Autodesk Inventor® Programming Fundamentals with iProperties

Brian Ekins – Autodesk, Inc.

This paper provides an introduction to Inventor's VBA programming environment and the basic concepts you need to understand when programming Inventor. These concepts are put into practice as we look in detail at the iProperties portion of the programming interface and develop some practical examples.

Key Topics:

- ❑ The basics of Visual Basic for Applications (VBA)
- ❑ Understand how Inventor provides a programming interface
- ❑ Object-oriented programming
- ❑ Inventor's object model
- ❑ The iProperties portion of the object model
- ❑ iProperty program examples

Target Audience:

Autodesk Inventor users who want to increase their productivity with Inventor by writing programs

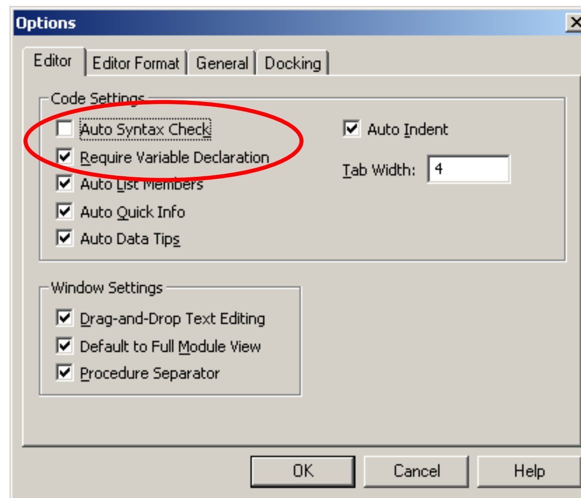
Introduction

The goal of this paper is to provide enough information that anyone will be able to write a program to automate working with Inventor's iProperties. This paper is not intended to be a comprehensive coverage of VBA programming or even iProperties, but focuses on the features that are most commonly used. We'll begin by looking briefly at VBA and the steps required to begin writing a program. Then we'll discuss the concepts around Inventor's programming interface. Next we'll look at the interface for programming iProperties. Finally, we'll look at several examples that demonstrate all of these concepts.

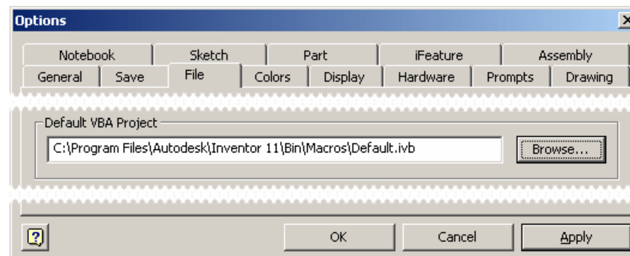
VBA Basics

Visual Basic for Applications (VBA) is a programming environment developed by Microsoft and licensed by other companies to integrate into their applications. Autodesk licenses VBA and includes it in AutoCAD and Inventor.. This provides all customers of Inventor with a free development environment. Below are the minimal steps to begin using VBA and writing a simple macro.

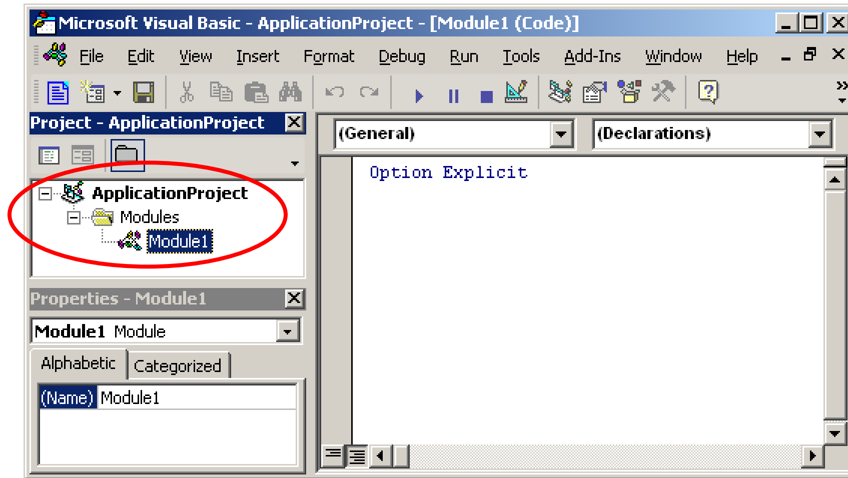
You access VBA through Inventor using the **Macro | Visual Basic Editor** command in the Tools menu, or by pressing Alt-F11. Once the VBA environment is open, the first thing I recommend you do is change some of the VBA settings. In the VBA environment run the **Options** command from the Tools menu. Change the **Auto Syntax Check** and **Require Variable Declaration** settings to those shown below.



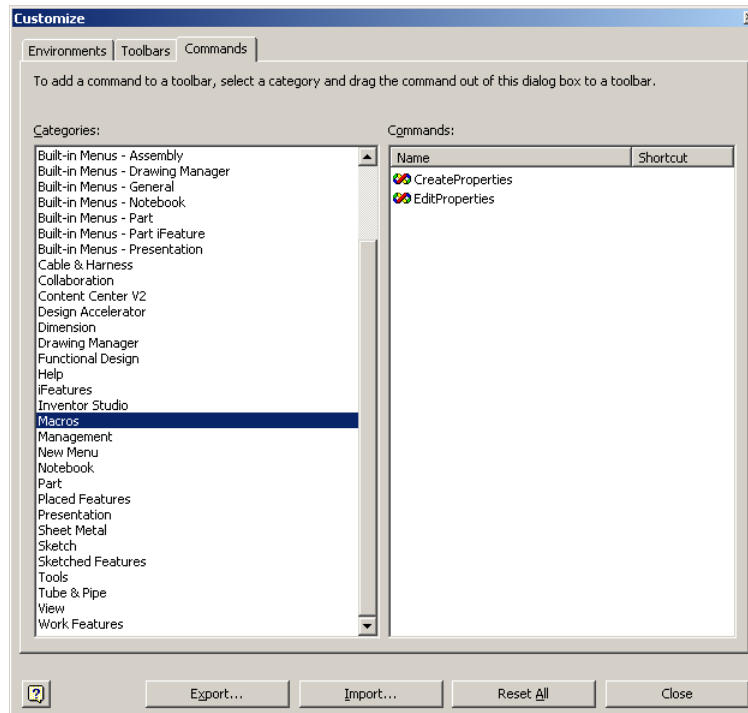
The VBA programs you write can be saved in Inventor documents or in external files. For 99% of the cases, saving it in an external file is best, so that's what we'll cover here. This external file is referred to as a VBA project and Inventor saves it as an .ivb file. A single project can contain any number of macros (programs). There is a default VBA project that Inventor loads at startup. The default VBA project is defined in the File tab of the Inventor application options, as shown below. This is where your programs will be saved.



A project can consist of many different types of modules. Macros are written in a code module within the VBA project. There is a code module automatically created for any new VBA project named "Module1". To add code to this module you can double-click on the module in the Project Explorer window, as shown below. This will cause the code window for that module to be displayed.



There are several ways to run a VBA macro. From the VBA editor you can position the cursor in the code window within the macro you want to run and execute the **Run Macro** command ▶ on the main toolbar. From within Inventor you can run the **Macro | Macros…** command from the Tools menu (or Alt+F8) which brings up a dialog where you can choose a macro and run it. Finally, you can use the Inventor **Customize** command to create a button that will run a macro. For frequently used macros this provides the most convenient interface for the end-user. This is done using the **Commands** tab of the Customize dialog, as shown below. Select "Macros" as the category. The available macros are listed on the right-hand side of the dialog. To create a button to execute the macro, just drag and drop the desired macro onto any existing toolbar.



The Basics of Inventor's Programming Interface

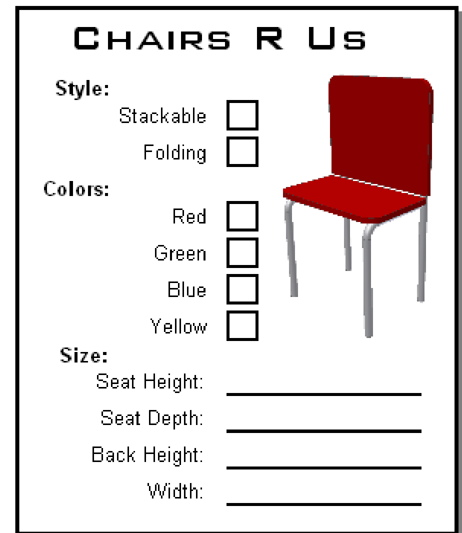
Inventor supports the ability for you to automate tasks by writing programs. Inventor does this using some Microsoft technology called COM Automation. This is the same technology used when you write macros for Word or Excel. The technology itself isn't important, but what is important is what it allows you to do.

COM Automation allows applications, like Inventor, to provide a programming interface in a fairly standard structure that can be used by most of the popular programming languages available today. This provides two benefits to you. First, if you've programmed other applications that use COM Automation (Word, Excel, AutoCAD) Inventor will have a lot of similarities. Second, it's very flexible in how you access the programming interface so you can choose your favorite programming language and integrate with other applications.

The topics below discuss the basic concepts you'll need to understand when working with Inventor (or any COM Automation programming interface).

Objects

A COM Automation interface exposes it functions through a set of *objects*. A programming object has many similarities to real-world objects. Let's look at a simple example to understand how object oriented programming terminology can be used to describe a physical object.



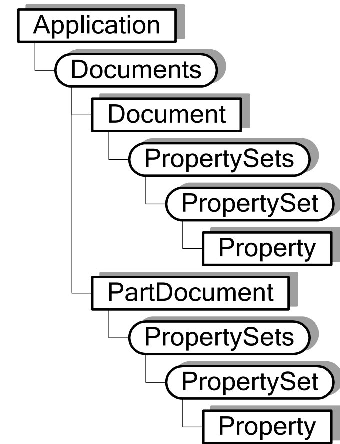
A company that sells chairs might allow a customer to design their own chair by filling out an order form, like the one shown to the right. The options on the order form define the various *properties* of the desired chair. By setting the properties to the desired values the customer is able to describe the specific chair they want.

In addition to properties, objects also support *methods*. Methods are basically instructions that the object understands. In the real world, these are actions you would perform with the chair. For example, you could move the chair, cause it to fold up, or throw it in the trash. In the programming world the objects are smart and instead of you performing the action you tell the object to perform the action itself; move, fold, and delete.

In Inventor, some examples of objects are extrude features, work planes, constraints, windows, and iProperties. Inventor's programming interface defines the set of available objects and their associated methods and properties. By obtaining the object you're interested in you can find out information about it by looking at the values of its properties and edit the object by changing these values or by calling the objects methods. The first step in manipulating any object is getting access to the object you want. This is explained in the next section.

The Object Model

As discussed above, Inventor's API is exposed as a set of objects. By gaining access to these objects through the API you can use their various methods and properties to create, query, and modify them. Let's look at how the *object model* allows you to access these objects. Understanding the concept of the object model is a critical concept to using Inventor programming interface.



The object model is a hierarchical diagram that illustrates the relationships between objects. A small portion of Inventor's object model is shown in the figure to the right. Only the objects relating to iProperties are shown. In most cases you can view these as parent-child relationships. For example, the Application is the parent of everything. The Document object is a parent for the various property related objects. To obtain a specific object within the object model you typically start at the top and then go down child-by-child to the object you want. For example, you would start with the Application object and from it get the Document that contains the iProperty you're interested in. From the Document you then get the PropertySet that is the parent of the iProperty and finally you get the desired Property object.

The Application Object

One of the most important objects in the object model is the *Application* object. This object represents the Inventor application and is the top-most object in the hierarchy. The Application object supports methods and properties that affect all of Inventor but its most important trait is that through it you can access any other Inventor object. You just need to know how to traverse the object model to get to the specific object you want. We'll look at how to write a program to do this and some shortcuts you can take to make it a little bit easier.

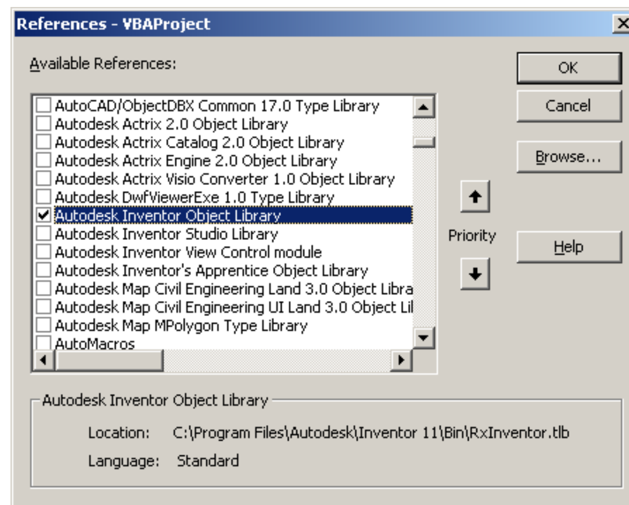
In order to use the Application object you first need to get it. How you get it depends on what programming language you're using. The code below illustrates three of the most common techniques.

Inventor VBA

When using Inventor's VBA there is the global variable called ThisApplication that always contains the Application object. So with Inventor's VBA you don't need to do anything but use this variable.

Visual Basic 6 or another VBA (i.e. Excel's VBA)

When accessing Inventor's API from Visual Basic 6 or VBA within another application there are a couple of steps you need to perform. The first is to make VB/VBA aware of Inventor's objects. You do this by referencing Inventor's type library. To reference a type library you use the **References** command. In VB the **References** command is found under the Project menu. In VBA it is found under the Tools menu. In both cases the References dialog, as shown below, is displayed. Find the entry "Autodesk Inventor Object Library", check the box next to it and click the "OK" button.



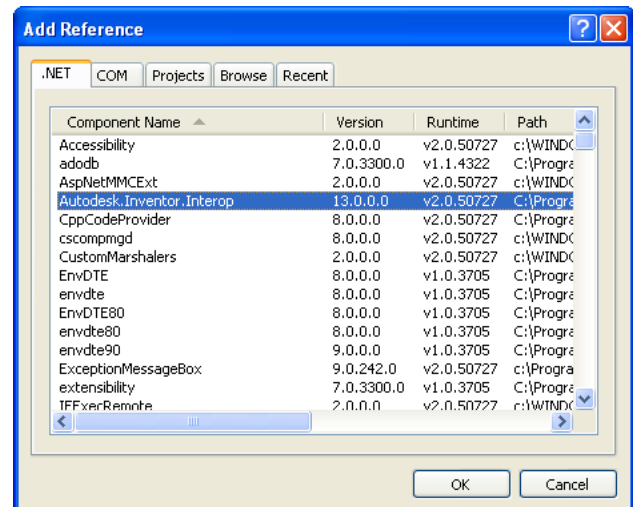
The next step is to connect to Inventor and get a reference to the Application object. The sample code below demonstrates this.

```
Public Sub InventorConnectSample()
    ' Declare a variable as Inventor's Application object type.
    Dim invApp As Inventor.Application

    ' Turn on error handling.
    On Error Resume Next

    ' Try to connect to a running instance of Inventor.
    Set invApp = GetObject(, "Inventor.Application")
    If Err.Number <> 0 Then
        ' Was unable to connect to a running Inventor.  Clear the error.
        Err.Clear

        ' Try starting Inventor.
        Set invApp = CreateObject("Inventor.Application")
        If Err.Number <> 0 Then
            ' Unable to start Inventor.  Display a message and exit out.
            MsgBox "Unable to connect to or start Inventor."
            Exit Sub
        End If

        ' When Inventor is started using CreateObject it is started
        ' invisibly.  This will make it visible.
        invApp.Visible = True
    End If

    ' Turn off error handling.
    On Error GoTo 0
End Sub
```

VB.Net (Visual Basic 2005 Express Edition)

The process of connecting to Inventor and getting the application object using VB.Net is similar to VB 6 and VBA. The differences are how you create the reference to the Inventor library and the style of error handling. (Although the On Error style of error handling is also supported so the previous code will also work in VB.Net.)

With VB.Net you can use the Inventor Primary Interop Assembly to gain access to all of the Inventor defined objects. To add a reference to this assembly use the **Add Reference** command found under the Project menu. In the dialog, shown to the right, "Autodesk Inventor Interop" item and click the "OK" button, as shown to the right.



The VB.Net code is shown below. It's identical to the VBA/VB code above except it uses a different style of error handling that only VB.Net supports.

```
Public Sub InventorConnectSample()
    ' Declare a variable as Inventor's Application type.
    Dim invApp As Inventor.Application

    ' Enable error handling.
    Try
        ' Try to connect to a running instance of Inventor.
        invApp = GetObject(, "Inventor.Application")
    Catch ex As Exception
        ' Connecting to a running instance failed so try to start Inventor.
        Try
            ' Try starting Inventor.
            invApp = CreateObject("Inventor.Application")

            ' When Inventor is started using CreateObject it is started
            ' invisibly.  This will make it visible.
            invApp.Visible = True
        Catch ex2 As Exception
            ' Unable to start Inventor.
            MsgBox("Unable to connect to or start Inventor.")
            Exit Sub
        End Try
    End Try
End Sub
```

Apprentice

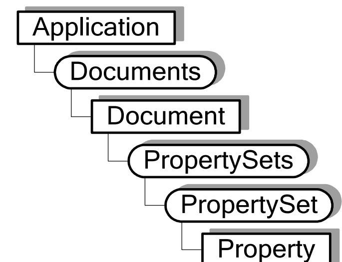
There's one other technique of accessing some of Inventor's functionality. There's a utility component referred to as *Apprentice* that provides a programming interface to a small portion of the information in an Inventor document. Apprentice doesn't have a user-interface and is exclusively a programming component. Design Assistant and Inventor View are both programs that use Apprentice to access document information. iProperties are one of the things that you have access to through Apprentice.

There are a couple of reasons to use Apprentice when possible. First, programs using Apprentice will run much faster than the equivalent program in Inventor. This is because Apprentice doesn't have a user-interface and it only provides partial access to information in the document so it doesn't need to load as much of the document as Inventor does. Second, Apprentice is freely available by installing Inventor View, (which includes Apprentice), from autodesk.com.

We'll see some examples of the use of apprentice in some of the samples.

Collection Objects

Another concept that's important to understand when working with Inventor's programming interface is *collection* objects. Collection objects are represented in the object model diagram by boxes with rounded corners. In the object model picture to the right the Documents, PropertySets, and PropertySet objects are collection objects.



The primary job of a collection object is to provide access to a group of related objects (the set of children for that parent). For example, the Documents object provides access to all of the documents that are currently open in Inventor. A collection provides access to its contents through two properties; *Count* and *Item*. The Count property tells you how many items are in the collection and the Item property returns a specific item. All collections support these two properties.

Typically, when using the Item property you specify the index of the item you want from the collection (i.e. Item 1, 2, 3 …). For example, the code below prints out the filenames of all of the documents currently open by using the Count and Item properties of the Documents collection object. (This and most of the following samples use Inventor's VBA and take advantage of the ThisApplication global variable.)

```
Public Sub ShowDocuments()
   ' Get the Documents collection object.
   Dim invDocs As Documents
   Set invDocs = ThisApplication.Documents

   ' Iterate through the contents of the Documents collection.
   Dim i As Integer
   For i = 1 To invDocs.Count
      ' Get a specific item from the Documents collection.
      Dim invDocument As Document
      Set invDocument = invDocs.Item(i)

      ' Display the full filename of the document in the Immediate window.
      Debug.Print invDocument.FullFileName
   Next
End Sub
```

Another technique of going through the contents of a collection is to use the For Each statement. This can also be more efficient and results in a faster program in many cases. The macro below accomplishes exactly the same task as the previous macro but uses the For Each statement.

```
Public Sub ShowDocuments2()
   ' Get the Documents collection object.
   Dim invDocs As Documents
   Set invDocs = ThisApplication.Documents

   ' Iterate through the contents of the Documents collection.
    Dim invDocument As Document
    For Each invDocument In invDocs
      ' Display the full filename of the document in the Immediate window.
      Debug.Print invDocument.FullFileName
   Next
End Sub
```

When the Item property is used with a value indicating the index of the item, the first item in the collection is 1 and the last item is the value returned by the collection's Count property. For some collections the Item property also supports specifying the name of the item you want. Instead of specifying the index of the item you can supply a String that specifies the name of the object. We'll see this later when working with iProperties.

Another important feature of many collections is the ability to create new objects. For example the Documents collection supports the Add method which is used to create new documents. It also supports the Open method which is used to open existing documents from the disk. These will be used in some of the samples that follow.

Derived Objects

The idea of *derived* and *base class* objects is usually a new concept for most end-users wanting to use Inventor's API. It's not a critical idea to understand but is a useful concept that we'll take advantage of when writing many of the following samples. To help describe this concept let's look at a close parallel; animal taxonomy or classification. For example, within the Animal kingdom you have insects, birds, mammals, etc. Within the mammal classification there are many different species but all of them share the same characteristics of a mammal; have hair, produce milk, etc. Even though an elephant and a dolphin are distinctly different they can both still be called mammals and share those same traits. This same idea can be used to understand the concept of derived and base class objects.

Let's look at how this concept applies to Inventor. Inventor has base class objects and derived objects. An example is the Document, PartDocument, AssemblyDocument, and DrawingDocument objects. The base class object is the Document object. The Document object supports all of the common traits that all documents share. The PartDocument, AssemblyDocument, and DrawingDocument objects are derived from the Document object and inherit these traits. They support everything the Document object supports plus they have additional methods and properties that are specific to that particular document type. For example, from the Document object you can get the filename, referenced documents, and iProperty information. From a PartDocument object you can get all of that, plus you can get sketches, features, and parameters.

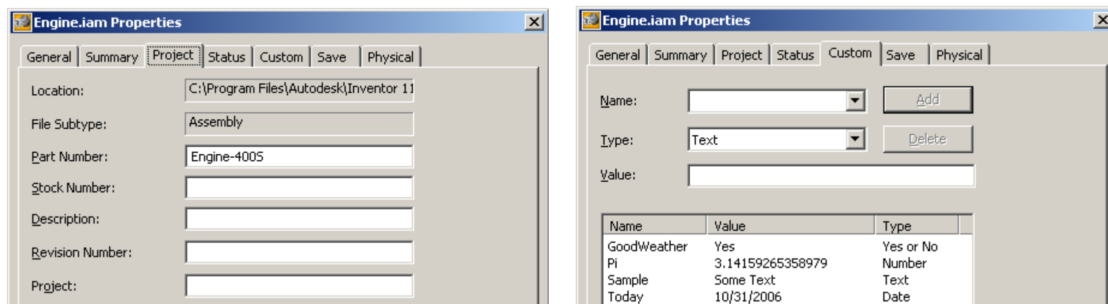
The usefulness of having derived objects is demonstrated in the previous sample code. Notice that the variable invDocument is declared to be type Document. The program iterates through the contents of the Documents collection and assigns each item to this variable. It doesn't matter if the document is a part, drawing, or an assembly since they are all derived from the Document object.

Programming Inventor's iProperties

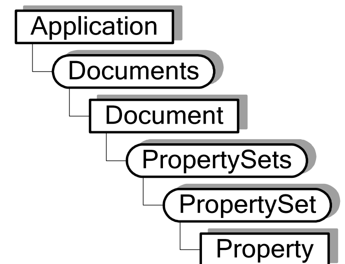
iProperties in the User-Interface

Before looking at the programming interface for iProperties let's do a quick review of iProperties from an end-user perspective. iProperties are accessed through the iProperties dialog. There are several tabs at the top that organize the available properties. There are also some tabs on this dialog that are not actually iProperties. For example, the General tab provides access to information about the document, the Save tab provides access to options that control how the thumbnail image is captured, and the Physical tab provides access to the various physical properties of the part or assembly document. If you need access to this information it is available through programming objects other than properties.

The picture below on the left illustrates a typical tab of the iProperties dialog where you have access to a specific set of iProperties. Through the dialog, you can view and edit the values of iProperties. The set of iProperties shown on each tab are predetermined by Inventor and cannot be changed. However, using the Custom tab (shown in the picture below on the right) you can add additional iProperties to the document. This allows you to associate any information you want with the document.



iProperties in Inventor's Programming Interface



The object model for iProperties is shown to the right. The diagram starts with the Application object and goes through the Documents collection to get the Document object. Remember that the Document object is the base class for all document types so it can represent any type of Inventor document. From the Document object we then access the PropertySets object which is the top-level property related object and provides access to the various iProperties associated with that particular document. Before we look at accessing the iProperties lets look at some different ways of accessing documents using the programming interface.

Accessing Documents

There are a few different ways you might want to access a document:

- Accessing through the Documents collection (which we've already seen in the previous samples).
- Access the document the end-user is currently working on in Inventor.
- Open a specific document that's been saved on disk.
- Create a new document
- Through a reference from another document.
- Open a document in Apprentice.

These are demonstrated in the code samples below. All of these samples, except for the Apprentice sample are shown using Inventor's VBA, and take advantage of the ThisApplication global variable it provides.

Accessing the Active Document

This sample gets the document currently being edited by the end-user. It uses the ActiveDocument property of the Application object.

```
Public Sub ActiveDocumentSample()
    ' Declare a variable to handle a reference to a document.
    Dim invDoc As Document

    ' Set a reference to the active document.
    Set invDoc = ThisApplication.ActiveDocument
    MsgBox "Got document: " & invDoc.DisplayName
End Sub
```

Opening a Document from Disk

This sample opens a document on disk and returns a reference to the open document. It uses the Open method of the Documents collection object.

```
Public Sub OpenDocumentSample()
    ' Declare a variable to handle a reference to a document.
    Dim invDoc As Document

    ' Open a document.
    Set invDoc = ThisApplication.Documents.Open("C:\Temp\Part1.ipt")
    MsgBox "Got document: " & invDoc.DisplayName
End Sub
```

Creating a New Document

This sample creates a new part document. It uses the Add method of the Documents collection and uses the FindTemplateFile method to get the filename of the standard part template.

```
Public Sub CreateDocumentSample()
    ' Declare a variable to handle a reference to a document.
    Dim invDoc As Document

    ' Create a new part document using the standard part template.
    Set invDoc = ThisApplication.Documents.Add(kPartDocumentObject, _
                  ThisApplication.FileManager.GetTemplateFile(kPartDocumentObject))
    MsgBox "Created: " & invDoc.DisplayName
End Sub
```

Through a Reference From Another Document

This is really beyond the scope of this paper but is presented here to demonstrate that it is possible to access documents that are referenced from another document. The example below gets the document associated with a part in the assembly named “Bracket:1”.

```
Public Sub ReferencedDocumentSample()
    ' Declare a variable to handle a reference to a document.
    Dim invDoc As Document

    ' Get the currently active document which is assumed to be an assembly.
    Dim invAssemblyDoc As AssemblyDocument
    Set invAssemblyDoc = ThisApplication.ActiveDocument

    ' Get the occurrence named "Bracket:1" from the assembly.  If a part
    ' of this name doesn't exist then an error will occur.  To simplify
    ' this sample, no error handling is done.
    Dim invOccurrence As ComponentOccurrence
    Set invOccurrence = _
            invAssemblyDoc.ComponentDefinition.Occurrences.ItemByName("Bracket:1")

    ' Get the document referenced by the occurrence.
    Set invDoc = invOccurrence.Definition.Document
    MsgBox "Got " & invDoc.DisplayName
End Sub
```

Opening a Document using Apprentice

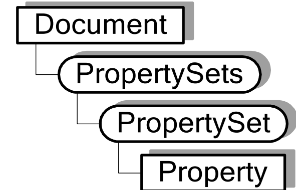
This example illustrates opening a document using Apprentice. Apprentice cannot be used within Inventor’s VBA, so this sample must be run from either Visual Basic or VBA within another product, i.e. Excel. Within Apprentice, documents are represented by the ApprenticeServerDocument object. It is slightly different than the Document object but serves the same purpose and provides access to the iProperties of the document.

```
Public Sub ApprenticeOpen()
    ' Declare a variable for Apprentice.  Notice that this uses the "New"
    ' keyword which will cause a new instance of Apprentice to be created.
    Dim invApprentice As New ApprenticeServerComponent

    ' Open a document using Apprentice.
    Dim invDoc As ApprenticeServerDocument
    Set invDoc = invApprentice.Open("C:\Temp\Part1.ipt")
    MsgBox "Opened: " & invDoc.DisplayName
```

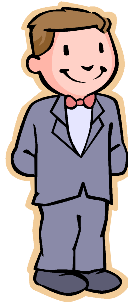
```
    ' Close the document and release all references.
    Set invDoc = Nothing
    invApprentice.Close
    Set invApprentice = Nothing
End Sub
```

Accessing iProperites

Accessing iProperties through Inventor's programming interface is reasonably simple. The object model for iProperties is shown to the right. The first thing to notice is that they're accessed through the Document object. Properties are owned by documents and to get access to properties you need to go through the document that owns them.



Even though the programming interface for iProperties is simple, people still tend to struggle with it. I believe this is primarily because of not understanding how to access a specific property. Before discussing iProperties in detail, a brief discussion of naming is appropriate to help describe the concepts Inventor uses. The picture below shows a person and three different ways to identify this person. His full legal name is a good way to identify him, although a bit formal. His social security number is good, but not very user friendly. His nickname, although commonly used, can have problems since it's not as likely to be unique Bill could decide tomorrow he would rather be called Billy or Will. The point is that there are three ways to identify this person, each one with its own pros and cons.



Legal Name: William Harry Potter

Nickname: Bill

SSN: 365-58-9401

When working with iProperties you'll also need to specify specific objects. Like Bill, iProperties also have several names. Understanding this single concept should help you overcome most of the problems other people have had when working with iProperties. The various iProperty objects, their names and best use suggestions are described below.

PropertySets Objects

The PropertySets object serves as the access point to iProperties. The PropertySets object itself doesn't have a name but is simply a utility collection object that provides access to the various PropertySet objects. Using the Item method of the PropertySets object you'll specify which PropertySet object you want. The Item method accepts an Integer value indicating the index of the PropertySet object you want, but more important, it also accepts the name of the PropertySet object you want. The next section discusses PropertySet objects and their various names.

PropertySet Objects

The PropertySet object is a collection object and provides access to a set of iProperties. The PropertySet object is roughly equivalent to a tab on the iProperties dialog. The Summary, Project, Status, and Custom tabs of the dialog contain the iProperties that are exposed through the programming interface. There are four PropertySet objects in an Inventor document; Summary Information, Document Summary Information, Design Tracking Properties, and User Defined Properties.

PropertySet objects are named so that you can find a particular PropertySet object. A PropertySet object has three names; Name, Internal Name, and Display Name. Using the analogy of Bill above, the Name is equivalent to his legal name, the Internal Name is equivalent to his social security number, and the Display Name is equivalent to his nickname. Let's look at a specific example to illustrate this. There is a PropertySet object that has the following names:

Name: Inventor Summary Information
Internal Name: {F29F85E0-4FF9-1068-AB91-08002B27B3D9}
DisplayName: Summary Information

Any one of these can be used as input to the Item method in order to get this particular PropertySet object. I would suggest always using the Name for the following reasons. The Name cannot be changed, is guaranteed to be consistent over time, and is an English human readable string. The Internal Name cannot be changed and will remain consistent but it is not very user-friendly and makes your source code more difficult to read and maintain. The Display Name is not guaranteed to remain constant. The Display Name is the localized version of the name and will change for each language. A chart showing the names of the four standard PropertySet objects is at the end of this paper.

Below is an example of obtaining one of the PropertySet objects. In this case the summary information set of iProperties.

```
Public Sub GetPropertySetSample()
    ' Get the active document.
    Dim invDoc As Document
    Set invDoc = ThisApplication.ActiveDocument

    ' Get the summary information property set.
    Dim invSummaryInfo As PropertySet
    Set invSummaryInfo = invDoc.PropertySets.Item("Inventor Summary Information")
End Sub
```

Property Objects

A Property object represents an individual property. Each Property object also has three names; Name, Display Name, and ID. Many of the same principles discussed for PropertySet objects applies here. The Name is an English string that is guaranteed to remain constant. The Display Name is the localized version of the Name and can change, so it's not a reliable method of accessing a particular property. The ID is a number and is similar to the Internal Name of the PropertySet object, but is a simple Integer number instead of a GUID. For consistency I would recommend using the name of the property when you need to access a specific one. Below is an example of the three identifiers for a particular property.

Name: Part Number
DisplayName: Part Number
ID: 5 or kPartNumberDesignTrackingProperties

The following code gets the iProperty that represents the part number.

```
Public Sub GetPropertySample()
    ' Get the active document.
    Dim invDoc As Document
    Set invDoc = ThisApplication.ActiveDocument

    ' Get the design tracking property set.
    Dim invDesignInfo As PropertySet
    Set invDesignInfo = invDoc.PropertySets.Item("Design Tracking Properties")

    ' Get the part number property.
```

```
    Dim invPartNumberProperty As Property
    Set invPartNumberProperty = invDesignInfo.Item("Part Number")
End Sub
```

You may see program samples that use identifiers like kPartNumberDesignTrackingProperties to specify a specific property. These identifiers are defined in the Inventor type library and provide a convenient way of specifying the ID of a property. For the part number, instead of specifying the ID as 5 you can use kPartNumberDesignTrackingProperties. This makes your code more readable. If you want to use the ID instead of the Name you need to use the ItemByPropId property instead of the standard Item property of the PropertySets object. As stated before, for consistency I would recommend using the Name in both cases.

Something else you'll see in a lot of sample code is where several property calls are combined into a single line. The code below does the same thing as the previous sample but it's getting the PropertySet object returned by the PropertySets Item property and immediately calling the Item property on it to get the desired property.

```
Public Sub GetPropertySample()
    ' Get the active document.
    Dim invDoc As Document
    Set invDoc = ThisApplication.ActiveDocument

    ' Get the part number property.
    Dim invPartNumberProperty As Property
    Set invPartNumberProperty = invDoc. _
            PropertySets.Item("Design Tracking Properties").Item("Part Number")
End Sub
```

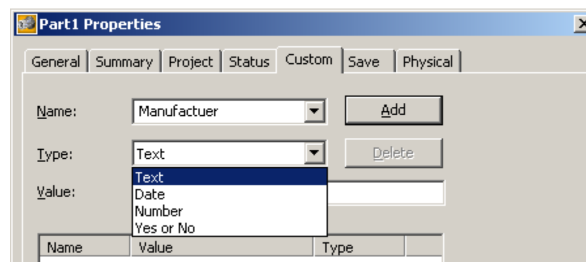
Now that we have a reference to a specific property we can use its programming properties to get and set the value. The Property object supports a property called Value that provides this capability. For example, the line below can be added to the previous sample to display the current part number.

```
MsgBox "The part number is: " & invPartNumberProperty.Value
```

This next line sets the value of the part number iProperty.

```
invPartNumberProperty.Value = "Part-001"
```

The Value property uses an interesting programming feature that is useful to understand when working with iProperties. The Value property is of type *Variant*. A Variant is a special type that can represent almost any other type. For example, a Variant can hold a String, Double, Date, or most any other type. iProperties take advantage of this since they allow the value to be one of many different types. You see this when working with custom (user defined) properties, as shown in the picture below. When creating or modifying a custom iProperty you define the type; Text, Date, Number, or Yes or No. This results in an iProperty being created where the value contains a String, Date, Double, or Boolean value.



When setting the value of any of Inventor's predefined iProperties, Inventor forces them to be the correct type and will convert them automatically, when possible. If you supply a value that can't be converted to the expected type, it will fail. In the table at the end of this paper the type of each property is listed. Most of the standard iProperties are Strings, with a few Date, Currency, Boolean, and Long values. There is also one other slightly unusual type called IPictureDisp. This type is used for the thumbnail picture associated with a document. Using this you can extract the thumbnail picture from a document.

Creating Properties

The primary reason to be aware of the Variant type is when you create your own custom iProperties. Interactively, as shown in the previous picture, you specify the type of property you're going to create; Text, Date, Number, or Yes or No. When you create them using Inventor's programming interface you don't explicitly specify the type but it is implicitly determined based on the type of variable you input.

New iPropertiies can only be created within the Custom (user defined) set of properties. New iProperties are created using the Add method of the PropertySet object. The sample below illustrates creating four new iProperties, one of each type.

```
Public Sub CreateCustomProperties()
    ' Get the active document.
    Dim invDoc As Document
    Set invDoc = ThisApplication.ActiveDocument

    ' Get the user defined (custom) property set.
    Dim invCustomPropertySet As PropertySet
    Set invCustomPropertySet = invDoc.PropertySets.Item( _
                                        "Inventor User Defined Properties")

    ' Declare some variables that will contain the various values.
    Dim strText As String
    Dim dblValue As Double
    Dim dtDate As Date
    Dim blYesOrNo As Boolean

    ' Set values for the variables.
    strText = "Some sample text."
    dblValue = 3.14159
    dtDate = Now
    blYesOrNo = True

    ' Create the properties.
    Dim invProperty As Property
    Set invProperty = invCustomPropertySet.Add(strText, "Test Test")
    Set invProperty = invCustomPropertySet.Add(dblValue, "Test Value")
    Set invProperty = invCustomPropertySet.Add(dtDate, "Test Date")
    Set invProperty = invCustomPropertySet.Add(blYesOrNo, "Test Yes or No")
End Sub
```

A common task is when you have a value you want to save as a custom property within a document. If the property already exists you just want to update the value. If the property doesn't exist then you need to create it with the correct value. The code below demonstrates getting the volume of a part and writing it to a custom property named "Volume". With the volume as an iProperty it can be used as input for text on a drawing. The portion of this macro that gets the volume and formats the result is outside the scope of this paper but helps to demonstrate a practical use of creating and setting the value of an iProperty.

```
Public Sub UpdateVolume()
    ' Get the active part document.
    Dim invPartDoc As PartDocument
    Set invPartDoc = ThisApplication.ActiveDocument

    ' Get the volume of the part.  This will be returned in
    ' cubic centimeters.
    Dim dVolume As Double
    dVolume = invPartDoc.ComponentDefinition.MassProperties.Volume

    ' Get the UnitsOfMeasure object which is used to do unit conversions.
    Dim oUOM As UnitsOfMeasure
    Set oUOM = invPartDoc.UnitsOfMeasure

    ' Convert the volume to the current document units.
    Dim strVolume As String
    strVolume = oUOM.GetStringFromValue(dVolume, _
                            oUOM.GetStringFromType(oUOM.LengthUnits) & "^3")

    ' Get the custom property set.
    Dim invCustomPropertySet As PropertySet
    Set invCustomPropertySet = _
               invPartDoc.PropertySets.Item("Inventor User Defined Properties")

    ' Attempt to get an existing custom property named "Volume".
    On Error Resume Next
    Dim invVolumeProperty As Property
    Set invVolumeProperty = invCustomPropertySet.Item("Volume")
    If Err.Number <> 0 Then
        ' Failed to get the property, which means it doesn't exist
        ' so we'll create it.
        Call invCustomPropertySet.Add(strVolume, "Volume")
    Else
        ' We got the property so update the value.
        invVolumeProperty.Value = strVolume
    End If
End Sub
```

Saving Properties

To save any changes you make to properties you need to save the document. The Save method of the Document object does this. However, when working with Apprentice it's possible to only save the iProperty changes to the document, which is much faster than saving the entire document. The FlushToFile method of the PropertySets object saves any iProperty changes. The sample below demonstrates this by opening a document using Apprentice, changing a property value, saving the change, and closing everything. Remember that Apprentice cannot be used from within Inventor's VBA. It must be from another application's VBA or from Visual Basic.

```
Public Sub ApprenticeUpdate()
    ' Declare a variable for Apprentice.
    Dim invApprentice As New ApprenticeServerComponent

    ' Open a document using Apprentice.
    Dim invDoc As ApprenticeServerDocument
    Set invDoc = invApprentice.Open("C:\Temp\Part1.ipt")

    ' Get the design tracking property set.
    Dim invDTProperties As PropertySet
    Set invDTProperties = invDoc.PropertySets.Item("Design Tracking Properties")

    ' Edit the values of a couple of properties.
    invDTProperties.Item("Checked By").Value = "Bob"
    invDTProperties.Item("Date Checked").Value = Now

    ' Save the changes.
    invDoc.PropertySets.FlushToFile

    ' Close the document and release all references.
    Set invDoc = Nothing
    invApprentice.Close
    Set invApprentice = Nothing
End Sub
```

Example Programs

There are some associated sample programs that provide more complete examples of the various topics covered in this paper and provide practical examples in various programming languages of how to use Inventor's programming interface.

PropertySamples.ivb

CopyProperties – Copies a set of properties from a selected document into the active document.

DumpPropertyInfo – Displays information about all of the property sets and their properties.

Properties.xls

An Excel file where each sheet contains a list of properties and values. There are two Excel macros that are executed using the buttons on the first sheet.

Set Properties of Active Document – Copies the properties of the selected Excel sheet into the active document.

Set Properties of Documents – Copies the properties of the selected Excel sheet into all of the Inventor documents in the selected directory. This sample uses Apprentice.

Inventor User Defined Properties, {D5CDD505-2E9C-101B-9397-08002B2CF9AE}			
Inventor Summary Information, {F29F85E0-4FF9-1068-AB91-08002B27B3D9}			
Property Name	**Id**	**Id Enum**	**Type**
Title	2	kTitleSummaryInformation	String
Subject	3	kSubjectSummaryInformation	String
Author	4	kAuthorSummaryInformation	String
Keywords	5	kKeywordsSummaryInformation	String
Comments	6	kCommentsSummaryInformation	String
Last Saved By	8	kLastSavedBySummaryInformation	String
Revision Number	9	kRevisionSummaryInformation	String
Thumbnail	17	kThumbnailSummaryInformation	IPictureDisp
Inventor Document Summary Information, {D5CDD502-2E9C-101B-9397-08002B2CF9AE}			
Property Name	**Id**	**Id Enum**	**Type**
Category	2	kCategoryDocSummaryInformation	String
Manager	14	kManagerDocSummaryInformation	String
Company	15	kCompanyDocSummaryInformation	String
Design Tracking Properties, {32853F0F-3444-11D1-9E93-0060B03C1CA6}			
Property Name	**Id**	**Id Enum**	**Type**
Creation Time	4	kCreationDateDesignTrackingProperties	Date
Part Number	5	kPartNumberDesignTrackingProperties	String
Project	7	kProjectDesignTrackingProperties	String
Cost Center	9	kCostCenterDesignTrackingProperties	String
Checked By	10	kCheckedByDesignTrackingProperties	String
Date Checked	11	kDateCheckedDesignTrackingProperties	Date
Engr Approved By	12	kEngrApprovedByDesignTrackingProperties	String
Engr Date Approved	13	kDateEngrApprovedDesignTrackingProperties	Date
User Status	17	kUserStatusDesignTrackingProperties	String
Material	20	kMaterialDesignTrackingProperties	String
Part Property Revision Id	21	kPartPropRevIdDesignTrackingProperties	String
Catalog Web Link	23	kCatalogWebLinkDesignTrackingProperties	String
Part Icon	28	kPartIconDesignTrackingProperties	IPictureDisp
Description	29	kDescriptionDesignTrackingProperties	String
Vendor	30	kVendorDesignTrackingProperties	String
Document SubType	31	kDocSubTypeDesignTrackingProperties	String
Document SubType Name	32	kDocSubTypeNameDesignTrackingProperties	String
Proxy Refresh Date	33	kProxyRefreshDateDesignTrackingProperties	Date
Mfg Approved By	34	kMfgApprovedByDesignTrackingProperties	String
Mfg Date Approved	35	kDateMfgApprovedDesignTrackingProperties	Date
Cost	36	kCostDesignTrackingProperties	Currency
Standard	37	kStandardDesignTrackingProperties	String
Design Status	40	kDesignStatusDesignTrackingProperties	Long
Designer	41	kDesignerDesignTrackingProperties	String
Engineer	42	kEngineerDesignTrackingProperties	String
Authority	43	kAuthorityDesignTrackingProperties	String
Parameterized Template	44	kParameterizedTemplateDesignTrackingProperties	Boolean
Template Row	45	kTemplateRowDesignTrackingProperties	String
External Property Revision Id	46	kExternalPropRevIdDesignTrackingProperties	String
Standard Revision	47	kStandardRevisionDesignTrackingProperties	String
Manufacturer	48	kManufacturerDesignTrackingProperties	String
Standards Organization	49	kStandardsOrganizationDesignTrackingProperties	String
Language	50	kLanguageDesignTrackingProperties	String
Defer Updates	51	kDrawingDeferUpdateDesignTrackingProperties	Boolean
Size Designation	52		String
Categories	56	kCategoriesDesignTrackingProperties	String
Stock Number	55	kStockNumberDesignTrackingProperties	String
Weld Material	57	kWeldMaterialDesignTrackingProperties	String