IM221410

The Power of iLogic Design Automation: How did we get Here?

Jason Hunt
FS-Elliott Co., LLC

Alex Curtin
FS-Elliott Co., LLC

Learning Objectives

- Explore the reasons to automate designs, via iLogic.
- How to incorporate iLogic into your work flows.
- Learn about some helpful tips and tricks in writing iLogic Code.
- Discover the importance and value in using iLogic for your business.

Description

The iLogic component of Autodesk Inventor software is a powerful tool that is utilized in automating the design process, improving efficiency and the quality of your work. In this class, we will discuss and demonstrate how powerful the iLogic tool is in allowing you to automate your design workflows. We will start our discussion with how we used iLogic when we started to learn how to apply it to our design process and how we progressed to more advanced techniques, as our knowledge of iLogic and the API grew. This is a success story where we will discuss the criteria we used to determine which parts of our designs to automate and how we incorporated iLogic into our workflows. We will then show you live demonstrations of our complex impeller and diffuser model programs; while emphasizing the flexibility of these programs to handle many variations of the designs. Don't miss this opportunity to learn how important and valuable the iLogic tool in Inventor can be for your business.

Speaker(s)

Jason Hunt is the NPD Group Leader in the CAD Designer Group for FS-Elliott Co., LLC. FS-Elliott is a leading manufacturer of oil-free centrifugal air and gas compressors with sales, service, and manufacturing locations around the world. Based in Orchard Park, NY, Jason provides lead design services to current NPD projects and helps drives current CAD standards and Best Practices amongst the NPD team. His experience involves 20+ years of compressor design, with an education background in Engineering from SUNY at Alfred, Industrial Engineering from SUNY Buffalo State and Business/Marketing from the McColl School of Business at Queens University of Charlotte.
jhunt@fs-elliott.com

Alex Curtin is a Product Marketing Engineer at FS-Elliott Co. primarily responsible for sales support and aero staging selections. He was formerly a member of the Airend New Product Development team responsible for CFD and FEA analysis of the compressor aerodynamic components. He is currently pursuing a Master's degree in Mechanical Engineering with an emphasis on fluid mechanics and thermodynamics from Penn State University. His research focus is on numerical investigation of aerodynamic excitations in turbomachinery. He holds Bachelor's degrees in Aerospace and Mechanical Engineering from West Virginia University.
acurtin@fs-elliott.com

Contents

Learning Objectives 1
Speaker(s) 2
Class Overview 5
Incorporating iLogic into your Intelligent Modeling Workflows 5
How FS-Elliott uses iLogic in Design 5
Why FS-Elliott uses iLogic in Design 6
Basic Overview of the 2019 Inventor iLogic Browser 7
Inventor API: 10
FS-Elliott Program Techniques 12
Using “If-Else Statements” 12
Using “For Next Loops” & “Try-Catch Statements” 12
Using “Do While Loops” 14
Using Spreadsheets to Assist in Writing Code 14
Parameter Creation Using the API 19
How FS-Elliott incorporates these iLogic Techniques into Their Workflows 19
Diffuser Design Program 20
Reasons to Create the Diffuser iLogic Program 20
How We Created the iLogic Program 20
Diffuser Design Efficiency Improvement 21
Pros & Cons to this Diffuser Design Methodology 21
Impeller Design Program 22
Reasons to Create the Impeller iLogic Program 22
How We Created the iLogic Program 22
Impeller Design Efficiency Improvement 23
Pros & Cons to this Impeller Design Methodology 24
Motor Mounting Hardware Design Program 25
Reasons to Create the Motor Mounting Hardware iLogic Program 25
How We Created the iLogic Program 26
Motor Mounting Hardware Design Efficiency Improvement 28
Pros & Cons to this Motor Mounting Hardware Design Methodology 28

Reasons for Moving to API Usage in Writing our Code 29
API Coding (Examples / Tips & Tricks) 30
Create Box Example (Extrude) 31
Create Cylinder with Form Example (Revolve) 36
Impeller Creation - Using the API 39
Creating the Impeller Blade 39
Creating the Full Impeller 44
Diffuser Creation - Using the API 45
Template Creation - Using the API 45
Update Inventor Drawings to the Latest FS-Elliott Standards – Using the API 47
Conclusion 48

Class Overview

Why does automating your design matter? There are many different reasons that may drive you to automate your design. You may have a need to automate designs because you may be dealing with repetitive, mundane tasks. Maybe you are finding that designers are making too many mistakes, when modeling common components or you may be trying to build consistency of the modeling output to your CNC department. Whatever the reason for automating your design, you need to find a way to do it in Autodesk Inventor.

This is a success story where we will discuss the criteria we used to determine which parts of our designs to automate and how we incorporated iLogic into our workflows. We will then show you live demonstrations of our complex impeller and diffuser model programs; while emphasizing the flexibility of these programs to handle many variations of the designs. This is not a class where we will deep dive into the iLogic or API, but a class to show you how FS-Elliott has been successful in using iLogic and API to improve our design workflows.

After the conclusion of this class, you will more than likely be thinking about your current workflows and how you can incorporate iLogic into them. This class will help introduce you to how iLogic has helped our company and get you started in thinking about how to apply iLogic and API in your company's workflows.

Incorporating iLogic into your Intelligent Modeling Workflows

How FS-Elliott uses iLogic in Design

Here is the main criteria we look for, when we are thinking about using iLogic in our design process.

FS-Elliott iLogic Usage Criteria

Repetitive tasks:
We search for repetitive tasks that we do on a regular basis that are pretty standard and straightforward. These types of tasks may be as simple as a design check rule or something more advanced, such as a rule checking for adherence to company standards.

Automate Complex Tasks:
We search for repetitive tasks that we do on a regular basis that are more complex in nature, requiring a lot of thought.

Tool Creation:
We search for opportunities for tool creation. These tools can be as simple as an iLogic Rule to populate a part weight in your drawing notes to a design tool for O-ring design.

Why FS-Elliott uses iLogic in Design

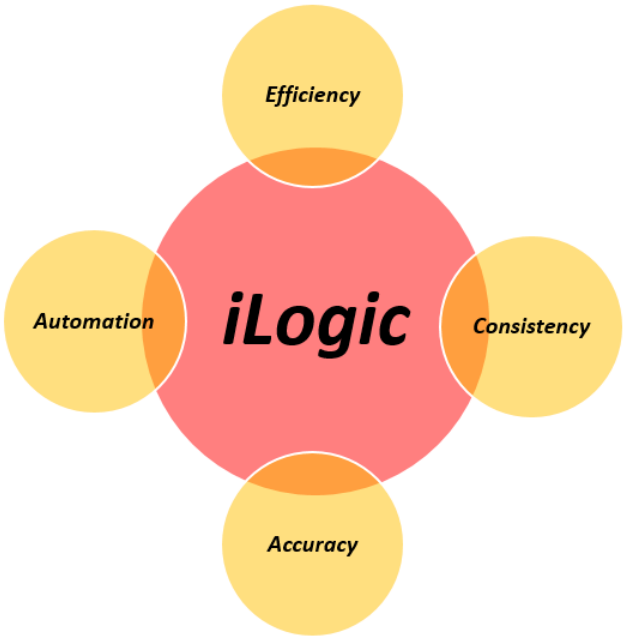
"iLogic is key to our continued success as being the premier global provider of centrifugal compressor solutions. We are able to utilize iLogic to drive complex model automation, increasing our efficiency and accuracy. Thus helping us achieve our goal of being the premier global provider of centrifugal compressor solutions."

FS-Elliott's 4 Pillars of why we use iLogic

Automation:
Automation allows our design and engineering team to focus on design activities that are not easily automated.

Efficiency:
iLogic has helped us become efficient in various aspects of our designs. Some examples, which we will discuss later, include some of our aero component designs. These consist of our impeller and diffusers that we design and make.

Consistency:
Using iLogic has helped our company design components in a consistent manner. Doing this has made our internal workflows and our suppliers workflows easy to determine. Producing designs in a consistent manner makes it easier to document an efficient workflow much easier.



Accuracy:
iLogic has afforded us the opportunity to create very accurate designs. We have taken complex designs that were modeled in various different ways and the output data varied by the user. Since we use iLogic for these designs, we ensure that we take the human factor out the design process for some of our designs. This has produced highly accurate models with no documented errors to this point. The only errors we found, is when the program has errors in the design. These types of errors are avoided by putting your written program through a rigorous testing program.

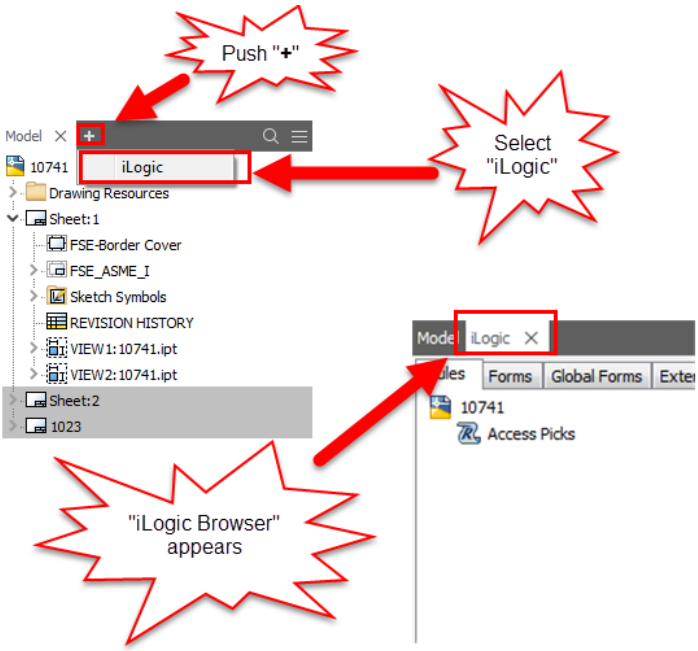
Basic Overview of the 2019 Inventor iLogic Browser

What is iLogic?

iLogic enables rule driven design that provides a simple way to capture and reuse your work. It allows the user to standardize and automate the design process.

iLogic allows you to become a coding expert without having to learn much actual code.

- In order to turn on your iLogic browser you need to go to the model tree and select the “+” (show tabs button) and select “iLogic”.
- The iLogic browser will then appear next to the model tree browser.



What are iLogic Rules?

A rule is a small Visual Basic program that can monitor and control other Inventor parameters, features, or components.

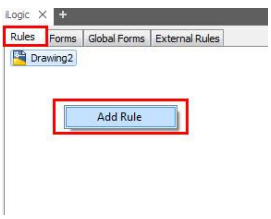
iLogic embeds rules as objects directly into a part, assembly and drawing documents. The rules determine and drive the design parameter and attribute values. By controlling these values, you can define the behavior of model attributes, features and components.

Knowledge is saved and stored directly in documents just like geometric design elements are stored.

Internal Rules

iLogic Rules saved within a document are known as Internal Rules.

- To create an internal iLogic rule go to the iLogic browser and click on the “Rules” tab.
- Next, right click in the open space and select “Add Rule” from the pop-up window.



When to use Internal Rules

The iLogic rule is only going to apply to one part and not globally.

- For example, if you have a part that can be any diameter, but can't exceed the maximum or minimum diameter due to material availability. You could write an internal iLogic rule to flag the user.

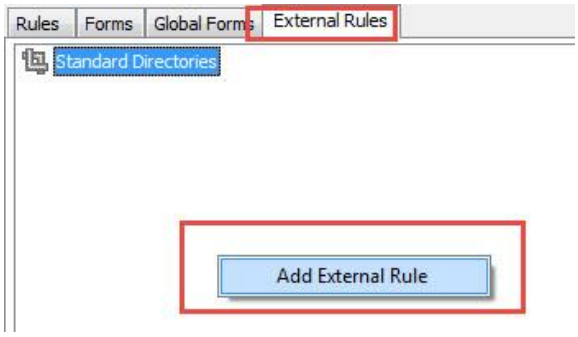
An **advantage** to an internal rule is that it is copied with the part and will always remain with the part, no matter where that part is used.

A big **disadvantage** is that if you need to edit or correct the code, you will have to track down every copy of that part to perform the edits. This can be time consuming.

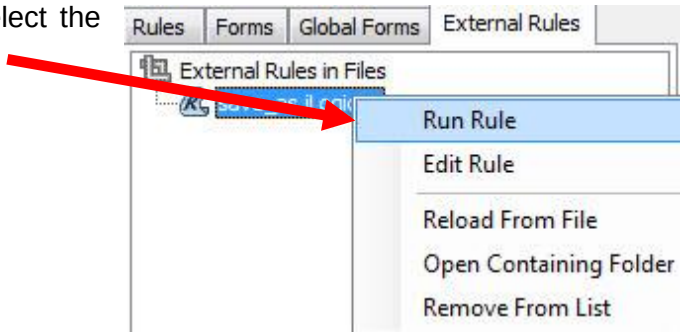
It is up to the designer to decide if an internal iLogic rule is the best approach.

External Rules

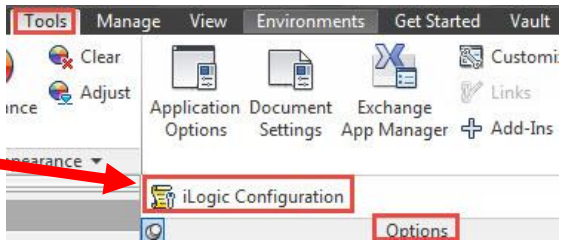
External Rules are saved on your local or network drive.



- To create an external iLogic rule go to the iLogic browser and click on the "External Rules" tab.
- Next, right click in the open space and select "Add External Rule" from the pop-up window.
- iLogic will look for Rules in the following places:
  - The folder in which the current Inventor document is located.
  - The current Inventor Project Workspace folder.
  - The list of folders set in iLogic Configuration.
- Next, you will have to select the External Rule and run it.

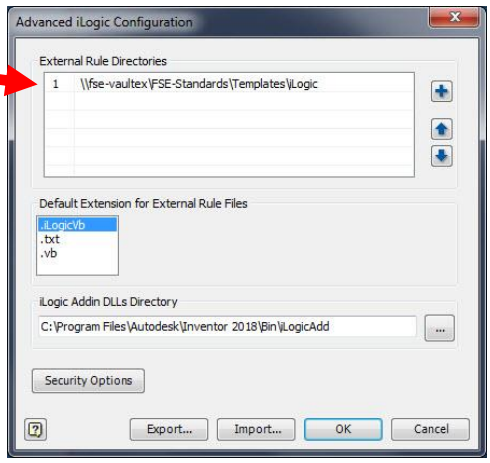


If you plan on sharing your Rules with others in your company, it is a good idea to add your company iLogic folder to the list of folders in the iLogic configuration setup.



To do this, go to "Tools > Options > iLogic Configuration"

A pop up window will appear so you can add your company iLogic Folder to the set up.



When to Use External Rules

External Rules are great to use when one wishes to apply the rule to many models.

- For example, if you have a company requirement that all panels will have set choices for material thickness, length, width, height and color. Then you could write an external iLogic rule to drive these choices through all the panels.

One advantage, in this case, is that any errors in your code can be fixed in the one source file and will automatically be applied, every time they are called in a document versus going into every model that has the rule embedded in it.

Tip: External rules are great for creating iLogic code 'Modules' that you can reuse for other tasks.

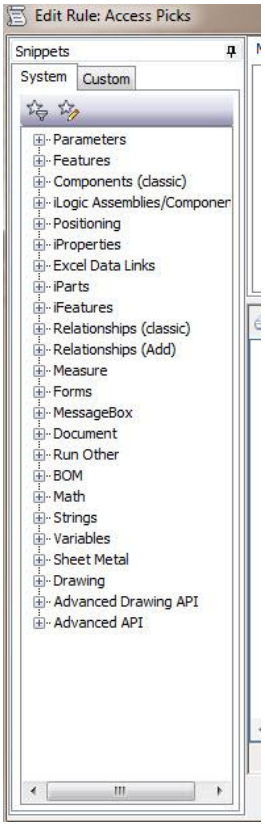
The only disadvantage is that you will need to remember to send your iLogic file with the document, if you want the iLogic Rule to be used elsewhere.

Best Practices for Writing iLogic Rules

- Use comments in your code to make it easier to understand what your code is doing. This will help you and others down the road to understand how it works.
- Don't overdo it by overcomplicating your rule. Sometimes more rules are better than putting all your iLogic code into one rule.

- Consider making your rules, so they can be reused in other projects. Why reinvent the wheel.
- When writing code, it is always good to be consistent in your methods.
- Did I mention to use comments?

iLogic Code Snippets



Code snippets provide the programmer shortcuts, for frequently used pieces of code. Using snippets allows the user to insert them into your code that you would normally have to type in manually. Using snippets also helps reduce the possibility of errors in your program, due to typographical errors.

You can access the available snippets from the “Snippets” area of the Edit Rule dialog box. This area features two tabs:

- The System tab includes a set of predefined snippets, arranged by category.
  - In order to display the tool tip, hold the cursor over each snippet to display its function in more detail.
- The Custom tab allows you to add your own snippets, or create custom copies of System snippets.
- Favorite snippets
  - Favorites allow you to choose which snippets appear on the System tab. You can mark specific snippets as favorites, and then toggle the display of the list to show only those snippets marked as favorites.

Inventor API:

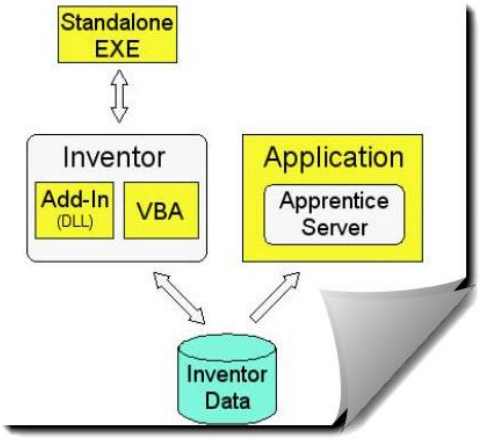
What is API?

API is a three-letter acronym meaning, Application Programming Interface. So, what is API? API is an interface that allows the user write a program that will perform the same types of operations that would normally be able to use when working in inventor interactively.

The use of API is important, because it allows the user to add functionality to meet their needs. This allows the user to optimize repetitive operations in their designs, as we will demonstrate in our live demonstrations in automating our diffuser and impeller designs.

How do I access API?

There are many different ways to access the API (VBA, Add-Ins, Standalone EXE and Apprentice Server). None of the methods is incorrect, so it is good to have a general understanding of the ways to connect to the API. Having this basic understanding will allow you to make the best decision about how to write your program. The diagram shown to the right, from the Inventor API Help site, illustrates the different ways to access Inventor's API.

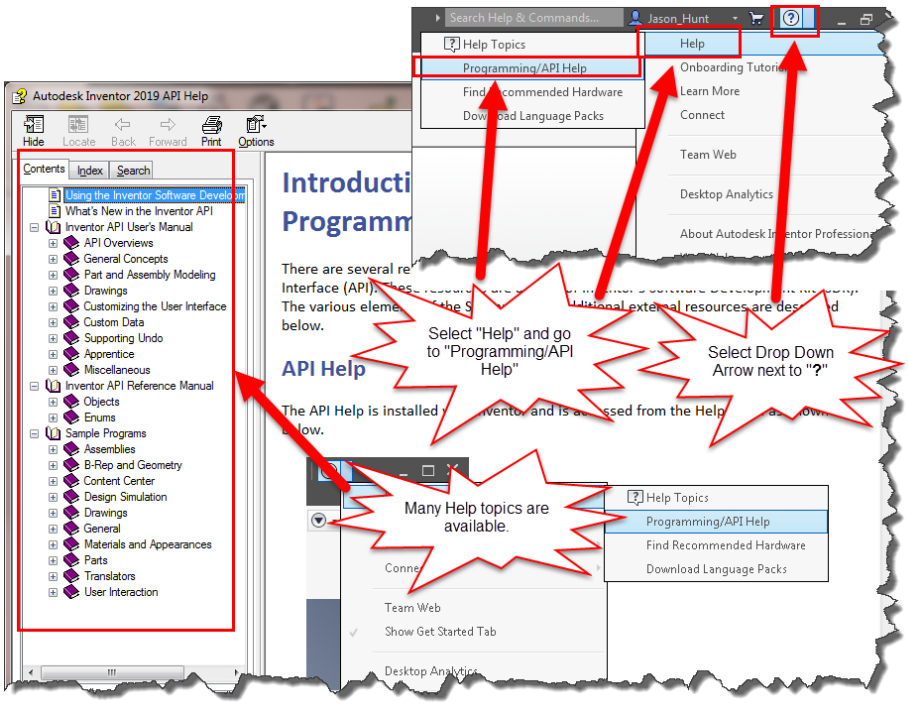


In order to understand the API and to get an overview of the methods to access API, you can view Inventor's API help document.

Accessing API Help

In order to access Autodesk Inventor's API help, you need to do the following.

- Go to the Help icon in the upper right hand corner of Inventor and select the drop down arrow.
- Select "Help" and go to "Programming/API Help"
- A help window will appear with many API related topics.



This is a great reference document to assist you in programing with the API. This document helps you develop code by showing you how to access Inventor functions and some examples, so you can use them in your code.

FS-Elliott Program Techniques

Using "If-Else Statements"

What is an "If-Else Statement"?

An "If Statement" is a very powerful coding tool.

An "If Statement" is typically used to check some logical statement and then execute some section of code, if the logical statement is true.

An "If Statement" has the following syntax:

```
If a > b Then
   'Do Something Here
End If
```

In the above statement if "a" is greater than "b", some code will be executed. If "a" is not greater than "b", then nothing will happen and the code will continue.

Another version of this is an "If-Else Statement" which has the following syntax:

```
If a > b Then
   'Do Something Here
Else
   'Do Something Else Here
End If
```

In this example, the code will still execute the first part if "a" is greater than "b". However, if "a" is not greater than "b", some other code will be executed. Code is executed either way, depending on the values of "a" and "b". The "Else" statement can also be an "ElseIf" which can contain additional logical checks.

Using "For Next Loops" & "Try-Catch Statements"

What is a "For Next Loop"?

The "For Next Loop" is one of the most frequently used loops in VBA.

The "For Next Loop" is typically used to move sequentially through a list of items and numbers. Let us take a closer look at this loop.

The "For Next Loop" has the following syntax:

```
For a_counter = start_counter To end_counter
   'Do Something Here
Next a_counter
```

What we are doing here is creating a loop that uses a variable a_counter as the "time keeper" of the loop. We set it to a value equal to start_counter at the beginning of the loop and then increment it by 1 during each loop. The loop will execute until the time the value of the a_counter becomes equal to the end_counter. The loop executes for the last time when both the values match and then stop.

What is a "Try-Catch" Statement?

The "Try-Catch" statement consists of a **try** block followed by one or more **catch** clauses, which specify handlers for different exceptions.

In the following code example, we see both a "For Next" loop and a "Try-Catch" statement. The code is doing the following:

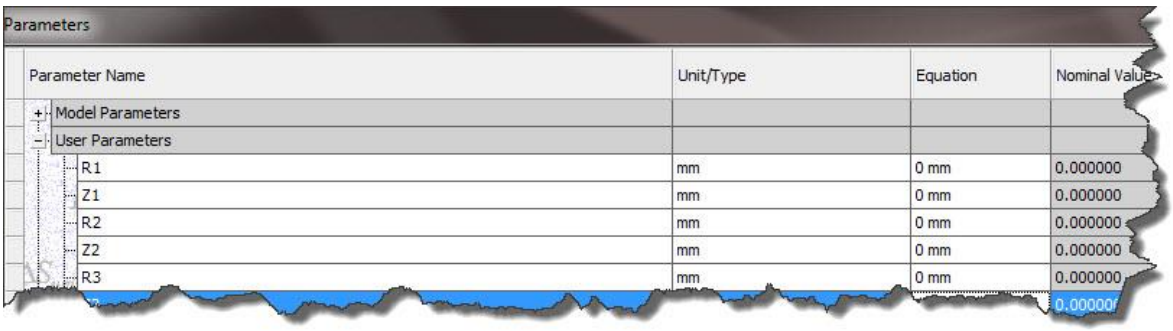
- The "For Next" loop iterates through all 29 user parameter scenarios to allow the "Try-Catch" statement to check if the user parameter has been created. If it hasn't been created the user parameter will be created through the code.
- The code will check to see if the 29 user parameters (R1, R2, R3,……R28, R29) have been created, using the try block.
- If an error exists from the Try block (the user parameter doesn't exist) then the Catch clause will create the user parameter.

Example of a "For Next Loop" with a "Try-Catch" Statement:

```
'----------------SHORTENS THE AMOUNT OF TEXT TO TYPE WHEN CREATING PARAMETER CODE
oMyParameter=ThisApplication.ActiveDocument.ComponentDefinition.Parameters.UserParameters
'----------------FOR STATEMENT TO GENEREATE ALL THE PARAMETERS
   For Y = 1 To 29
'--------------------R DATA POINT PARAMETER CHECK
      Try
         oTest1 = Parameter("R" & Y)
      Catch
'--------------------SETTING UP R DATA POINT USER PARAMETERS AS MILLIMETERS
      oParameter=oMyParameter.AddByExpression(R & Y, "0", UnitsTypeEnum.kMillimeterLengthUnits)
      End Try
'----------------INCREMENTS THE DATA POINT PARAMETERS FOR Z & R
   Next
```

The output of the above code would be to create 29 user parameters:

- Parameters named as follows: R1, R2, R3, R4, R5, ……. R28, R29
- The unit of measure would be in millimeters.

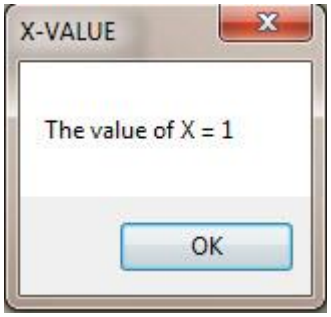


Using "Do While Loops"

What is a "Do While Loop"?

The "Do While Loop" provides a way to continue iterating through your code, while one or more conditions are true.

For Example: A "Do While Loop" can have one or more conditions in its expression. In the below example, there is one condition that continues iterating while the Variable "X" is from 1 to less than or equal to 5. The below code will launch a message box stating what the current X value is, as long as the condition of X is from 1 to less than or equal to 5 is met.




```
Dim X As Integer
X = 1
Do While X <= 5
   MessageBox.Show("The value of X = " &X, "X-VALUE")
   X = X+1
Loop
```

The downside to this type of loop is that it can easily lead to infinite loops if you are not careful. My suggestion would be to make sure you save your file before executing your rule, or you may end up killing your session before a save and chance losing a lot of work.

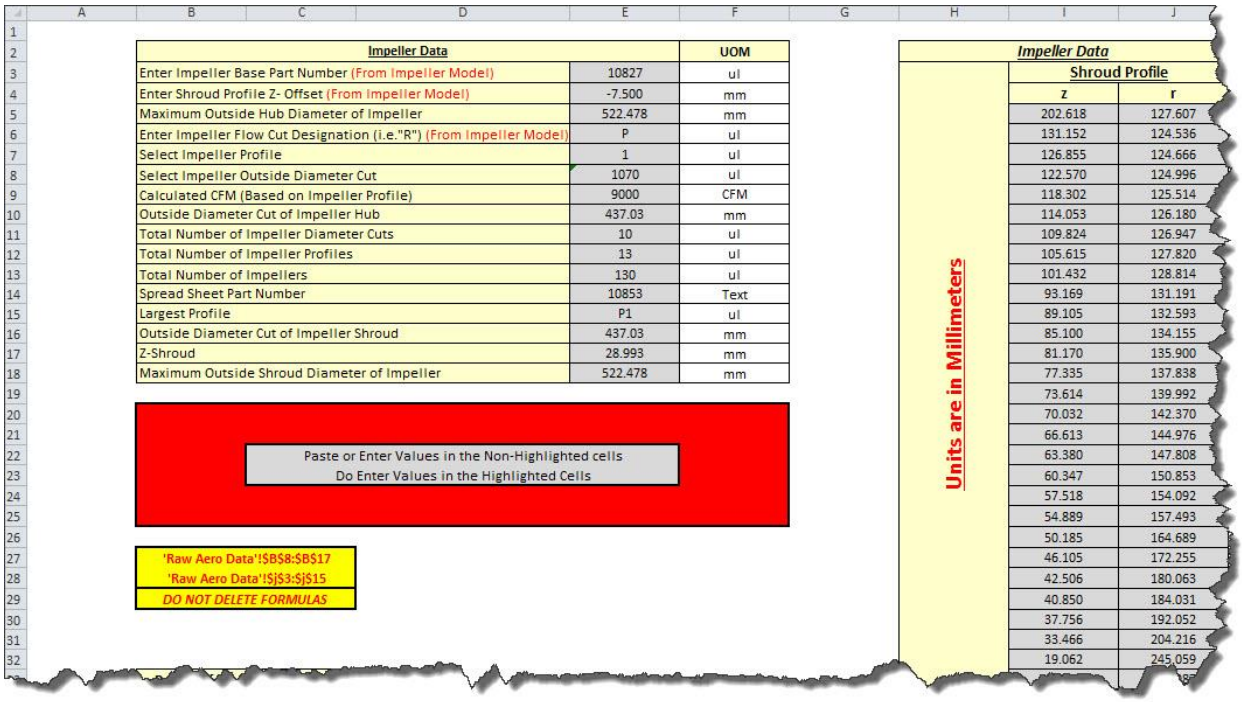
Using Spreadsheets to Assist in Writing Code

Why Use Spreadsheets?

Using a spreadsheet in the engineering world is nothing new, this practice has been around at least since Microsoft® Excel has been around. I know from experience that it is common practice to use a spreadsheet to store design data and to use it to calculate engineering data outputs.

Using iLogic with spreadsheets has advantages:

- Data can be read from Excel to push to Inventor
- Data can be pushed to Excel to allow the spreadsheet to process the inputted data
- Excel can process data and uses various functions within the spreadsheet that would normally be rather difficult to code, using iLogic. This is an advantage to the novice programmer.
- Allows data inputs and outputs to be entered and read in a consistent manner.
- You can read and input data from any sheet within the spreadsheet.



Impeller Data		UOM
Enter Impeller Base Part Number (From Impeller Model)	10827	ul
Enter Shroud Profile Z- Offset (From Impeller Model)	-7.500	mm
Maximum Outside Hub Diameter of Impeller	522.478	mm
Enter Impeller Flow Cut Designation (i.e."R") (From Impeller Model)	P	ul
Select Impeller Profile	1	ul
Select Impeller Outside Diameter Cut	1070	ul
Calculated CFM (Based on Impeller Profile)	9000	CFM
Outside Diameter Cut of Impeller Hub	437.03	mm
Total Number of Impeller Diameter Cuts	10	ul
Total Number of Impeller Profiles	13	ul
Total Number of Impellers	130	ul
Spread Sheet Part Number	10853	Text
Largest Profile	P1	ul
Outside Diameter Cut of Impeller Shroud	437.03	mm
Z-Shroud	28.993	mm
Maximum Outside Shroud Diameter of Impeller	522.478	mm

Paste or Enter Values in the Non-Highlighted cells
Do Enter Values in the Highlighted Cells

'Raw Aero Data'!$B$8:$B$17
'Raw Aero Data'!$J$3:$J$15
DO NOT DELETE FORMULAS

Impeller Data

Units are in Millimeters

Shroud Profile	
z	r
202.618	127.607
131.152	124.536
126.855	124.666
122.570	124.996
118.302	125.514
114.053	126.180
109.824	126.947
105.615	127.820
101.432	128.814
93.169	131.191
89.105	132.593
85.100	134.155
81.170	135.900
77.335	137.838
73.614	139.992
70.032	142.370
66.613	144.976
63.380	147.808
60.347	150.853
57.518	154.092
54.889	157.493
50.185	164.689
46.105	172.255
42.506	180.063
40.850	184.031
37.756	192.052
33.466	204.216
19.062	245.059

These advantages allow the user to use Microsoft Excel to define your design intent when you are creating something as simple as door panels to something more advanced, like an impeller. The only thing a user needs to understand is how to read and manipulate the data from Excel and incorporate it within the iLogic program to produce the desired result.

In order to learn more about Excel iLogic functions, follow the Autodesk link below.

iLogic Excel Functions

Methods to Specify the Excel Spreadsheet:

There are three methods to specify the Microsoft Excel data. You can embed the file internally to the inventor file, link it to an Autodesk Inventor file, or use it as an external file. They all will work the same way. It is just a matter of deciding what works best for your design intent. The only slight difference is the code used to read and write data to and from the Excel spreadsheet. These functions require either a filename or a specification of a linked or embedded Excel file.

The supported filename extensions for Excel spreadsheets are as follows:

- .xls
- .xlsx
- .xlsm
- .xlsb

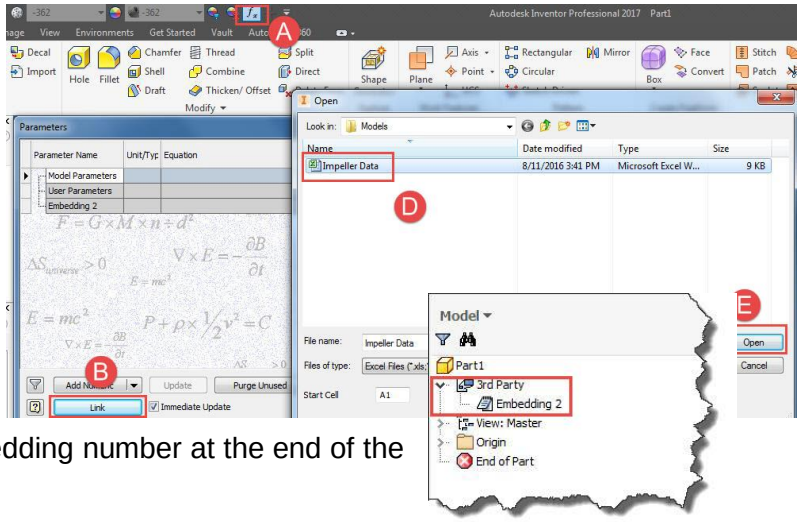
Embedded Spreadsheets (Method 1)

There are two ways to embed a spreadsheet with an Inventor file.

Parameter Icon Method:

In order to embed the spreadsheet into your inventor file you will need to do the following.

A. Click on the Parameter icon.
B. Press the “Link” button.
C. Select the “Embed” radial button.
D. Browse to the desired Excel file.
E. Push the “Open” button.

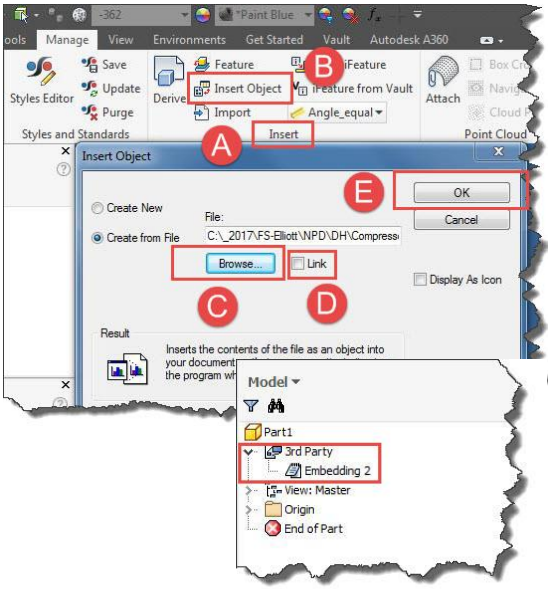


The result is found in the “3rd Party” (found in the model tree) with the file called “Embedding 2”.

- Please note that the Embedding number at the end of the filename can vary.

Ribbon Panel Method:

A. Go to the Manage tab and Insert panel.
B. Press the “Insert Object” button.
C. Browse to the desired Excel file.
D. Do not select the “Link” check box.
E. Push the “OK” button.



The result is found in the “3rd Party” (found in the model tree) with the file called “Embedding 2”.

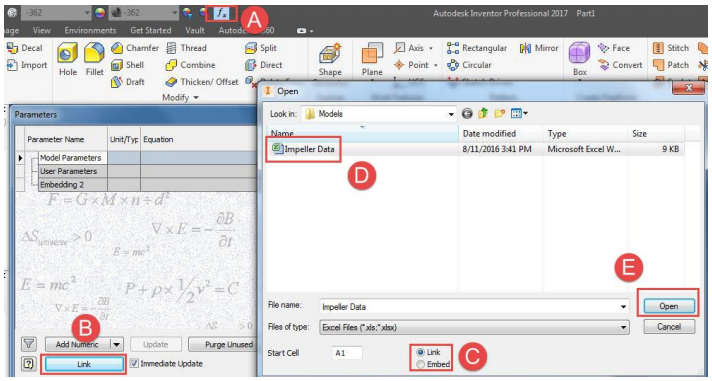
- Please note that the Embedding number at the end of the filename can vary.

Special Notes:

- If you use an embedded table, embed it using **Link** on the Parameters dialog box. Do not change the embedded table name from the default name given to it by Autodesk Inventor (for example, **Embedding 1**). `GoExcel` requires the original name.
- The syntax to use for writing code for an embedded spreadsheet is "`3rd Party:Embedding#`" for embedded spreadsheets.
  - **GoExcel.CellValue**("3rd Party:Embedding 1", "Sheet1", "A2")

Linked Spreadsheets (Method 2)

There are two methods to link a spreadsheet with an Inventor file.

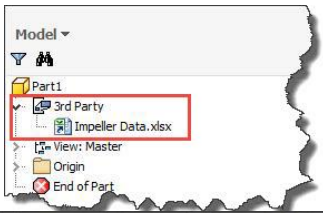


Parameter Icon Method:

In order to link the spreadsheet into your inventor file you will need to do the following.

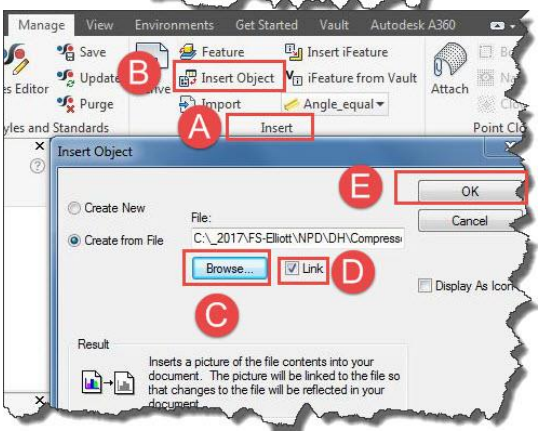
A. Click on the Parameter icon.
B. Press the "Link" button.
C. Select the "Link" radial button.
D. Browse to the desired Excel file.
E. Push the "Open" button.

The result is found in the "3rd Party" (found in the model tree) with the file called, for this example, "Impeller Data.xlsx".

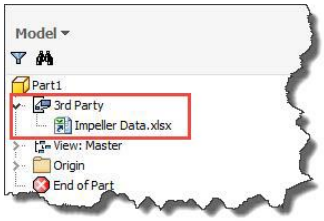


Ribbon Panel Method:

A. Go to the Manage tab and Insert panel.
B. Press the "Insert Object" button.
C. Browse to the desired Excel file.
D. Select the "Link" check box.
E. Push the "OK" button.



The result is found in the "3rd Party" (found in the model tree) with the file called, for this example, "Impeller Data.xlsx".



Special Notes:

- The syntax to use for writing code for a linked spreadsheet is "`3rd Party:LinkedName.xls`" for linked spreadsheets.
  - **GoExcel.CellValue(**"3rdParty:filename.xls", "Sheet1", "A2"**)**
- Specify the name that displays in the Autodesk Inventor Model tree, under 3rd Party.

External Path (Method 3)

Another option for reading data into iLogic program is to link the spreadsheet to the inventor file by writing code to specify the path to the Excel file. For a filename, you can either specify a relative or absolute path.

Relative Path:

If you do not specify a path, iLogic assumes that the Excel document is in the same folder as the current Inventor document. A relative path is assumed to be in the same folder as the Inventor file. iLogic also searches for the file under the project Workspace path. You can use a relative path under the project Workspace path.

- The syntax to use for writing code for a spreadsheet that utilizes a Relative Path is:
  - **GoExcel.CellValue(**"filename.xls", "Sheet1", "A2"**)**

Absolute Path:

An absolute path is entered in the iLogic code to read data directly from the spreadsheet file location. Although, using an absolute path can make it difficult to send the model to another user on another computer or if the file moves it could make all your code invalid.

- The syntax to use for writing code for a spreadsheet that utilizes an Absolute Path is:
  - **GoExcel.CellValue(**"C:\BOMs\filename.xls", "Sheet1", "A2"**)**

There are ways to write code to make the path a parameter. This will allow you to update the code quickly, if the path changes. Even better, you could write code to browse for the file location and once you select the file; the parameter for the path will update the code. An example of this will be shown later.

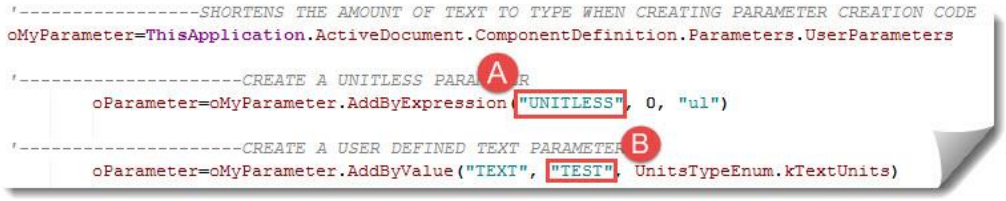
Parameter Creation Using the API

Some API expressions to create parameters are as follows:

- The following code snippet allows any parameter creation code to be simpler and more compact, with the same outcome. Writing this expression will save typing time and this code snippet only needs to be defined once in your code.

```
'----------------SHORTENS THE AMOUNT OF TEXT TO TYPE WHEN CREATING PARAMETER CODE
oMyParameter=ThisApplication.ActiveDocument.ComponentDefinition.Parameters.UserParameters
```

- The following lines of code create user parameters and their associated values.
  - The boxed in area by "**A**" is the parameter name "UNITLESS", with a value of "0".
  - The boxed in area by "**B**" is the value "TEST" for user parameter "TEXT".




```
'----------CREATE A UNITLESS PARAMETER
oParameter=oMyParameter.AddByExpression("UNITLESS", 0, "ul")

'----------CREATE A USER DEFINED TEXT PARAMETER
 oParameter=oMyParameter.AddByValue("TEXT", "TEST", UnitsTypeEnum.kTextUnits)

'----------CREATE A USER DEFINED PARAMETER WITH MILLIMETER UNITS
 oParameter=oMyParameter.AddByExpression("MM", "0", UnitsTypeEnum.kMillimeterLengthUnits)

'----------CREATE A USER DEFINED PARAMETER WITH INCH UNITS
oParameter=oMyParameter.AddByExpression("IN", "0", UnitsTypeEnum.kInchLengthUnits)

'----------CREATE A USER DEFINED ANGLE PARAMETER
oParameter=oMyParameter.AddByExpression("ANGLE", "0", UnitsTypeEnum.kDefaultDisplayAngleUnits)
```

How FS-Elliott incorporates these iLogic Techniques into Their Workflows

We have discussed and reviewed some of the iLogic techniques that we use at FS-Elliott, along with the criteria we use in deciding to use iLogic in our design workflows. In this section we want to introduce to you where and how we incorporated these techniques in our workflows. In this early learning phase of iLogic, we did not fully understand the API or know how to apply it to our designs, so we focused on writing programs that hinged on pure iLogic.

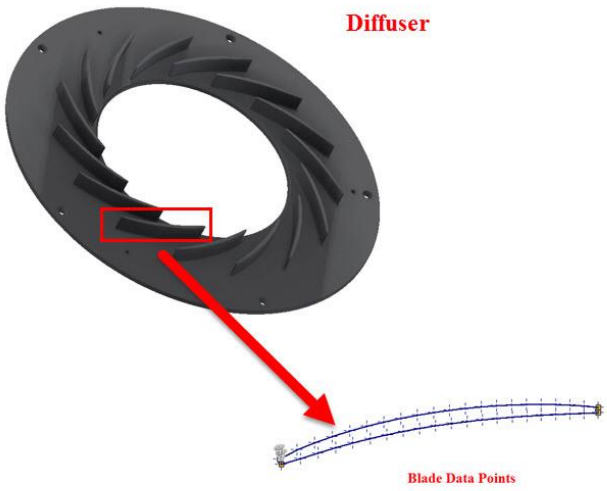
Later on in this paper, we will discuss how we took the impeller, diffuser and template programs to the next level with the API.

Diffuser Design Program

In the early phase in our knowledge of iLogic, we recognized a need to automate the design of our diffusers. This component was one we felt could be easily created through iLogic programming. The diffuser has many design variations across product platforms, but many similarities in the design. Here are the reasons we selected the diffuser to be created though an iLogic program.

Reasons to Create the Diffuser iLogic Program

- Diffuser geometry changes across design frames, but it is consistent in the type of information needed to design.
    - The diffuser outside diameter varies, but is a consistent input.
    - The diffuser inside bore diameter varies, but is a consistent input.
    - Blade geometry varies and is driven by blade data points, as shown to the right.
- The diffuser design is controlled with parameters. This makes it easy to make geometry changes with an iLogic program, when the model is intelligent.
- We wanted our diffuser models to be an automated design to create consistent and accurate models for our CNC department.



How We Created the iLogic Program

- We created a master model up front, meaning we made the model.
    - The model is comprised of parameter driven dimensions, for every key feature.
    - A set number of 29 blade data points, was determined to be the standard.
- The design data and all other family related design variations are stored in an external Excel spreadsheet for extrapolation, when the iLogic program runs.
- The iLogic Code is rather simple, as the main part of the code is a “Do While Loop” that reads and pushes data from Excel to change the parameters and create the new model. Below is a snippet from some of the diffuser code that we use.

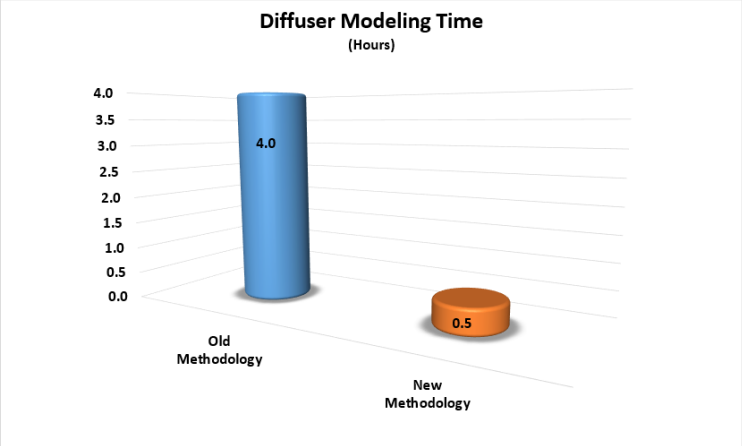
```
Do While Z <=NDP
        PROFILE = GoExcel.CellValue("3rd Party:Embedding 1", "Diffuser Outputs", "H17")      'PUSHES PROFILE TO EXCEL
        PDC = GoExcel.CellValue("3rd Party:Embedding 1", "Diffuser Outputs", "H18")
        If PDC<999 Then
                PDC = "0"&PDC
        Else
                PDC = GoExcel.CellValue("3rd Party:Embedding 1", "Diffuser Outputs", "H18")
        End If
        D_EW = GoExcel.CellValue("3rd Party:Embedding 1", "Diffuser Outputs", "H9")          'BLADE HEIGHT
        I_CA = GoExcel.CellValue("3rd Party:Embedding 1", "Diffuser Outputs", "H10")         'INLET CHAMFER ANGLE
        I_CW = GoExcel.CellValue("3rd Party:Embedding 1", "Diffuser Outputs", "H11")         'INLET CHAMFER WIDTH
        WBH = GoExcel.CellValue("3rd Party:Embedding 1", "Diffuser Outputs", "H21")          'D VALUE ON DRAWING
        DPN=DBPN+"-"+PROFILE+"-"+PDC+".IPT"                 'THIS IS FILE NAME GENERATION
```

```
PART=DBPN+"-"+PROFILE+"-"+PDC                    'TABLE PART NUMBER GENERATION
TPROFILE=PROFILE+"-"+PDC
iLogicVb.RunRule("UPDATE MODEL")
iProperties.Value("Summary", "Comments") = TPROFILE   'Fills out iproperties value
RuleParametersOutput()                           'NEEDED FOR PART NUMBER/PROFILE CUT TO BE RIGHT
InventorVb.DocumentUpdate()                      'NEEDED FOR PART NUMBER/PROFILE CUT TO BE RIGHT
FileToSave = ThisDoc.Path + "\" + DBPN + "\" + DPN       'Generates the path name
ThisDoc.Document.SaveAs(FileToSave, True)        'Saves at the above pathname
iLogicVb.RunRule("DRAWING DATA")
Z=Z+1                                            'INCREMENTS THE PROFILE
PROFILE = DP & Z                                 'PROFILE DESIGNATION CREATION
GoExcel.CellValue("3rd Party:Embedding 1", "Diffuser Outputs", "H17") = PROFILE      'PUSHES PROFILE TO EXCEL
Loop
```

Diffuser Design Efficiency Improvement

The key takeaway from this Diffuser Model iLogic program was the efficiency improvement in the design task our design team is seeing.

- The Original design methodology, without the iLogic program, would take a designer about 4 hours to complete, including the drawing.
- New Methodology now takes about .5 hours to complete, including the drawing.
- This is an 87.5 % improvement in efficiency.



Diffuser Program	
Original Time (Hours)	4.0
New Time (Hours)	0.5
Time Delta (Hours)	3.5
Percent Improvement	87.50%

Pros & Cons to this Diffuser Design Methodology

With any workflow or program, one can always find advantages and disadvantages. This iLogic program is no different. Here are the Pros and Cons to this program.

Pros

- Create as many diffuser models as needed, in a short amount of time.
- Models are accurate and created in a consistent manner.
- CNC department gets the correct data they need every time.

- Huge improvement in efficiency, allowing the design team to focus on other key design tasks.
- Getting the diffuser design finished faster helps bring your design to market sooner than your competition.

Cons

- The program is not flexible. The design must have 29 blade data points.
  - This means that the aero engineer is limited in the number of points needed to optimize their design. Sometimes they may need less than 29 blade data points and sometimes they may need more than 29 blade data points.
  - The program is also not useable for older designs that do not use 29 points.
- You have to model the diffuser up front as a master model.
  - This means there is still an upfront cost in creating the model, but it is a minimal cost with a quick payback period.
  - Depending on variation in designs, we would need to create and maintain multiple master models.

Impeller Design Program

In the early phase in our knowledge of iLogic, we recognized a need to automate the design of our impellers. This component was one we felt could be easily created through iLogic programming. The impeller has many design variations across product platforms, but many similarities in the design. Here are the reasons we selected the impeller to be created though an iLogic program.

Reasons to Create the Impeller iLogic Program

- The impeller is a complex design with many design variations, but it is consistent in the type of information needed to design.
  - The impeller outside diameter varies, but is a consistent design input.
  - The impeller profile cuts vary, but is a consistent design input.
  - Blade geometry varies and is driven by blade data points.
- The impeller design is controlled with parameters. This allows changes to be easily made with an iLogic program.
  - There are many user parameters created to drive the model.
- We wanted our impeller models to be an automated design to create consistent and accurate models for our CNC department and our external suppliers.

How We Created the iLogic Program

- We created a master model up front, meaning we made the model up front.
  - Model is comprised of parameter driven dimensions, for every key feature.
  - A set number of 29 blade points, was determined to be the standard.
- The design data and all other family related design variations are stored in an external Excel spreadsheet for extrapolation, when the iLogic program runs.

- The iLogic Code is rather simple as the main part of the code is a “Do While Loop” that reads and pushes data from Excel to change the parameters and create the new model. Below is a snippet from some of the impeller code that we use to create the diameter and blade profile cuts.

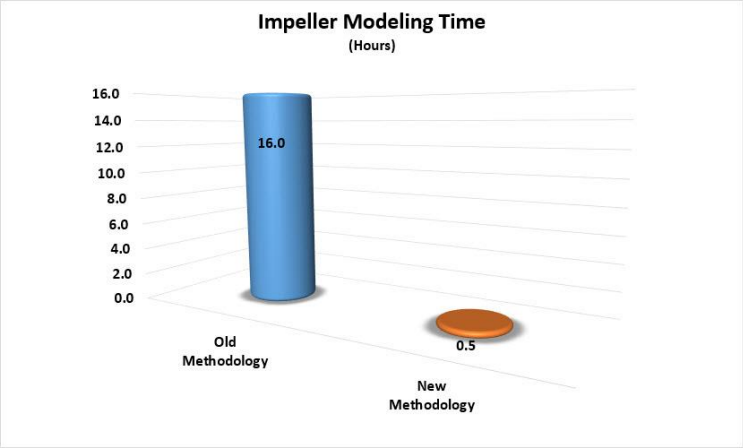
```
'---------------------------------------IMPELLER PROFILE CUT & DIAMETER CUT CREATION-----------
Do While J<=L                                                                   'DIAMETER CUT DO LOOP
        COLLET = GoExcel.CellValue("3rd Party:"+EXCEL+".xlsx", "Impeller CAD Shifted Data Point", "N68")
        IC = GoExcel.CellValue("3rd Party:"+EXCEL+".xlsx", "Impeller CAD Shifted Data Point", COLLET & "72")
        GoExcel.CellValue("3rd Party:"+EXCEL+".xlsx", "Impeller CAD Shifted Data Point", "E8") = IC      'PUSH DIAMTER CUT TO EXCEL
        SOD = Round(GoExcel.CellValue("3rd Party:"+EXCEL+".xlsx", "Impeller CAD Shifted Data Point", "E10"),3)
        Do While z <= NP                                                        'Impeller Flow Cuts
                GoExcel.CellValue("3rd Party:"+EXCEL+".xlsx", "Impeller CAD Shifted Data Point", "E7") = CIE
                iProperties.Value("Summary", "Author") = iProperties.Value("Project", "Designer")       'UPDATE AUTHOR iProperty
                CFM = GoExcel.CellValue("3rd Party:"+EXCEL+".xlsx", "Impeller CAD Shifted Data Point", "E9")
                MOD = Round(Impeller_OD,3)                                      'MAXIMUM DIAMTER CUT PARAMETER
        '------------------READ ALL PROFILE CUT VALUES FROM EXCEL SPREAD SHEET--------------------
                WWW = 0
                ZZ = WWW
                For NN = 5 To 33
                        WWW = WWW + 1
                        ZZ = WWW
                        Parameter ("R" + ZZ) = GoExcel.CellValue("3rd Party:"+EXCEL+".xlsx", "Impeller CAD Shifted Data Point", "J"&NN)
                        Parameter ("Z" + ZZ) = GoExcel.CellValue("3rd Party:"+EXCEL+".xlsx", "Impeller CAD Shifted Data Point", "I"&NN)
                Next
'---------------------------------------CLOSE AND SAVE EXCEL DOCUMENT-------------------------
                RuleParametersOutput()                  'NEEDED FOR PART NUMBER/PROFILE CUT TO BE RIGHT
                InventorVb.DocumentUpdate()                     'NEEDED FOR PART NUMBER/PROFILE CUT TO BE RIGHT
'-------------------------------------------------------------------------------------
                iLogicVb.RunRule("IMPELLER CUT")                'IMPELLER CUT MODEL CHECK
                iLogicVb.RunRule("UPDATE MODEL")
                P=K+"-"+IC                                      'PROFILE CUT NAME i.e. "DR1-1100"
                O=IPN+"-"+P                                     'P/N DESIGNATION  i.e "10802-DR1-1100"
                N=O+".ipt"                                      'THIS IS FILE NAME GENERATION i.e. "10802-DR1-1100.ipt"
                PART = O                                        'PART NUMBER PARAMETER
                PROFILE = P                                     'PROFILE NAME PARAMETER
                iProperties.Value("Summary", "Comments") = PROFILE      'PUSH PROFILE NAME TO iProperties
                iLogicVb.UpdateWhenDone = True
                RuleParametersOutput()                  'NEEDED FOR PART NUMBER/PROFILE CUT TO BE RIGHT
                InventorVb.DocumentUpdate()                     'NEEDED FOR PART NUMBER/PROFILE CUT TO BE RIGHT
                GoExcel.DisplayAlerts = False
                iLogicVb.RunRule("INTERNAL DRAWING TABLE")      'POPULATE DATA FOR DRAWING TO EXCEL
                iLogicVb.RunRule("UPDATE MODEL")
                '------------------------Below saves Model to specific folder-----------------------------------------------
                FileToSave = ThisDoc.Path & "\"+IPN+"\" + N
                ThisDoc.Document.SaveAs(FileToSave, True)
                '-------------------------------------------------------------------------------------------------------
                z=z+1                                   'COUNT UP PROFILE NUMBER
                CIE = z                                 'PROFILE NUMBER PARAMETER
                K = PD & z                              'NEW PROFILE DESIGNATION
                CI = K                                  'PROFILE DESIGNATION PARAMETER
                RuleParametersOutput() 'NEEDED FOR PART NUMBER/PROFILE CUT TO BE RIGHT
                InventorVb.DocumentUpdate()             'NEEDED FOR PART NUMBER/PROFILE CUT TO BE RIGHT
        Loop
        z = 1
        CIE = z
        K = PD & z                                      'NEW PROFILE DESIGNATION
        CI = K                                          'PROFILE DESIGNATION PARAMETER
        J = J+1
        GoExcel.CellValue("3rd Party:"+EXCEL+".xlsx", "Impeller CAD Shifted Data Point", "M68")          = J
        RuleParametersOutput()          'NEEDED FOR PART NUMBER/PROFILE CUT TO BE RIGHT
        InventorVb.DocumentUpdate()             'NEEDED FOR PART NUMBER/PROFILE CUT TO BE RIGHT
Loop
```

Impeller Design Efficiency Improvement

The key takeaway from this Impeller Model iLogic program was the efficiency improvement in the design task, our design team is seeing.

- The Original design methodology, without the iLogic program, would take a designer about 16 hours to complete, including the drawing.
- New Methodology now takes about .5 hours to complete, including the drawing.

- This is a 96.9 % improvement in efficiency.



Impeller Program	
Original Time (Hours)	16.0
New Time (Hours)	0.5
Time Delta (Hours)	15.5
Percent Improvement	96.88%

Pros & Cons to this Impeller Design Methodology

With any workflow or program, one can always find advantages and disadvantages. This iLogic program is no different. Here are the Pros and Cons to this impeller design program.

Pros

- Create as many impeller models as needed, in a short amount of time.
- Models are accurate and created in a consistent manner.
- CNC department gets the correct data they need every time.
- Huge improvement in efficiency, allowing the design team to focus on other key design tasks.
- Getting the impeller design finished faster, helps bring your design to market sooner than your competition.

Cons

- The program is not flexible. The design must have 29 blade data points.
  - This means that the aero engineer is limited in the number of points needed to optimize their design. Sometimes they may need less than 29 blade data points and sometimes they may need more than 29 blade data points.
  - The program is also not useable for older designs that do not use 29 points.
- You have to model the impeller up front as a master model.
  - This means there is still an upfront cost in creating the model, but it is a minimal cost with a quick payback period
  - Depending on variation in designs, we would need to create and maintain multiple master models.

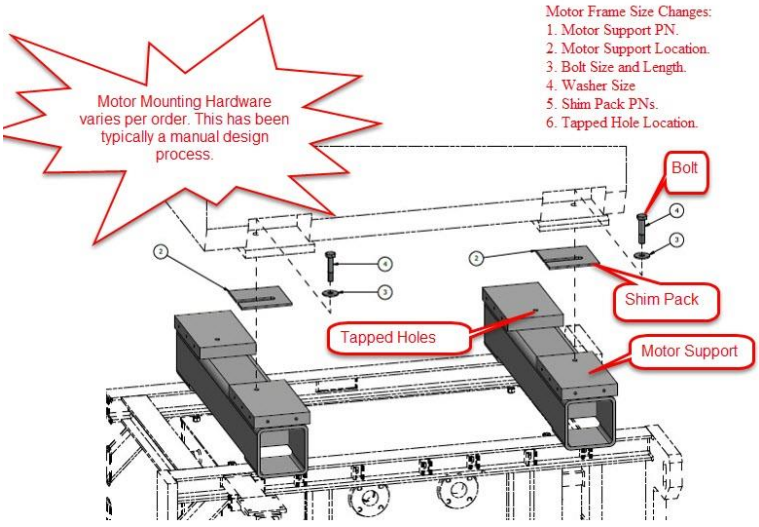
Motor Mounting Hardware Design Program

The more we learned about iLogic, the more we started to look at other tasks that were repetitive, but also time-consuming. We recognized a need to automate the design of our motor mounting hardware on one of our product lines. This task took our design team on average 20 hours to complete, from start to finish. This time included the drawings required to document the design.

Almost every order we receive, the motor is different. The difference could be a motor frame size, motor type or a different supplier. This means that the motor mounting hardware is always changing and needs to be selected and designed for almost every order. This causes changes to the following:

- Motor support PN and motor support mounting location.
- Motor mounting bolt size and washer size will change along with the bolt length changing.
- Shim pack PNs will change per motor
- Tapped hole locations for mounting the motor to the support will change.

With all these variations and the amount of time needed to be spent on designing the proper configuration, we decided to create a program to create the design, BOM and update the drawings for each customer order.



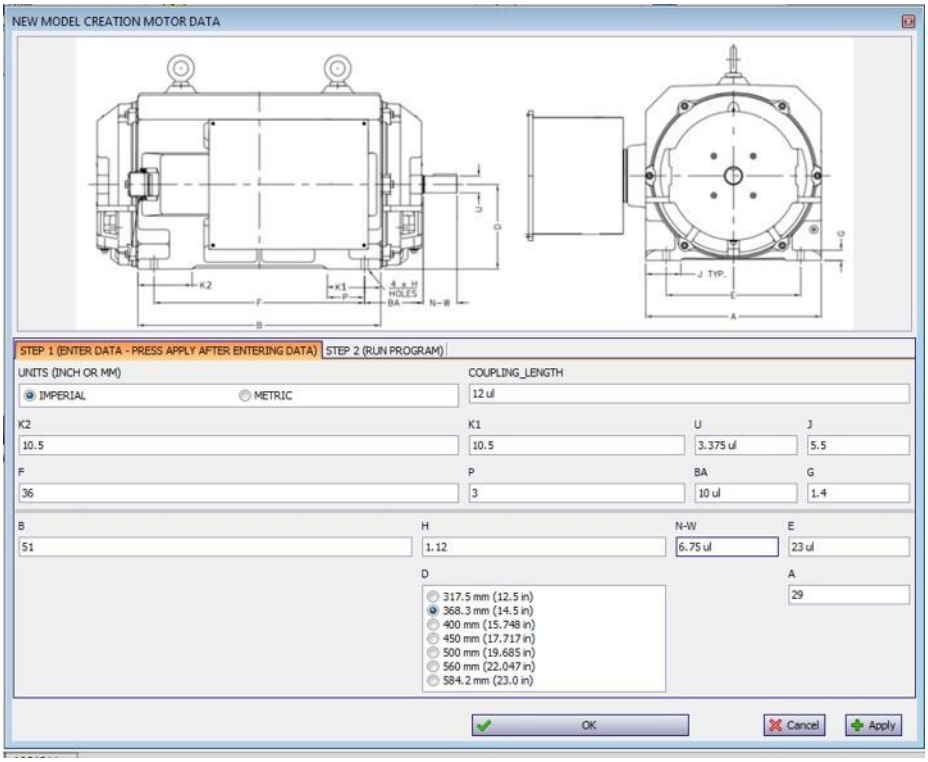
Reasons to Create the Motor Mounting Hardware iLogic Program

- The design team spent, on average, 20 hours to design the proper motor mounting configuration.
    - Motor Support PN and motor support mounting location.
    - Motor Mounting Bolt size and washer size will change along with the bolt length changing.
    - Shim Pack PNs will change per motor.
    - Tapped hole locations for mounting the motor to the support will change.

    - Many calculations are required.
- The design team has made errors in the past that have cost us a delay in assembly. We wanted an efficient and accurate tool to churn out the design, so our design team can focus on other important tasks.

How We Created the iLogic Program

- We created a master model up front of a motor mounting hardware design, meaning we made the models and drawings up front.
- Motor data, specific to the order, is entered in an iLogic Form.
    - This data is stored as user parameters in the master model.



- A rule then pushes the inputs to Excel that calculates all the proper design variables.
    - The code checks to see if the design already exists, in order to avoid duplicate PNs. The program will then inform you of the PN to use.
    - If the design does not currently exist, then the data is sent back to the model, through the iLogic Code.
- A new model is created and the drawing is updated through the iLogic Code
- The iLogic Code is rather simple, as the main part of the code is just pushing and reading data from Excel and running other rules with similar code. The code in the other rules will replace components based on the motor data inputted in the form. Below is a snippet from some of the code we use.

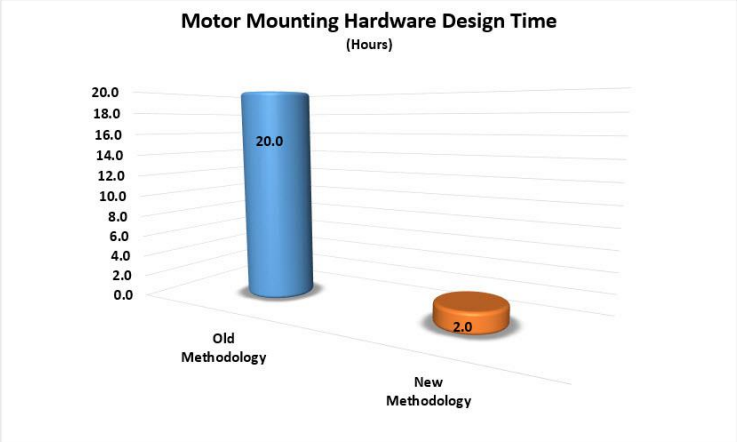
```
InventorVb.DocumentUpdate()
GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "D2")=UNITS
If UNITS = "METRIC" Then
		If D2="317.5 mm (12.5 in)" Then
			D1=317.5
		ElseIf D2="368.3 mm (14.5 in)" Then
			D1=368.3
		ElseIf D2="400 mm (15.748 in)" Then
			D1=400
		ElseIf D2="450 mm (17.717 in)" Then
			D1=450
		ElseIf D2="500 mm (19.685 in)" Then
			D1=500
		ElseIf D2="560 mm (22.047 in)" Then
			D1=560
		Else
			D1=584.2
		End If
Else
		If D2="317.5 mm (12.5 in)" Then
			D1=12.500
		ElseIf D2="368.3 mm (14.5 in)" Then
			D1=14.500
		ElseIf D2="400 mm (15.748 in)" Then
			D1=15.748
		ElseIf D2="450 mm (17.717 in)" Then
			D1=17.717
		ElseIf D2="500 mm (19.685 in)" Then
			D1=19.685
		ElseIf D2="560 mm (22.047 in)" Then
			D1=22.047
		Else
			D1=23.000
		End If
End If
GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "D18")=B1	'SETS EXCEL CELLS TO INPUTED PARAMETER VALUES
GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "D19")=F1
GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "J20")=K1
GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "D20")=K2
GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "J18")=BA
GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "J19")=H1
GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "J21")=P1
GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "P19")=N_W
GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "P20")=U1
GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "V18")=A1
GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "V19")=E1
GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "V20")=G1
GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "V21")=J1
GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "M23")=COUPLING_LENGTH
GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "P18")=D1

'AFTER CALCULATION ARE DONE THE VALUES ARE SENT BACK TO THE PARAMETERS
MOTOR_MOUNT_THICKNESS_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "G1")
BOLT_PN_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "G2")
NUT_PN_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "G3")
WASHER_PN_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "G4")
MOUNT_PN_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "G5")
FJBL_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "G6")
RJBL_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "G7")
MOTOR_MOUNT_HEIGHT_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "V1")
FROM_GC_CENTER_TO_FM_CENTER_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "V2")
FROM_GC_CENTER_TO_RM_CENTER_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "V3")
FROM_FM_CENTER_TO_HOLE_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "V4")
FROM_RM_CENTER_TO_HOLE_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "V5")
HOLE_TO_HOLE_WIDTH_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "V6")
HOLE_DIAMTER_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "V7")
TOTAL_MOTOR_WIDTH_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "V8")
MOUNT_THREAD_SIZE_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "V9")
WASHER_LOC_1 = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "G8")
ALARM_1=GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "A10")
ALARM_2=GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "A11")
DWG_TABLE_THREAD_SIZE = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs", "V12")
If ALARM_1<>"" Then 					'IF WIDTH IS EXCEEDED
		MessageBox.Show(ALARM_1, "WIDTH ALARM")
End If
If ALARM_2<>"" Then 					'IF LENGH EXCEEDS THE STANDARD BASE
```

```
                MessageBox.Show(ALARM_2+" REQUIRED", "LENGTH ALARM")
End If
BGD = GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "AQ55")
Mount_Width= GoExcel.CellValue("3rd Party:Embedding 2", "Inputs (2)", "AQ35")
RuleParametersOutput()
InventorVb.DocumentUpdate()
iLogicVb.RunRule("PART NUMBER CHECK")                    'RUNS THE RULE TO CHECK IF GEOMETRY ALREADY EXISTS
```

Motor Mounting Hardware Design Efficiency Improvement

Once again, the key takeaway from this Motor Mounting Hardware iLogic design program is the efficiency improvement in the design task, our design team is seeing and the accuracy in the design.

- The Original design methodology, without the iLogic program, would take a designer about 20 hours to complete, including the drawing.
- New Methodology now takes about 2 hours to complete, including the drawing.
- This is a 90 % improvement in efficiency.



Motor Mounting	
Original Time (Hours)	20.0
New Time (Hours)	2.0
Time Delta (Hours)	18.0
Percent Improvement	90.00%

Pros & Cons to this Motor Mounting Hardware Design Methodology

With any workflow or program, one can usually find advantages and disadvantages. We have not seen any cons to this iLogic program. Here are the Pros to this iLogic program we created.

Pros

- We took a very involved repetitive design task and automated it to create accurate designs.
- Generates BOMs, Models and MFG assembly drawings all at once.
- Huge improvement in efficiency, allowing the design team to focus on other key design tasks.

Reasons for Moving to API Usage in Writing our Code

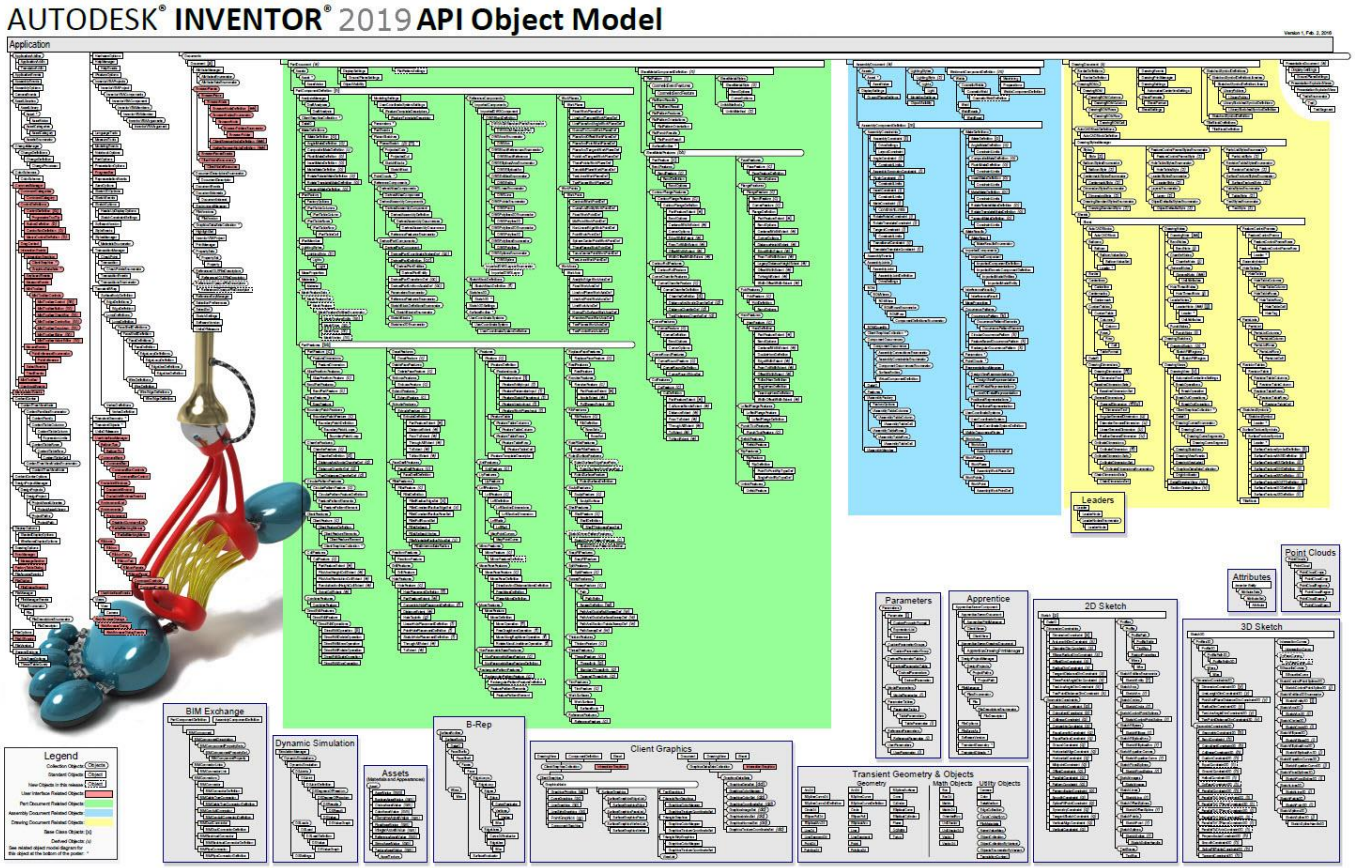
Simple iLogic codes are an excellent tool for repetitive designs with small variations. Using existing parameterized models with iLogic can allow a user to automatically manipulate parts. While this can work well, it does not allow for much flexibility. Some of the more advanced API functions allow sketches and features to be completely modeled using code, allowing for whatever flexibility the user desires to program.

For example, an impeller can have several variations such as the inclusion of splitter blades or different attachment methods such as an interference or polygon fit. When using iLogic, this led FS-Elliott to have multiple “Master Models” to maintain, each of which had slight differences in the code and their own embedded spreadsheets for the data, which consisted of the following:

- Main Blades Only, Interference Fit
- Main Blades Only, Polygon Fit
- Main Blades and Splitter Blades, Interference Fit
- Main Blades and Splitter Blades, Polygon Fit

In addition to having four Master Models, as mentioned previously since these are already created models with parameters, the number of aerodynamic points that can be used is fixed. This was not good for future designs since it restricts the flexibility of the designer. It was also not accommodating of previous designs, many of which use varying number of points to define the geometry.

By utilizing the functionality of the API and smart use of the Loops and If statements that were discussed previously, a single program (or subset of modules) can be used with a single spreadsheet. Since the API allows for sketches and features to be created from scratch, the designer is not limited to any set number of points and the program can accommodate variations in older designs.



API Coding (Examples / Tips & Tricks)

Before getting started with the API functionality, it can be incredibly helpful to review the API help section from within Inventor that was introduced previously. This help section includes the API User's Manual, which includes helpful instructions on how to utilize various aspects of the API. It includes the API Reference Manual, which includes all of the API Objects and their associated properties, as well as the available Enumerators, which are used to define aspects of certain objects. Finally, the help includes several Sample Programs that show real examples of uses for the API with the code in VBA. The API Object Model shown above is also a helpful reference and can be found in the help or online.

The API commands can be accessed in a number of ways including using the VBA editor built into Inventor, or by creating an add-in or standalone .exe in something like VB.NET. The API commands can also be accessed through the iLogic Rule editor, which can be convenient for embedding rules within a part or assembly file or for creating external rules that can be shared across files. In the 2019 release of Inventor, the rule editor contains helpful tool tips when typing API expressions. For further ease of use, forms can be created within the iLogic browser with click buttons to run rules and user fields to edit parameters. For ease of access, the following codes are executed through embedded Rules.

Create Box Example (Extrude)

While the API help contains example codes, we want to show some simple examples with detailed explanations of the code. The first example will show how to extrude a simple box.

Note: This file has been uploaded as “Create Box.ipt” and includes an embedded rule with the code.

Below is the start of the code snippet used to do this including comments:

```
'Creates a reference to the active document
Dim oDoc As Document = ThisApplication.ActiveDocument
'Specifies the active document is a part file type
Dim oPartDoc As PartDocument
'Creates a reference to the part document type
oPartDoc = ThisApplication.ActiveDocument
'Creates a component definition reference. This is used when creating
sketches, parameters, planes, etc.
Dim oCompdef As ComponentDefinition = oDoc.ComponentDefinition
'Creates a part specific component definition. This is used for part specific
features like lofts or revolves.
Dim oPartDef As PartComponentDefinition
'Creates reference between part component definition and part document type.
oPartDef = oPartDoc.ComponentDefinition
```

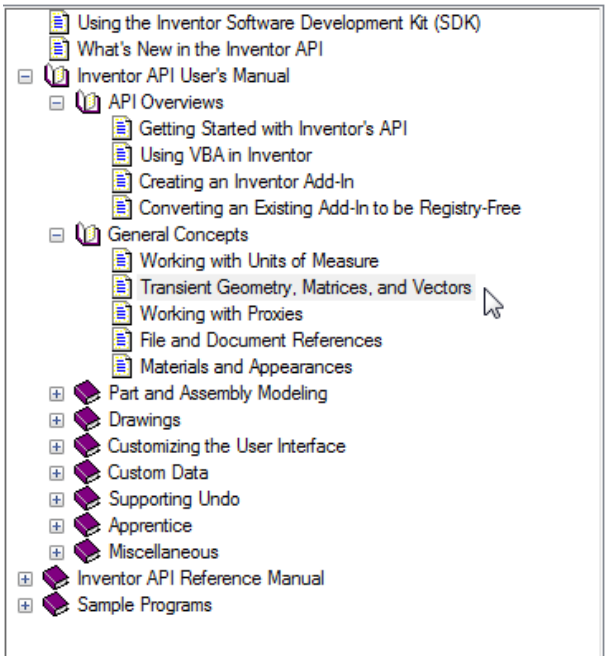
The above code is helpful to include whenever utilizing API commands, as it creates references to several properties within Inventor and is used in various ways in the remaining code. Next, some code is included that deletes all of the features and sketches in the current part.

```
'Delete Features, The below loop deletes all the features in the model to
start from scratch
Dim oFeat As PartFeature
For Each oFeat In oCompdef.Features
      oFeat.Delete()
      On Error Resume Next
Next

'Delete 2D Sketches, Deletes all the 2D sketches to start from scratch
Dim oSketch As PlanarSketch
For Each oSketch In oCompdef.Sketches
      oSketch.Delete()
      On Error Resume Next
Next

'Turns Error Checking Back On
On Error GoTo 0
```

In the above code, within the loops error checking is turned off so that the loops will continue to delete the features and sketches even if an error is encountered. It is then turned back on at the end of the snippet. This code can also be reused in various programs and can be helpful when

the desire of the program is to start from a blank model and create the sketches and features from scratch.

The next small snippet of code declares a “transient geometry” reference. Transient geometry is very useful when working with the API as it allows you to create and access points in space without actually creating a sketch point or work point. This tg variable will be used shortly. The API help within inventor includes a detailed discussion on transient geometry and how it can be used within the User Manual under the General Concepts section.




```
'Declares a transient geometry reference.
This is used to create points and other
objects without creating a visible instance
of the object.
Dim tg As TransientGeometry
'Links transient geometry to the open
instance of inventor.
tg = ThisApplication.TransientGeometry
```

Next, a 2D sketch is declared, created, and renamed. As mentioned in the comment within the code the sketch is made on the XZ plane which is accessed using WorkPlanes(2). The # of the work plane follows the order they appear in the model tree. This is also true for work axes.

```
'Declares an inventor planar sketch aka 2D sketch
Dim oSketch1 As Inventor.PlanarSketch
'Creates a 2D sketch on the XZ Plane. WorkPlanes(#) are in the same order
they appear in the model tree under origin so YZ=1, XZ=2, and XY=3.
oSketch1 = oPartDef.Sketches.Add(oPartDef.WorkPlanes(2))
'Renames the sketch
oSketch1.Name = "Box Sketch"
```

With the sketch created, next several sketch lines are declared. When creating sketches through the API, lines and other sketch items can be declared once and reused, or a separate item can be declared for each instance as is shown below. Declaring them separately is preferred, especially when working with dimensions and constraints as this allows each sketch object, in this case lines, to be referenced individually.

```
'Declares a few 2D sketch lines
Dim oLine1 As Inventor.SketchLine
Dim oLine2 As Inventor.SketchLine
Dim oLine3 As Inventor.SketchLine
Dim oLine4 As Inventor.SketchLine
'Declares an Offset Dimension Constraint, this is like manually adding a
dimension from a line to another line
Dim oLineLength As OffsetDimConstraint
```

The offset dimension constraint declared above is useful for creating a sketch dimension between a line and some other sketch entity such as another line, a point, a circle, etc. Another useful dimension type not used here is a “TwoPointDistanceDimConstraint” which defines a distance between two points.

The next snippet of code actually creates the sketch for the box and adds two dimensions for the height and width. Refer to the comments within the code for how “tg” is used and how the lines are made to be coincident to one another.

```
'Create Box Sketch
'Here the previously declared tg is used to reference coordinates on the
plane without actually creating a sketch point or work point.
oLine1 = oSketch1.SketchLines.AddByTwoPoints(tg.CreatePoint2d(0, 0),
tg.CreatePoint2d(1, 0))
'The .EndSketchPoint property makes the previous line coincident to this new
line.
oLine2 = oSketch1.SketchLines.AddByTwoPoints(oLine1.EndSketchPoint,
tg.CreatePoint2d(1, 1))
oLine3 = oSketch1.SketchLines.AddByTwoPoints(oLine2.EndSketchPoint,
tg.CreatePoint2d(0, 1))
'The .StartSketchPoint property connects the final line to the first point of
the first line.
oLine4 = oSketch1.SketchLines.AddByTwoPoints(oLine3.EndSketchPoint,
oLine1.StartSketchPoint)

'Creates a dimension between two of the parallel lines
oLineLength = oSketch1.DimensionConstraints.AddOffset(oLine1, oLine3,
tg.CreatePoint2d(-0.5, 0.5), False)
'Creates a dimension between the other two parallel lines
oLineLength = oSketch1.DimensionConstraints.AddOffset(oLine2, oLine4,
tg.CreatePoint2d(0.5, -0.5), False)
```

Within the code to create the dimensions, following “AddOffset”, the first two entries are the two entities the dimension is created between, followed by a transient geometry point that locates the text of the dimension on the sketch. The “False” makes the dimension from one entity to the other. Changing this to “True” will make this a linear diameter dimension centered on the first line chosen. This is similar to making a dimension to a centerline manually in a sketch. Finally, an optional input is not shown that when set to “True” will make the dimension driven. See the DimensionConstraints Object and the AddOffset Method in the help for more information.

While adding the dimensions sets the distance between each side of the box, the sketch is still not fully constrained. In order to fully constrain the sketch some additional geometric constraints are used. First, the X-axis and Z-axis are projected to use for constraints later on.

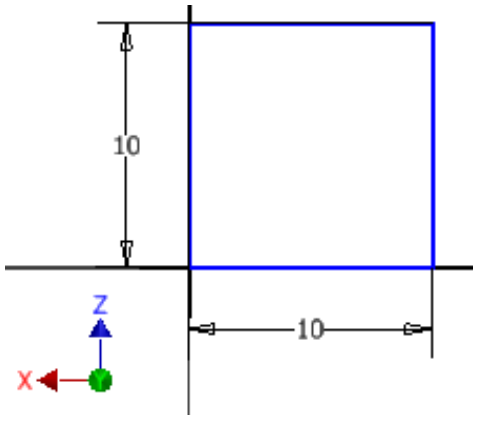
```
'Declares a sketch point
Dim oXaxis As SketchLine
Dim oZaxis As SketchLine
'Projects the X and Y axes on the sketch. WorkAxes(1) is the X axis.
WorkAxes(3) is the Z axis.
oXaxis = oSketch1.AddByProjectingEntity(oCompdef.WorkAxes(1))
oZaxis = oSketch1.AddByProjectingEntity(oCompdef.WorkAxes(3))
```

The “AddByProjectingEntity” command can be used to project any of the origin axes or planes using the “WorkAxes(#)” or “WorkPlanes(#)”. This command can also be used to project named sketch entities from one sketch to another, even from 3D sketches to 2D sketches.

Next several geometric constraints are used to constrain the sketch including a perpendicular constraint, a coincident constraint, and a horizontal constraint. Creating these through code is identical to doing it manually; you choose which type of constraint you would like to make, and then choose the sketch entities you would like to constrain. See the GeometricConstraints Object within the help for more types of constraints that can be used.

```
'Create a Perpendicular Geometric constraint between line 1 and line 4
oSketch1.GeometricConstraints.AddPerpendicular(oLine1, oLine4)
'Create a Coincident Geometric constraint between the start point of line 1
and the projected X-axis.
oSketch1.GeometricConstraints.AddCoincident(oLine1.StartSketchPoint, oXaxis)
'Create a Coincident Geometric constraint between the start point of line 1
and the projected Z-axis.
oSketch1.GeometricConstraints.AddCoincident(oLine1.StartSketchPoint, oZaxis)
'Create a Horizontal Geometric constraint on line 1 to fully constrain the
sketch.
oSketch1.GeometricConstraints.AddHorizontal(oLine1)
```

The above code makes the sketch shown here, which is now fully constrained. Note that the dimensions shown are in millimeters, whereas the default for values entered using the API are in centimeters, therefore the values of 1 used above correspond to 10 mm.



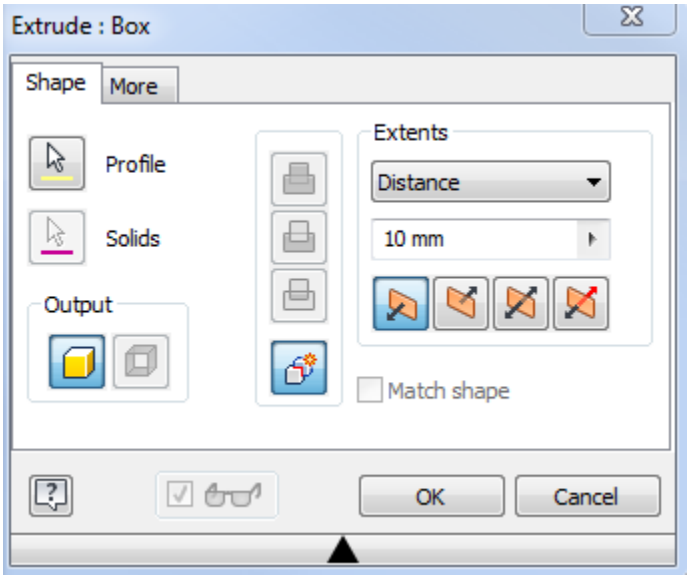
With the sketch completed and fully constrained, the next step is to define a profile for use with an extrude feature. This code simply defines the sketch that was just created as a Profile object. The AddForSolid is used to create a solid feature. AddForSurface can also be used to create a surface feature.

```
'Declares a profile.
Dim oProfile1 As Profile
'Adds the box sketch to the profile.
oProfile1 = oSketch1.Profiles.AddForSolid
```

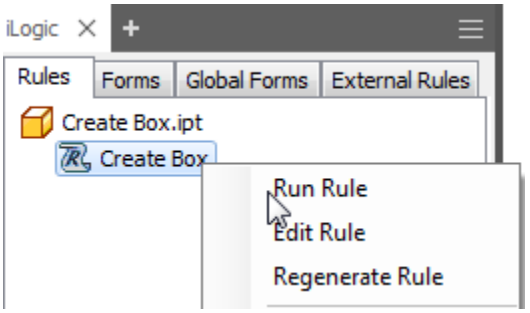
Finally, the extrude feature can be created. The below code declares an extrude feature, creates an extrude feature, and renames the feature in the model tree. Creating the feature works the same way as if you were creating the feature manually within inventor, you choose which profile to extrude, the distance you want to extrude it, the direction you want extrude it, and what type of operation it should be (i.e. join, cut, etc.) Since this is the first feature created in this model, join functions the same as creating a new body, but this code can be replaced with kNewBodyOperation to create a new body as well.

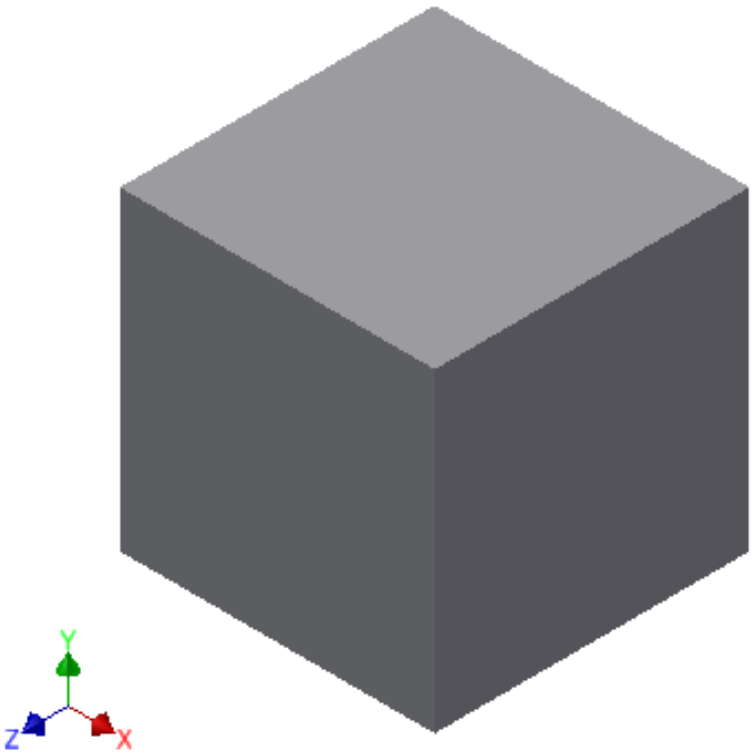
```
'Declares an extrude feature
Dim oExtrude1 As ExtrudeFeature
'Creates the extrude features using the create profile, a distance of 1 cm
(or 10 mm), extrudes in the positive direction, and does a join operation.
oExtrude1 = oPartDef.Features.ExtrudeFeatures.AddByDistanceExtent(oProfile1,
1, kPositiveExtentDirection, kJoinOperation)
'Renames extrude feature
oExtrude1.Name = "Box"
```

From the code above it can be seen that creating the extrude feature with code is equivalent to creating it manually and is identical to making the selections shown in the screen shot below.



This completes the code necessary to extrude a box from a fully constrained sketch. Running the rule, by right clicking on the Create Box rule in the iLogic browser and selecting Run Rule will execute the code and create the box shown below.





Create Cylinder with Form Example (Revolve)

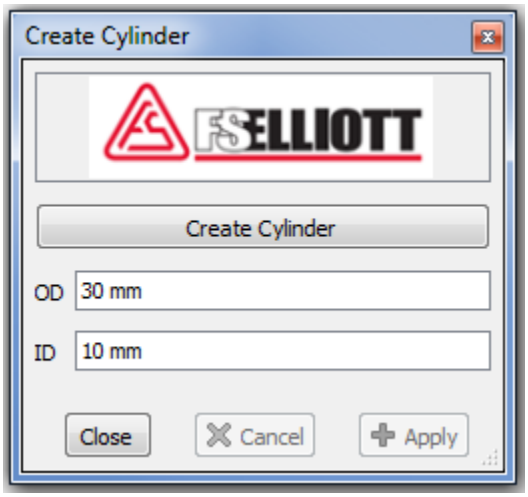
Another simple example is creating a cylinder with a bore cut. This example will use an iLogic form to update parameters that control the model and to run the rule itself. Forms are very useful for these types of tasks. This handout will not cover how to create a form, however there is useful help on the Autodesk Knowledge Network on working with forms for unfamiliar users.

Working with Forms

Note: This file has been uploaded as “Create Cylinder.ipt” and includes an embedded rule with the code and the form.

Much of the code for this example is very similar to the previous example for creating a box, however now there are parameters used. In this case, the parameters were created manually through inventor and are updated through the form. Parameters can also be created and updated through code, as will be shown later on.

Here a very simple form is used. The button titled “Create Cylinder” is used to execute the code. The text boxes are used to update parameters for the OD or outside diameter of the cylinder and the ID or inside diameter of the ring, which also corresponds to the OD of the bore. Changing these parameters and then running the code will create cylinders with bore cuts of various sizes.



In order the access the values of the parameters within the rule, very simple code is used, which is shown below.

```
'The below code creates variables based on the
values of the parameters defined on the form.
The division by 10 converts the parameters,
which are in millimeters, to centimeters for
use with the API.
Cyl_OD = Parameter("OD")/10
Cyl_ID = Parameter("ID")/10
```

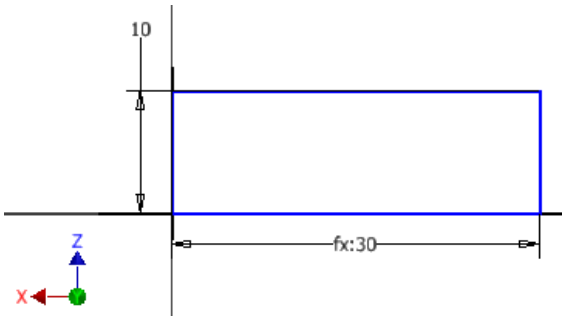
In order to use these values when creating the sketches and therefore size the cylinder, the value of these variables are used when creating the lines in the sketch as shown below.

```
'Here the previously declared tg is used to reference coordinates on the
plane without actually creating a sketch point or work point.
oLine1 = oSketch1.SketchLines.AddByTwoPoints(tg.CreatePoint2d(0, 0),
tg.CreatePoint2d(Cyl_OD, 0))
```

Similar code to the above is used when creating the other sketch line for the cylinder and the bore (with Cyl_ID). This would be enough the size the cylinder based on the parameters, however also included in the code is how to set the value of a dimension to a parameter which is shown below.

```
'Declares a model parameter.
Dim oModelParam As ModelParameter
'Creates a dimension between the two parallel lines
oLineLength = oSketch1.DimensionConstraints.AddOffset(oLine2, oLine4,
tg.CreatePoint2d(Cyl_OD / 2, -0.25), False)
'Sets the dimension value to a parameter.
oModelParam = oLineLength.Parameter
'Sets the parameters expression to "OD".
oModelParam.Expression = "OD"
```

In the above code, the dimension is created in the same way it was previously, but now the dimension is set to be a parameter and ultimately changed to be driven by the “OD” parameter. With this included in the code, the model can now be driven directly from the form. Updating either parameter, and clicking apply will update the model. The sketch, with the parameter defined “OD” is shown below.

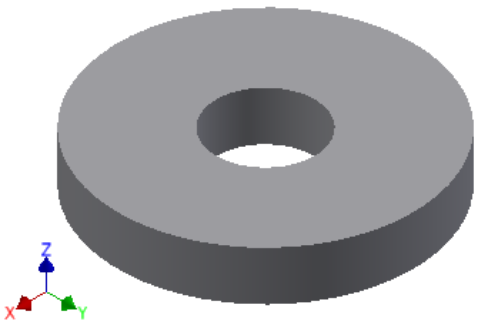


The only other major difference in code between this example and the box example, is that creating a cylinder involves using a revolve feature.

```
'Declares a Work Axis Object
Dim oWorkAxis As Object
'Makes the object a work axis defined by the Z axis
oWorkAxis = oPartDef.WorkAxes.Item(3)

'Declares a profile
Dim oProfile1 As Profile
'Adds the box sketch to the profile
oProfile1 = oSketch1.Profiles.AddForSolid

'Revolve Feature
'Declares a revolve feature
Dim oRevFeature1 As RevolveFeature
'Revolves the 2D sketch around the Z axis, creating a new body.
oRevFeature1 = oPartDef.Features.RevolveFeatures _
    .AddFull(oProfile1, oWorkAxis, kJoinOperation)
'Renames the revolve feature
oRevFeature1.Name = "Cylinder"
```

Creating the revolve works almost identically to creating an extrude, except an axis is defined to perform the revolve around. The created z work axis could also be replaced with the projected z-axis used in the sketch.

Running the code creates the model shown below. As mentioned previously, since some of the dimension values were set to parameters, the model can be controlled from the form by updating the values of these parameters.



Impeller Creation - Using the API

Now that we have shown some simple API examples, we will show how we have incorporated the API into some of our actual modeling workflows, starting with the impeller.

Creating the Impeller Blade

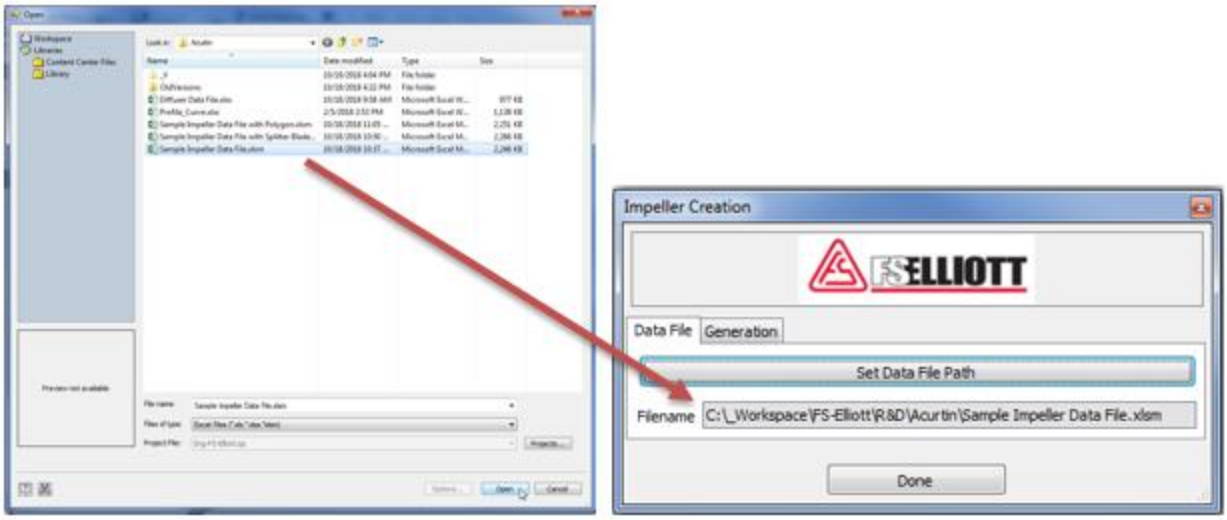
Before covering the entire impeller, some snippets from creating the impeller blade will be shown.

Note: This file has been uploaded as “Impeller Blade.ipt” and includes an embedded rule with the code and the form.

As has been discussed previously, a spreadsheet can be useful for getting values to drive a model. The below code shows how to get the file path for a spreadsheet saved outside of inventor and sets the path name to a parameter.

```
'Declare Iventor File Dialog
Dim oFileDlg As Inventor.FileDialog = Nothing

'Run File Selection Code
    InventorVb.Application.CreateFileDialog(oFileDlg)

'Allow Selection Of Excel Files
    oFileDlg.Filter = "Excel Files (*.xls;*.xlsx;*xlsm)|*.xls;*.xlsx;*xlsm"

'Get File Path, setting the initial path to the same folder that the model is
saved in.
    oFileDlg.InitialDirectory = ThisDoc.Path
    oFileDlg.CancelError = True
    On Error Resume Next

'Show Open File Dialog Window.
    oFileDlg.ShowOpen()

'Set Filename parameter to the path.
    Parameter("Filename") = oFileDlg.FileName
```

Running the above rule from the form opens the Open File dialog box and allows you to “Open” an excel spreadsheet. Choosing a file sets that files path to the parameter “Filename” as shown on the form.



Note: Not all of the code will be covered given the length; however, it is included in the embedded rule “Create Blade” within the model.

With the path to the spreadsheet set, the impeller blade can be modeled. First, the path to the spreadsheet is defined using the parameter. Given the desire to have flexibility in the number of points that are used to define the geometry, the number of points can vary and is counted in the spreadsheet and defined here. Finally, a variable is created to define whether to create dimensions or not, which is shown later on. The spreadsheet comes with this value set to “Yes” so the dimensions are created.

```
'Sets the excel reference to the parameter Filename.
Impeller_Coordinates = Parameter("Filename")
'Determines how many main blade points there are.
Blade_Count = GoExcel.CellValue(Impeller_Coordinates, "Main Blade", "AM3")
'Determines if the dimensions for the blade are going to be created.
Dimension_Value = GoExcel.CellValue(Impeller_Coordinates, "Inputs", "B21")
```

The next snippet to discuss is the creation of one of the splines that defines the blade shape. This code also shows the creation of a 3D sketch. The below code is shown as a picture to keep the formatting; the actual code can be found in the model.

```
' Blade Hub Sketch
'Declares sketch1 as a 3D sketch
Dim sketch1 As Sketch3D

'Creates the 3D sketch in inventor
sketch1 = oPartDef.Sketches3D.Add
'Names the sketch "Blade Hub"
sketch1.Name = "Blade Hub"

'Declares the collection of the spline fit points.
Dim oFitPoints As Inventor.ObjectCollection = ThisApplication.TransientObjects.CreateObjectCollection
'Declares a generic 3D point for the creation of each point.
Dim points As Inventor.SketchPoints3D = sketch1.SketchPoints3D
'Declares an array for each point to be added to and recorded.
Dim pointArray1(Blade_Count) As Inventor.SketchPoint3D
j = 0      'Creates a counter to record the points in order
a = 1      'Creates a counter to name the dimension parameters for each point correctly

'The blade coordinates start in row 29 in the spreadsheet. Loops through all the points until the blade count is reached.
For i = 29 To 29 + (Blade_Count - 1)
    'Gets the X-Coordinates on the hub on Side 1. /10 because API defaults to centimeters.
    XP = GoExcel.CellValue(Impeller_Coordinates, "Main Blade", "AC" & (i)) / 10
    'Gets the Y-Coordinates on the hub on Side 1.
    YP = GoExcel.CellValue(Impeller_Coordinates, "Main Blade", "AD" & (i)) / 10
    'Gets the Z-Coordinates on the hub on Side 1.
    ZP = GoExcel.CellValue(Impeller_Coordinates, "Main Blade", "AE" & (i)) / 10
    'Adds the point to the point array.
    pointArray1(j) = points.Add(tg.CreatePoint(XP, YP, ZP))
    'Checks to see if the dimensions should be created.
     'Checks to see if the dimensions should be created.
    If Dimension_Value = "Yes" Then
        'Creates dimension from the point to the YZ Plane.
        DimCon = sketch1.DimensionConstraints3D.AddPointAndPlaneDistance(pointArray1(j), oCompdef.WorkPlanes.Item(1))
        'Sets the dimension to a model parameter.
        oModelParam = DimCon.Parameter
        'Creates a parameter equal to the X coordinate.
        oParameter = oMyParameter.AddByValue("X1H_" & a, XP, UnitsTypeEnum.kMillimeterLengthUnits)
        'Makes the expression of the dimension equal to the parameter.
        oModelParam.Expression = "X1H_" & a
        'Creates dimension from the point to the XZ Plane.
        DimCon = sketch1.DimensionConstraints3D.AddPointAndPlaneDistance(pointArray1(j), oCompdef.WorkPlanes.Item(2))
        'Sets the dimension to a model parameter.
        oModelParam = DimCon.Parameter
        'Creates a parameter equal to the Y coordinate.
        oParameter = oMyParameter.AddByValue("Y1H_" & a, YP, UnitsTypeEnum.kMillimeterLengthUnits)
        'Makes the expression of the dimension equal to the parameter.
        oModelParam.Expression = "Y1H_" & a
        'Creates dimension from the point to the XY Plane.
        DimCon = sketch1.DimensionConstraints3D.AddPointAndPlaneDistance(pointArray1(j), oCompdef.WorkPlanes.Item(3))
        'Sets the dimension to a model parameter.
        oModelParam = DimCon.Parameter
        'Creates a parameter equal to the Z coordinate.
        oParameter = oMyParameter.AddByValue("Z1H_" & a, ZP, UnitsTypeEnum.kMillimeterLengthUnits)
        'Makes the expression of the dimension equal to the parameter.
        oModelParam.Expression = "Z1H_" & a
    End If
    'Adds the current point to the object collection.
    oFitPoints.Add(pointArray1(j))
    'Adds to the counter for a
    a = a + 1
    'Adds the counter for j
    j = j + 1
Next

' Create Side 1 Spline
'Declares a 3D Spline
Dim Spline1 As SketchSpline3D
'Creates a 3D spline in 3D sketch "Blade Hub" connected the fit points for Hub Side 1.
Spline1 = sketch1.SketchSplines3D.Add(oFitPoints)
'Makes the fit method of the spline use the Auto CAD fit method.
Spline1.FitMethod = SplineFitMethodEnum.kACADSplineFit
```

In the above code, a loop is used to create all of the points necessary for the spline shape. As each point is created, a dimension is created to each origin axis. Once each point is created, it is

added to an object collection used to create the spline, which uses the AutoCAD spline fit method. The standard and minimum energy methods can also be used.

The above code is repeated to create the three remaining splines used to define the blade and the start and end points of the respective splines are closed with 3D lines. In order create a feature from these sketches they are added to profiles in the same way was done for the box and cylinder. The blade is created with a loft feature and is defined by a loft definition, which is created with code as shown below.

```
'Create Loft Sections
'Declares an object collection for the sections of the loft. This is done
since lofts use multiple profiles.
Dim oSections As ObjectCollection
oSections = ThisApplication.TransientObjects.CreateObjectCollection
'Adds the profiles to the section collection.
Call oSections.Add(Profile1)
Call oSections.Add(Profile2)

'Loft Definition
'Declares the loft definition.
Dim oLoftDefinition As LoftDefinition
'Adds the sections to the loft definition. kNewBodyOperation adds it as a new
body.
oLoftDefinition =
oPartDef.Features.LoftFeatures.CreateLoftDefinition(oSections,
kNewBodyOperation)
```

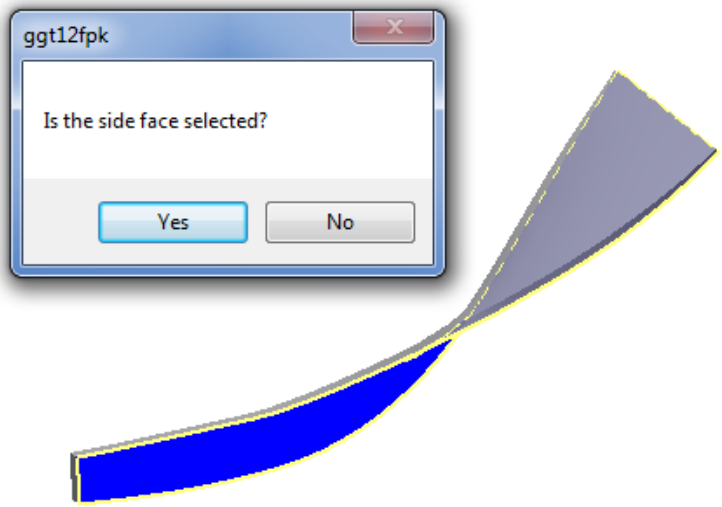
Rails are used between the loft sections in order to accurately create the blade geometry. The code for this is shown below and is very similar to the creation of the splines, however, in this case lines are created between the points and each one is added to the loft definition.

```
' Create Blade Rails on Side 1
j = 1
a = 1
Dim Profile3 As Profile3D
For i = 29 To 29 + (Blade_Count - 1)
    sketch3 = oPartDef.Sketches3D.Add
    sketch3.Name = "Blade Rails" & j
    XP = GoExcel.CellValue(Impeller_Coordinates, "Main Blade", "AC" & (i)) / 10
    YP = GoExcel.CellValue(Impeller_Coordinates, "Main Blade", "AD" & (i)) / 10
    ZP = GoExcel.CellValue(Impeller_Coordinates, "Main Blade", "AE" & (i)) / 10
    XS = GoExcel.CellValue(Impeller_Coordinates, "Main Blade", "AI" & (i)) / 10
    YS = GoExcel.CellValue(Impeller_Coordinates, "Main Blade", "AJ" & (i)) / 10
    ZS = GoExcel.CellValue(Impeller_Coordinates, "Main Blade", "AK" & (i)) / 10
    Dim line5 As SketchLine3D
    line5 = sketch3.SketchLines3D.AddByTwoPoints(tg.CreatePoint(XP, YP, ZP), tg.CreatePoint(XS, YS, ZS))
    If Dimension_Value = "Yes" Then
        DimCon = sketch3.DimensionConstraints3D.AddPointAndPlaneDistance(line5.StartSketchPoint, oCompdef.WorkPlanes.Item(1))
        oModelParam = DimCon.Parameter
        oModelParam.Expression = "X1H_" & a
        DimCon = sketch3.DimensionConstraints3D.AddPointAndPlaneDistance(line5.StartSketchPoint, oCompdef.WorkPlanes.Item(2))
        oModelParam = DimCon.Parameter
        oModelParam.Expression = "Y1H_" & a
        DimCon = sketch3.DimensionConstraints3D.AddPointAndPlaneDistance(line5.StartSketchPoint, oCompdef.WorkPlanes.Item(3))
        oModelParam = DimCon.Parameter
        oModelParam.Expression = "Z1H_" & a
        DimCon = sketch3.DimensionConstraints3D.AddPointAndPlaneDistance(line5.EndSketchPoint, oCompdef.WorkPlanes.Item(1))
        oModelParam = DimCon.Parameter
        oModelParam.Expression = "X1S_" & a
        DimCon = sketch3.DimensionConstraints3D.AddPointAndPlaneDistance(line5.EndSketchPoint, oCompdef.WorkPlanes.Item(2))
        oModelParam = DimCon.Parameter
        oModelParam.Expression = "Y1S_" & a
        DimCon = sketch3.DimensionConstraints3D.AddPointAndPlaneDistance(line5.EndSketchPoint, oCompdef.WorkPlanes.Item(3))
        oModelParam = DimCon.Parameter
        oModelParam.Expression = "Z1S_" & a
    End If
    Profile3 = sketch3.Profiles3D.AddOpen
    oLoftDefinition.LoftRails.Add(Profile3)
    j = j + 1
    a = a + 1
Next
```

With the rails created and added to the loft definition, the loft feature is created using the code shown below.

```
'Create Loft
'Declares the loft feature.
Dim oLoft As LoftFeature
'Adds the loft definition to the loft feature and creates the loft
oLoft = oPartDef.Features.LoftFeatures.Add(oLoftDefinition)
'Names the loft feature "Main Blade" in the model tree.
oLoft.Name = "Main Blade"
```

The leading edge of the blade is defined in this program using a simple full round fillet. Depending on the geometry and rotation direction of the impeller, the faces of the blade are not always defined consistently. For this reason the creation of the leading edge fillet includes code that cycles through the available faces on the blade and prompts the user to choose the side face, center face, and other side face needed to define the full round fillet using simple Yes/No message boxes. The full code for this can be found within the model. This can be useful for when you need the user to choose a face on the model when executing the program. An example of what the prompt looks like is shown below.



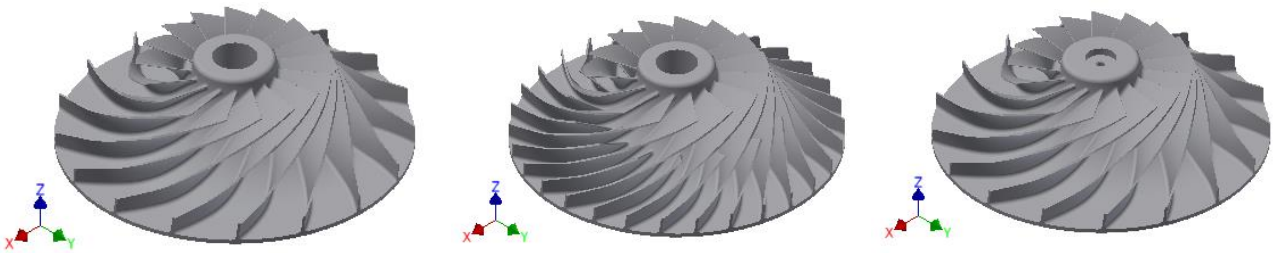
Creating the Full Impeller

Note: This file has been uploaded as “Full Impeller.ipt” and includes embedded rules with the codes and the form. Spreadsheet data files have been included for a base impeller, an impeller with splitter blades, and an impeller with a polygon attachment.

The code for the full impeller is long and complicated and will not be shown here. The program includes the code for creating the blade along with “If statements” used to determine if the impeller uses splitter blades and if it has points that define the leading edge rather than a full round. The code includes additional “If statements” used to determine what the “attachment” method is. It includes many of the feature types previously covered including revolves, extrudes, lofts, fillets, etc.

In terms of time savings, the Impeller API program offers essentially the same efficiency improvements as the iLogic program when compared to modeling an impeller manually from scratch. However, the API program offers much greater flexibility. It eliminates the need to maintain the various master models and allows the design to be defined by as many points as the user desires or an old design requires.

For the spreadsheets included here, three different impeller designs can be made with the same program. By setting the file path to different spreadsheet files, different impeller models are made.



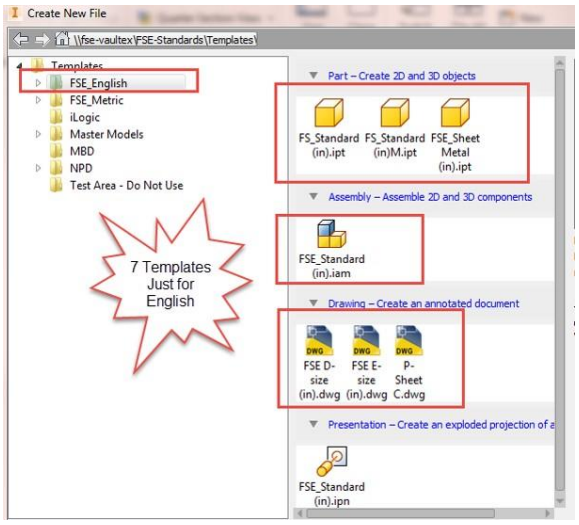
Diffuser Creation - Using the API

Note: A sample diffuser spreadsheet has been uploaded titled “Diffuser Data File.xlsx”. Sample models with embedded rules have been uploaded to make and pattern just the diffuser blades “Diffuser Blade.ipt” and to make a full diffuser “Full Diffuser.ipt”.

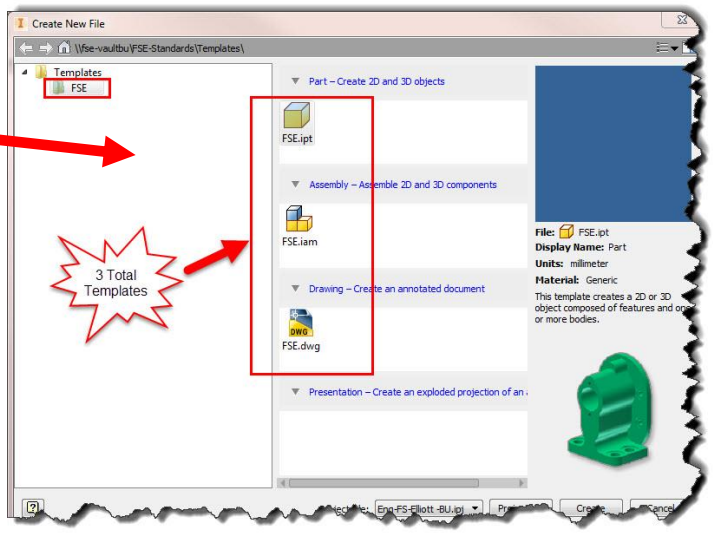
The diffuser program uses many of the same code features as the impeller program and have been presented previously. For this reason, it will not be covered in detail here; however, example models and a data file spreadsheet have been uploaded. The main advantage over the iLogic program is again that using the API functions and creating the model from scratch allows for much greater flexibility, especially in the number of data points that can be used.

Template Creation - Using the API

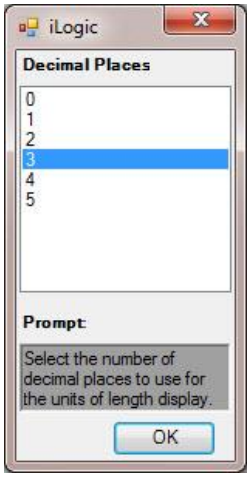
As we learned more about the API, we explored different ways to enhance our workflows. Another area we looked to improve upon was our templates. FS-Elliott at one point had 7 templates for Imperial units and 7 templates for Metric units. These templates were needed for different styles, drawing format type and various other reasons. This gave us 14 Inventor templates to maintain. We found that the templates were not always updated to the latest industry and FS-Elliott standards.

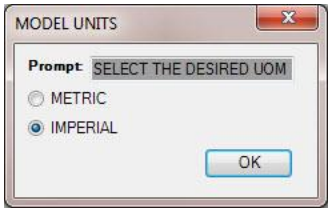


Without going into the API and how we did it, we created 3 templates that replaced the 14. These templates have rules that run as soon as the templates are opened.

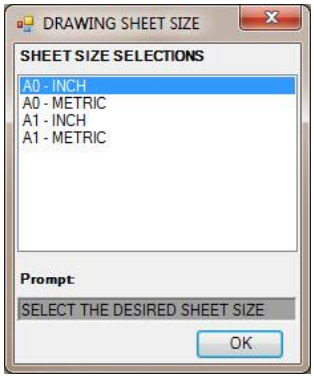


For the FSE.ipt & FSE.iam templates, the program prompts the user to select the model units and the precision of the decimal places to use within the model.

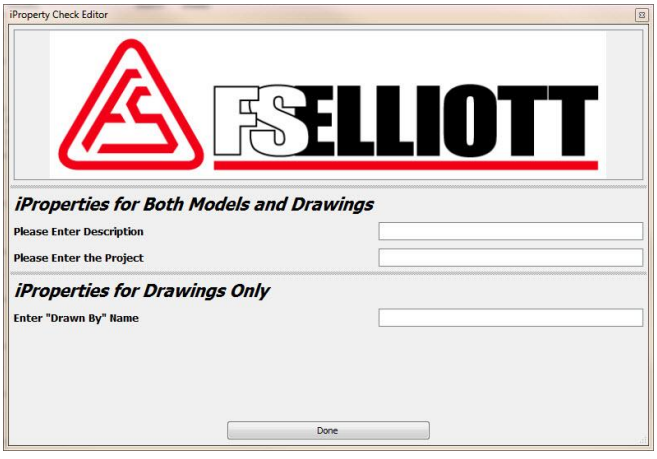




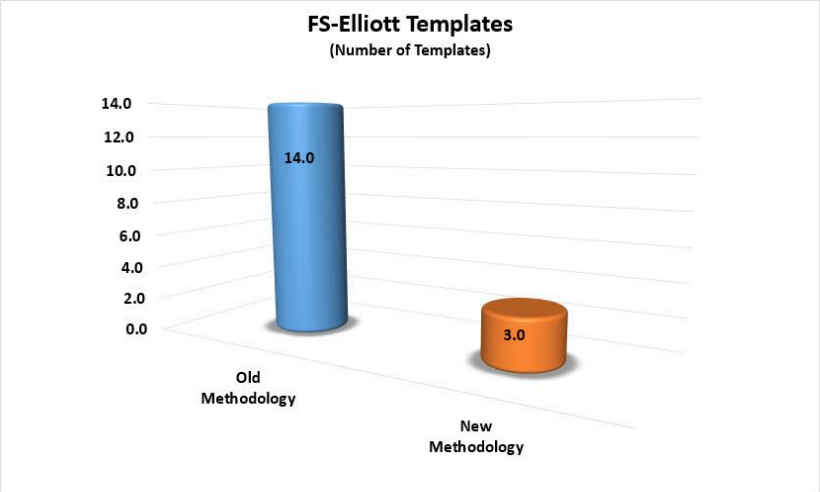
The FSE.dwg template program prompts the user to select the drawing size and the dimensional units to use on the drawing.



All the template programs prompt the user to enter mandatory iProperties and the program is set up as a loop until they enter the required iProperties.



Setting up the templates was a big reduction in the number of templates to maintain. We took the total from 14 to 3 templates, with the number of templates being reduced by 78.6 %. Not only was this a reduction, but we were able to make the templates flexible for our workflows.



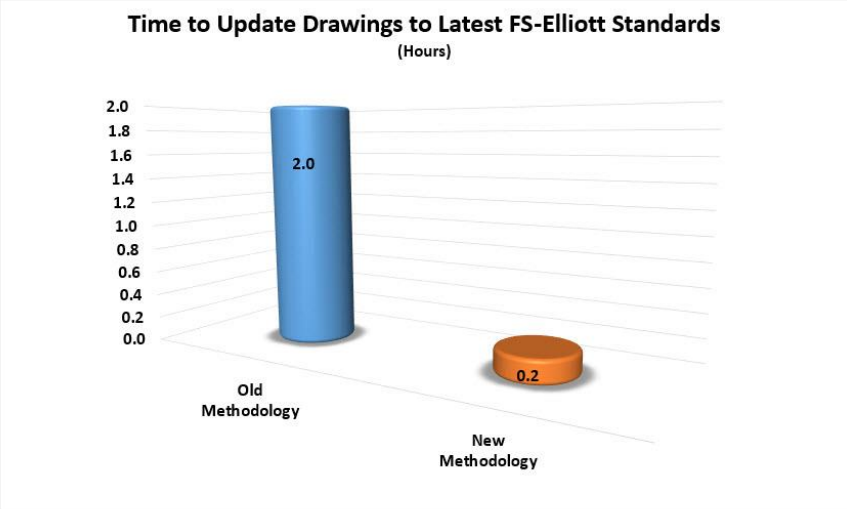
Template Program	
Original Number of	14
Number of Templates	3
Delta of Templates	11
Percent Improvement	78.57%

Update Inventor Drawings to the Latest FS-Elliott Standards – Using the API

When we created the template files using API to support them, we also took it a step further to create iLogic programs to update our FS-Elliot drawings to the latest standards. Unfortunately, there is too many rules and too much code written to show or demonstrate. We just wanted to show you the endless possibilities for efficiency improvements, which iLogic and the API can do for your company. The program is iLogic form driven and updates the following:

- Drawing Format
- iProperties
- Dimension Styles
- Drawing Weight
- Converts drawing Dimensions, labels, FCS and more to the correct styles.

For example, in order to update a 6-page drawing without using the update program, it would take the Inventor user approximately 2 hours. With the drawing update program, it now takes an Inventor user approximately .2 hours, which is a 91.5% efficiency improvement.



Update Drawings	
Original Time for 6 Sheet	2.0
New Time (Hours)	0.2
Time Delta (Hours)	1.8
Percent Improvement	91.50%

Conclusion

If your company does not use iLogic or the API to improve the efficiencies of their design workflows, maybe this class has intrigued you to dig into the possibilities of using iLogic. We have shown you many ways that FS-Elliott has used iLogic and the API to improve our workflows. It is hard to ignore the numbers we presented in how valuable this tool is to our company and how it could be for yours.

There are a number of ways to write code to automate your design processes. We have shown you how automation has helped FS-Elliott. We hope that this has started to get the thought process churning to think of ways that you can automate your design processes.

Diffuser Program	
Original Time (Hours)	4.0
New Time (Hours)	0.5
Time Delta (Hours)	3.5
Percent Improvement	87.50%

Impeller Program	
Original Time (Hours)	16.0
New Time (Hours)	0.5
Time Delta (Hours)	15.5
Percent Improvement	96.88%

Motor Mounting	
Original Time (Hours)	20.0
New Time (Hours)	2.0
Time Delta (Hours)	18.0
Percent Improvement	90.00%

Template Program	
Original Number of	14
Number of Templates	3
Delta of Templates	11
Percent Improvement	78.57%

Update Drawings	
Original Time for 6 Sheet	2.0
New Time (Hours)	0.2
Time Delta (Hours)	1.8
Percent Improvement	91.50%

Remember these questions to ask yourself to see if you should automate a design.

- Is the work repetitive?
- Will automation save time, so a designer can focus on other tasks?
- Is there a need or desire to improve consistency?
- Do you want to improve efficiency?
- Does the data output need to be provided in a certain manner, every time?
- Do you need to create a lot of models in a short amount of time?

Follow these Best Practices, when writing iLogic Code.

- Use comments in your code to make it easier to understand what your code is doing. This will help you and others down the road to understand how it works.
- Don't overdo it by overcomplicating your rule. Sometimes more rules are better than putting all your iLogic code into one rule.
- Consider making your rules so they can be reused in other projects. Why reinvent the wheel.
- When writing code it is always good to be consistent in your methods.
- Did we mention to use comments?

Remember this; experience with 3D CAD applications is an important factor when it comes to developing good workflows that are geared towards your company's deliverables. Product designs help produce new workflows and better ideas to assist in accuracy and speed to market.

My advice to you all is stay current with best practice tips and tricks by reading blogs, articles on-line, and magazines. Also, be sure to stay current with:

Autodesk Websites / Forums:

Autodesk Community Forums: https://forums.autodesk.com/
Autodesk Exchange Apps: https://apps.exchange.autodesk.com/en

Inventor Blogs:

From the Trenches with Autodesk Inventor (Curtis Waguespack):
http://inventortrenches.blogspot.com/

The CAD Setter Out (Paul Munford):
https://cadsetterout.com/