methods, be sure that you put the statement **using System.Math** at the very top of your code in the **Form1.cs** tab.

Method	Types	Description
Abs(n)	(Decimal, Double, Int16-64, Sbyte, Single).IN/OUT	Returns absolute value of n
Acos(n)	double.IN/OUT	Returns angle in radians whose cosine is n
Asin(n)	double.IN/OUT	Returns angle in radians whose sine is n
Atan(n)	double.IN/OUT	Returns angle in radians whose tangent is n
BigMul(a,b)	int32.IN, int64.OUT	Multiply 32bit int a,b to 64bit result

Ceiling(n)	(decimal, double).IN/OUT	Rounds upto nearest integer
cos(n)	double.IN/OUT	angle in radians
cosh(n)	double.IN/OUT	angle in radians
Exp(n)	double.IN/OUT	natural number e to the specified power
Floor(n)	(decimal, double).IN/OUT	rounds down to the nearest integer
Log(n)	double.IN/OUT	natural log base (e) of n
Log(n,b)	double.IN/OUT	specified log base b of n
Log10(n)	double.IN/OUT	log 10 of n
Max(a,b)	(int16-64,Sbyte,Single,UInt16-64,Decimal,Double,Byte).IN/OUT	returns the larger of the two inputs
Min(n)	(int16-64,Sbyte,Single,UInt16-64,Decimal,Double,Byte).IN/OUT	returns the smaller of the two inputs
Pow(n,p)	double.IN/OUT	p to the power of n
Round(n)	(decimal, double).IN/OUT	rounds to nearby integer
Round(n,d)	(decimal, double).IN/OUT, (Int32).Param	rounds to specified decimal place
Round(n,mp)	(decimal, double).IN/OUT, (MidpointRounding).Param	rounding with midpoint mode specified
Round(n,d,mp)	(decimal, double).IN/OUT, (Int,MidpointRounding).Param	rounding to specified decimal place with midpoint mode specified
Sign(n)	(decimal,double,int16-64,byte, single).IN/OUT	returns sign of number
Sin(n)	double.IN/OUT	angle in radians
Sinh(n)	double.IN/OUT	angle in radians
Sqrt(n)	double.IN/OUT	square root function
Tan(n)	double.IN/OUT	angle in radians
Tanh(n)	double.IN/OUT	angle in radians
Truncate(n)	(Decimal, Double).IN/OUT	truncates to decimal place

For example, the following code uses the **Abs** method of the **Math class** to compute the absolute value of a number. However, these functions are not supported in the main C# class. If you want to use them, you must first import all the math classes into our main program. As explained before, you can import **Math class** methods with inserting **using** statement at the first line of your program (as shown below).

```
using System.Math;
//Above line should be at the top of the file
//Code below must be in a method
double MyNumber;
MyNumber = Math.Abs(+50.3);  //Returns 50.3
MyNumber = Math.Abs(-50.3); //Returns 50.3
```

2.4 Decision Structures

2.4.1 Conditional Expressions

A **conditional expression** is part of a program statement that asks a true or false question about a property, a variable, or another piece of data in the program code. You can use the following comparison operators in a conditional expression:

== Equal to
!= Not equal to
> Greater than

< Less than
>= Greater than or equal to
<= Less than or equal to

NOTE: As in C, "=" is the assignment operator, "==" is the conditional operator. Using "=" as a conditional will result in a broken conditional.

2.4.2 If Decision structures

You can use an *If(conditional) {code;}* decision structure to evaluate a condition in the program and take a course of action based on the result. In its simplest form, an *If(conditional) code;* decision structure is written on a single line:

```
if (score >= 90) Label1.Text = "A+";
```

Is an *If* decision structure that uses the conditional expression **score>=90** to determine whether the program should set the **text property** of the **Label1** object to "A+". If the **score** variable contains a value that's greater than or equal to 90, the **Label1.text** sets as "A+"; otherwise nothing happens.

When a multiple-line *If...Else* is encountered, condition is tested. If condition is **True**, the statements following *code* is executed. If condition is **False**, each *Else If* statement is evaluated in order. When a **True** *Else If* condition is found, the statements immediately following are executed. If no *Else If* condition evaluates to **True**, or if there are no more *Else If* statements, the statements following *Else* are executed. After executing the statements following *If*, *Else If*, or *Else*, execution continues with the statement following end bracket.

```
if(condition1)
{
    statements;
}
else if(condition2)
{
    statements;
}
else if(condition3)
{
    statements;
}
else (condition4)
{
    statements;
}
```

Note: C# requires a conditional statement for integers and numbers and will not implicitly convert them into Boolean expressions. The following code is invalid and type error from visual studio will be reported.

```
int test = 0;
//BAD IF STATEMENT
if (test)
{
    MessageBox.Show("Huh, You will never see me");
}
```

To correct this, it simply requires an explicit conditional.

```
int test = 0;
//GOOD IF STATEMENT
if (test == 1)
{
```

```
        MessageBox.Show("Huh, You will never see me");
    }
```

2.4.3 Switch Case Decision structures

You can also use *switch case* decision structure to control the execution of the statements. The syntax for a *switch case* structure looks like this:

```
int caseSwitch = 1;
switch (caseSwitch)
{
    case 1:
        Console.WriteLine("Case 1");
        break;
    case 2:
        Console.WriteLine("Case 2");
        break;
    default:
        Console.WriteLine("Default case");
        break;
}
```

2.4.4 Iteration (Loops)

Loops allow you to perform repetitive operations on a set of data or stream of incoming data in a compact coding style. Note that the *foreach* loop is preferred were possible as the complier will decide the best format for you and will minimize loop definition errors.

foreach Loop

For Each Loops will iterate automatically through a series of objects in accordance with the following format.

```
foreach (ObjectType tempName in Group.SubGroup.etc)
{
    statements;
}
```

As shown previously the for each loop can be used to affect all button controls on the code's current class (Note the this refers to the class the code belongs to which is the windows form)

```
foreach (Button target in this.Controls)
{
    target.Click += form_click;
}
```

*Note: You cannot delete, re-assign the iterated object in a foreach loop.

for Loop

For Loops will iterate through a series of objects in accordance with the following format.

```
for (ObjectType counter = initVal; counter conditional; counter inc/dec)
{
    statements;
}
```

The following code uses an integer i set to 1 initially and increments it till the counter is less than 4. (Note that using the for(;;) style will cause an infinite loop)

```
for (int i = 1; i < 4 ; i++)
{
    statements;
}
```

while Loop

While Loops will iterate through a series of objects in accordance with the following format.

```
while (counter conditional)
{
    statements;
}
```

The following code uses an integer i and so long as it is less than 30 then the while loop will continue. (Note this loop can result in infinite loops)

```
while (i < 30)
{
    statements;
}
```

do while Loop

Do While Loops will iterate through a series of objects in accordance with the following format. Do while loops will occur once even if the conditional statement is false.

```
do
{
    statements;
} while (counter conditional);
```

The following code uses an integer i and loops as long as i is less than 5.

```
do
{
    statements;
} while (i < 5);
```

(Note: For code readability, it is best not to use single letter identifiers)

2.5 The Timer Control

The **Timer** control is an invisible stopwatch that gives you access to the system clock in your programs. Using the **Timer,** you can trigger the execution of subroutines at specific **Intervals**. Although timer objects are not visible at run time, each timer is associated with an event procedure that runs every time the timer's preset interval has elapsed, which is known

as a timer tick. You set a timer's interval by using the **Interval** property, which is measured in milliseconds. You can start the timer by setting the timer's **Enabled** property to true. Once the timer is enabled, its event handler routine will be executed when its preset interval has elapsed. Note the accuracy of timers on a PC is limited. Timer intervals less than 100 ms are often inaccurate.

2.6 Error Handling using Try & Catch & Finally Structure

Since C# 2010, an improved approach to handle unanticipated run-time errors (also called run-time exceptions) has been adopted. Unanticipated run-time errors, such as missing or unconnected hardware, are handled using the *Try...Catch...Finally* structure as follows.

```
try
{
    throw new Exception();  //Do something dangerous things that might throw exceptions
                            //NEVER use try/catch/finally around everything
                            //These statements have a performance penalty
}
catch(Exception e)
{
    handleException(e);     //Handle a specific exception type
                            //Process the exception
}
catch
{
    handleAnythingElse();   //Handle anything
                            //(Do not catch errors blindly this will make debugging impossible)
}
finally
{
    NoMatterWhatCode();     //Perform critical actions like closing port/file/lock
                            //Used for actions that should occur no matter what
}
```

The try statement allows you to execute code that may fail such as opening a file, port, lock, etc…

If the try statement throws an exception, then the catch statements can attempt to handle either a specific exception defined in the catch statement or all exceptions.

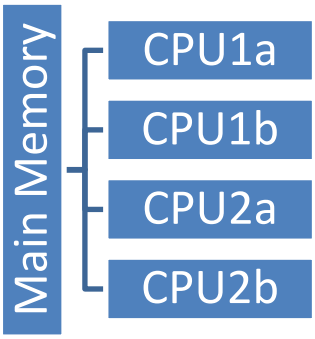
Note that you should never suppress all unknown exceptions as this may make debugging very difficult

The finally statement is always executed can be used to attempt to tidy up after a crash regardless of the state of the program. Typically used to close ports/files even if they did not open properly or release locks the program may be holding.

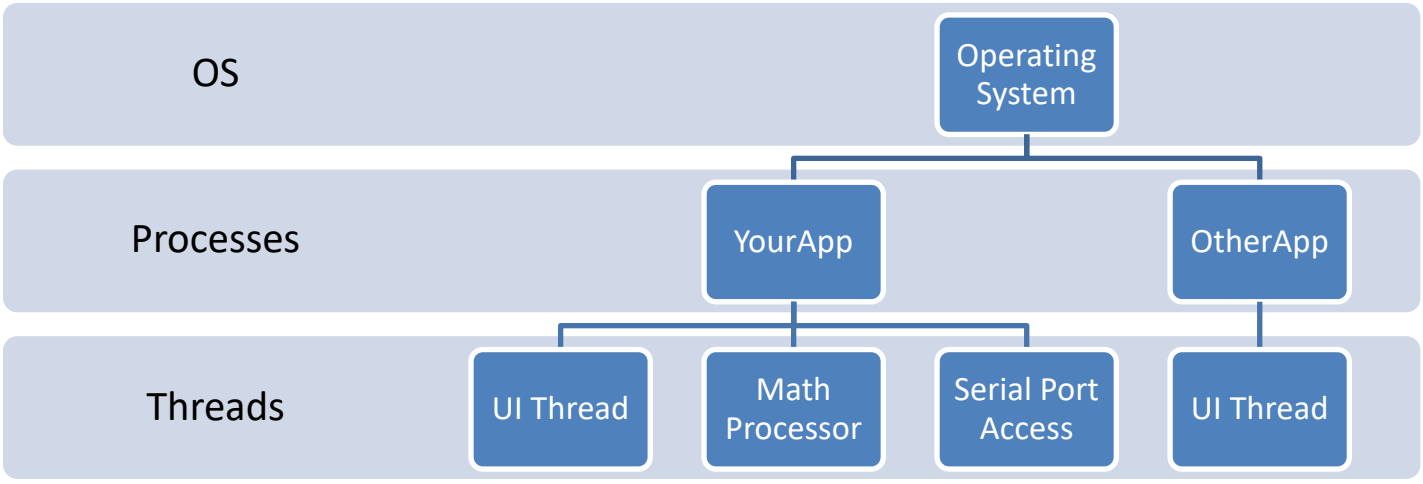
2.7 Multi-Threading

With modern computers often using at least two physical processors and top of the line systems containing up to 24+ logical CPUs, threading has become the primary way to improve performance and program responsiveness. In this

course's case the interface between hardware and user interface software prime area for threaded operation.



By default, the user interface generated in a windows form application resides in its own thread called the Graphical User Interface Thread. If there is any calculation that requires an extended period of time this can make the application appear to be unresponsive or slow. To prevent this most data intensive operations are separated onto separate threads. In addition, the commonly used SerialPort module used in many of these programs to communicate with the microcontroller implicitly creates its own thread to handle incoming data in a timely manner. If the serial port thread resided on the UI thread any latency in the program interface would cause data to be lost.



While the multi-thread organization scheme allows for highly responsive programs, it introduces the difficult and complex issue of synchronization and race conditions. By default, attempting to update the UI from another thread will cause an exception as this can result in unpredictable UI element states (program can randomly crash).

2.7.1 Creating Threads

To create your own thread, you first need to import the following libraries with the using statements.

```
using System;
using System.Collections.Concurrent;
using System.Threading;
using System.Threading.Tasks;
```

Threads can be created anonymously (*i.e.* without a named reference) as below. In this case, a Messagebox is created on a separate thread operating in parallel to the main form thread. The Messagebox created in this manner operates concurrently to the form, which allows the form to stay active while the user interacts with the Messagebox. The thread is destroyed once the user closes the Messagebox. The code structure below greatly simplifies the many thread related operations and is called a Lambda expression. The part between the curly braces { } can contain as many lines of code as you want and will automatically start the thread and execute the code contained within the expression.

```
new Thread(() =>
{
    Messagebox.Show("Boo Radley");
}).Start();
```

Threads can also be created with a reference as below. In this case, you can control the thread as a standard object.

```
Thread yourThread;
yourThread = new Thread(objectName.methodName);
yourThread.Start();
```

2.7.2 Cross-thread Calls

In this class, we will need cross-thread calls in order to communicate between the UI thread and serial reader in a safe and efficient manner. To do so we will need to use an inline Lambda expression below, which allow for the simplification of complex statements in a compact manner. Specifically, in a non-UI thread, such as a serial reader or a thread you created yourself, calls to the GUI should follow this format.

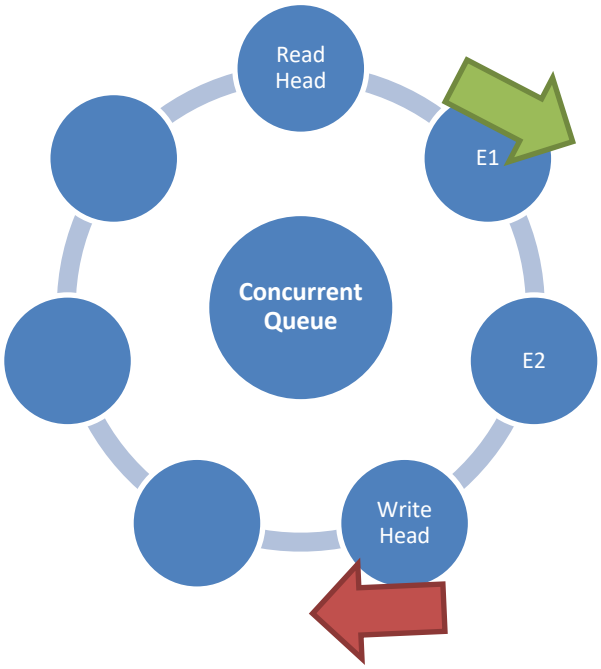
```
string newText = "Cross Thread Update";
this.Invoke((MethodInvoker)delegate { button1.Text = newText; });
```

The above code, for example, can be executed in the serial reader to allow you to access properties in the controls of the main form. Note that Invoke will block the current thread.

In the following section we will discuss using thread-safe objects to allow different threads to synchronize and share data correctly. If you are not careful, multi-threading code bugs can crash your computer, visual studio, or just your program.

2.8 Queues (Producer Consumer)

Queues are often used in data handling to acquire data in a buffer and thereby service consuming threads. For simplicity sake the model used in this course involves a single producer and consumer, which greatly simplifies the synchronization and control requirements.



Buffering is necessary between the two components to allow the consumer to more efficiently handle blocks of data from the producer.

```
int testInt;                          //Holding Variable
ConcurrentQueue<int> cqueue = new ConcurrentQueue<int>();   //Definition
cqueue.Enqueue(19);                   //Add Item
cqueue.TryDequeue(out testInt);       //Remove Item
```

Note: This structure is part of the System.Collections.Concurrent module (you need to include it to use it) and allows multiple threads to access the data contained in the queue. As the entire collection is "thread safe" no additional locking is required. MSDN Documentation for Concurrent Queue (http://msdn.microsoft.com/en-us/library/dd267265.aspx).

This collection can be of any type and allows you to add objects to the end, take objects from the start which implements a first in first out type of queue were old data is removed first. Additionally, this queue can be of a limited size or dynamically expanded as needed (elements can be added as required).

3 Mini-Projects

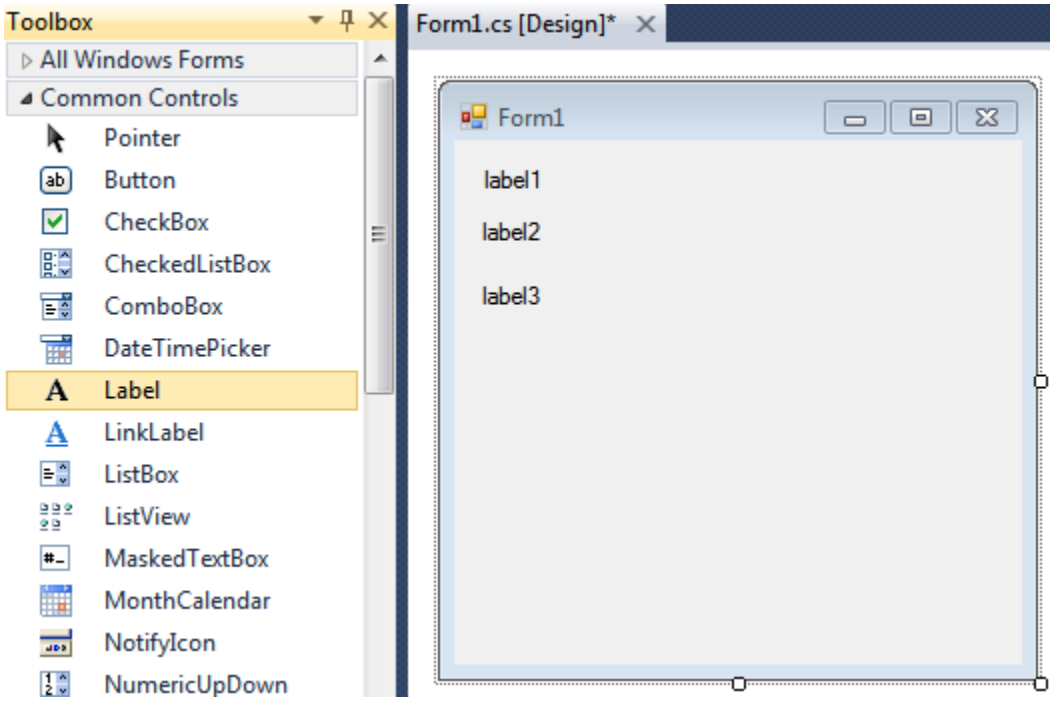
3.1 Mini-Project #1: the Visual C# programming interface

Goal: Write a C# program that shows mouse position on the screen in Textboxes and stores the time between each mouse movement in a Listbox.

In this mini-project we will write a simple C# program to track the horizontal and vertical location of the mouse cursor. Such a program could be used as part of a graphics program or games. We will also report the time interval between each movement of the mouse by the user using the PC clock.

3.1.1 Procedure for Mini-Project #1

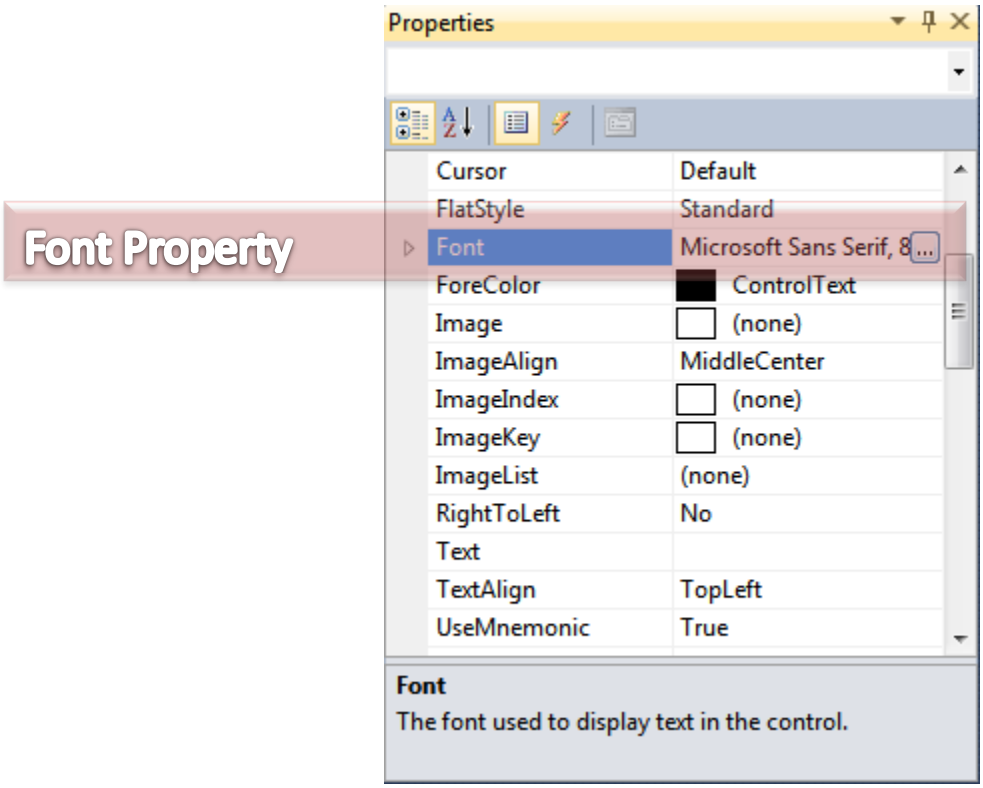
1. Start **New Project** and select Visual C# and Windows Forms Application
2. Name the project **Project 1**
3. Create three **labels** and select all three labels by holding the left mouse button and drag selecting or click on one label and then click on each other label with the control key held down on the keyboard.



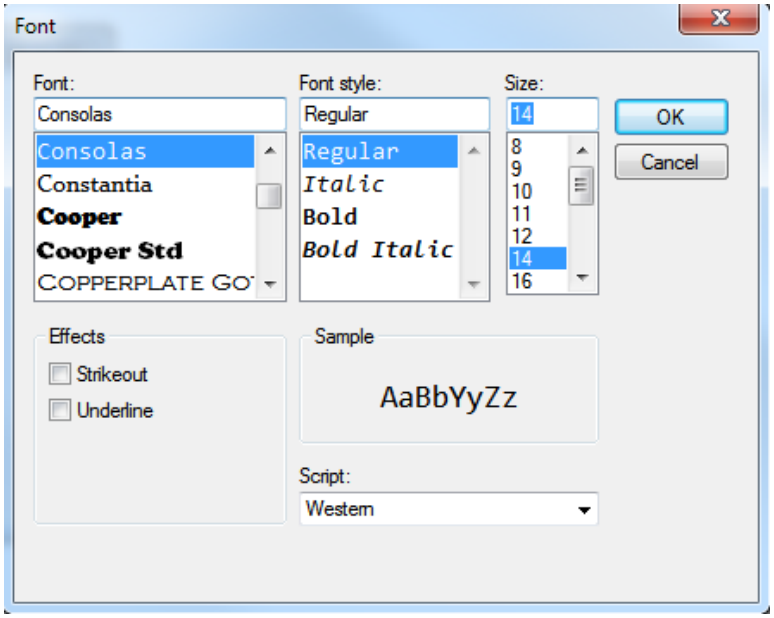
(Note with the control click selection method be careful not to accidentally select the form itself)

(Clicking a selected object while holding down the control key will unselect it)

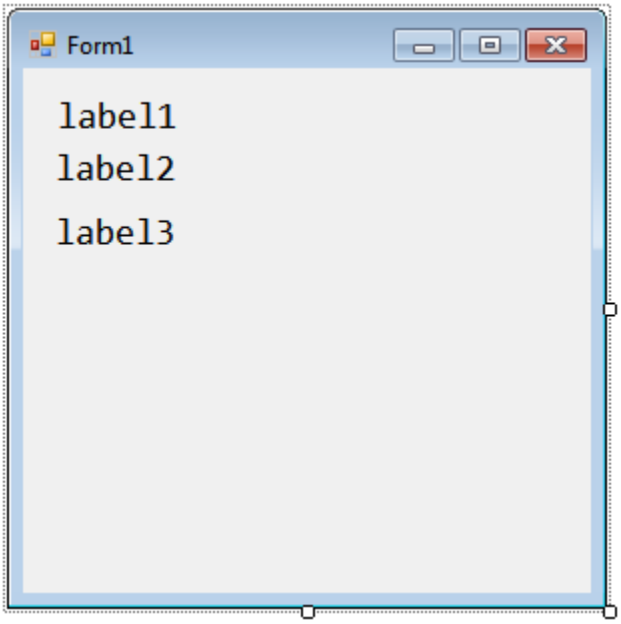
4. Now look at the properties pane for the labels and click on the "…" button on the font line.



5. Now look at the properties pane for the labels and click on the "…" button on the Font line. Change the font to **Consolas** and the size to **14** point.



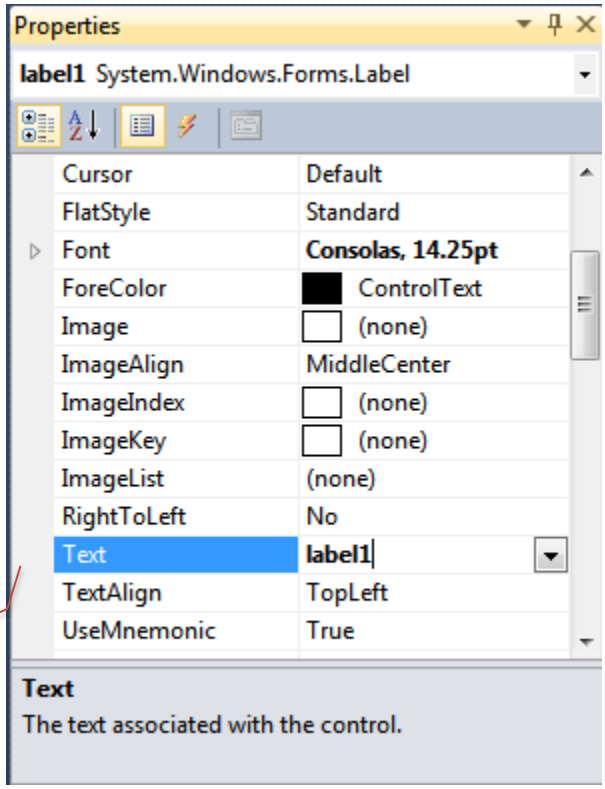
6. The **Form1.cs [Design]** tab should now look like this with all the selected labels updated to the new font property. Now click on the form to deselect the labels for the next step.



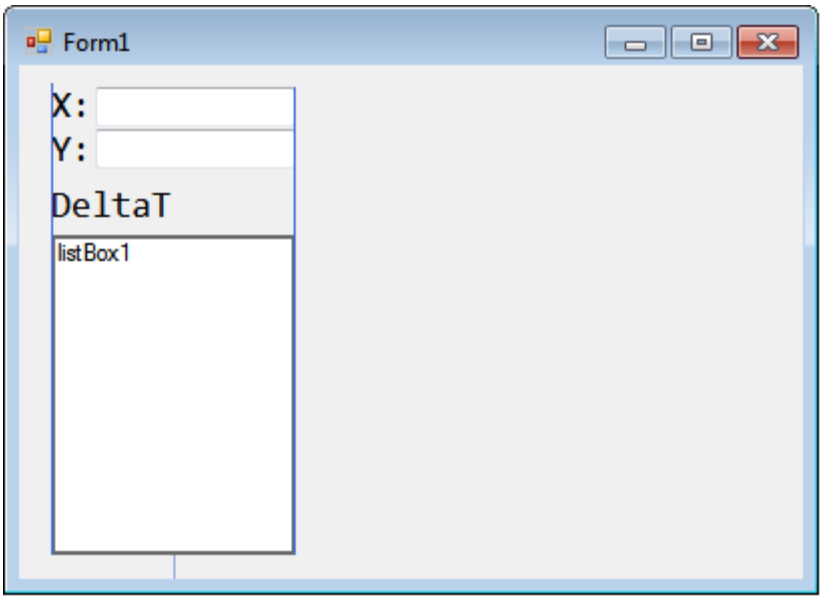
7. Change the **Text property** of the label1, Label2 and Label3 to X:, Y: and Delta T, respectively.

Note you should the control name for each selected label

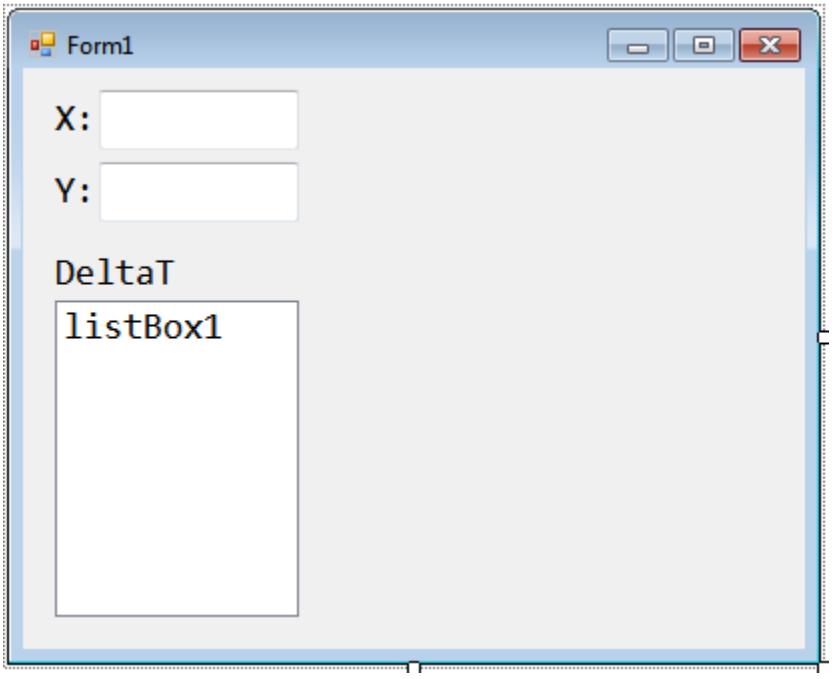
Changing the Text Property will change what is displayed on the UI



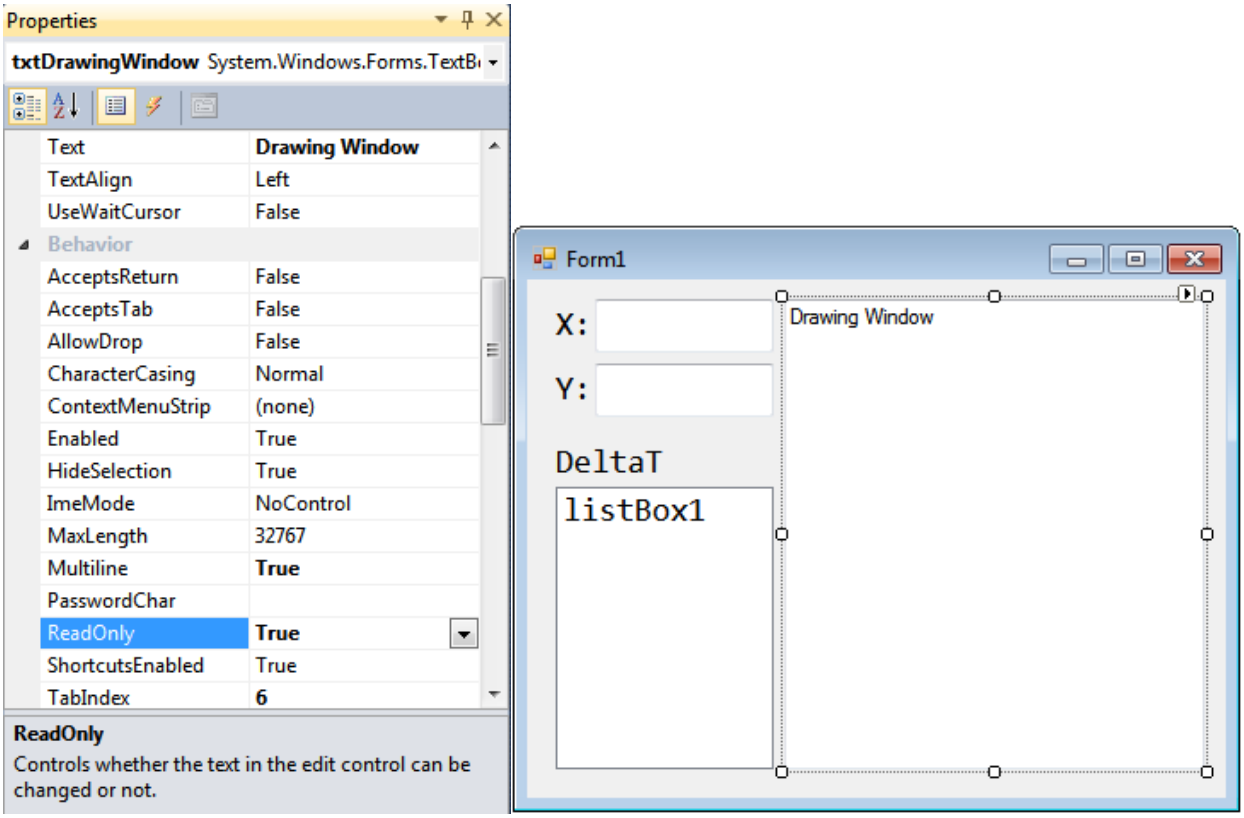
8. Add two **TextBoxes** and a **ListBox** and put them beside “X:”, “Y:” and below “Delta T”.



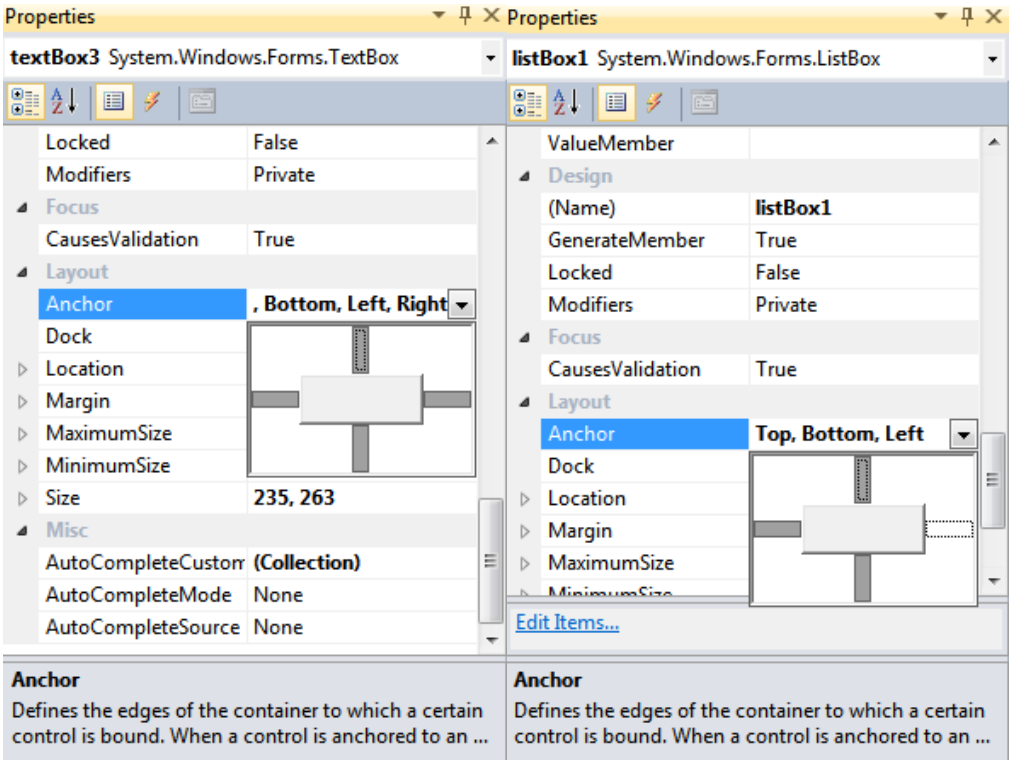
9. Change the size of the **Font** property of the boxes to **14** and **Consolas**. You will have to re-align the objects when you change the font size and the form should appear as follows.



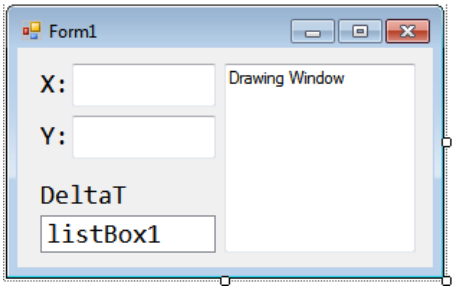
10. Now create a square textbox and enter the displayed text as “Drawing Window” and select **Multiline** and **ReadOnly** to true.



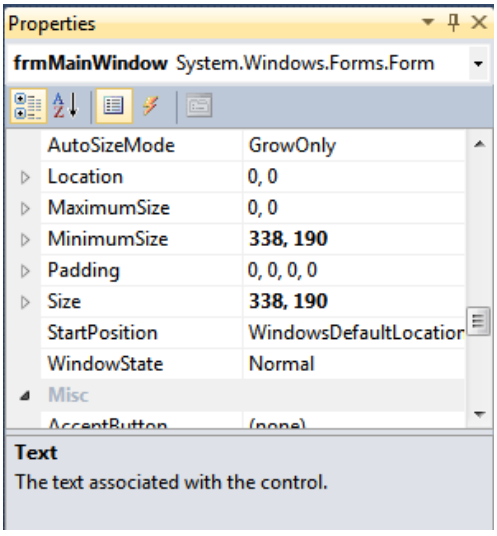
11. Then Select the textbox and change the **Anchor** to highlight all four directions. This will allow the drawing window to expand when the form window size changes. Also select the listbox and change its **Anchor** to “Top, Bottom, Left”.



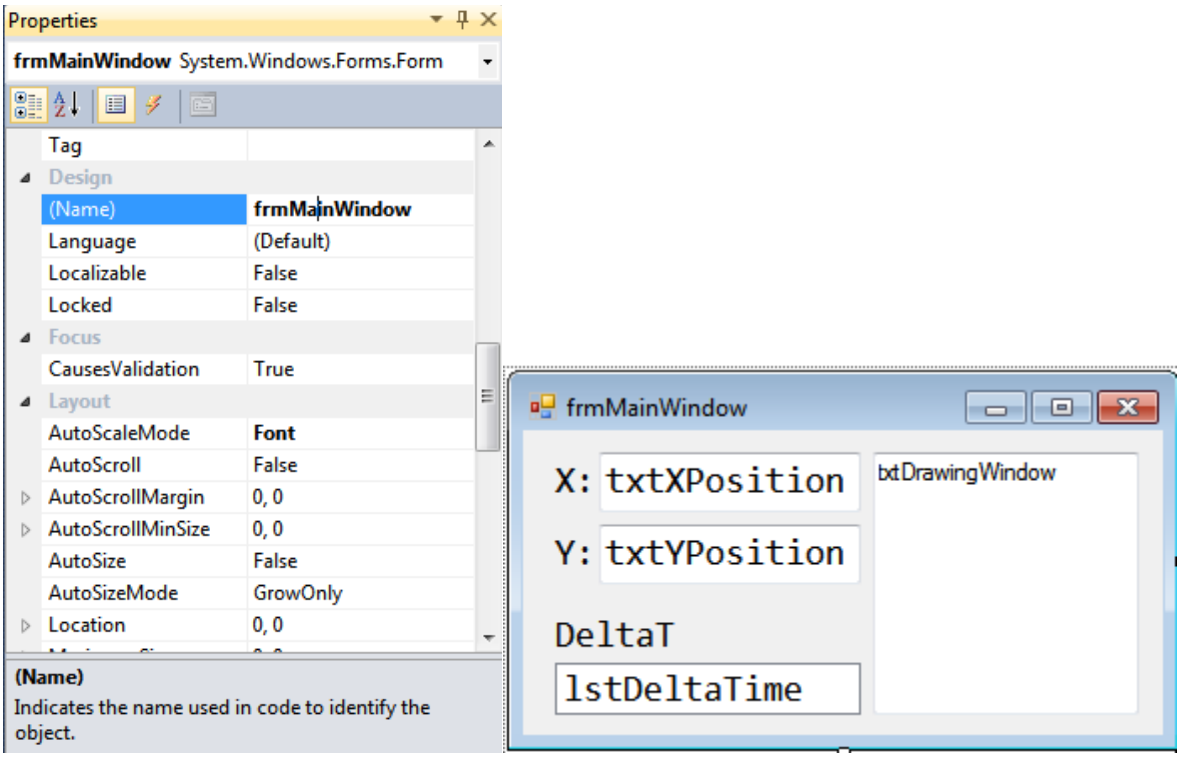
12. Try dragging the lower left corner of the form window to change the size in the design view and the listbox and drawing window should follow the size of the form. Then make the window as small as possible for the controls to still appear normally and the drawing window to be roughly square.



13. Select the form and change the **MinimumSize** property to match the current **Size** property displayed.

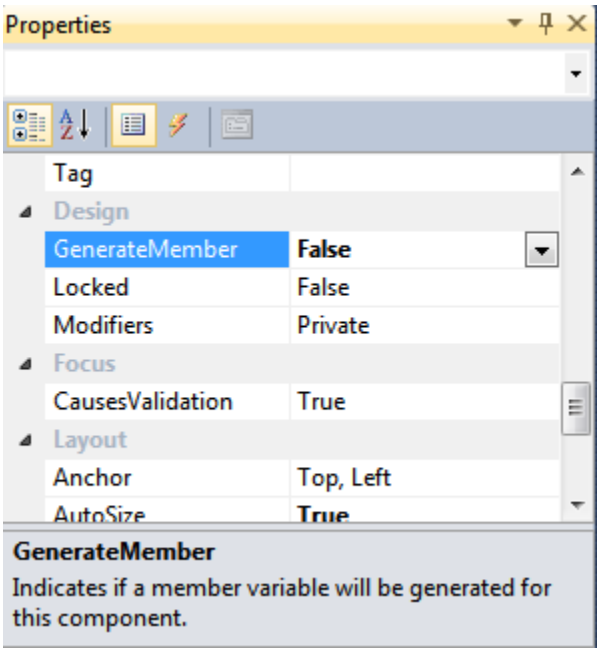


14. Before we continue coding, you should always give controls names that make sense by changing the design name (see top of next page). **Note that within your code you should not use prefixes to denote variable types as this is no longer considered acceptable coding standard under the latest .NET guidelines.** However, for UI elements using pre-fixes will help you code faster and designer generated elements are typically treated separately from your own code (http://support.microsoft.com/kb/173738).



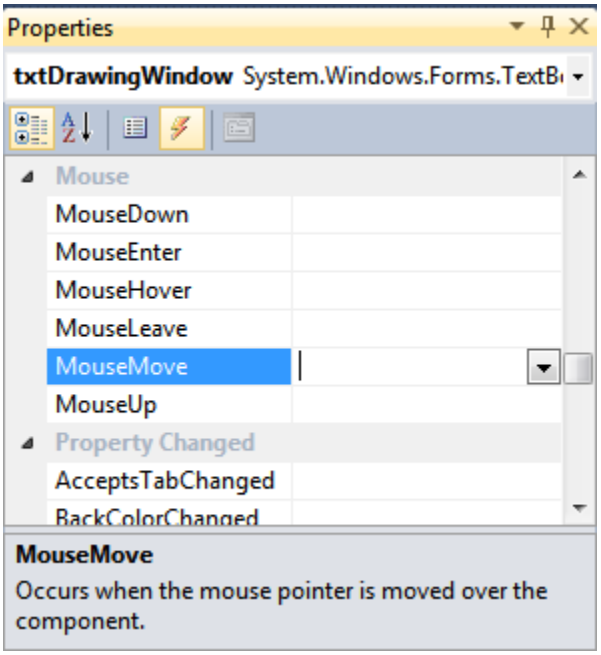
NOTE: The text fields were also changed to show what the design name should be you do not need to change the text property only the design name

15. Note that if you do not provide a name to a designer control you should change the following property to prevent clutter in your code. **(Disabling GenerateMember will prevent you from making changes to the control from code) Select the three labels and set generate member to false.**



16. Add **MouseMove** event (Double Click On Blank Event Entry). Then go to the **Form1.cs** tab and insert the following code inside **Private Sub Form1_MouseMove**.

```
txtXPos.Text = e.X.ToString("0000");
txtYPos.Text = e.Y.ToString("0000");
```



Libraries Used For Program

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
```

Automatically Generated Code

```
namespace CS_Project_1
{
    public partial class frmMainWindow : Form
    {
        public frmMainWindow()
        {
            InitializeComponent();
        }
```

Form Initialization (Don't Delete InitializeComponent())

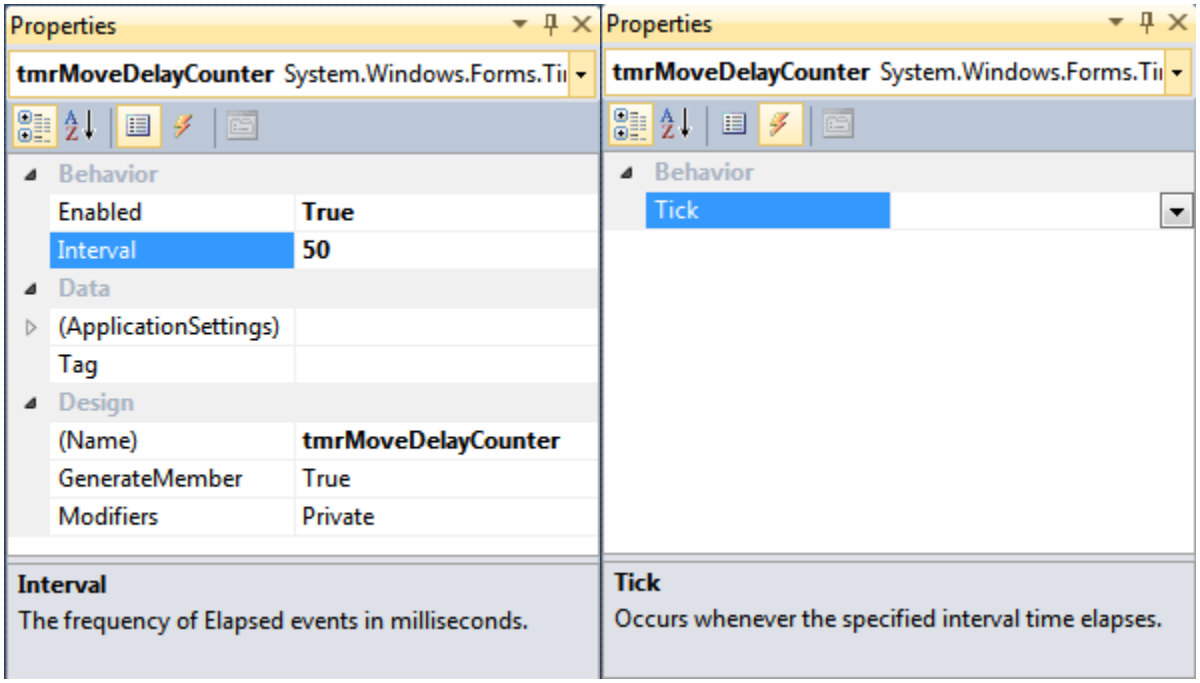
Event Handler

```
        private void txtDrawingWindow_MouseMove(object sender, MouseEventArgs e)
        {
            txtXPos.Text = e.X.ToString("0000");
            txtYPos.Text = e.Y.ToString("0000");
        }
    }
}
```

<Insert This Code Here>

e.X and **e.Y** are the arguments of event **MouseMove** that show the horizontal and vertical location of the mouse, respectively. Inserting that code, upon movement of the mouse, the X and Y position of the mouse is a shown in **txtXPos** and **txtYPos**.

17. Switch to your design page (**Form1.cs [Design]** tab) again and add **Timer** to your program from the components. From the **Properties** window, change the **Enabled** to true and **Interval** to 50 ms. And name it "tmrMoveDelayCounter". Then on the event pane window double click on the Tick Event.



18. Now you want to know the *Δt* between each movement of the mouse and print it on the **lstDeltaTime**. One way to do this, is to use the **tmrMoveDelayCounter_tick** for calculating the time and derive the difference between each movement. For this purpose add these variables in your code (in the **Form1.cs** tab), under (public partial class frmMainWindow : Form)

```
//Class Variables
int timeCounter;
int currentTime;
int itemLimit = 20;
```

The *counter* will be used to count the times that timer ticks with the interval of *the timer*. To do this, double click on the timer in your form, and under **tmrMoveDelayCounter_Tick** event add the line below.

```
timeCounter++;
```

Then you need to measure the time between each movement of the mouse. For this purpose, we modify what we put in **txtDrawingWindow_MouseMove** to:

```
//Calculate time delay between mouse moves
currentTime = timeCounter * tmrMoveDelayCounter.Interval;
lstDeltaTime.Items.Add(currentTime.ToString("000000"));
timeCounter = 0; //Restart Count
```

This code will put the actual time after mouse movement in *currentTime*. For showing *Δt* amounts in the **Listbox1**, we **add** them as **Lisbox1.Items** with the *listbox1.Items.Add* command. To get the delay to the next time the counter is reset after the update.

19. Now run the program.

20. You can limit the amount of the information you store in the list box and select the latest item with the following lines of code:

```
//Limit listbox items to item limit and remove old entries
while (lstDeltaTime.Items.Count > itemLimit)
{
    lstDeltaTime.Items.RemoveAt(0);
}

//Select the latest entry
lstDeltaTime.SelectedIndex = lstDeltaTime.Items.Count - 1;
```

The *lstDeltaTime.Items.Count* will count the number of the items that have been added to the **lstDeltaTime** and if it is greater than the itemLimit it will remove the oldest items from the **Listbox** with the *ListBox1.Items.RemoveAt (0)*.

Note that if you compute the difference in time instead of resetting the timer your program will crash after the integer overflows. (If the counter continuously increments even after mouse moves nothing will clear it and after 3 to four years the program will always crash) In this case the problem is trivial but in actual engineering systems you need to be careful as many installations may never be reset for their entire operating lifetimes.

In addition, the timer is not a reliable measure of time as windows is not a real time operating system and the consistent operation of the timer is not guaranteed.

3.1.2 Code for Mini Project #1

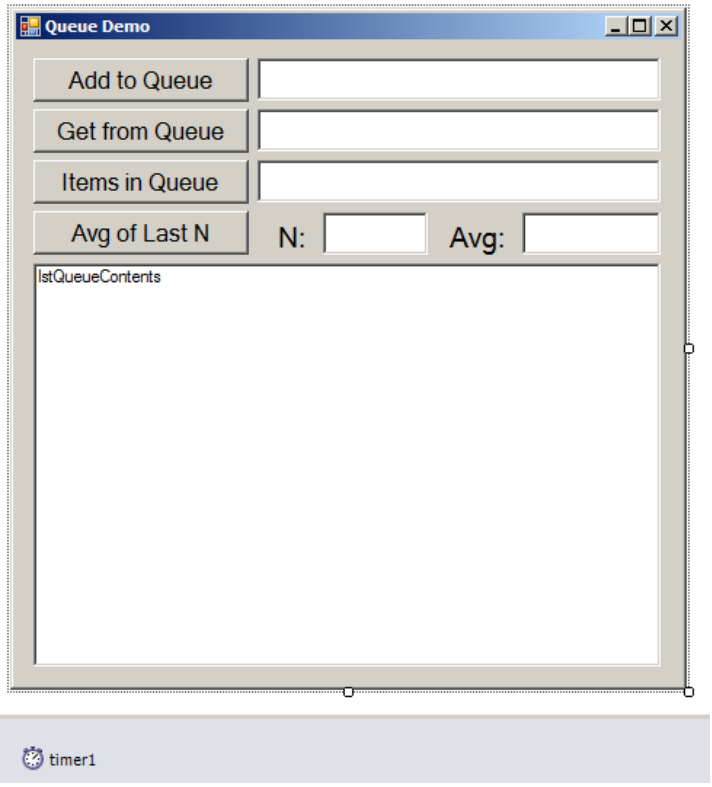
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace CS_Project_1
{
    public partial class frmMainWindow : Form
    {
        //Class Variables
        int counterValue;
        int currentTime;
        int itemLimit = 20;

        public frmMainWindow()
        {
            InitializeComponent();
        }

        private void txtDrawingWindow_MouseMove(object sender, MouseEventArgs e)
        {
            //Output mouse position to UI (Top Left corner is 0,0)
            txtXPos.Text = e.X.ToString("0000");
            txtYPos.Text = e.Y.ToString("0000");

            //Calculate time delay between mouse moves
            currentTime = counterValue * tmrMoveDelayCounter.Interval;
            lstDeltaTime.Items.Add(currentTime.ToString("000000"));
            counterValue = 0; //Restart Count

            //Limit listbox items to item limit and remove old entries
            while (lstDeltaTime.Items.Count > itemLimit)
            {
                lstDeltaTime.Items.RemoveAt(0);
            }

            //Select the latest entry
            lstDeltaTime.SelectedIndex = lstDeltaTime.Items.Count - 1;

        }

        //Increment the counter
        private void tmrMoveDelayCounter_Tick(object sender, EventArgs e)
        {
            counterValue++;
        }
    }
}
```

3.2 Mini-Project #2 Buffering Data

Goal: Write a circular buffer program in C# to get input characters from the user, add the characters into a Textbox, and then retrieve characters from the circular buffer.

Queues are a common structure in programs and this method will simplify circular buffer coding. (Easier)

1. Start a **new project** in C#.
2. Design your form like the one shown below.



3. Don't forget to name your UI elements with your own human readable labels.

4. Create a strongly typed concurrent queue called `dataQueue`.

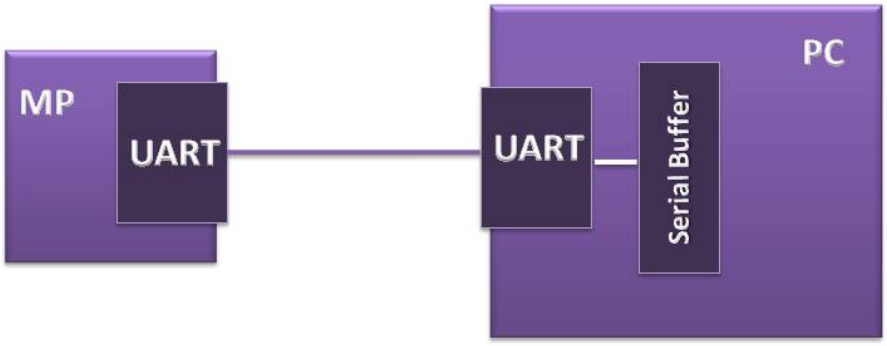
5. Using these support functions tie each button and the limit textbox to these methods.
    a. Add a tick event to update the list box with the contents of the queue,
    b. Add events for the buttons to add, remove, count, average items in the queue.

3.3 RS-232 Serial communications

Serial ports are one of the most common ways to communicate between PCs and peripheral devices. For example, a microprocessor can send sensor data for a PC to analyze and display. Or a PC can send commands to control robots or other devices. We now focus on the PC side of serial-port communications to use C# to access serial ports, read data from serial ports, and displaying it on the PC.

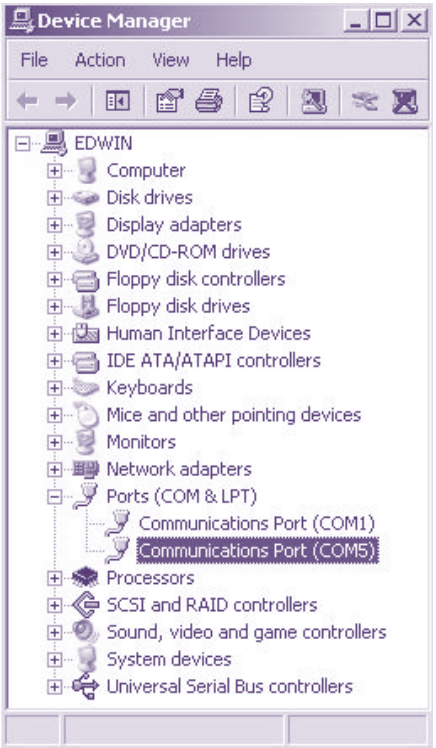
3.3.1 Transmitting and Receiving Serial Data

A universal asynchronous receiver/transmitter (UART) is a digital circuit block that manages serial communications between digital devices such as microprocessors and PCs. A UART is typically included within the integrated circuits of both microprocessors and PCs. At the transmission end, a UART takes bytes of data and transmits the individual bits in a sequential fashion. At the receiving end, a second UART assembles these bits back into complete bytes.



3.3.2 Serial Ports and COM Ports

A serial port refers to the UART interface built into a PC. Older PCs typically have separate serial interfaces where each interface is assigned a COM port number. Most newer PCs rely exclusively on USB ports, which can also be configured to operate as a virtual serial port and assigned a COM port number.



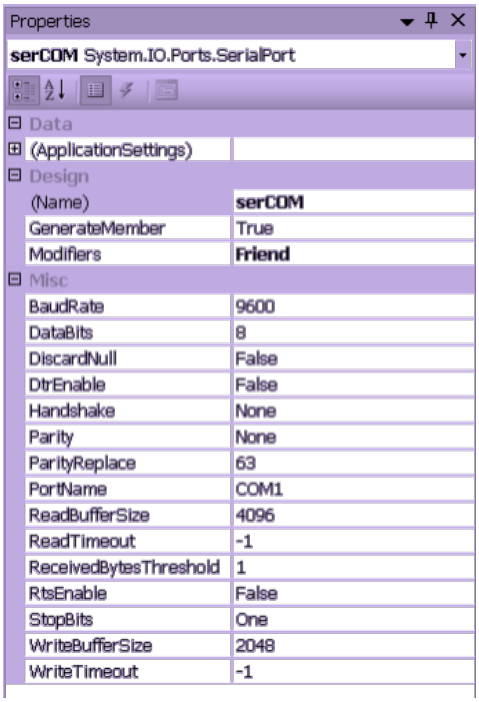
3.3.2.1 SerialPort Objects in C#

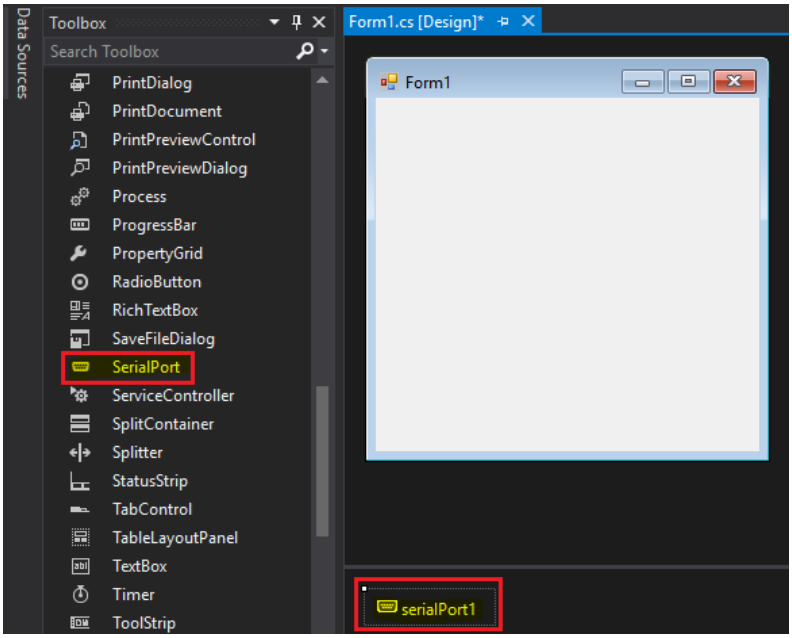
Serial ports on your computer can be accessed through the C# Framework classes in the **System.IO.Ports** namespace which contain classes for controlling serial ports. Data coming into the PC serial ports is first stored in its read buffer. The size of the read buffer is controlled by the **SerialPort.ReadBufferSize** property. When the number of data bytes in the read buffer exceeds the **SerialPort.ReceivedBytesThreshold**, the **SerialPort.DataReceived** event is triggered and the event handler is called. The SerialPort.Read method retrieves the data bytes from the serial port read buffer. The **SerialPort.Write** method writes a string to the serial port, while the **SerialPort.WriteLine** method writes a string with a newline value to the output buffer. For closing the port connection, the **SerialPort.Close** method can be used that sets the **IsOpen** property of the port to false.

3.3.2.2 SerialPort Properties in Detail

Serial port tool is added from the toolbox and treated like other objects like forms and listboxes. You can change the name of it to “serCOM” for instance and set the desirable properties. The serial port exposes members that you can see the widely used one in the table below. These properties can be changed in the property box of the “serCOM” object and defined explicitly in the C# code.

Name	Description
BaudRate	Gets or sets the serial baud rate.
BytesToRead	Gets the number of bytes of data in the receive buffer.
BytesToWrite	Gets the number of bytes of data in the send buffer.
PortName	Gets or sets the port for communications, including but not limited to all available COM ports.
DataBits	Gets or sets the standard length of data bits per byte.
DiscardNull	Gets or sets a value indicating whether null bytes are ignored when transmitted between the port and the receive buffer.
DtrEnable	Gets or sets a value that enables the Data Terminal Ready (DTR) signal during serial communication.
WriteBufferSize	Gets or sets the size of the serial port output buffer.
WriteTimeout	Gets or sets the number of milliseconds before a time-out occurs when a write operation does not finish.
Parity	Gets or sets the parity-checking protocol.
ParityReplace	Gets or sets the byte that replaces invalid bytes in a data stream when a parity error occurs.
ReadBufferSize	Gets or sets the size of the SerialPort input buffer.
ReadTimeout	Gets or sets the number of milliseconds before a time-out occurs when a read operation does not finish.
StopBits	Gets or sets the standard number of stopbits per byte.
IsOpen	Gets a value indicating the open or closed status of the SerialPort object.

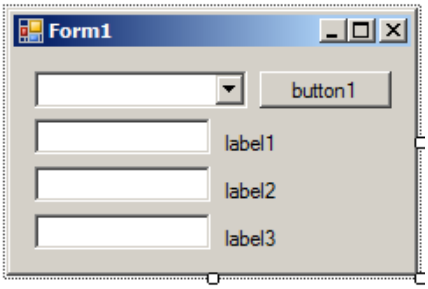




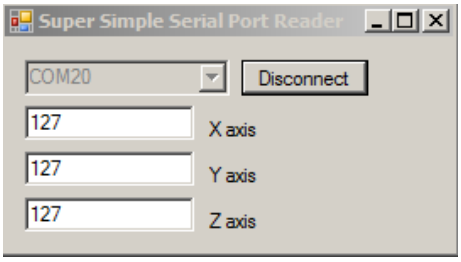
3.4 Mini-Project #3: Super Simple Serial Port Reader

This mini-project is designed to quickly enable you to read data from the default Gumstick firmware.

1. Start a project called “SerialReaderSuperSimple”.
2. Create an interface with a com port selection dropdown (**ComboBox**), a connect/disconnect **Button,** and a display for the current x,y,z values from the Gumstick.

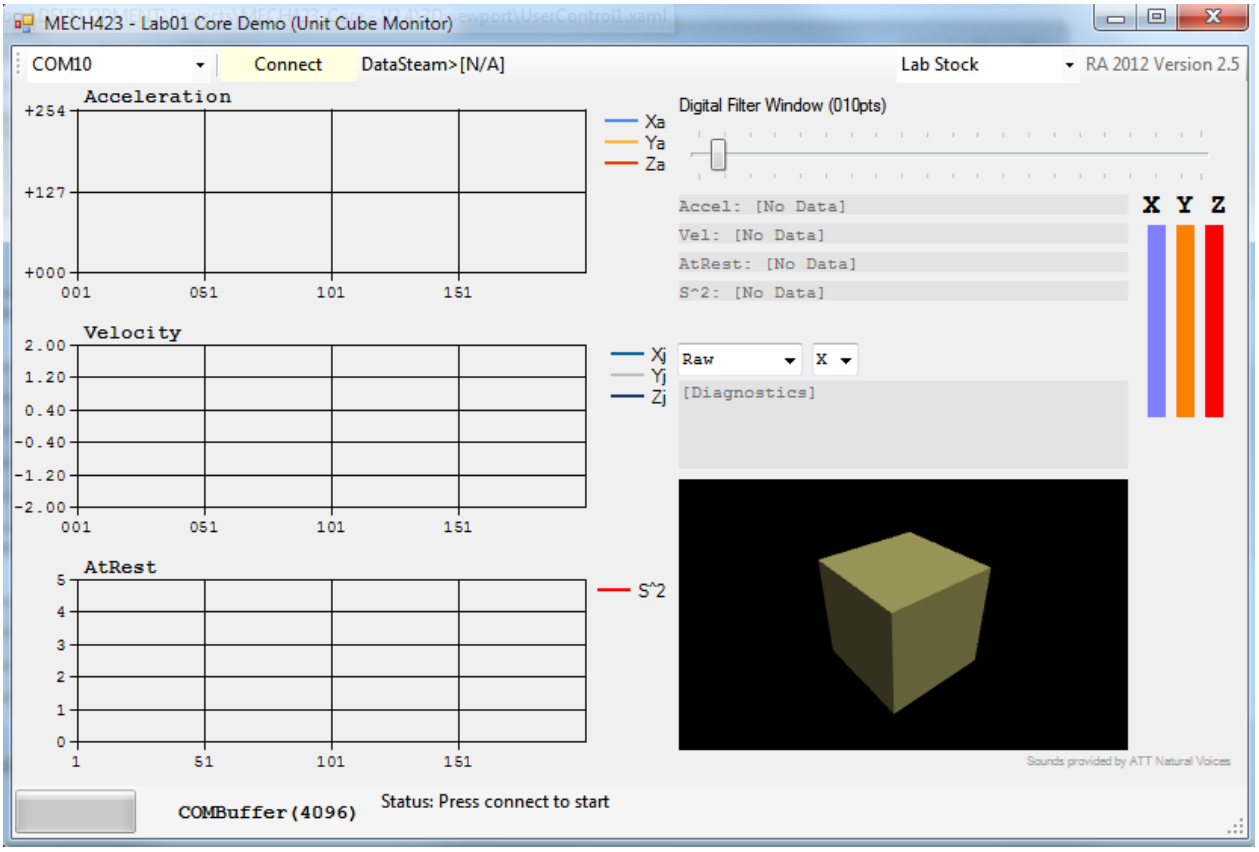


3. Use proper naming and create a **SerialPort** and **Timer** for data processing.
4. For the com port detection, use relevant events to update the drop down with the current list of ports using the following method: `System.IO.Ports.SerialPort.GetPortNames().ToArray()`
5. When the user presses the connect/disconnect button, check if the serial port is open/closed and change the text accordingly and perform the correct action depending on the state of the serial port.
6. Make sure you close the port if the program is closed before the user presses disconnect.
7. Use the serial port events to fill data into a queue and use the timer to flush the queue to the display.
8. Now test the program and make sure that the numbers make sense.



3.4.1 Putting it All Together

Now we are ready to combine all three mini-projects to create a final data acquisition program. Follow the instructions from the lab document. Feel free to make the user interface however you like. Here is an example:



4 Frequently Observed Issues

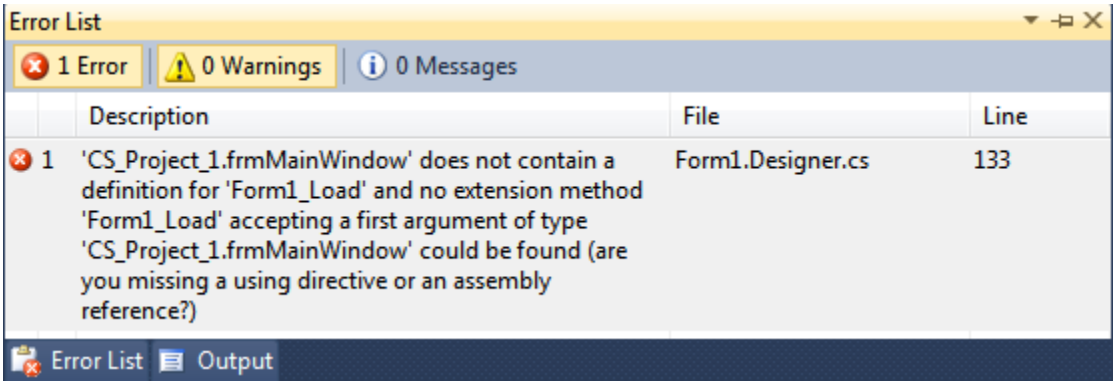
Here are some frequently observed issues and how to fix them.

4.1 Removing a designer generated event handler

Removing the generated code for an event handler will cause a complier error as there is some additional code in the background that connects the event handler code you deleted to the actual event.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace CS_Project_1
{
    public partial class frmMainWindow : Form
    {

        public frmMainWindow()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            //IF YOU DELETE ME HERE I WILL GET Visual Studio TO COMPLAIN
        }

        private void txtDrawingWindow_MouseMove(object sender, MouseEventArgs e)
        {
            txtXPos.Text = e.X.ToString("0000");
            txtYPos.Text = e.Y.ToString("0000");
        }
    }
}
```

As shown above the Form1_Load event will be deleted but the following error will occur.

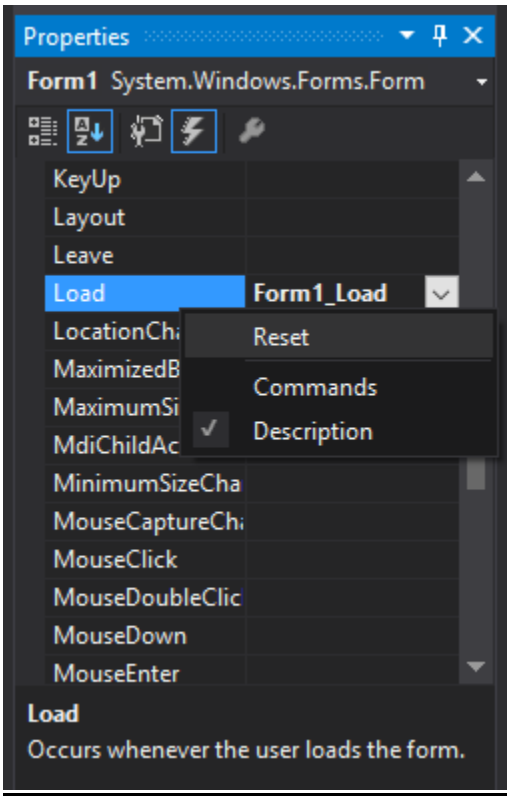


Double click on the error and you will automatically be taken to a new **Form1.Designer.cs** tab with highlight the line containing the error highlighted.

```
this.Load += new System.EventHandler(this.Form1_Load);
```

(WARNING: EDITING THE DESIGNER CODE CAN CAUSE THE DESIGNER VIEW TO STOP WORKING. MAKE CHANGES HERE VERY CAREFULLY, ONLY WHEN NECESSARY)

To properly delete an accidentally created event, go back to the **Form1.cs [Design]** tab, locate the event in **Properties**, right-click the event and click **Reset**.



4.2 Program hangs when closing

If your using the **SerialPort** object there can be many issues that can cause a deadlock to occur. Because of the "design" of the object, it doesn't respond or wait for itself to close properly when requested. This can cause issues when the serial port is interacting with the GUI or the form is trying to close the port.

To get around this issue, do not attempt to close the form with the port open (The user must have the port closed first).

This can be done using the form_closing even to close the port and cancel the close operation. After the port is closed then the program can be exited normally.