



**AUTODESK  
UNIVERSITY  
2005**

Walt Disney World Swan and Dolphin Resort  
Orlando, Florida

## **Automating Autodesk Inventor® Parameters with the API**

Brian Ekins - Autodesk

**MA31-4** This class provides a detailed look at Inventor's parameter system as exposed through Inventor's programming interface. The ability to access Inventor's parameters from a programming language opens doors to all kinds of automation -- from tasks as simple as exporting a set of parameters to an Excel spreadsheet to automating the motion of a complex assembly using custom rules. We will cover the concepts you need to know to create these types of programs. This class will benefit Inventor users who want to learn about available tools to add custom functionality to Inventor and increase productivity.

### **About the Speaker:**


Brian has been involved in the CAD industry for the past 25 years, with the past 10 years focused on various CAD programming interfaces. He has designed the APIs for Autodesk Inventor and other CAD applications. Brian is now working with Autodesk Consulting Services where he is able to combine his past engineering experience with his detailed knowledge of the Autodesk Inventor API to solve customers' problems.

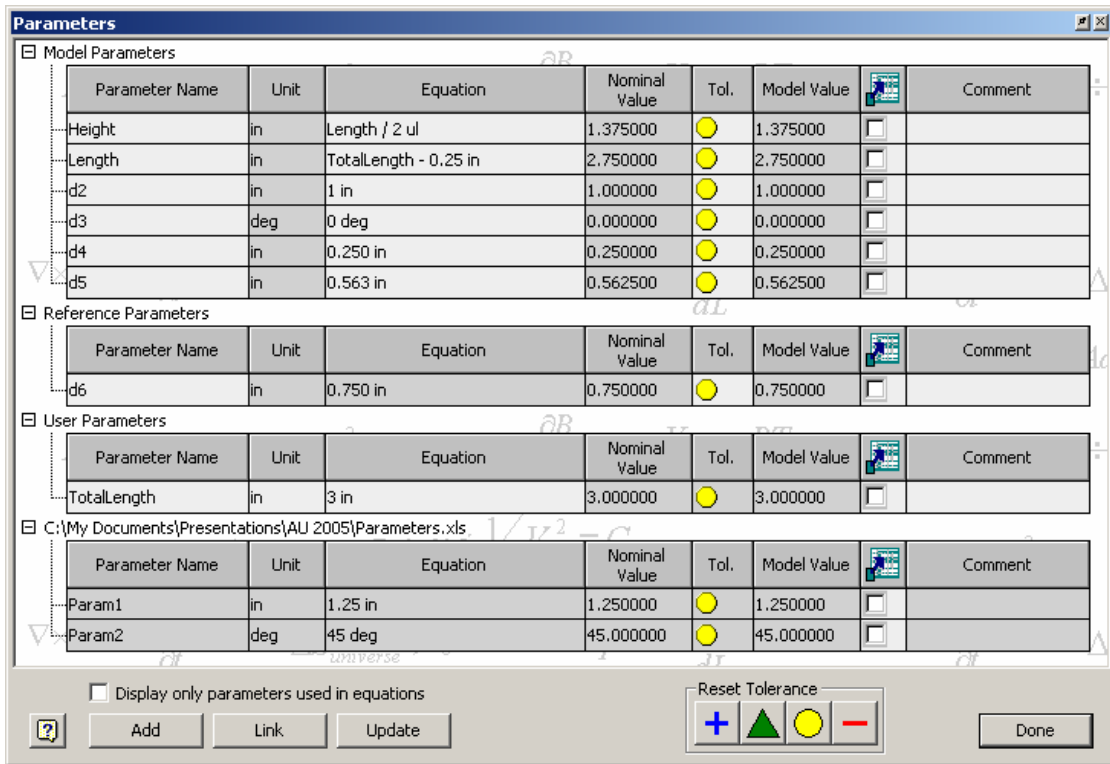
**brian.ekins@autodesk.com**



Autodesk Inventor is a *parametric modeler* which implies that parameters are a key component of the system. A primary use of parameters is to control the sizes of parts and the positions of parts within an assembly, although they can also be used in other ways. Before looking at the programming interface that provides access to the parameters lets first go through a brief overview of how parameters work from the perspective of an end-user.

## Parameter Functionality

Each part or assembly document has its own set of parameters. You access these parameters using the **Parameters** command , which displays the Parameters dialog as shown below. Let's look at the general functionality supported by all parameters.



**Parameter Name** – All parameters have a unique name. These names are editable by the end-user (except for table parameters where the name is controlled by the spreadsheet).

**Unit** – All parameters have a specified unit type. The unit type is editable for user parameters. For table parameters it is controlled by the spreadsheet. For all other types, the unit type is based on the current default units defined for the document.

**Equation** – The equation defines the value of the parameter. The equation is editable by the end-user for all parameter types except for reference and table parameters. The equation specified for a parameter must result in units that match the unit specified for that parameter. For example, if the parameters a, b, and c exist and are defined to be inch units, the equation “a \* b” is invalid for c because the resulting units of multiplying two length units together (inch) will be an area (square inches). The equation “a + b” is valid because the resulting unit is still a length (inch).

Equations can also call functions. For example “d0 \* sin(d5)” is a valid equation, assuming the parameters d0 and d5 exist and are the correct unit type. To find a description of all of the supported functions search for “functions in edit boxes” in Inventor’s online help. All of the functions listed can be used within parameter equations.

**Nominal Value** – The nominal value displays the resulting value of the equation.

**Tolerance** – The tolerance setting in the Parameters dialog specifies which value to use as the model value within the part or assembly. You can choose to use the nominal (which is the default), the upper, lower, or median value.

**Model Value** – The model value shows the resulting value after the tolerance setting has been applied to the nominal value. This value will be used when recomputing the part or assembly.


**Export Parameter** – The export parameter check box defines whether this parameter is exported as an iProperty or not. If checked, this parameter is exported as a custom iProperty and is also selectable when deriving a part.

**Comment** – The comment field allows the end-user to add a description to a parameter. The comment is editable for all parameter types except for table parameters which are controlled by the spreadsheet.

## Parameter Types

The parameters are grouped by type within the dialog. Let's look at each of these parameter types and their functionality.

**Model Parameters** – Model parameters are created automatically whenever you create something within Inventor that is dependent on a parameter, i.e. sketch dimensions, features, assembly constraints, iMates, etc. You cannot edit the unit type or delete model parameters.

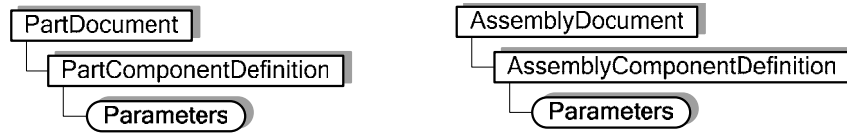
**Reference Parameters** – Reference Parameters are created automatically whenever a sketch dimension is created and is designated as a *driven* dimension. By default, dimension constraints are *driving* dimensions. That is, they drive or control the geometry. By toggling a dimension constraint to be *driven*, the geometry controls it. A reference dimension constraint can be created by placing a dimension constraint that will over-constrain the sketch. A warning dialog pops up, notifying you of the problem but allows you to proceed with the placement of the dimension as a driven dimension. You can also toggle any existing dimension constraint from driving to driven using the **Driven Dimension** command, . You cannot change the unit type or the equation of a reference parameter and you cannot delete a reference parameter.

**User Parameters** - User parameters are created by the end-user using the **Add** button on the Parameters dialog. You have complete control over user parameters and can edit the unit and equation and can also delete them.

**Table Parameters** – Table parameters are created automatically whenever an Excel spreadsheet is linked to the parameters using the **Link** button on the Parameters dialog. Each linked spreadsheet is listed in the parameters dialog with the parameters defined in that spreadsheet nested within that sheet in the Parameters dialog. You can't edit any of the information associated with a table parameter except to specify that you want to export it. Everything else is defined the by the spreadsheet.

## Parameters Programming Interface

Now let's look at the programming interface for parameters. The parameter functionality is exposed through the Parameters object. This object is obtained using the Parameters property of either the PartComponentDefinition or AssemblyComponentDefinition object as shown below.



The VBA code shown below obtains the Parameters object. It will work for either a part or assembly document.

```
Dim oParameters As Parameters
Set oParameters = ThisApplication.ActiveDocument.ComponentDefinition.Parameters
```

Below is an example that displays the number of parameters in the active document. It also incorporates error handling when attempting to get the Parameters object. The other samples have omitted error handling to make them easier to read.

```
Public Sub GetParameters()
    ' Get the Parameters object. This uses error handling and will only be successful
    ' if a part or assembly document is active.
    Dim oParameters As Parameters
    On Error Resume Next
    Set oParameters = ThisApplication.ActiveDocument.ComponentDefinition.Parameters
    If Err Then
        ' Unable to get the Parameters object, so exit.
        MsgBox "Unable to access the parameters. A part or assembly must be active."
        Exit Sub
    End If
    On Error Goto 0

    ' Do something with the Parameters object.
    MsgBox "The document has " & oParameters.Count & " parameters in it."
End Sub
```

Let's look at a simple program that uses the Parameter object to change the value of a parameter named "Length". Although it is simple, this program demonstrates the parameter functionality that is most commonly used and is frequently the only parameter functionality needed for many programs.

```
Public Sub SetParameter()
    ' Get the Parameters object. Assumes a part or assembly document is active.
    Dim oParameters As Parameters
    Set oParameters = ThisApplication.ActiveDocument.ComponentDefinition.Parameters


    ' Get the parameter named "Length".
    Dim oLengthParam As Parameter
    Set oLengthParam = oParameters.Item("Length")

    ' Change the equation of the parameter.
    oLengthParam.Expression = "3.5 in"







    ' Update the document.
    ThisApplication.ActiveDocument.Update
End Sub
```

This program starts by getting the Parameters object, just as the previous sample demonstrated. The Parameters object is a typical collection object. It supports the Count property which returns the number of items in the collection and it supports the Item property which lets you access the items within the collection. The Item property will accept a Long value indicating the index of the Parameter within the collection you want or it will also accept a String, which specifies the name of the Parameter you want. When specifying the name it is case sensitive. If the name you specify does not exist the call will fail. In this example, as long as there is a parameter named "Length", the call of the Item property will return the parameter and assign it to the variable oLengthParam.

In the next line the Expression property of the Parameter object is used to set the expression. The Expression property provides read and write access to the equation of the parameter. The terms "Expression" and "Equation" are synonymous when working with parameters. The expression is set using a String that defines a valid equation. The same rules apply here as when setting a parameter interactively in the Parameters dialog.

Just as when you edit parameters interactively, you need to tell the model to update to see the results. Interactively, you run the Update command . In the API, the equivalent is the Update method of the Document object.

Below is a chart that shows the parameter functionality and the equivalent API function.

<table border="1"> <tr><td>Parameter Name</td></tr> <tr><td>Height</td></tr> </table>	Parameter Name	Height	<p>Parameter.Name</p> <p>Ex: Parameter.Name = "Length"</p>
Parameter Name			
Height			
<table border="1"> <tr><td>Unit</td></tr> <tr><td>in</td></tr> </table>	Unit	in	<p>Parameter.Units</p> <p>Ex: Parameter.Units = "mm"</p>
Unit			
in			
<table border="1"> <tr><td>Equation</td></tr> <tr><td>Length / 2 ul</td></tr> </table>	Equation	Length / 2 ul	<p>Parameter.Expression</p> <p>Ex: Parameter.Expression = "Height/2 + 5 in"</p>
Equation			
Length / 2 ul			
<table border="1"> <tr><td>Nominal Value</td></tr> <tr><td>1.375000</td></tr> </table>	Nominal Value	1.375000	<p>Parameter.Value</p> <p>Ex: MsgBox "Nominal Value: " &amp; Parameter.Value &amp; " cm"</p>
Nominal Value			
1.375000			
<table border="1"> <tr><td>Tol.</td></tr> <tr><td></td></tr> </table>	Tol.		<p>Parameter.ModelValueType</p> <p>Ex: Parameter.ModelValueType = kUpperValue</p>
Tol.			
			
<table border="1"> <tr><td>Model Value</td></tr> <tr><td>1.375000</td></tr> </table>	Model Value	1.375000	<p>Parameter.ModelValue</p> <p>Ex: MsgBox "Model Value: " &amp; Parameter.ModelValue &amp; " cm"</p>
Model Value			
1.375000			
<table border="1"> <tr><td></td></tr> </table>		<p>Parameter.ExposedAsProperty</p> <p>Ex: Parameter.ExposedAsProperty = True</p>	
			
<table border="1"> <tr><td>Comment</td></tr> </table>	Comment	<p>Parameter.Comment</p> <p>Ex: Parameter.Comment = "This is the comment."</p>	
Comment			

There are a few new concepts to understand when working with parameters through the API versus using them in the user-interface. Some of these are also general concepts that apply to other areas of the API as well. Below is a discussion of some of the properties above and these new concepts.

## Expression

We looked at the Expression property in an earlier example. Remember that this is exactly the equivalent of the Equation field in the parameters dialog. Setting the Expression property has the same rules as entering it manually in the parameters dialog.

## Value

As shown in the chart above, the Value property is the equivalent of the Nominal Value field in the parameters dialog. However, there are some differences between the Nominal Value field in the dialog and the Value property of the API. First, whenever values are passed in the API, either in or out, they use Inventor's internal units; distance **values are always in centimeters** and **angle values are always in radians**. This isn't true of the Expression property since it is just a String that's describing a value. The Value property returns a Double value and is the real internal value of that parameter after its expression has been evaluated. This concept also applies when reading the ModelValue property.

The idea of a consistent internal unit system applies throughout the entire API and is not limited to parameters. For example if you get the length of a line through the API it will always be in centimeters regardless of the unit type the end-user has specified in the Document Options dialog.

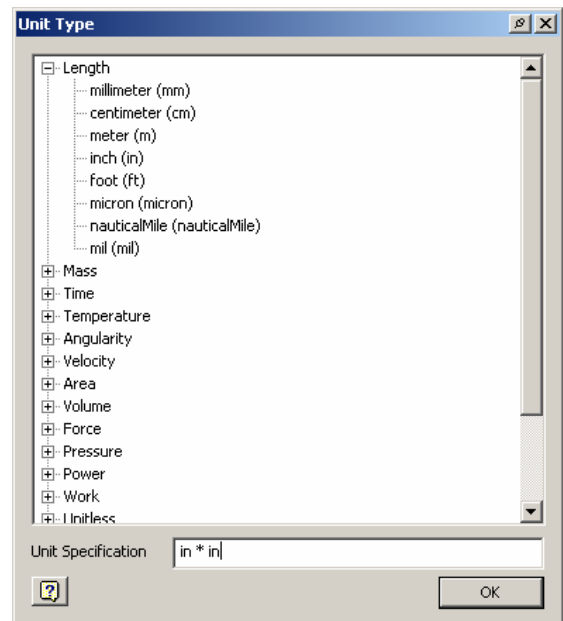
Another difference between the Nominal Value field in the dialog and the Value property of the API is that the Value property is editable. That is you can set the parameter using a Double value instead of having to create a string that defines your value. The same concept of internal units applies when setting the Value property; it is always in internal database units.

## Unit

The Unit property returns a String identifying the unit type associated with the parameter. The Unit property can be set using a String that defines a valid unit or you can use one of the predefined unit types in UnitsTypeEnum. The two statements below are both valid and have the same result.

```
Parameter.Unit = "in"
Parameter.Unit = kInchLengthUnits
```

The UnitsTypeEnum contains a list of the commonly used units. It's the same list of units that are available if you click on the Units field in the parameters dialog and the "Unit Type" dialog is displayed, as shown to the right. The primary benefit of using a String is that you can define unit types that are not available in the enum list. For example, square inches are not a predefined unit type but by specifying the string "in \* in" or "in ^ 2" you can define square inch units. A unit for acceleration could be defined as "(m / s) / s" or "m / (s^2)". Almost any engineering unit can be defined by combining the predefined base units.



## Tolerance

The Tolerance property of the Parameter object returns a Tolerance object. Using this object you can get and set the various tolerance options for the parameter. The Tolerance dialog shown below illustrates setting a tolerance for a parameter interactively. (You access this dialog interactively by selecting the Equation field of a parameter in the Parameters dialog and clicking the arrow at the right of the field to select the "Tolerance..." option.) This example sets a deviation type of tolerance with a

+0.125/-0.0625 tolerance value. It also sets the evaluated size to be the lower value and a precision of 4 decimal places. The code below uses the Tolerance object to define the same tolerance.

The line below sets the tolerance using Strings to define the tolerance values. When using a String the unit can be defined as part of the String or it will default to the document units.

```
Call oParam.Tolerance.SetToDeviation("0.125 in", "0.0625 in")
```

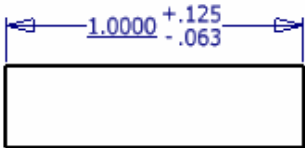
The line below accomplishes exactly the same result but specifies the tolerances using Double values. When providing a Double value, instead of a String, it is always in internal units. In this case the dimension is a length so it is in centimeters.

```
Call oParam.Tolerance.SetToDeviation(2.54 / 8, -2.54 / 16)
```

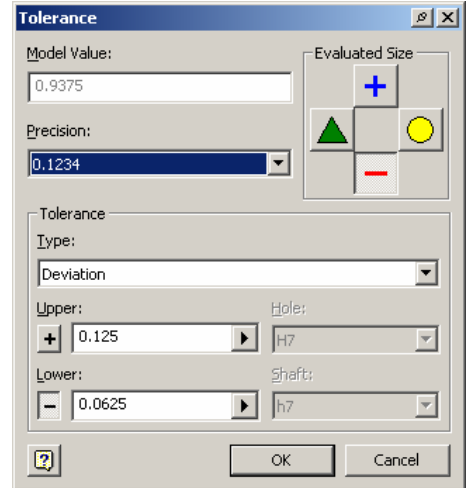
These last two lines define the number of decimal places to display for the model value and specifies that the lower value is to be used as the evaluated sized.

```
oParam.Precision = 4
oParam.ModelValueType = kLowerValue
```

Here's the result in the part.



Parameter Name	Unit	Equation	Nominal Value	Tol.	Model Value		Comment
d0	in	1 in	1.000000	-	0.937500		



### Creating Parameters

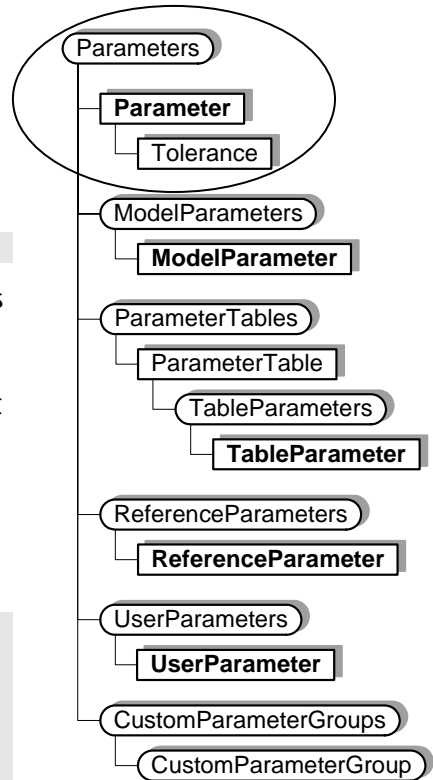
You may have noticed that so far in the discussion of the API we have not discussed the different types of parameters (model, user, reference, or table). The code we've looked at deals with all parameter types as a single generic "Parameter" type. The line of code shown below will work regardless of what type of parameter "Length" is.

```
Set oLengthParam = oParameters.Item("Length")
```

Shown to the right is the complete object model for parameters. The previous samples have used just the circled portion. The Item property of the Parameters collection object provides access to all of the parameters in the document, regardless of their type. Also from the Parameters collection object you can access other collections that contain the parameters of a specific type. The Item property of those collections will return only the parameters of that specific type. It's also through these type specific collections that you create new parameters. The sample below illustrates creating a user parameter. Notice that it uses the UserParameters collection object.

```
Dim oUserParams As UserParameters
Set oUserParams = oCompDef.Parameters.UserParameters

Dim oParam As Parameter
Set oParam = oUserParams.AddByExpression("NewParam1", "3", _
    kInchLengthUnits)
```



## Automating Autodesk Inventor® Parameters with the API

The example above uses the `AddByExpression` method of the `UserParameters` collection to create a new parameter called "NewParam1". The value of the parameter is "3" and the units are inches. Because the units are specified to be inches the value of "3" is interpreted to be 3 inches. Below is a similar example.

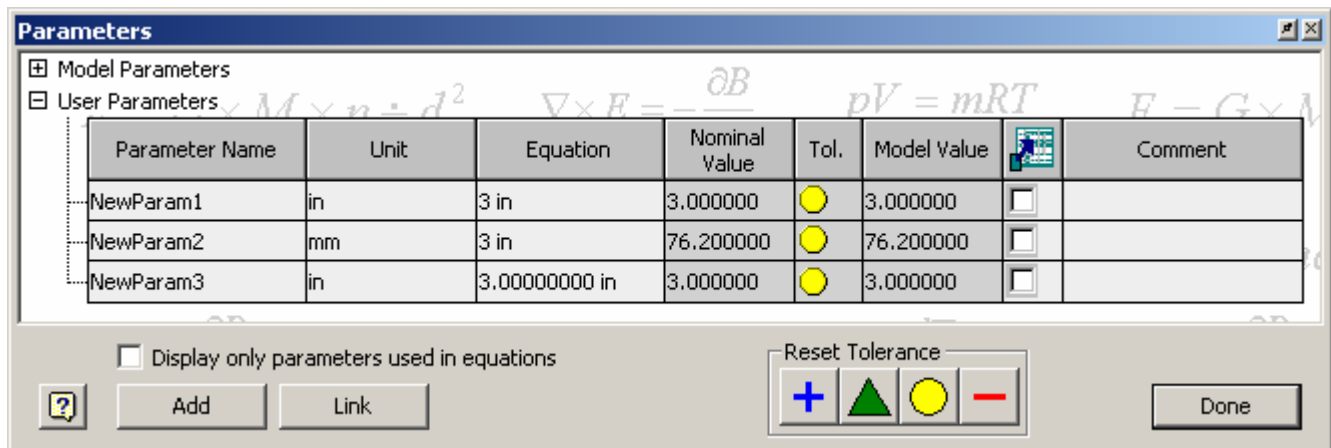
```
Set oParam = oUserParams.AddByExpression("NewParam2", "3 in", _  
                                         kMillimeterLengthUnits)
```

The example above uses the same method to create the parameter named "NewParam2". In this case the units are specified to be millimeters but because the expression also defines the unit, the resulting parameter value will still be 3 inches, but the parameter unit will be millimeters. The code below uses the `AddByValue` method to create a new parameter.

```
Set oParam = oUserParams.AddByValue("NewParam3", 3 * 2.54, kDefaultDisplayLengthUnits)
```

The `AddByValue` method is similar to the `AddByExpression` method except instead of the second argument being a String that defines the expression of the parameter it is a Double that defines the parameter value using internal units. This means that for lengths the value specified is **always** in centimeters. The units for `AddByValue` are defined just the same as they were for `AddByExpression`. However, this sample demonstrates the use of a special enum value within the `UnitsTypeEnum`. The unit type `kDefaultDisplayLengthUnits` specifies that the API is to use whatever the default length unit is for the document. This is the length that the user specifies in the Document Options dialog. When you use this enum, or the equivalent `kDefaultDisplayAngleUnits` for angles, you're not specifying a particular unit but only the type of unit; typically length or angle. The specific unit type is picked up from the document. Since the value you specify is always in internal units, (centimeters or radians), it doesn't matter to you what specific units the end-user has chosen to work in.

The end result of the previous lines of code creates the parameters shown below. The default units for the document are inches. That's why NewParam3 has the unit type "in".



Through the user-interface you can only create user and table parameters. (Table parameters are created by linking an Excel spreadsheet.) Model and reference parameters are created automatically as you add dimension constraints to the model. Through the API it is possible to create model and reference parameters. They're created the same way as user parameters except you use the `Add` methods of the `ModelParameters` and `ReferenceParameters` objects. For most programs you will want to create user parameters. Model and reference parameters are typically created by applications that want to expose values to the end-user with the behavior of model or reference parameters.

## Units of Measure

There's another area of the API that can be useful when working with parameters. The UnitsOfMeasure object provides the equivalent functionality of the Units tab of the Document Settings dialog. In addition to this it supports several utility methods to allow you to work with equations in your programs. The UnitsOfMeasure object is obtained using the UnitsOfMeasure property of the Document object. Each document has its own UnitsOfMeasure object since each document can have different default units defined. The following illustrates getting the UnitsOfMeasure object when a form is initialized and assigning it to a global variable within the form.

```
Private m_oUOM As UnitsOfMeasure

Private Sub UserForm_Initialize()
    Set m_oUOM = ThisApplication.ActiveDocument.UnitsOfMeasure
End Sub
```

Typically the UnitsOfMeasure object is used whenever you need to interact with the end-user and have them input values or you need to display a value back to the end-user. For example, perhaps your program displays a dialog to allow the end-user to specify a length in a part. If you want your program to have the same behavior as standard Inventor commands you should support units and equations. For example, if the end-user types "4", how would you handle it? Or if they type "4 cm + d0" what would you do? It would be a lot of work to correctly handle these inputs if you had to do it on your own. Instead you can take advantage of the UnitsOfMeasure object and let it do the work for you. The examples below described and illustrate the most common uses of this object.

The code below illustrates using the CompatibleUnits method to check if an expression defines a valid length. It's implemented in the Change event of a text box. If the expression isn't valid it changes the text color of the text box to red to indicate to the end-user the expression is invalid.

```
Private Sub TextBox1_Change()
    On Error Resume Next
    ' Check that the expression is a valid length.
    Dim bCompatible As Boolean
    bCompatible = oUOM.CompatibleUnits(TextBox1.Text, kDefaultDisplayLengthUnits, _
        "1", kDefaultDisplayLengthUnits)

    ' Check if it was successful and change the text to the appropriate color.
    If Err Or Not bCompatible Then
        TextBox1.ForeColor = vbRed
    Else
        TextBox1.ForeColor = vbWindowText
    End If
End Sub
```

The code below illustrates using the GetValueFromExpression method to evaluate an equation and get the resulting value. The first argument is the String that represents the equation. The second argument specifies the unit type that the expression should represent. In the example below it's specifying that the expression should be whatever the current default length units for the document are. You would use kDefaultDisplayAngleUnits if the expression represents an angle. The value passed back is in database units.

```
' Get the real value of the input string.
Dim dValue As Double
dValue = oUOM.GetValueFromExpression(txtBox1.Text, kDefaultDisplayLengthUnits)
```

Let's look at an example of input and the corresponding output from the previous example. If the user enters "5" in your dialog's text box and the current length units for the document are inches it is interpreted by the `GetValueFromExpression` method to be 5 inches. The value returned is 12.7 which is the equivalent of 5 inches in centimeters. The great thing about this method is that you don't need to do anything special to handle the various units and equations. You just pass any equation into the function and as long as Inventor can evaluate it, it will pass back the result.

Another common use of the `UnitsOfMeasure` object is to display results back to the end-user. If you perform a calculation in your program that is in a known unit you can use the `UnitsOfMeasure` object to format a `String` to display to the end-user. The example below illustrates this.

```
MsgBox "Length: " & oUOM.GetStringFromValue(dValue, kDefaultDisplayLengthUnits)
```

This example uses the `GetStringFromValue` method to get a correctly formatted `String` to display to the end-user. For example, if you've calculated the length of an object and have the value, represented by the variable `dValue` in the sample, this method will return a `String` that represents that value in the current default length units of the document. Remember that the units for a length input value will always be centimeters and for an angle will always be radians. In this case if you passed in the value 12.7 and the current length unit of the document is inches, Inventor will return the string "5.00 in".

This also helps to illustrate the power of the `UnitsOfMeasure` object and how easy it makes interacting with the end-user. In this case you don't need to worry about any of the document units. The end-user can specify any unit type for the document and it doesn't matter to you. You just do your calculations in internal units and anytime you need to display a result to the end-user you pass in the value and Inventor provides the correctly formatted `String` to display to the end-user.

### Putting it all Together

The parameters portion of the API is fairly simple but it can become powerful when you show your creativity in applying this functionality towards solving your specific problems. The power in using the API with parameters is that it allows you to quickly change one or more parameters based on any input you choose. Here are some examples of potential uses of the parameter portion of the API.

Delivered with Inventor is a programming sample whose primary function is to change parameter values. It also takes advantage of other API functionality to create a complete set of functionality. The sample program is delivered in the `SDK\Samples\VB\AddIns\Analyze` directory where Inventor was installed. The `readme.txt` file describes how to run the sample. In this sample you choose from the parameters in the active document and for each parameter choose a starting and ending value and the number of steps. The program then computes intermediate values for the parameters between the starting and ending values. It assigns the values to the parameters for each step and updates the assembly. The result is that you see the assembly animating as it is driven by the parameters.

Another common use of the parameter portion of the API is to drive the sizes of parts and assemblies based on external input. Most commonly this is done for some type of configuration application. For example, an order for a particular product can be made and communicated to you. Based on this order you are able to determine which parts are needed to create the specified product. This logic is up to you but could be based on information in a database or just logic within your program. You can then edit the parameters of parts and assemblies to get the desired sizes and create the desired finished product.

