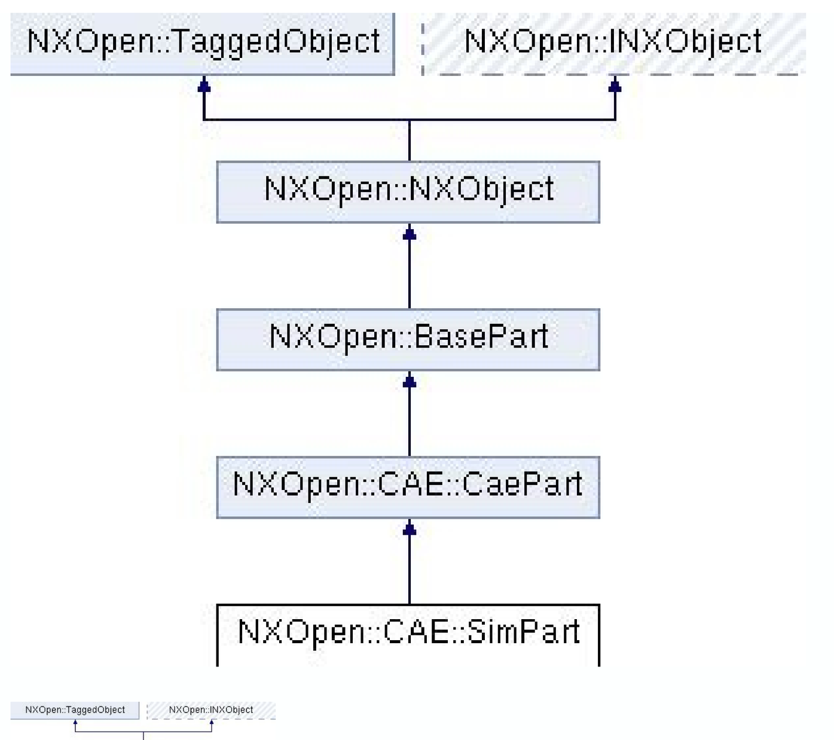
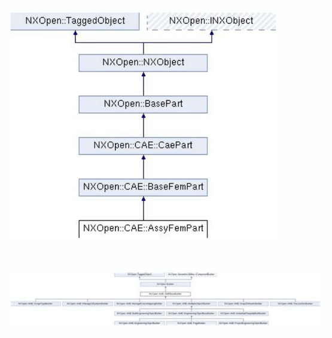
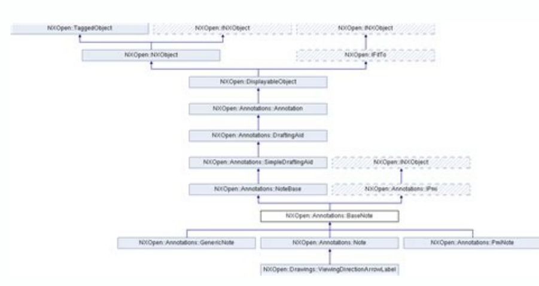
Nxopen c reference guide







Getting Started with NX Open Revision 12.0. September 2017 Table of Contents Chapter 1: Introduction ■ What Is NX Open NX Open is an Application Programming Interface (API) that lets you write programs to customize or extend NX. The benefit is that applications created this way can often speed up repetitive tasks, and capture important design process knowledge. There is a broad range of NX Open functions, which provide capabilities like Creating part geometry, assemblies, drawings, and CAE and CAM objects Cycling through the objects in a part file, reading information or performing various operations on them Creating custom user interfaces that allow users to select objects and enter data Some typical applications of these functions are: Creating part geometry or drawings according to your local standards Importing data from other sources, outside of NX Reading data from objects in a part file, and writing it out in some form of report Building custom applications to make processes faster or easier to understand Of course, these are just a few examples of what is possible. You can probably think of many little repetitive processes that you would like to automate to speed up your work or standardize your output. If you'd like a little more background information, please continue reading here. If you can't wait, and you just want to start writing code immediately, please skip to chapter 2, where we show you how to proceed. ■ Purpose of this Guide This guide is a beginner's introduction to programming using NX Open. It will get you started in writing your first few applications, and give you a sample of some of the things that are possible with NX Open. You don't need to have any programming experience to read this document, but we assume you have some basic knowledge of NX and Windows. If you are an experienced programmer, the only benefits of this document will be the descriptions of programming techniques specific to NX. The variant of NX Open that we're describing here is just a .NET library, so it can be used with any .NET-compliant language. In this document, we focus on the Visual Basic (VB) language, but in most cases it will be obvious how to apply the same techniques in other .NET languages, such as C#, IronPython, F#, etc. Other versions of NX Open are available for use with C++, Java, and Python. ■ Where To Go From Here The next two chapters show you how to write programs in two different environments. If you have no programming experience, you won't understand much of the code you see. That's OK — the purpose of these two chapters is to teach you about the programming environments and their capabilities, not about the code. Chapter 2 discusses programming using the NX Journal Editor. The only real advantage of this environment is that it requires no setup whatsoever — you just access the Journal Editor from within NX, and you can start writing code immediately. But, by the time you reach the end of the examples in chapter 2, you will probably be growing dissatisfied with the Journal Editor, and you will want to switch to a true "Integrated Development Environment" (IDE) like Microsoft Visual Studio. Chapter 3 discusses Microsoft Visual Studio. We explain how to download and install a free version, and how to use it to develop NX Open programs. If you have some programming experience, and you already have Visual Studio installed on your computer, you might want to skip through chapter 2 very quickly, and jump to chapter 3. Chapter 4 provides a very quick and abbreviated introduction to the Visual Basic (VB) programming language. A huge amount of material is omitted, but you will learn enough to start writing NX Open programs in VB. If you already know Visual Basic, or you have a good book on the subject, you can skip this chapter entirely. Unrestricted Getting Started with NX Open Chapter 1: Introduction Page 1 In Chapter 5, we provide a brief overview of NX Open concepts and architecture. It's not really necessary for you to know all of this, but understanding the underlying principles might help you to learn things more quickly. Brief descriptions of some NX Open functions are given in chapters 6 through 15, along with examples of their uses. We focus on basic techniques and concepts, so we only describe a small subset of the available functions. You can get more complete information from the NX Open Reference Guide. Chapter 16 discusses "exceptions", and, finally, in chapter 17, we tell you how to deal with some common problems, if they should arise. ■ Other Documentation The definitive source of information about the capabilities of NX Open is the NX Open Reference Guide, which you can find in the NX documentation set in the location shown below: The document is fully indexed and searchable, so we hope you'll be able to find the information you need. It describes all NX Open functions in detail. If you get tired of clicking through all the security warnings that appear when you access the NX documentation, you can fix this. In Internet Explorer, choose Tools Internet Options Advanced. Scroll down to the Security set of options near the bottom of the list, and check "Allow active content to run in files on My Computer". In Visual Studio, another option is to use the Object Browser, which you can access from the View menu: Unrestricted Getting Started with NX Open Chapter 1: Introduction Page 2 The Object Browser won't let you see the example programs and explanatory remarks that are in the Reference Guide, but it might be easier to access while you're in the middle of writing some code. Actually, you may find that you don't need either the NX Open Reference Guide or the Visual Studio Object Browser, because all the information you need about calling a function is given by Visual Studio "intellisense" as you type. If you have some experience with the GRIP language, then there's a document called "SNAP and NX Open for GRIP Enthusiasts" that might be helpful to you. It explains SNAP and NX Open programming in terms that are likely to be familiar to people who have used GRIP, and shows you how to map GRIP functions to SNAP and NX Open ones. You can find that document in the standard NX documentation set, in roughly the same place that you found this one. ■ Example Code Once you understand the basic ideas of NX Open, you may find that code examples are the best source of help. You can find example programs in several places: In this guide. There are about a dozen example programs in chapters 2 and 3, along with quite detailed descriptions. Also, the later chapters contain many "snippets" of code illustrating various programming techniques. There are some examples in [...NX]\UGOPEN\NXOpen\Examples. There are two folders: the one called "Getting Started Examples" contains the examples from this guide, and the "More Examples" folder contains some larger examples that try to do more useful things. Here, and in the remainder of this document, the symbol [...NX] denotes the folder where the latest release of NX is installed, which is typically C:\Program Files\Siemens\NX 12, or something similar. The GTAC web page has a large collection of example programs that you can search through to find useful code. Log in with your webkey username and password. From the main menu choose "Symptom/Solution Information Query", and then "Search Solution Center". Enter a search string that includes a phrase like "sample program", and click on the "Search" button. A list of results will appear, which you can filter by document type, software product, and publish date. Set the document type filter to "nx_api" to find sample programs, and filter further by programming language if you want to. If you've read everything, and you're still stuck, you can contact Siemens GTAC support, or you can ask questions in the NX Customization and Programming Forum at the Siemens PLM Community site . Finally, you can often get help at NXJournaling.com and in the NX forum at eng-tips.com. Unrestricted Getting Started with NX Open Chapter 1: Introduction Page 3 Chapter 2: Using the NX Journal Editor In this chapter, we will discuss creation of simple programs using the NX Journal Editor. This is not a very supportive environment in which to write code, but it's OK for very simple programs, and it requires no setup. In the next chapter, we will discuss the use of Microsoft Visual Studio, instead. This requires a small preparation effort, but provides a much nicer development environment. ■ System Requirement — The .NET Framework To use NX Open with NX 12, you need version 4.6 of the .NET Framework, or newer. It's possible that you have several versions installed (which is quite OK) — you can use the "Programs and Features" Control Panel to check: If you don't have version 4.6 or later, please download and install it from this Microsoft site. ■ Typographic Conventions In any document about programming, it's important to distinguish between text that you're supposed to read and code that you're supposed to type (which the compiler will read). In this guide, program text is either enclosed in yellowish boxes, as you see on the next page, or it's shown in this blue font. References to filenames, pathnames, functions, classes, namespaces, and other computerish things will sometimes be written in this green color, if this helps clarify an explanation. ■ Licensing You don't require any special license to record .NET journals and play them in the NX Journal Editor, as described in this chapter. Alternatively, you can compile your journal code to produce an "executable" (an EXE or DLL file), as described in the next chapter. Working in Journal Editor imposes some restrictions: all of your code must be in one file, and you can only call functions that reside in a small set of special libraries (the NX Open DLLs, the SNAP DLL, and a few basic Windows DLLs). If the restrictions cause trouble for you, then you can purchase an "author" license (dotnet_author) that makes it more convenient to work with compiled code. Specifically, the author license allows you to: Conveniently write large compiled programs whose code is distributed across several files, and which can call any function in any .NET DLL. "Sign" the compiled programs you write (so that other people can run them more easily) Run compiled programs that call NXOpen functions, even if they have not been signed When an NX Open program is running, it consumes licenses in the same way as an interactive NX session. So, if your NX Open program calls some drafting function, for example, then it will consume a drafting license. Unrestricted Getting Started with NX Open Chapter 2: Using the NX Journal Editor Page 4 ■ The Guide Functions There are many places where we use certain "helper" functions to make the example code in this document shorter and easier to understand. Since their only purpose is to improve the readability of this guide, we call these functions Guide functions. For instance, we will often need to write out text to the NX Info window. Rather than repeating the three of four lines of code required to do this, we have captured that code in the simple Guide.InfoWriteLine function. This function is used in the first example below, and in many other places. The Guide functions are described in detail in an Appendix, and in the NX Open Reference Guide. They are very simple and limited, because our primary goal was to make them easy to call. Though you may find uses for them in the code you write, their intended purpose is purely educational. ■ Example 1: Hello World We will start by creating a journal to print "Hello World" to the NX Information Window. Run NX, create a new part file, and then choose the Developer tab. If you do not see the Developer tab in the NX Ribbon bar, please activate it by selecting it from the Ribbon Bar context menu. The Developer Tab contains several groups related to NX programming, including the Journal Group. The Journal Group contains commands to record, play, and edit journals, as well as some commands to add comments or code to a journal as it is recorded. Choose Developer tab → Journal group → Edit. In the Journal Editor dialog, Click Open in the Journal Editor toolbar and open the file NXOpenSample.vb, which you can find in [...NX]\UGOPEN\NXOpenExamples\VB\Templates . Remember that [...NX] is just shorthand for the location where NX is installed, which is typically somewhere like C:\Program Files\Siemens\NX 12. You should see some text like this: This journal just gets the NX session. Any text in a Visual Basic .NET file to the right of an apostrophe is treated as comment text by the compiler. Now we will add some code to print "Hello World" in the NX Information Window. In your journal, replace the line of text that says 'Your code goes here with the following line: Guide.InfoWriteLine("Hello, World!") Unrestricted Getting Started with NX Open Chapter 2: Using the NX Journal Editor Page 5 In the Journal Editor, click Play, (the red triangular arrow icon) to play the journal. You should see the Information Window appear containing the text "Hello, World!". If you receive some sort of error, rather than the output shown above, here are some possible causes: Maybe you typed something incorrectly, in which case the compiler will probably complain that it can't understand what you wrote. An error message will tell you in which line of code the problem occurred. The description of the error might not be very helpful, but the line number should be. Maybe you don't have an up-to-date version of the .NET framework installed, as mentioned above. This may cause a mysterious error that reports an "Invalid attempt to load library". Maybe you neglected to delete the quotation mark at the beginning of the line "Your code goes here", in which case your code will run without any errors, but the NX Information window will not appear There is a troubleshooting guide in chapter 17 that will help you figure out what went wrong, and get it fixed. Fortunately, you will only have to go through the troubleshooting exercise once. If you can get this simple "hello world" program to work, then all the later examples should work smoothly, too. ■ Example 2: Collections NX can create parts and assemblies with complex geometry and product structure. Sometimes you will need to perform operations on a collection of objects in your parts or assemblies. Using a journal to cycle through a collection will often make these tasks easier. We will start by creating some simple journals to understand how to cycle though object collections. An NX part has several collections, each holding objects of a certain type. For example, each part has a CurveCollection that holds all the curve objects in that part. The property workPart.Curves accesses the CurveCollection of the work part. You can use a CurveCollection to cycle over all types of curves in an NX part. Choose File tab → Open to open the part file curves.prt, which you can find in [...NX]\UGOPEN\NXOpenExamples\ExampleParts. This part file contains several types of curves (lines, arcs, general conics, and splines) that we can cycle through to understand how collections work in NX Open. Open the file NXOpenSample.vb in the Journal Editor, just like the steps in example 1. Unrestricted Getting Started with NX Open Chapter 2: Using the NX Journal Editor Page 6 Imports NXOpen Module NXJournal Sub Main () Dim theSession = NXOpen.Session.GetSession() 'Your code goes here End Sub End Module Replace the comment text 'Your code goes here with the following lines: Dim workPart = theSession.Parts.Work Dim numCurves As Integer = 0 Dim curveLength As Double For Each cur As curve In workPart.Curves numCurves = numCurves + 1 curveLength = cur.GetLength Guide.InfoWriteLine("Curve " & numCurves & " has length " & curveLength) Next cur Guide.InfoWriteLine("Work part has " & numCurves & " curves.") You can print information about each of the curves as you cycle through the curve collection. The Curve class has a method GetLength that returns the length of the curve. This code is cycling through the curves in the part and printing the length of each curve to the information window. Once we finish cycling through the curves, we also print out the number of curves to the information window. The meanings of the more interesting lines of code are as follows: Lines of code Explanation Dim workPart = theSession.Parts.Work Declares a variable "workPart" and initializes it to the work part of the current NX session. Since we are using Option Infer On in our journals, we do not have to declare the type of this variable. The .NET Framework infers its type from the return type of the property theSession.Parts.Work. For Each curve In workPart.Curves Next curve This is a repetitive "loop" process. The statements between the "For" statement and the "Next curve" statement are executed for each curve in workPart.Curves, which is the CurveCollection of the work part. curveLength = curve.GetLength() Get the length of the curve by calling the GetLength method on the curve. Guide.InfoWriteLine ("Work part has " & numCurves & " curves.") Unrestricted Getting Started with NX Open Output a line of text to the information window. The "&" character takes two strings and combines them into one. The Integer variable "numCurves" is converted to a string before it is combined with the other strings. Chapter 2: Using the NX Journal Editor Page 7 You should see the following output in your listing window if you used the curves.prt part file: ■ Example 3: Creating Simple Geometry Some collections have additional methods to create objects. A CurveCollection has several methods for creating curves. One method, CreateLine, will create a line in the work part when you enter the start point and end point for the line. There are two versions of the CreateLine method, one that takes two point objects for the start and end points and one that takes two sets of coordinates for the start and end points. Create a new part in NX, and then open the file NXOpenSample.vb in the Journal Editor just like the steps in example 1. Replace the comment text 'Your code goes here with the following lines: Dim workPart = theSession.Parts.Work Dim p1 As New Point3d(50, -100, 0) Dim p2 As New Point3d(50, 100, 0) Dim line1 = workPart.Curves.CreateLine(p1, p2) Guide.InfoWriteLine("Line created with length " & line1.GetLength) This journal creates a line from (50, -100, 0) to (50, 100, 0) with a length of 200. The meanings of the more interesting lines of code are as follows: Lines of code Explanation Dim p1 As New Point3d(50, -100, 0) Declares a variable "p1" as an object of type Point3d. A Point3d is a structure that contains three double values named "X", "Y", and "Z" representing the x, y, z coordinates of the point. The coordinates are initialized to the values (50, -100, 0) Dim line1 = workPart.Curves.CreateLine(p1, p2) Create a line between "p1" and "p2" using the CreateLine method of the CurveCollection. The CurveCollection of a part is represented by the property Curves. You can use journals to create curves programmatically in a pattern that would be difficult to do interactively in NX. Unrestricted Getting Started with NX Open Chapter 2: Using the NX Journal Editor Page 8 For example, the following journal creates a diagram of a parabolic mirror. It shows how rays of light are reflected off the mirror towards a focus point. Dim Dim Dim Dim Dim workPart = theSession.Parts.Work vertex As New Point3d(0,0,0) focus As New Point3d(100,0,0) axisX As New Vector3d(1,0,0) axisY As New Vector3d(0,1,0) Dim focLength = focus.X Dim h = 100.0 Dim p1, p2 As Point3d Dim lens = workPart.Curves.CreateParabola(vertex, axisX, axisY, focLength, -h, h) For y = -h to h Step 10.0 Dim x = (y*y)/(4.0*focLength) p1 = New Point3d(x,y,0) p2 = New Point3d(250,y,0) workPart.Curves.CreateLine(focus, p1) workPart.Curves.CreateLine(p1, p2) Next y Running this code should produce the following output: ■ Example 4: Reading Attributes You can attach attributes to any NX object to store information about it. Open the part Bracket.prt, which you can find in [...NX]\UGOPEN\NXOpenExamples/ExampleParts, and then open the file UserAttributesOnBodies.vb in the Journal Editor just like the steps in example 1. Play the journal to see what it does: Dim theSession = Session.GetSession() Guide.InfoWriteLine("Outputting list of attributes in each body in the work part:") Dim bodies = theSession.Parts.Work.Bodies For Each bod As body in bodies Dim attributes = bod.GetUserAttributes() For Each attr As NXObject.AttributeInformation in attributes Guide.InfoWriteLine(attr.Title & " = " & attr.StringValue) Next attr Next bod Guide.InfoWriteLine("") This journal cycles through the bodies in the work part, and prints all the attributes on each body to the Information window. Attributes can be defined to be certain types, such as Integer, Number, Time, and String, but you will always be able to get a string representation of the attribute through the StringValue property. Unrestricted Getting Started with NX Open Chapter 2: Using the NX Journal Editor Page 9 The meanings of the more interesting lines of code are as follows: Lines of code Explanation Dim bodies = theSession.Parts.Work.Bodies Gets the BodyCollection of the work part. Dim attributes = bod.GetUserAttributes Get all the attributes defined on the body. The attributes are returned in an array of AttributeInformation structures. Guide.InfoWriteLine(attribute.Title & " = " & attribute.StringValue) Prints out the attribute title and string value of the attribute to the information window. Running this code on Bracket.prt should produce the following Information window output: This part only has one body, so the listed attributes are from that single body. Other NX objects may have attributes attached to them. Open the journal file UserAttributesOnGeometry.vb to list any user attributes attached to bodies, faces, and edges in the work part. The journal looks like this: Imports NXOpen Module NXJournal Sub Main () Dim theSession = Session.GetSession() Guide.InfoWriteLine("Outputting list of user attributes on geometry in the work part:") Dim bodies = theSession.Parts.Work.Bodies For Each bod As body in bodies PrintAttributes(bod) Dim edges = bod.GetEdges() For Each edg As edge in edges PrintAttributes(edg) Next edg Dim faces = bod.GetFaces() For Each f As face in faces PrintAttributes(f) Next f Next bod Guide.InfoWriteLine("") End Sub Sub PrintAttributes (obj As NXObject) Dim attributes = obj.GetUserAttributes() For Each attr As NXObject.AttributeInformation in attributes Guide.InfoWriteLine(attr.Title & " = " & attr.StringValue) Next attr End Sub End Module Unrestricted Getting Started with NX Open Chapter 2: Using the NX Journal Editor Page 10 The code that prints out the attributes of a given body might be re-usable elsewhere. To make the re-use easier, we have placed this code in a new "subroutine", (denoted by the keyword "Sub"). We call this subroutine in our main subroutine when we want to print out the attributes for any NXObject; in our case bodies, faces, or edges. Running this code on Bracket.prt should produce the following listing window output: ■ Example 5: WinForms (The Hard Way) The .NET framework provides a wide variety of tools for designing user interface dialogs. These dialogs are called Windows Forms (WinForms, for short). The NX Block UI Styler has similar tools, and produces dialogs that are more consistent with the rest of NX, as explained in chapter 14. But WinForms are more flexible, and you may find them useful in some cases. Designing WinForm-based user interfaces is actually much easier if you use an IDE like Visual Studio, and we will see how to do this in the next chapter. For now, we will create a very simple WinForm, to show the basic concepts. Copy and Paste the following code into the file NXOpenSample.vb: Sub Main() Dim myForm As New System.Windows.Forms.Form() NXOpenUI.FormUtilities.SetApplicationIcon(myForm) NXOpenUI.FormUtilities.ReparentForm(myForm) myForm.BackColor = System.Drawing.Color.Red myForm.Opacity = 0.5 myForm.Text = "Hi there" myForm.ShowDialog() End Sub Unrestricted Getting Started with NX Open 'Create a Windows form 'Use an NX icon for the application icon 'Set NX as the parent of our form 'Color our form red 'Make our form translucent 'Change the title of our form 'Display our form Chapter 2: Using the NX Journal Editor Page 11 When you run this application, you should see a WinForm appear, like this: The WinForm is pretty boring, but it does have all the standard Windows functionality — you can move it around, resize it, minimize it, and so on, in the usual way. The code calls some methods in a special FormUtilities class in the NXOpenUI namespace to make our WinForm a bit more NX-specific. The method SetApplicationIcon creates the form with an NX icon in its top left corner, which will help the user understand that it's associated with NX. Also, the method ReparentForm sets the main NX Window as the "parent" of our new form, which means that our form will be minimized and restored along with the NX window, and will never get hidden underneath it. Actually, in the current scenario, our form is "modal", which means that you have to close it before you do anything with the NX window, so the parenting arrangement doesn't have much value. We got this modal behavior because we called myForm.ShowDialog to display our form. There is also myForm.Show, which creates a non-modal form, but this doesn't work in the Journal Editor. The next few lines of code adjust various properties of the form — we give it a red color, make it 50% transparent, and put the words "Hi there" in its title bar. There are dozens of properties that influence the appearance and behavior of a WinForm, but it's best to wait until the next chapter to explore these, because it's very easy using Visual Studio. To stop your code running and get back to the Journal Editor, you need to close the WinForm. You do this in the usual way — click on the "X" in the top right corner. Unrestricted Getting Started with NX Open Chapter 2: Using the NX Journal Editor Page 12 Next, let's add a button to our WinForm. Modify the code in NXOpenSample.vb as follows: Imports NXOpen, NXOpenUI Imports System, System.Drawing.Color, System.Windows.Forms Module NXOpenSample Dim WithEvents myButton As Button Dim theSession As Session Dim rand As Random 'A variable to hold a button 'A variable to hold the NX Session 'A variable to hold a random number generator Sub Main() theSession = Session.GetSession rand = New Random Dim myForm As New Form myForm.Text = "Create Random Spheres" FormUtilities.SetApplicationIcon(myForm) FormUtilities.ReparentForm(myForm) myButton = New Button myButton.BackColor = Yellow myButton.Text = "Click me" myForm.Controls.Add(myButton) myForm.ShowDialog End Sub 'Get the NX Session 'Create a random number generator 'Create a Windows form 'Use an NX icon for the application icon 'Set NX as the parent of our form 'Create a button 'Color it yellow 'Put some text on it 'Add it to our form 'Display our form End Module First, note that we have added another line of "Imports" statements at the top of the file. These allow us to abbreviate the names in our code. So, for example, we can refer to Yellow instead of the full name System.Drawing.Color.Yellow, and we can refer to Random instead of System.Random. As you can see, we used the "New" keyword when creating the random number generator, the form, and the button. We didn't use the "New" when we created NX line objects in earlier examples, and you may be wondering why these two types of objects get treated differently. The answer is given in chapter 5, in the section entitled "Factory Objects". Don't worry about it for now — just accept that the "New" keyword isn't needed when you're creating NX objects. Or, if the curiosity is overwhelming, you can read about this topic in chapter 5. Try running this code. You will see that the form is displayed, but nothing happens if you click on the yellow button. To change this, place the following code down near the bottom, just before the line that says "End Module". Sub Handler(sender As Object, e As EventArgs) Handles myButton.Click Dim x = 100 * rand.NextDouble 'Get a random x-coordinate between 0 and 100 Dim y = 100 * rand.NextDouble 'Get a random y-coordinate between 0 and 100 Dim z = 100 * rand.NextDouble 'Get a random z-coordinate between 0 and 100 Guide.CreateSphere(x, y, z, 10) 'Create a sphere at (x,y,z) with diameter 10 End Sub This is a new "subroutine" (denoted by the keyword "Sub"). So, as in the previous example, we now we have two subroutines — one called "Main" and one called "Handler". This is a fairly typical situation — as your code gets longer, it's easier to understand if you break it up into several subroutines. The "Handler" function is an event handler for the "click" event of the yellow button. In other words, this code gets executed whenever you click on the yellow button in the form. As you can see, every time you click the button, the code will create a randomly-located sphere. The completed sample is in the file CreateRandomSpheres.vb Unrestricted Getting Started with NX Open Chapter 2: Using the NX Journal Editor Page 13 Designing buttons and writing event handlers is much easier in Visual Studio, as we will see in the next chapter. ■ What Next? The examples in this chapter have given you a brief glimpse at some of the things you can do with NX Open. Using the Journal Editor, we were able to start programming immediately, and we saw that NX Open allows us to build simple user interfaces, do calculations, and create NX geometry. If you liked what you saw in this chapter, you'll probably like the next one, too. It shows you some further examples of NX Open capabilities, and also some much easier and more pleasant ways to write code. Unrestricted Getting Started with NX Open Chapter 2: Using the NX Journal Editor Page 14 Chapter 3: Using Visual Studio Express In the previous chapter, we developed code using the NX Journal Editor. This is a convenient starting point, since it requires no setup, but it is really a fairly primitive environment. Except for very short programs, it is far better to use a more powerful "integrated development environment" (IDE). Microsoft Visual Studio Express 2015 for Windows Desktop is a free-ware lightweight version of the Microsoft Visual Studio IDE used by many professional programmers. The idea, according to Microsoft, is to provide a streamlined, easy-to-use IDE for less serious users, such as hobbyists, students, and people like you. You can use this "Express" version of Visual Studio with the Visual Basic, C#, and C++ programming languages. In this chapter, we will be focusing on using Visual Basic. NOTE: It should be noted that the NXOpen Wizards provided by NX will not work with Visual Studio Express. ■ Installing Visual Studio If you already have some version of Visual Studio 2015 installed on your computer, and you are familiar with it, you can skip this section and proceed directly to the first example. If not, then the first step is to install Visual Studio 2015 Express for Windows Desktop, which you can download from here, or several other places. If you can't find the web page (because the Microsoft folks have moved it again), just search the internet for "Visual Studio 2015 Express". Make sure you get the "for Windows Desktop" version. A common mistake is to download the "for Windows" version, instead, but this is for building Windows store apps, so it's not what we want. After selecting download, choose the executable downloaded, click next, and follow the installation instructions in order to complete the installation. After you're done, you should see Microsoft Visual Studio 2015 Express on your Programs menu, and you should see a folder called Visual Studio 2015 in your My Documents folder. If you run into trouble, it might help to watch this video. Older versions of Visual Studio will not work because they don't allow you to use version 4.6 of the .NET Framework. Unfortunately, the Visual Studio Express download is much larger than it was when it was first released — it has grown from around 80 MB to over 1.1 GB. If you don't have the patience or disk space to handle a package this large, you can try the SharpDevelop IDE, instead. It's only around 15MB, and provides everything you need. The instructions you read in this document won't match SharpDevelop exactly, but it should be fairly easy to adapt. In the examples in this chapter, we'll provide step-by-step instructions for writing the code, just as we did in chapter 2, so it should be easy to follow. But if you'd like to get some additional information about the Visual Basic language or Visual Studio, then one good place to start is this series of videos. There is a huge amount of other tutorial material available on the internet, and you might find other sources preferable, especially if your native language is not English. Unrestricted Getting Started with NX Open Chapter 3: Using Visual Studio Express Page 15 ■ Installing NX Open Templates After installing Visual Studio, you should install two custom templates that we will be using as convenient starting points when developing NX Open programs. You will find two zip files in [...NX]\UGOPEN\NXOpenExamples\VB\Templates. Again, remember that [...NX] is just shorthand for the location where NX is installed, which is typically somewhere like C:\Program Files\Siemens\NX 12. The names of the files are NXOpenTemplateVB.zip and NXOpenWinFormTemplateVB.zip. Copy these two zip files into the folder [My Documents]\Visual Studio 2015\Templates\ProjectTemplates\Visual Basic. For added clarity, here are the same instructions in pictorial form: Unfortunately, experience has shown that people often do this step wrong, so we're going to yell at you... NOTE: please do not extract the contents from the zip files; just copy the zip files themselves. NOTE: please note that the NX Open Templates are different than the NX Wizards. The NX Templates will work with both Visual Studio and Visual Studio Express. While the NXOpen Wizards will only work with the Visual Studio. For more information about the NX Open Wizards please see the Programming Tools section of the Technical Documents. ■ Licensing Issues Again You will need an NX Open .Net Author license to run NX Open dlls built from Visual Studio. If you do not have an NX Open .Net Author license, you can edit NX Open Visual Basic journals in Visual Studio and replay the journal in NX as long as the journal is arranged to be in one file. ■ Example 1: Hello World Again Our first exercise is to create a "Hello World" application again. Sorry, we know it's boring, but it's a tradition. After you get Visual Studio Express installed and running, choose New Project from the File menu. A "project" is the name Visual Studio uses for a collection of related files. You will see a list of available project templates Unrestricted Getting Started with NX Open Chapter 3: Using Visual Studio Express Page 16 Choose the "NX Open Application" template. This is a special custom template designed to serve as a convenient starting point for certain kinds of NX Open applications. Also, give your project a suitable name — something like "HelloApp" would be good. The NX Open Application template gives you a framework for a simple NX Open application, as shown here: In the left-hand pane, you can see some familiar VB code, which the template has placed in a file called MyProgram.vb for you. We need to make a couple of changes to this code: add Option Explicit Off at the top, and add a line that outputs some message to the Information window, as shown here: You should type the new code, rather than just copying and pasting it, because some interesting things happen as you type (as you saw in the tutorial videos, if you watched them). In fact, it's interesting to type the entire 7 lines of code. You will find that you actually only have to type 5 lines — Visual Studio will type the other two for you. Generally, Visual Studio helps you by suggesting alternatives, completing words, correcting mistakes, showing you documentation, and so on. To accept the highlighted alternative, you can either press Tab, or type another character, like a period or a parenthesis. All of this is called "Intellisense" by Microsoft's marketeers. Despite its dubious name, you'll find it very helpful as your programming activities progress. Also, notice that Visual Studio automatically makes comments green, literal text red, and language keywords blue, to help you distinguish them. Next, you are ready to compile (or "build") your code into an executable application. To do this, go to the Build menu and choose Build HelloApp, or press Ctrl+Shift+B, which will send your code to the VB compiler. The compiler will translate your code into an executable form that your computer can run, and will store this in a file called HelloApp.dll. The extension "dll" stands for "Dynamic Link Library", which is a type of file that holds executable code. You should get the good news about the build succeeding down at the bottom left: Unrestricted Getting Started with NX Open Chapter 3: Using Visual Studio Express Page 17 On the other hand, if you're unlucky, you might get some error messages like these: It's not very likely that this problem will occur, so we don't want to interrupt the flow by discussing all the details here. The possible causes and corrective actions are described in chapter 17. At some point, you should save your project by choosing Save All from the File menu. Visual Studio will offer to save in your Projects folder, whose path is typically something like [My Documents]\Visual Studio 2015\Projects. Now, we are ready to run our new application. From within NX, choose File Execute NX Open (or press Ctrl+U). Your version of the NX user interface might not have the Execute option installed in the File menu, but the Ctrl+U shortcut will work anyway. A dialog will appear that allows you to find your executable. As mentioned earlier, it will be called HelloApp.dll, and it will be located in [My Documents]\Visual Studio 2015\Projects\HelloApp\HelloApp\bin\Debug along with two other files that you don't need to worry about. To see HelloApp.dll, make sure you set the "Files of type" filter in the NX dialog to "Dynamic Loadable Libraries (*.dll)". Double-click on HelloApp.dll, and a friendly greeting should appear in your NX Listing window. If you can't find your application, try looking in the bin\Release folder, rather than the bin\Debug folder. If you still can't find it, it's probably because you forgot to save it, or you didn't set the file type filter correctly. There's a useful trick that allows you to locate your executable quickly. When you build the application, some text like this will appear in the "Output" pane at the bottom of your Visual Studio window: If the output pane is not visible, press Ctrl+Alt+O to display it (that's the letter O, not the number zero). You can then just copy the pathname of the newly-created application (highlighted in yellow above) and paste it into the "Execute" dialog within NX. This technique is highly recommended — it avoids all the hunting around folders that we described above, and it ensures that you are running the code that you just built. You only have to do this once per NX session, because NX will remember the location for you. Unrestricted Getting Started with NX Open Chapter 3: Using Visual Studio Express Page 18 ■ Example 2: Declaring Variables In this example, we will do some vector calculations to compute the radius of a circle through three points. We will focus on the topic of "declaring" the variables we use, to see how this affects things. If your previous project is still open in Visual Studio, close it by choosing File Close Project. Then choose New Project from the File menu, use the NX Open Application template to create a project, and give it the name ThreePointRadius, or something like that. As before, add the line Option Explicit Off at the top of the file. For reasons explained below, this is the last time we're going to do this in our examples. Then, replace the line "Your code goes here" with the following: Dim sel = NXOpen.UI.GetUI.SelectionManager Dim myView As View Dim p1, p2, p3 As Point3d sel.SelectScreenPosition("Specify first point", myView, p1) ' Get first point from user sel.SelectScreenPosition("Specify second point", myView, p2) ' Get second point sel.SelectScreenPosition("Specify third point", myView, p3) ' Get third point u As New Vector3d(p2.X - p1.X, p2.Y - p1.Y, v As New Vector3d(p3.X - p1.X, p3.Y - p1.Y, uu = u.X * u.X + u.Y * u.Y + u.Z * u.Z uv = u.X * v.X + u.Y * v.Y + u.Z * v.Z vv = v.X * v.X + v.Y * v.Y + v.Z * v.Z det = uu * vv - uv * uv alpha = (uu * vv - uv * vv) / (2 * det) beta = (uu * vv - uu * uv) / (2 * det) rx = alpha * u.X + beta * v.X ry = alpha * u.Y + beta * v.Y rz = alpha * u.Z + beta * v.Z radius = Math.Sqrt(rx*rx + ry*ry + rz*rz) p2.Z - p1.Z) ' Vector3d from p1 to p2 p3.Z - p1.Z) ' Vector3d from p1 to p3 ' Dot product of vectors ' Determinant for solving linear equations ' Bad code !! Should check that det is not zero Guide.InfoWriteLine(radius) ' Output to Info window ' Radius vector components ' Radius is length (norm) of this vector Again, you can gain some experience with Intellisense if you type this code, rather than copying and pasting it. The only thing that's new here is the function SelectScreenPosition, which allows you to get a screen point location from the user. As before, you can save this project, build it, and run it from within NX using File Execute NX Open (or Ctrl+U). Now let's see what happens if you make a typing error. Change the line that calculates "det" to read det = uu * vv - uv * u In other words, change the last term from "uv" to "u". Then build the project and try running it again. It will still build successfully, but when you run it from within NX, you'll get an error message like this: If you choose Help Log File from within NX, and hunt around the NX System Log, you will find some more error messages about 50 lines from the bottom, most notably these ones Caught exception while running: Main System.InvalidCastException: Operator '*' is not defined for type 'Double' and type 'Vector3d'. at Microsoft.VisualBasic.CompilerServices.Operators.InvokeObjectUserDefinedOperator ... blah blah blah at Microsoft.VisualBasic.CompilerServices.Operators.MultiplyObject(Object Left, Object Right) at ThreePointRadius.ThreePointRadius.Main() in C:\Users\ ... \ThreePointRadius.vb:line 21 Unrestricted Getting Started with NX Open Chapter 3: Using Visual Studio Express Page 19 Obviously it would be much better to discover errors like this earlier, as you're writing the code, rather than when you run the application. And, in fact, you can, if you change the way you write the code, and give the compiler a little more information. The key is a process called "declaring" variables, which lets us tell the compiler about their types. To see how this works, change your code to read: Dim Dim Dim Dim Dim u As New Vector3d(p2.X v As New Vector3d(p3.X uu As Double = u.X * u.X uv As Double = u.X * v.X vv As Double = v.X * v.X p1.X, p1.X, + u.Y + u.Y + v.Y p2.Y - p1.Y, p2.Z - p1.Z) p3.Y - p1.Y, p3.Z - p1.Z) * u.Y + u.Z * u.Z * v.Y + u.Z * v.Z * v.Y + v.Z * v.Z The phrase "Dim u As New Vector3d" tells the compiler that the variable u is supposed to hold a Vector3d object, and so on. So, the compiler now knows that u and v are vectors, and uu, uv, and vv are numbers (doubles). So the expression uv*u is trying to multiply a vector by a number, which is not a legal operation in this context. So we get a "squiggly underline" error indicator, and we know immediately that we have made a mistake. And, if you hover your mouse over the mistake, a message will appear telling you what you did wrong: Up until now, our applications have been very simple, so there was not much justification for the extra effort of declaring variables. But, as you start to write more complex applications, you will definitely want the compiler to help you find your mistakes. And it can do this very effectively if you declare your variables. Actually, many programming languages require you to declare all variables. Visual Basic is an exception — if you use the "Option Explicit Off" directive at the start of your code, as we have been doing, then you don't have to. But declaring variables is a good thing, so we're going to do it

from now on. For further discussion of declaring variables (and avoiding or shortening declarations), please see chapter 4. ■ Example 3: WinForms Again One of the nice things about Visual Studio is the set of tools it provides for designing user interface dialogs using Windows Forms (WinForms, for short). We're going to recreate the "Create Random Spheres" dialog from the previous chapter, but it will be much easier this time, using Visual Studio, and the dialog will look nicer. Run Visual Studio Express, and choose New Project from the File menu. Instead of choosing the NX Open Application template this time, choose the NX Open WinForm Application template this time. Call your new project NXOpenWinFormApp. Your new project will look something like this: Unrestricted Getting Started with NX Open Chapter 3: Using Visual Studio Express Page 20 You may need to double-click on NXOpenWinForm.vb to see the new Windows form in the left-hand pane. In the lower right-hand pane, all the "properties" of the new WinForm are listed, along with their values. As you can see, the form has a property called "Text", and this property currently has the value "NX Open WinForm". This property actually represents the text in the title bar of the dialog. Edit this text to read "Create Sphere". When you do this, you will see that the dialog title bar changes, too. Next, as before, we're going to add a button to our form. On the left-hand side of the Visual Studio window, you should see a Toolbox containing various types of user interface objects. If you don't see the Toolbox, choose it from the View menu, or press Ctrl+Alt+X. Click on the "Button" object. The cursor will change to a small "+" sign, and you can then use it to graphically draw a button on the form. Initially, the button will be labelled with the text "Button1", but you can change this to "Click me" or whatever you want by editing the text property of the button, just as we edited the text property of the form. You can also change the font size and the background color. Your result might be something like this: Also, you can adjust the sizes of the button and the form by dragging on their handles. Next, let's make the button do something useful. Double-click the button, and a code window will appear. Add code to the button's "click" event. Currently, it should look like this: Private Sub Button1_Click(sender As System.Object, e As System.EventArgs) Handles Button1.Click End Sub Unrestricted Getting Started with NX Open Chapter 3: Using Visual Studio Express Page 21 To cause your code to run when the button is clicked, add code in between the lines, so that it looks like this: Private Sub Button1_Click(sender As System.Object, e As System.EventArgs) Handles Button1.Click Guide.InfoWriteLine("Creating a sphere") Guide.CreateSphere(0, 0, 0, 10) End Sub When we created this dialog manually, in the previous chapter, you may recall that we wrote code like this: myForm.Text = "Create Random Spheres" myButton = New Button myButton.BackColor = Color.Yellow myButton.Text = "Click me" myForm.Controls.Add(myButton) ' Create a button 'Color it yellow 'Put some text on it 'Add it to our form This same sort of code exists in our current project, too, but it was written for us by Visual Studio, and it's somewhat hidden, because you're not supposed to edit it. To see this code, click on the Show All Files button at the top of the Solution Explorer window, and then double-click on the file named NXOpenWinForm.Designer.vb. To display our dialog, we have a couple of lines of code in Sub Main in the file MyProgram.vb: Public Shared Sub Main() Dim form As NXOpenWinForm = New NXOpenWinForm() form.ShowDialog() End Sub As before, we're using form.ShowDialog to display the dialog, so it will be "modal", which means that we can't do anything else until we close the form. There is also form.Show, which creates a non-modal form, but to use this, you have to change the GetUnloadOption function in the file Unload.vb. Specifically, you have to modify this function to return NXOpen.UFSession.AtTermination instead of NXOpen.UnloadOption.Immediately. If you fail to do this, your dialog will flash on the screen briefly, and then disappear again. Run your project, and run it from within NX, as usual. When your dialog appears, you can click on your button to create spheres. When you get bored with this, click the "X" to close your dialog. If you want to learn more about creation of WinForm-based user interfaces, there are many books and on-line tutorials available on the subject, including this tutorial. ■ Example 4: Hello World Yet Again (the Hard Way) Sorry, but we're going to create a "Hello World" application yet again. This time, we're going to do it without getting any assistance from the NX Open template we used last time. This will help you understand what is happening Unrestricted Getting Started with NX Open Chapter 3: Using Visual Studio Express Page 22 "behind the scenes" so that you will know what to do if you run into problems later. If you're not interested in this, you can skip to the next example. Run Visual Studio Express, and choose New Project from the File menu. You will see the available set of project templates. But, this time, instead of choosing the NX Open Application template, choose the Class Library one: You might be thinking that you could use the "Console Application" template, instead. Unfortunately, there are some technical reasons why this will not work — on some systems, it will lead to a mysterious "failed to load image" error when you try to run your application from within NX. Please see chapter 17 for more details. This Class Library template gives you a framework for a Visual Basic class definition. We will see a file called Class1.vb that contains a couple of lines of code. Delete this code and paste (or type) the contents of NXOpenSample.vb in its place. Delete the first line (the one that says Option Explicit Off). Also, delete the second line if it contains "Imports System". We'll replace it with NX Open code. Now the code goes here, and replace it by Guide.InfoWriteLine("Hello world!"), as we have done several times before. You should end up with something that looks like this: As you're typing, you might notice that the usual "Intellisense" doesn't work. This is the first indication that something is wrong. Also, you will see several squiggly underlines, and some error and warning messages in the list at the bottom of the window: Most of the problems arise because our code is using the NX Open libraries, and these are not connected in any way to our project. In other words, our project doesn't know anything about NX Open, or the NXOpen.Guide.InfoWriteLine Unrestricted Getting Started with NX Open Chapter 3 Using Visual Studio Express Page 23 function. To fix this, we need to add a "reference" to the NX Open libraries. From the Project menu, choose Add Reference. In the dialog that appears, click on the Browse tab, and navigate to the folder [...NX]\NXBIN\managed: You will see a number of DLLs. We only need the NX Open DLLs in this example. Select five DLLs, as shown above, and click OK. Your project now has references to the NX Open libraries, and these should remove the complaints about them "containing no public members". Now you can build and run the application, as usual. The NX Open Application template that we used previously already includes the references for you, so it didn't have to add them manually this time. But, it's useful to know how to do this when you need to. In particular, if you want to use some .NET Framework functions in any journals you write, you may have to add references to the assemblies where they reside. If you forget to do this, you will get "type not defined" errors, like the ones we saw above. Please see chapter 17 for more information about problems with references. A project based on the Class Library template has another deficiency — it doesn't include a GetUnloadOption function. This means that NX won't know how to "unload" your code after it has finished executing — in some sense, NX "holds onto" your code, and won't let it go. So, if you try to change your code and rebuild the project, you'll get an error message telling you that you "can't access the file because it is being used by another process". The other process is NX, and you'll have to terminate NX to get it to release its hold on your DLL so that you can rebuild it. The NX Open Application template provides a GetUnloadOption function for you, so you won't have these sorts of problems. Writing your own GetUnloadOption function is fairly simple. The code is as follows: Public Shared Function GetUnloadOption(ByVal dummy As String) As Integer Return CType(NXOpen.Session.LibraryUnloadOption.Immediately, Integer) End Function It's convenient to place this code in the same class or module as your "Main" function — in our case, this means inside the NXOpenSample class that we created earlier. So, you just need to paste this code immediately before the line that says "End Class". Please look up GetUnloadOption in the NX Open Programmer's Guide for more information about unloading code. ■ Example 5: Editing a Recorded Journal Our next example covers how to edit a recorded journal to create a reusable application. We will use a simple workflow to show the general procedure for editing the recorded journal to remove the specific recorded selections and replace them with selection operations that will work on generic parts. We have an example file you Unrestricted Getting Started with NX Open Chapter 3: Using Visual Studio Express Page 24 can use in ChangeLayerOfBody.vb where we recorded a journal that changes the layer of a selected body to layer 45. This journal is listed below: ' NX 12.0.0.8 Imports System Imports NXOpen Module NXJournal Sub Main(ByVal args() As String) Dim theSession As NXOpen.Session = NXOpen.Session.GetSession() Dim workPart As NXOpen.Part = theSession.Parts.Work Dim displayPart As NXOpen.Part = theSession.Parts.Display ' -- ' Menu: Format->Move to Layer... ' -- Dim markId1 As NXOpen.Session.UndoMarkId markId1 = theSession.SetUndoMark(NXOpen.Session.MarkVisibility.Visible, "Move Layer") Dim objectArray1(0) As NXOpen.DisplayableObject Dim body1 As NXOpen.Body = CType(workPart.Bodies.FindObject("CYLINDER(2)"), NXOpen.Body) objectArray1(0) = body1 workPart.Layers.MoveDisplayableObjects(45, objectArray1) ' -- ' Menu: Tools->Journal->Stop Recording ' -- End Sub End Module After the lines creating an undo mark for the operation is a line that specifies that you want to move to layer 45. This is the line that we need to edit. The line Dim body1 As NXOpen.Body = CType(workPart.Bodies.FindObject("CYLINDER(2)"), NXOpen.Body) looks for the body in the current part named "CYLINDER(2)". We will replace this line by a selection. In the code below we have edited our journal to ask the user to select a body to edit: ' NX 12.0.0.8 Imports System Imports NXOpen Module NXJournal Sub Main(ByVal args() As String) Dim theSession As NXOpen.Session = NXOpen.Session.GetSession() ' Get UI object Dim theUI As NXOpen.UI = NXOpen.UI.GetUI() Dim workPart As NXOpen.Part = theSession.Parts.Work Dim displayPart As NXOpen.Part = theSession.Parts.Display ' -- ' Menu: Format->Move to Layer... ' -- Dim markId1 As NXOpen.Session.UndoMarkId markId1 = theSession.SetUndoMark(NXOpen.Session.MarkVisibility.Visible, "Move Layer") Dim objectArray1(0) As NXOpen.DisplayableObject 'Dim body1 As NXOpen.Body = CType(workPart.Bodies.FindObject("CYLINDER(2)"), NXOpen.Body) ' Ask user to select one body Dim selObj As TaggedObject Dim cursor As Point3d Dim response As Selection.Response = theUI.SelectionManager.SelectTaggedObject("Select", "Select Object", Unrestricted Getting Started with NX Open Chapter 3: Using Visual Studio Express Page 25 SelectionScope.UseDefault, False, False, selObj, cursor) If resp Response.Back And resp Response.Cancel Then objectArray1(0) = CType(selObj, DisplayableObject) workPart.Layers.MoveDisplayableObjects(45, objectArray1) End If displayModification1.Dispose() ' -- ' Menu: Tools->Journal->Stop Recording ' -- End Sub End Module Although we didn't change a lot of lines in the recorded journal, we have ended up with a journal that asks the user to pick a body to move to layer 45. We can make this a little neater by moving the selection code into a separate function and calling it from our code. Using the example from the selection chapter, we have placed it in a Function called SelectBody. This function will return the selected body if the user selects a body, or Nothing if the user presses Back or Cancel. ' NX 12.0.0.8 Imports System Imports NXOpen Module NXJournal Sub Main(ByVal args() As String) Dim theSession As NXOpen.Session = NXOpen.Session.GetSession() Dim workPart As NXOpen.Part = theSession.Parts.Work Dim displayPart As NXOpen.Part = theSession.Parts.Display ' -- ' Menu: Edit->Object Display... ' -- Dim markId1 As NXOpen.Session.UndoMarkId markId1 = theSession.SetUndoMark(NXOpen.Session.MarkVisibility.Visible, "Move Layer") Dim objectArray1(0) As NXOpen.DisplayableObject 'Dim body1 As NXOpen.Body = CType(workPart.Bodies.FindObject("CYLINDER(2)"), NXOpen.Body) ' Ask user to select one body Dim body1 As Body = SelectBody() If body1 IsNot Nothing Then objectArray1(0) = body1 workPart.Layers.MoveDisplayableObjects(45, objectArray1) End If displayModification1.Dispose() ' -- ' Menu: Tools->Journal->Stop Recording ' -- End Sub ' Function to ask user to select a body Unrestricted Getting Started with NX Open Chapter 3: Using Visual Studio Express Page 26 Function SelectBody() As Body Dim theUI As NXOpen.UI = NXOpen.UI.GetUI() ' Get the UI.SelectionManager Dim objects to NXOpen.DisplayableObject Dim selObj As TaggedObject Dim cursor As Point3d Dim message As String = "Select Body" Dim title As String = "Selection" Dim resp As Response = sel.SelectTaggedObject(message, title, SelectionScope.UseDefault, False, False, selObj, cursor) If resp Response.Back And resp Response.Cancel Then If TypeOf selObj Is Body Then body1 = CType(selObj, Body) End If End If Return Nothing End Function End Module The selection function checks if the selected object is a Body. If it is not, then the function returns Nothing for the selected object. You can use selection filters to allow you to filter the selected objects to specific types. If you are interested in filtering, you can read ahead in chapter 15, which covers the NX Open selection API in more detail. ■ Debugging in Visual Studio The full version of Visual Studio (but not the Express edition) provides an excellent debugger that lets you step through your code one line at a time, watching what's happening to variables as it executes. Setting up your code to debug is a little bit complicated for two reasons. This is a very good way to find problems, obviously. The techniques used with SNAP and NX Open programs are a little unusual because you can't run your code directly, you have to run it from within NX. The easiest technique is to call the function System.Diagnostics.Debugger.Launch when your program starts. You have to be careful about where you put this code, since you need to do it before your code has started doing anything that you want to debug. If you put the debugger call in a place that isn't executed, the debugger will never start. If your code has a "Main" function, then the best place to put this call is as the first line of "Main", and this is what we normally do. Using Debugger.Launch First, you should place a breakpoint somewhere near the beginning of your code, and then you run your application in the normal way using File Execute NX Open. When execution reaches the Debugger.Launch call, the Just-In-Time Debugger dialog will appear, asking you which debugger you want to use: Double-click on the debugger for your current project, as shown in the picture above, and you will be taken back to Visual Studio with your code "paused" at the Debugger.Launch line, ready to begin stepping through it. Using Attach To Process Within Visual Studio, choose Tools Attach to Process (or press Ctrl+Alt+P), and double-click on the NX process (ugraf.exe) in the list of available processes. Again, run your application using File Execute NX Open, and you will arrive back in Visual Studio with your code "paused" at the first breakpoint. Unrestricted Getting Started with NX Open Chapter 3: Using Visual Studio Express Page 27 Regardless of which of the two approaches you use, you are now ready to step through your code. The available options are shown in the Debug menu or on the Debug Toolbar within Visual Studio. For example, you can step into the debugger facilities, please consult one of the many tutorials available on the internet. Unrestricted Getting Started with NX Open Chapter 3: Using Visual Studio Express Page 28 Chapter 4: The Visual Basic Language One of the strengths of NX Open is that it is based on standard mainstream programming languages. This means there are many excellent tools you can use (like Visual Studio), and there's lots of tutorial and help material available. This chapter provides an introduction to the Visual Basic language (which we have been using for all of our examples). There are many places where you can learn more about Visual Basic (like this series of videos, for example), so our description here will be very brief. When looking for books and on-line tutorials, you should be aware that the Visual Basic language has evolved significantly over the years. What we are using here is Visual Basic for .NET. Older versions (like Visual Basic 6, for example), are quite different. So, when you start reading, make sure you are using fairly modern materials. If you really want the complete story, you can read the Microsoft documentation on this web page. If you prefer to use the C# language, instead of Visual Basic, then these videos should be helpful. ■ The Development Process The basic process of creating a program in Visual Basic (or any other language) is shown below source code object code run compiler The process is quite simple, but unfortunately it typically involves quite a lot of programmer jargon. The Visual Basic statements you write are known as "source code". This code is typically contained in one or more text files with the extension ".VB". Your source code is then sent to a compiler, which converts it into "object code" that your computer can actually understand and run. The object code is sometimes referred to as an "executable" or a "library", or an "assembly", and is held in a file with the extension ".EXE" or ".DLL". ■ Structure of a Visual Basic Program Visual Basic has standard building blocks, typically presented in the following sequence: Option statements Imports statements The Main procedure Class and Module definitions Option Statements Option statements establish ground rules for subsequent code. Option Explicit On ensures that all variables are declared, which may make debugging easier. Option Strict On applies more strict rules to variable type conversions, which helps prevent problems that can occur when you transfer information between variables of different types. Option Infer On asks the compiler to try to guess the types of your variables, which reduces the need for declarations, as explained a little later, in the section entitled Omitting Declarations. If you place Option statements in your source code, they must be placed at the beginning of a source file, and they apply only to the source file in which they appear. Another approach is to specify compilation options in the properties of a Visual Studio project, in which case they apply to the entire project. This is often more convenient. Imports Statements and Namespaces Placing an Imports statement at the beginning of a source file allows you to use abbreviated names within that file (rather than longer "fully qualified" ones), which reduces your typing effort. For example, suppose you want to print something using the System.Console.WriteLine function to output text. If you write Imports System at the beginning of your source file, then you will be able to refer to this function as simply WriteLine whenever you need it. Unrestricted Getting Started with NX Open Chapter 4: The Visual Basic Language Page 29 In Visual Basic, the thing that appears in an Imports statement can be either a class or a namespace. Classes are explained later in this chapter. Namespaces help you to organize large quantities of code into related subgroups, and to distinguish different uses of the same name. Suppose you had a large application that performed operations on both fish and musical instruments. This probably isn't very likely, but it provides a convenient illustration. You might invent two namespaces called Instruments and Fish to hold your code. You could use the same Bass within both of these namespaces, because Instruments.Bass and Fish.Bass would be different names. If you wrote both Imports Instruments and Imports Fish at the top of a code file, you could use the name Bass to refer to either one. Of course, this could cause a problem, of course, because the name Bass would be ambiguous. The Main Procedure The Main procedure is the "starting point" for your application — the first procedure that is accessed when you run your code. Main is where you would put the code that needs to be accessed first. Classes, Modules, and Files Each line of executable code must belong to some class or module. Classes are explained near the end of this chapter. For now, you can consider a class to be a related collection of code and data fields, often representing some generic type of object. A module is really a special simplified type of class. Modules are not as flexible as classes, and they are not used as much in real-world applications, but we use them in this document because they provide a convenient way to temporarily manage smallish snippets of code. As you may recall, the NX Journaling function always produces code that is packaged into a Module. Many people advocate placing each class in its own source file, and giving this source file the same name as the class, but you can place several classes in a single file, if you want to. Conversely, you do not have to put an entire class within a single file — by using the "partial class" capability, you can split a class definition into several files, which is often useful. ■ An Example Program The listing below shows a simple program containing most of the elements mentioned above. Option Infer On Imports System, NXOpen Module MyProgram Sub Main() Dim radius As Double = 3.75 Dim area As Double = CircleArea(radius) Dim message As String = "Area is: " Guide.InfoWriteLine(message & area) End Sub ' Main function Function to calculate the area of a circle Function CircleArea(r As Double) As Double Dim pi As Double = System.Math.PI Dim area As Double = pi * r * r Return area End Function End Module The program starts with an Option statement and an Imports statement. Then there is a single module called "MyProgram" that holds all the executable code. Inside this module there is a "Main" procedure, as always, and then another function called CircleArea. Unrestricted Getting Started with NX Open Chapter 4: The Visual Basic Language Page 30 The following table shows a detailed description of this code: Line(s) Explanation Option Infer On Tells the compiler that it should try to guess (infer) the types of variables. Imports System, NXOpen Allows you to refer to functions in the NXOpen namespace using their short names, rather than fully qualified ones. Sub Main Main procedure Dim radius As Double = 3.75 Declares a variable of type Double, gives it the name radius, and stores the value 3.75 in it. Dim area As Double Declares another variable of type Double, and names it area area = CircleArea(radius) Calls a function named CircleArea, which is defined below. The variable radius is used as the input to this function, and the output returned from the function is written into the variable named area. Dim message As String = "Area is: " Declares and initializes a variable of type String Guide.InfoWriteLine(message & area) Calls the InfoWriteLine function to write text to the NX Info window. This function lives in the NXOpen namespace, so its full name is NXOpen.Guide.InfoWriteLine. We can use the shortened name here because we wrote "Imports NXOpen" above ' Function to calculate circle area This is a "comment". Comments are descriptive text to help you and other readers understand the code. They are ignored by the compiler. Function CircleArea(r As Double) As Double This is the beginning of the definition of the function named CircleArea. The text in parentheses says that, when this function is called, it should receive as input a variable of type Double, which will be referred to as "r". As output, the function will return an item of type Double. Dim pi As Double = Math.PI Defines a variable called pi and gives it the value π (accurate to around 16 decimal places). The full name of the item on the right is System.Math.PI, but we have Imports System at the top of our file, so we can use the shortened name Math.PI. Dim area = pi * r * r Calculates the area, and stores it in a newly declared variable called area. We do not need to write "As Double" because the compiler can infer this. Return area Returns the value of area as the output of the function ■ Lines of Code Generally, you place one statement on each line of your source code. But you can put several statements on a single line if you separate them by the colon (:) character. So, for example, you might write x1 = 3:x2 = 1 : : y1 = 5:y2 = 2 : : z1 = 7:z2 = 9 A statement usually fits on one line, but when it is too long, you can continue it onto the next line by placing a space followed by an underscore character (_) at the end of the first line. For example: Dim identityMatrix As Double(,) = New Double(,) { {1, 0, 0}, _ {0, 1, 0}, _ {0, 0, 1} } Chapter 4: The Visual Basic Language Page 31 Actually, in modern versions of Visual Basic, the underscores are often unnecessary, since the compiler can figure out by itself when a line of code is supposed to be a continuation of the one before it. Note that "white space" (space and tab characters) don't make any difference, except in readability. The following three lines of code do exactly the same thing, but the first is much easier to read, in my opinion: y = 3.5 * (x + 3/(x - 1)) y=3.5*(x+3/(x-1)) y = 3.5 * (x + 3 / (x - 1)) ■ Built-In Data Types In Visual Basic, as in most programming languages, we use variables for storing values. Every variable has a name, by which we can refer to it, and a data type, which determines the kind of data that the variable can hold. Some of the more common built-in data types are shown in the following table: Type Description Examples Approximate Range of Values Integer A whole number 1, 2, 999, –2, 0 –2,147,483,648 through 2,147,483,647 Double Double-precision floating-point number 1.5, –3.27, 3.56E+2 4.9 × 10–324 to 1.8 × 10308, positive or negative Char Character "x"c, "H"c, "@"c Any Unicode character String String of characters "Hello", "Rubbish" Zero up to about 2 billion characters Boolean Logical value True, False True or False Object Holds any type of data Anything Note that variables of type Double can use scientific notation: the "E" refers to a power of 10, so 3.56E+2 means 3.56 × 102, which is 356, and 3.56E–2 means 0.0356. There are many other built-in data types, including byte, decimal, date, and so on, but the ones shown above are the most useful for our purposes. ■ Declaring and Initializing Variables To use a variable, you first have to declare it (or, more precisely, you have to declare its type). You can also choose to give the variable some initial value at the time you declare it. Generally, a declaration/initialization takes the following form: Dim As = So, some examples are: Dim Dim Dim Dim Dim n As Integer = 45 i As Integer = 3*n bigger As Integer = 3*n biggestNumberExpected As Integer = 999 diameter As Double = 3.875 companyName As String = "Acme Incorporated" For more complex data types, you use the "New" keyword and call a "constructor" to declare and initialize a new object. For example: Dim v As New Vector(1,0,0) Dim myButton As New System.Windows.Forms.Button() A valid variable name can contain only letters, numbers, and underscores, and it must begin with either a letter or an underscore (not a number). Variable names are NOT case sensitive, so companyName and CompanyName are the same thing. Also, variable names must not be the same as Visual Basic keywords (like Dim or Integer). Unrestricted Getting Started with NX Open Chapter 4: The Visual Basic Language Page 32 There are some ways to omit or shorten variable declarations, as explained in the next section. ■ Omitting Variable Declarations When you're just experimenting with small programs, declaring variables is sometimes not very helpful, and the extra typing and text just interfere with your thought process. If you put Option Explicit Off at the beginning of your program, then you will prevent the compiler from complaining about missing declarations, and this might make your life easier (for a while, anyway). On the other hand, as we saw in chapter 3, declaring variables helps the compiler find mistakes for you, so it's valuable. When you write Option Explicit Off, the compiler will assume the types of undeclared variables, so it assumes that they are all of type System.Object. As we will see later, all objects in Visual Basic are derived (either directly or indirectly) from System.Object, so a variable of this type can hold any value whatsoever, and any assignment statement will work, no matter how peculiar: vec vec vec x = 3.75 = "Hello" = New Vector3d(3,2,1) vec X ' ' ' vec is of type System.Object so this strange assignment works and so does this works, too But you don't want this kind of thing in your real programs. Option Infer On If you type Option Infer On at the beginning of your program, the compiler will try to infer the types of variables, based on what you assign to them. So, for example, you can write Dim n = 5 instead of Dim n As Integer = 5, and the compiler will figure out that n must be an Integer. In the code below, the compiler will infer that x is an Integer, but you'll still want to use Option Strict On to make sure that type conversions happen as you intend. You can see the types of variables, and whether the compiler is guessing correctly, using Visual Studio Intellisense: when you hover the mouse over a variable, a tooltip will tell you its type. Option Infer On Dim Dim Dim Dim Dim n = 3 i = 3 * n greeting = "Hello" vec = New Vector3d(3, 2, 1) ' ' ' ' Compiler guesses that n is of type Integer Compiler guesses that greeting is of type String vec is of type Vector3d Intellisense helps us, now The word Dim before a variable is what prompts the compiler to start guessing. You are still declaring the variables x, y, greeting, and vec, but you don't have to tell the compiler their types, because it can guess from the context. This can cut down on a lot of repetition, and make your code much easier to read. In the following, the second and third lines of code are much clearer than the first three, and just as safe: Dim p1 As NXOpen.Point3d = New NXOpen.Point3d(2,3,4) Dim p1 = New NXOpen.Point3d(2,3,4) Dim p1 As New NXOpen.Point3d(2,3,4) In the examples later in this document, we will sometimes leave out variable declarations. You can use Option Infer On to make the code shorter and easier to read. But you have to be a little careful, sometimes, because the guessing isn't foolproof. Consider the following code: Option Infer On Dim x = 3 ' x is an Integer ' Error or unwanted rounding The compiler will infer that x is an Integer. So, in the second line of code, we're trying to assign a Double value to an Integer variable, and we'll either get an error message or the value of x will be rounded to 3 (instead of 3.14159...) when it's stored in the variable x. To avoid this sort of problem, you can write Dim x = 5.0 in the first line, which will tell the compiler that x is supposed to be a Double. Unrestricted Getting Started with NX Open Chapter 4: The Visual Basic Language Page 33 ■ Data Type Conversions Conversion is the process of changing a variable from one type to another. Conversions may either be widening or narrowing. A widening conversion is a conversion from one type to another type that is guaranteed to be able to contain it (from Integer to Double, for example), so it will never fail. In a narrowing conversion, the destination variable may not be able to hold the value (an Integer variable can't hold the value 3.5), so the conversion may fail. Conversions can be either implicit or explicit. Implicit conversions occur without any special syntax, like this: Dim weightLimit As Integer = 500 Dim weight As Double = weightLimit ' Implicit conversion from Integer to Double Explicit conversions, on the other hand, require so-called "cast" operators, as in the following examples. Dim weight As Double = 500.637 Dim roughWeight As Integer roughWeight = CInt(weight) roughWeight = CType(weight, Integer) ' Cast weight to an integer (rounding occurs) ' Different technique, but same result You can perform casts with the general CType function, or with more specific functions like CInt. The result is exactly the same — the weight value is rounded and we get roughWeight = 501. The set of allowable implicit conversions depends on the Option Strict setting. If you use Option Strict On, only widening conversions may occur implicitly. With Option Strict Off, both widening and narrowing conversions may occur implicitly. ■ Arithmetic and Math Arithmetic operators are used to perform the familiar numerical calculations on variables of type Integer and Double. The only operator that might be slightly unexpected is "^", which performs exponentiation (raises a number to a power). Here are some examples: Dim m As Integer = 3 Dim n As Integer = 4 Dim p1, p2, p3, p4, p5 As Integer p1 = m + n ' p1 now has the value p2 = 2*m + n - 1 ' p2 now has the value p3 = 2*(m + n) - 1 ' p3 now has the value p4 = m / n ' p4 now has the value p5 = m ^ n ' p5 now has the value 7 9 13 1. Beware !! 81 Even though m and n are both integers, performing a division produces a Double (0.75) as its result. But then when you assign this value to the Integer variable p4, it gets rounded to 1. With either Integer or Double data types, dividing by zero will cause trouble, of course. The System.Math namespace contains all the usual mathematical functions, so you can write things like: Dim rightAngle As Double = System.Math.PI / 2 Dim cosine As Double = System.Math.Cos(rightAngle) Dim x, y, r, theta As Double theta = System.Math.Atan2(3, 4) ' theta is about 0.6345 (radians) x = System.Math.Cos(theta) ' x gets the value 0.8 y = System.Math.Sin(theta) ' y gets the value 0.6 r = System.Math.Sqrt(x*x + y*y) Note that the trigonometric functions expect angles to be measured in radians, not in degrees. Other useful tools include hyperbolic functions (Sinh, Cosh, Tanh), logarithms (Log and Log10), and absolute value (Abs). Visual Studio Intellisense will show you a complete list as you type. Unrestricted Getting Started with NX Open Chapter 4: The Visual Basic Language Page 34 In floating point arithmetic (with Double variables), small errors often occur because of round-off. For example, calculating 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1 (10 times) won't give you 1.0, you'll get 0.99999999999999989, instead. Tiny errors like this usually don't matter in engineering applications. But, in cases where they do, you can use the Decimal data type, instead of Double. Arithmetic is much slower with Decimal variables, but more precise. ■ Logical Values & Operators Visual Basic provides a set of relational operators that perform some comparison between two operands and return a Boolean (true or false) result. Briefly, these operators are, =, , =, . Their meanings are fairly obvious, except perhaps for the last one, which means "is not equal to". Also, there are some logical operators that act on Boolean operands. They are: And: the result is True when both of the operands are True Or: the result is True when at least one of the operands is True Xor: the result is True when exactly one of the operands is True Not: this is a unary operator. The result is True if the operand is False Using these operators, we can construct complex conditions for use in If statements and elsewhere: Dim Dim Dim Dim Dim b1 b2 b3 b4 b5 b6 four As five As six As m, n As b1, b2, = = = = = = Integer Integer Integer Integer b3, b4, = 4 = 5 = 6 b5, b6 As Boolean (four = five) (six < five) (four five) ("four" < "five") (four < five) And (five < six) (m < n) Or (m >= n) ' ' ' ' ' ' Result Result Result Result Result Result is is is is is is False False True False. String comparison is alphabetical ! True True (regardless of values of m and n) ■ Arrays An array is a collection of values that are related to each other in some way, and have the same data type. Within an array, you can refer to an individual element by using the name of the array plus a number. This number has various names: index, offset, position, or subscript are some common ones. The term "offset" is perhaps the best, since it highlights the fact that the numbering starts at zero — the first element of the array has an offset of zero. In the following code, the first line declares and initializes an array variable that holds the number of people who work on each floor of an office building. It says that 5 people work on the ground floor, 27 on the first floor, and so on. Then the second and third lines read values from the people array. Dim people As Integer() = {5, 27, 22, 31} Dim groundFloorPeople As Integer = people(0) Dim firstFloorPeople As Integer = people(1) ' 5 people work on the ground floor ' 27 people work on the first floor Note that the style of array declaration shown here is perfectly legal, but it is not the usual one. Most VB programmers would write Dim people() As Integer, but I think the style shown above makes more sense — it says that people is an Integer( ) (i.e. it is an Integer array). If you want to declare and initialize the array separately, then you write something like: Unrestricted Getting Started with NX Open Chapter 4: The Visual Basic Language Page 35 Dim people As Integer() people = New Integer(3) {} people(0) = 5 people(1) = 27 people(2) = 22 people(3) = 31 ' Declares people as an array of integers ' Initialises the "people" array variable ' Initialise the elements of the array, one by one In this case, you need to place an integer between the parentheses in the declaration. Note that the number you use is the upper bound of the array (the highest index), which is one less than the number of elements in the array. So, in the example above, the "New Integer(3)" gave us an array of four integers with indices 0, 1, 2, 3. If you have experience with C-style programming languages, this can be very confusing, so please beware. You can also create two-dimensional (and higher dimension) arrays using declarations like Dim identityMatrix As Double(,) = { {1,0,0}, {0,1,0}, {0,0,1} } The .NET framework provides many useful functions for working with arrays. For example: The Length property returns the total number of elements in the array The GetUpperBound method returns the highest index value for the specified dimension The Sort method sorts the elements of a one-dimensional array The Find and FindIndex methods allow you to search for specific items ■ Other Types of Collections The .NET Framework includes the System.Collections namespace, which provides many useful "collections" that are more general than the arrays described above. For example, there are Lists, Dictionaries (Hash Tables), Queues, Stacks, and so on. You should use a List (rather than an array) when you don't know in advance how many items you will need to store. Here is a simple example: Dim nameList As New List(Of String) ' Create a list of strings Dim name As String Do ' Loop to collect names name = GetName() ' Get the next name, somehow nameList.Add(name) ' Add it to our list Loop Until name = "" ' Keep going until a blank name is encountered There is also a general collection called an ArrayList, which can hold elements of different types. So, you can write: Dim myList As New ArrayList myList.Add("apple pie") myList.Add(System.Math.PI) Dim x as Double = myList(1) ' Gives x the value 3.14159625 etc. Like a List, an ArrayList expands dynamically as you add elements. Though the ArrayList type is more general, you should use the List type, where possible, since it is faster and less error-prone. Most of the "collection" types support the same capabilities as arrays, such as indexing, counting, sorting, searching, and so on. ■ Strings A String is essentially an array of characters. You can declare and initialize a string with one statement like: Dim myString As String = "Hello, World!" Unrestricted Getting Started with NX Open Chapter 4: The Visual Basic Language Page 36 You can extract characters from a String just as if it were an array of characters: Dim Dim Dim Dim alphabet As String = "ABC" c0 As Char = alphabet(0) c1 As Char = alphabet(1) c2 As Char = alphabet(2) ' Sets c0 equal to "A" ' Sets c1 equal to "B" ' Sets c2 equal to "C" You can "concatenate" two strings (join them together into one) using either the "+" or "&" operators. Also, there are many useful functions available for working with strings; some of them are: Trim, ToUpper, ToLower, SubString, StartsWith, Compare, Copy, Split, Remove and Length. For example: Dim firstName As String = "Jonathon" Dim lastName As String = "Smith" Dim nickName As String = firstName.Substring(0, 3) Dim fullName As String = firstName & " " & lastName Dim greeting As String = "Hi, " & nickName ' Sets nickName = "Jon" ' Sets fullName = "Jonathon Smith" ' Sets greeting = "Hi, Jon" Strings are immutable, which means that once you assign a value to one, it cannot be changed. Whenever you assign another value to a string, or edit it in some way, you are actually creating a new copy of the string variable and deleting the old one. If you are doing a lot of modifications to a string variable, use the StringBuilder type, instead, because it avoids this deletion/recreation and gives much better performance. Any .NET object can be converted to String form using the ToString method. So, for example, this code Dim pi As Double = System.Math.PI Dim piString As String = pi.ToString will place the string "3.14159265358979" in the variable piString. ■ Enumerations Enumerations provide a convenient way to work with sets of related constants. You can give names to the constants, which makes your code easier to read and modify. For example, in NX Open, there is an enumeration that represents the various types of line font that can be assigned to an object. In shortened form, its definition might look something like this: Enum ObjectFont Solid = 0 Dashed = 1 Dotted = 2 End Enum Having made this definition, the symbol ObjectFont.Dotted now permanently represents the number 2. The benefit is that a statement like myFont = ObjectFont.Dotted is much easier to understand than myFont = 2. ■ Nothing Some of the data types we have discussed above can have a special value called Nothing (or "null" in some other programming languages). For example, strings, arrays, and objects can all have the value Nothing. Visual Basic provides a special function called IsNothing to make it easy to test for this value. Note that Nothing does not indicate a string with no characters, or an array with zero length, as the following code illustrates: Dim nullString As String = Nothing Dim zeroLengthString As String = "" ' A String variable with value = Nothing ' A String with zero length (no characters) Dim b1 As Boolean = IsNothing(nullString) Dim b2 As Boolean = IsNothing(zeroLengthString) Unrestricted Getting Started with NX Open ' True ' False Chapter 4: The Visual Basic Language Page 37 Simple data types like Integer, Double, Vector3d and Point3d cannot have the value Nothing, ordinarily — there is no such thing as a null integer or a null Point3d. This is actually quite inconvenient, at times. For example, in a function that computes the point of intersection of two curves, it would be natural to return Nothing if the curves don't actually intersect. Fortunately, recent versions of Visual Basic provide a solution via a technology called "nullable value types": by placing a question mark (?) after a variable type, you can indicate that it should be allowed to hold the value Nothing, in addition to its "regular" values. Then you can use the HasValue function to find out whether or not the variable holds a "real" value, rather than Nothing. Actually, Point3d? is an abbreviation for Nullable(Of Point3d), and you may see the longer form in documentation, sometimes. ■ Decision Statements Simple decisions can be implemented using the If Then Else construct, as shown in the following tax computation. It assumes that we have already defined two variables called income and tax If income < 27000 Then tax = income * 0.15 ElseIf income < 65000 Then tax = 4000 + (income - 27000) * 0.25 Else tax = 4000 + (income - 65000) * 0.35 End If ' 15% tax bracket ' 25% tax bracket ' 35% tax bracket If there were only two tax brackets, we wouldn't need the ElseIf clause, so our code could be simpler: If income < 27000 Then tax = income * 0.15 Else tax = 4000 + (income - 65000) * 0.35 End If ' 15% tax bracket ' 35% tax bracket This could be simplified even further: tax = income * 0.15 If income > 27000 Then tax = 4000 + (income - 65000) * 0.35 End If ' 15% tax bracket ' 35% tax bracket Finally, we can compress the If statement into a single line, if we want to: If income > 27000 Then tax = 4000 + (income - 65000) * 0.35 ' 35% tax bracket ■ Looping It is often useful to repeat a set of statements a specific number of times, or until some condition is met, or to cycle through some set of objects. These processes are all called "looping". The most basic loop structure is the For ... Next loop, which takes the following form For i = 0 To n a(i) = 0.5 * b(i) c(i) = a(i) + b(i) Next The variable i is called the loop counter. The statements between the For line and the Next line are called the body of the loop. These statements are executed n+1 times, with the counter i set successively to 0, 1, 2, ..., n. It is often convenient to declare the counter variable within the For statement. Also, you can append the name of the counter variable to the Next statement, which sometimes improves clarity, especially in "nested" loops like this: Unrestricted Getting Started with NX Open Chapter 4: The Visual Basic Language Page 38 For i As Integer = 0 To m For j As Integer = 0 To n c(i, j) = a(i) + b(j) Next j Next i Several other looping constructs are available, including: The For Each...Next construction runs a set of statements once for each element in a collection. You specify the loop control variable, but you do not have to determine starting or ending values for it. The Do...Loop construction allows you to test a condition at either the beginning or the end of a loop structure. You can also specify whether to repeat the loop while the condition remains True or until it becomes True. ■ Functions and Subroutines In many cases, you will call a "function" to perform some task. For example, you call the Math.Sqrt function to calculate the square root of a number, or you call the NX Open CreatePoint function to create a Point. Sometimes the function is one that you wrote yourself, but, more often, it's part of some library of functions written by someone else (like NX Open). You pass inputs to a function when you call it, the code inside the function is executed, and then (sometimes) it returns some value to you as output. The function provides a convenient place to put a block of code, so that it's easy to re-use. Here are some examples of function calls: ' Some calls to the Math.Sqrt function Dim x, y, z As Double x = 3 y = Math.Sqrt(x) z = Math.Sqrt(5) Dim root2 As Double = Math.Sqrt(2) ' Some calls to the NXOpen.Guide.InfoWriteLine function Dim greeting As String = "Hello" NXOpen.Guide.InfoWriteLine(greeting) NXOpen.Guide.InfoWriteLine("Goodbye") ' Some more calls to NXOpen functions Dim p As New NXOpen.Point3d(3, 5, 7) workPart.Points.CreatePoint(p) In Visual Basic, a function that does not return a value is called a "Subroutine" or just a "Sub". In the code above, NXOpen.Guide.InfoWriteLine is a subroutine, but Math.Sqrt and CreatePoint are not. Even if a function does return a value, you are not obligated to use this value. For example, in the code above, we didn't use the value returned from the CreatePoint function. A function can have any number of inputs (or "arguments") including zero. Near the start of this chapter, we saw an example of a function (CircleArea) that you might have written yourself: Function CircleArea(r As Double) As Double Dim pi As Double = 3.14 Dim area As Double = pi * r * r Return area End Function Since you have the source code of this function, you could just use this code directly, instead of calling the function, but we would not recommend this approach; calling functions makes your code less repetitive, easier to read, and easier to change. The general pattern for a function definition is: Function (arguments) As End Function Some further examples are: Unrestricted Getting Started with NX Open Chapter 4: The Visual Basic Language Page 39 Function RectangleArea(width As Double, height As Double) As Double ' Area of a rectangle Function Average(m As Double, n As Double) As Double ' Average of two numbers Function Average(values As Double()) As Double ' Avg of list of numbers Function Cube(center As Position, size As Double) As NXOpen.Body ' Create a cube Note that it's perfectly legal to have several functions with the same name, provided they have different types of inputs. This technique is called "overloading", and the function name is said to be "overloaded". For example, the function name "Average" is overloaded in the list of function definitions above. When you call the function, the compiler will decide which overload to call by looking at the types of inputs you provide. ■ Classes In addition to the built-in types described earlier, Visual Basic allows you to define new data types of your own. The definition of a new user-defined data type is held in a block of code called a class. The class represents a generic object, and a specific concrete object of this type is called an "instance" of the class. So, for example, we might have a "Sphere" class that represents spheres in general, and the specific sphere object with center at (0,0,0) and radius = 3 would be an instance of this Sphere class. New objects defined by classes have fields, properties and methods. Fields and properties can be considered as items of data (like the radius of a sphere), and a method is a function that does something useful with an object of the given class (like calculating the volume of a sphere). Properties are described in the next section, but, for now, you can think of a property as just a field with a smarter and safer implementation — it provides controlled read/write access to a hidden field. A class typically includes one or more functions called "constructors" that are used to create new objects. So, a typical class definition might look like this: Public Class Ball Public Center As Position Public Radius As Double ' Constructor, Sub New(center Me.Center = Me.Radius = End Sub ' Field to hold center point (should be a property, really) ' Field to hold radius value (should be a property, really) given a position and a radius As Position, r As Double) center r ' Constructor, given center coordinates and radius Sub New(x As Double, y As Double, z As Double, r As Double) MyClass.New(New Position(x, y, z), r) End Sub ' Function (method) to calculate volume Public Function Volume() As Double Return (4 / 3) * System.Math.PI * Me.Radius ^ 3 End Function ' Function (method) to draw a ball Public Sub Draw() ' Code omitted End Sub End Class Note that the constructors are "overloaded" — there are two of them, with different inputs. To create a Ball object, you call a constructor using the New keyword. Properties and methods are both accessed using a "dot" notation. As soon as you type a period in Visual Studio, Intellisense will show you all the available fields, properties and methods. Unrestricted Getting Started with NX Open Chapter 4: The Visual Basic Language Page 40 In this class, Center and Radius are both public fields, so you can access them directly. It would be safer to make them private fields and provide properties to access them. By doing this, we could prevent the calling code from making balls with negative radius, for example. Code to use the Ball class looks like this: Dim myBall As New Ball(x, y, z, r) myBall.Radius = 10 Dim mass As Double = density * myBall.Volume() myBall.Draw ' ' ' ' Create a ball named "myBall" Change its Radius property (or field) Use the Volume method Display the ball Note that the first line of code uses a convenient shorthand notation. The full form would have been Dim myBall As Ball = New Ball(x, y, z, r) Also, note how empty parentheses can simply be omitted when you call a function that has no arguments: so, we wrote myBall.Draw instead of myBall.Draw(). Similarly, in the line above, we could have just written myBall.Volume. ■ Shared Functions In the example above, we had a class called "Ball", and this class contained functions (methods) like Volume and Draw that operated on balls. This is the "object-oriented programming" view of life — the world is composed of objects that have methods operating on them. This is all very nice, but some software doesn't fit naturally into this model. Suppose for example that we had a collection of functions for doing financial calculations — for calculating things like interest, loan payments, and so on. The functions might have names like SimpleInterest, and LoanPayment, etc. It would be natural to gather these functions together in a class named FinanceCalculator. But the situation here would be fundamentally different from the "ball" class. The SimpleInterest function lives in the FinanceCalculator class, but it doesn't operate on FinanceCalculator objects. Saying it another way, the SimpleInterest function is associated with the FinanceCalculator class itself, not with instances of the FinanceCalculator class. Functions like this are called "Shared" functions in Visual Basic (or "static" functions in many other languages). You have already seen this word many times before because the "Main" function is always Shared. By contrast, the functions Volume and Draw in the Ball class are called Member functions or Instance functions. So, in short, the FinanceCalculator class is simply a collection of Shared functions. This is a common situation, so Visual Basic has a special construct to support it — a class that consists entirely of Shared functions is called a Module. Calls to Member functions and Shared functions look the same in our code, but they are conceptually different. For example, look at: Dim myBall As New Ball(x, y, z, r) Dim v As Double = myBall.Volume() Dim payment As Double = FinanceCalculator.LoanPayment(20000, 4.5) Both the second and third lines use the "dot" notation to refer to a function. But, in these two cases, the thing that comes before the "dot" is different. In myBall.Volume on the second line, myBall is an object (of type Ball), but in FinanceCalculator.LoanPayment on the third line, FinanceCalculator is a class. ■ Object Properties Each type of object we create in a VB program typically has a set of "properties" that we can access. For example, a point has a position in space, an arc has a center and a radius, a curve has a length, and a solid body has a density. In all cases, you can read (or "get") the value of the property, and in many cases, you can also write (or "set") the value. Setting a property value is often a convenient way to modify an object. If you are familiar with the GRIP language, these properties are exactly analogous to GRIP EDA (entity data access) symbols. Each property has a name. To get or set the property, you use a "dot" followed by the name of the property. So, if myCircle is an arc, then you refer to its center as myCircle.Center, and its radius as myCircle.Radius. Unrestricted Getting Started with NX Open Chapter 4: The Visual Basic Language Page 41 If p1, p2, p3 are three given positions, then (conceptually) we can write code like this: c1 = Circle(p1, p2, p3) r = c1.Radius c1.Center = p2 ' Creates a circle through three positions p1, p2, p3 ' Gets the radius of the circle ' Moves the circle, placing its center at position p2 ■ Hierarchy & Inheritance Object methods and properties are hierarchical. In addition to its own particular properties, a given object also has all the properties of object types higher up in the object hierarchy. So, for example, since a Line is a kind of Curve, it has all the properties and methods of the Curve type, in addition to the particular ones of lines. We say that the Line type "inherits" properties and methods from the Curve type. A portion of the hierarchy is shown below: System.Object Integer Double String NXObject Vector Point Curve Body Line Arc Spline As you can see, every object is derived from System.Object, and therefore inherits certain mysterious properties from it (like the Finalize, GetHashCode, and MemberwiseClone functions). The tables in the following chapters indicate the types of objects we will be using, and their properties. You might think you will need to keep these tables handy as you are writing code, so that you know what properties are available. But, this is not the case assuming you are using a modern IDE (Integrated Development Environment) to write your code. In a good IDE (like Visual Studio), as soon as you type a dot, a list of available properties and methods will appear, and all you have to do is choose the one you want. Some enthusiasts like to say that "the code writes itself" Unrestricted Getting Started with NX Open Chapter 4: The Visual Basic Language Page 42 Chapter 5: Concepts & Architecture This chapter describes the overall structure of NX Open, and some of the underlying principles. The standard NX Open Reference Guide tells you how to call any of the thousands of functions available in NX Open, but many people find it hard to see the "big picture", so they don't know where to start. This chapter explains the conceptual model behind NX Open programming, to make it easier to find the functions you need. ■ The Levels of NX Open The programming interfaces for NX have evolved over many years. Earlier generations are still supported and still work, even though they have been superseded by newer APIs and are no longer being enhanced. These older tools included an API called "User Function" or "UFUNC" that was designed to support applications written in the Fortran or C languages. The name of the User Function C API was subsequently changed, and it is now known as the NX Open C API, or sometimes just the Open C API. This API is old-fashioned, by today's standards, but it is extremely rich, fairly well documented, and still widely used. A large part of the NX Open .NET API (a portion called the NXOpen.UF namespace) was actually created by building "wrappers" around NX Open C functions. The NXOpen.UF functions are not used in recorded journals, so it's easy to forget about them, but they are very useful. Newer NX Open .NET functions are built directly on top of internal NX functions, so they by-pass the NX Open C layer. The SNAP layer is built on top of NX Open .NET. SNAP NX/Open .NET API functions. Some of these (left) call User Function Others (right) call NX internal functions NX/Open .NET NX/Open C API User Function NX A B C D E F NX internal functions ■ More About NXOpen.UF As mentioned above, there are many useful functions in the NXOpen.UF namespace. Some examples are: NXOpen.UF.Curve: many functions for working with curves NXOpen.UF.Disp: display functions (colors, grids, view names, etc.) NXOpen.UF.Draw: functions for working with drawings and drawing views NXOpen.UF.Drf: functions for working with drafting objects like dimensions NXOpen.UF.UFEval: information about curves and surfaces (points, tangents, normal, etc.) NXOpen.UF.UFModl: a large number of part modeling functions NXOpen.UF.UFPath: functions for working with NC toolpaths NXOpen.UF.UFSf: functions for working with finite element models (nodes, elements, meshes) You can use these functions alongside the newer functions, but mixing requires some simple conversions, as explained later in this chapter in the section entitled "Objects and Tags". Unrestricted Getting Started with NX Open Chapter 5: Concepts & Architecture Page 43 ■ The NX Open Inheritance Hierarchy As in most modern software systems, NX object classes are arranged in a hierarchical structure, with lower-level items inheriting from higher-level ones. There are hundreds of different object types, so the complete picture is difficult to understand (or even to draw). The simplified diagram below shows us the path from the top of the hierarchy down to some of the simple commonly-used objects. NXRemotableObject TaggedObject NXObject So, we see that a Point is a kind of "SmartObject", which is a kind of "DisplayableObject", and so on. The details are given later, but briefly, here are the roles of the more important object types: DisplayableObject Body DatumAxis DatumPlane Edge Face FacetedBody Sketch SmartObject Point Curve Line Conic RemotableObject Used for collections of preferences and also as the basis of the "UF" classes TaggedObject Used for lists of objects, for selections, and for "builders" (to be described later) Arc Ellipse Spline CoordinateSystem Axis Direction Plane Scalar Xform Feature BodyFeature Sphere NXMatrix Expression BasePart Part Builder FourPointSurfaceBuilder FeatureBuilder BooleanBuilder SphereBuilder NXObject Used for Part objects, and for objects that live inside NX part files, but are not displayed — views, layouts, expressions, lights, and so on. NXObjects have names and other non-graphical attributes. DisplayableObject Includes most of the object types familiar to users. Things like annotations, bodies, facetted bodies, datum objects, CAE objects. Displayable objects have colors, fonts, and other appearance attributes. Note that NX features are not displayable objects. SmartObject Includes points, curves, and some object types used as components of other objects when implementing associativity. BaseSession Session TaggedObjectCollection BaseFeatureCollection PartCollection FeatureCollection MathUtils ■ Sessions and Parts Typical NX objects (the ones we're discussing, here, anyway) reside in part files. So, if we want to create a new object, the first thing we must do is identify a part file in which this object will be created. Very often, you will want to create new objects in the current work part, so the required code is: Dim mySession As NXOpen.Session = NXOpen.Session.GetSession Dim parts As NXOpen.PartCollection = mySession.Parts Dim workPart As NXOpen.Part = parts.Work ' Get the current NX session ' Get the session's PartCollection ' Get the Work Part As you can see, we first get the current NX session object by calling the GetSession function. Every session object has a PartCollection object called "Parts" which we obtained in the second line of code. Then we get the Work Part Unrestricted Getting Started with NX Open Chapter 5: Concepts & Architecture Page 44 from this PartCollection. Of course, as always, we could have reduced our typing by putting Imports NXOpen at the top of our code file. In addition to the Work Part, there are other useful objects that you will probably want to initialize at the beginning of your program. Examples are the Display Part, the "UI" object, the "Display" object, the UFSession object, and so on. So, you will see code like this near the top of many NX Open programs: Dim Dim Dim Dim Dim Dim theSession As NXOpen.Session = NXOpen.Session.GetSession theWorkPart As NXOpen.Part = theSession.Parts.Work theDisplayPart As NXOpen.Part = parts.Display theUfSession As NXOpen.UF.UFSession = NXOpen.UF.UFSession.GetUFSession theDisplay As NXOpen.DisplayManager = theSession.DisplayManager theUI As NXOpen.UI = NXOpen.UI.GetUI ■ Objects and Tags As we mentioned earlier, there are many useful functions in the NXOpen.UF namespace. But the NXOpen.UF functions do not use the object types described earlier; they use object "tags" instead. Typical code looks like this: Dim coords As Double() = { 1.5, 2.5, 7 } Dim pointTag As NXOpen.Tag theUfSession.Curve.CreatePoint(coords, pointTag) theUfSession.Obj.SetLayer(pointTag, 30) Notice how the CreatePoint function does not return an NXOpen.Point object, it returns the NXOpen.Tag (pointTag) of the point it created. Then, whenever we want to refer to this point in subsequent code, we use this tag. So, for example, in the last line of code, pointTag is used as input to the SetLayer function. We can contrast this with code that does the same operations using newer object-based functions: Dim coordsPt As New NXOpen.Point3d(1.5, 2.5, 7) Dim myPoint As NXOpen.Point = theWorkPart.Points.CreatePoint(coordsPt) myPoint.Layer = 30 Of course, there will be times when you want to use a mixture of NXOpen.UF functions and newer ones, so it's important to understand how objects and tags relate to one another. In one direction, the correspondence is very simple: if you have an NX object called myObject, then myObject.Tag gives you its tag. So, we could do this: Dim myPoint As NXOpen.Point = theWorkPart.Points.CreatePoint(coordsPt) Dim pointTag As NXOpen.Tag = myPoint.Tag theUfSession.Obj.SetLayer(pointTag, 30) In the opposite direction (from tag to object), the process is slightly more complicated: Dim pointTag As NXOpen.Tag theUfSession.Curve.CreatePoint(coords, pointTag) Dim obj As NXOpen.TaggedObject = NXOpen.Utilities.NXObjectManager.Get(pointTag) Dim myPoint As NXOpen.Point = CType(obj, NXOpen.Point) myPoint.Layer = 30 As you can see, calling the NXObjectManager.Get function gives us an NXOpen.TaggedObject, and then we cast this to type NXOpen.Point. In practice, you would probably shorten this by using an implicit cast, like this: Dim myPoint As NXOpen.Point = NXOpen.Utilities.NXObjectManager.Get(pointTag) Unrestricted Getting Started with NX Open Chapter 5: Concepts & Architecture Page 45 ■ Factory Objects In NX Open, an object is usually not created by calling a constructor function. Instead, you use a "create" function that is a member function of some "factory" object. The "factory" concept is well-known in the software engineering field — just as in real life, a factory is a place where you produce new items. Different types of objects use different types of factories. Typically you can get a suitable factory object from an NXOpen.Part object (usually the work part), or from an NXOpen.Session object. So, suppose for example, that we want to create a point in a part named myPart. The relevant factory object is a PointCollection object called myPart.Points. So, the CreatePoint function can be found in the PointCollection class, and you use it as follows to create a point: Dim coords As new Point3d(3, 5, 0) Dim points As PointCollection = myPart.Points Dim p1 As Point = points.CreatePoint(coords) ' Define coordinates of point ' Get the PointCollection of the myPart ' Create the point Sometimes the factory object provides functions for directly creating objects, as in the example above. Other times, the factory object provides functions for creating "builder" objects, instead, as discussed in the next section. The following table shows some common examples of factory objects, how you obtain instances of these factory objects, and what sorts of creation functions they provide: Type of Factory Object Instance of Factory Object Example Creation Functions PointCollection BasePart.Points CreatePoint CreateQuadrantPoint CurveCollection BasePart.Curves CreateLine CreateArc CreateEllipse Features.FeatureCollection Part.Features CreateSphereBuilder CreateDatumPlaneBuilder CreateExtrudeBuilder CreateStudioSplineBuilder Annotations.AnnotationManager Part.Annotations CreateNote CreateLabel CreateDraftingNoteBuilder CAE.NodeElementManager CAE.BaseFEModel.NodeElementMgr CreateNodeCreateBuilder CreateElementCreateBuilder CAM.OperationCollection CAM.CAMSetup.CAMOperationCollection CreateHoleMakingBuilder CreateFaceMillingBuilder CAM.NCGroupCollection. CAM.CAMSetup.CAMGroupCollection CreateMillToolBuilder CreateDrillSpotfaceToolBuilder DexManager Session.DexManager CreateIgesImporter CreateStep203Creator PlotManager BasePart.PlotManager CreateCgmBuilder CreatePrintBuilder Unrestricted Getting Started with NX Open Chapter 5: Concepts & Architecture Page 46 Here are some further examples showing factory objects and their simple creation functions: Dim coords As New NXOpen.Point3d(3, 5, 9) Dim pointFactory As NXOpen.PointCollection = workPart.Points Dim myPoint = pointFactory.CreatePoint(coords) Dim Dim Dim p1 As New NXOpen.Point3d(1, 6, 5) p2 As New NXOpen.Point3d(3, 2, 7) curveFactory As NXOpen.CurveCollection = workPart.Curves myLine = curveFactory.CreateLine(p1, p2) Dim text As String() = { "Height", "Diameter", "Cost" } Dim origin As New NXOpen.Point3d(3,8,0) Dim horiz As NXOpen.AxisOrientation = NXOpen.AxisOrientation.Horizontal Dim noteFactory As NXOpen.Annotations.AnnotationManager noteFactory = workPart.Annotations Dim myNote = noteFactory.CreateNote(text, origin, horiz, Nothing, Nothing) In this code, we have explicitly defined the factory objects, to emphasize the role that they play. But, in typical code, they would not be mentioned explicitly, and you'd just write: Dim myPoint = workPart.Points.CreatePoint(coords) Dim myLine = workPart.Curves.CreateLine(p1, p2) Dim myNote = workPart.Annotations.CreateNote(text, origin, horiz, Nothing, Nothing) ■ Object Collections In many cases, the factory object described in the previous section has the word "collection" in its name. This is because, in addition to its object creation duties, the factory object also provides us with a way of cycling through objects of a specific type in a part file. This is useful if you want to cycle through the objects, performing some operation on each of them. For example, to perform some operation on all the points in a given part file (myPart), you can write: For Each pt As Point In myPart.Points ' Do something with pt Next Speaking more formally, the PointCollection class implements the IEnumerable interface, which means it's a collection of items that you can cycle through using a For Each loop. ■ The Builder Pattern We saw above how various factory objects provide functions like CreatePoint, CreateLine, and CreateNote that let us create simple objects directly. However, if we tried to handle more complex objects this way, we would need creation functions with huge numbers of input arguments. To avoid this complexity, we first create a "builder" object, and then we use this to create the object we want. As an example, here is the code to build a sphere feature: [1] [2] [3] [4] [5] [6] [7] [8] [9] Dim nullSphere As NXOpen.Features.Sphere = Nothing Dim mySphereBuilder As NXOpen.Features.SphereBuilder mySphereBuilder = workPart.Features.CreateSphereBuilder(nullSphere) mySphereBuilder.Type = mySphereBuilder.Center = mySphereBuilder.Diameter = mySphereBuilder.BooleanOption.Type = Dim myObject As NXOpen.NXObject = mySphereBuilder.Commit mySphereBuilder.Destroy Unrestricted Getting Started with NX Open Chapter 5: Concepts & Architecture Page 47 The code uses the so-called "Builder Pattern", which is a well-known software engineering technique for creating complex objects. The general approach is to Create a "builder" object — line [3] Modify its properties as desired — lines [4], [5], [6], [7] "Commit" the builder to create a new object — line [8] So, as we can see, along with the Sphere class, there is a corresponding SphereBuilder class, and a function called CreateSphereBuilder that produces a basic SphereBuilder object. The NX Open Reference Guide will help you find the classes and functions you need. Specifically The Sphere class refers you to the SphereBuilder class The SphereBuilder class refers you to the CreateSphereBuilder function The CreateSphereBuilder function indicates that it is provided by the factory class "FeatureCollection" The FeatureCollection class tells you to obtain an instance from a Part object (e.g. workPart.Features) You can actually use the CreateSphereBuilder function for either creation or editing purposes: if you input an existing Sphere object, then the Commit method will edit this sphere; if you input Nothing, then the Commit method will create a new sphere object, as in our code above. The meanings of the various builder properties that you need to set are fairly obvious in this simple case. But,

whenever you're in doubt about the meaning of a builder property, you can look at the corresponding feature creation dialog in interactive NX. You will see that the builder properties are closely related to the options that appear in the dialog. builder.Type = Features.SphereBuilder.Types.CenterPointAndDiameter builder.CenterPoint = centerPoint builder.Diameter.RightHandSide = diamString builder.BooleanOption.Type = NXOpen.GeometricUtilities.BooleanOperation.BooleanType.Create The commit function returns an NXOpen.NXObject, which is not immediately useful, in most situations. You typically have to cast to a more specific type (NXOpen.Features.Sphere in our example above) before making further use of the object. Builders for "feature" objects also have a CommitFeature method. This returns a very general NXOpen.Features.Feature object, so a cast will still be necessary in many cases. You can either perform the cast explicitly, or do it implicitly with an assignment statement, as shown here: ' Two steps: commit and explicit cast Dim myObject As NXOpen.Features.Sphere = builder.Commit Dim mySphere1 As NXOpen.Features.Sphere = CType(myObject, NXOpen.Features.Sphere) ' One step: commit and cast implicitly Dim mySphere2 As NXOpen.Features.Sphere = builder.CommitFeature In some cases, both the Commit and CommitFeature methods return Nothing, and you have to use the GetCommittedObjects function to obtain the created object(s). Unrestricted Getting Started with NX Open Chapter 5: Concepts & Architecture Page 48 ■ Exploring NX Open By Journaling The NX Open API is very rich and deep — it has thousands of available functions. This richness sometimes makes it difficult to find the functions you need. Fortunately, if you know how to use the corresponding interactive function in NX, the journaling facility will tell you which NX Open functions to use. Here is the basic approach: Choose Developer tab Journal Record to start recording, run through the desired series of steps, and then choose Developer tab Journal Stop Recording. The code generated by journaling is verbose and is often difficult to read. But it's worth persevering, because hidden within that code is an example call showing you exactly the function you need. You can indicate which language should be used in the recorded code by choosing File tab Preferences User Interface Tools Journal. The available choices are C#, C++, Java, Python, and Visual Basic. ■ The "FindObject" Problem When you use a journal as the starting point for an application program, one of the things you need to do is remove the "FindObject" calls that journaling produces. This section tells you how to do this. A journal records the exact events that you performed during the recording process. If you select an object during the recording process, and do some operations on it, the journal actually records the name of that object. So, when you replay the journal, the operations will again be applied to this same named object. This is almost certainly not what you want — you probably want to operate on some newly-selected object, not on the one you selected during journal recording. Very often, objects with the original recorded names don't even exist when you are replaying the journal, so you'll get error messages. To clarify further, let's take a specific example. Suppose your model has two objects in it — two spheres named SPHERE(23) and SPHERE(24). If you record a journal in which you select all objects in your model, and then blank them, then what gets recorded in the journal will be something like this: Dim objects1(1) As DisplayableObject Dim body1 As Body = CType(workPart.Bodies.FindObject("SPHERE(23)"), Body) objects1(0) = body1 Dim body2 As Body = CType(workPart.Bodies.FindObject("SPHERE(24)"), Body) objects1(1) = body2 theSession.DisplayManager.BlankObjects(objects1) If you replay this code, it's just going to try to blank SPHERE(23) and SPHERE(24) again, which is probably useless. There's a good chance that SPHERE(23) and SPHERE(24) won't exist at the time when you're replaying the journal, and, even if they do, it's not likely that these are the objects you want to blank. Clearly we need to get rid of the "FindObject" calls, and add some logic that better defines the set of objects we want to blank. There are a few likely scenarios: Maybe we want to blank some objects that were created by code earlier in our application Maybe we want to blank some objects selected by the user when our application runs Maybe we want to blank all objects in the model, or all the objects that have certain characteristics The first of these is easy: if we created the objects in our own code, then presumably we assigned them to program variables, and they are easy to identify: Dim myBall0 As NX.Body = Sphere(1,2,1, 5).Body Dim myBall1 As NX.Body = Sphere(1,4,3, 7).Body Dim objects1(1) As DisplayableObject objects1(0) = myBall0 objects1(1) = myBall1 theSession.DisplayManager.BlankObjects(objects1) For the second case, we need to add a selection step to our code as outlined in chapter 15, and then blank the objects the user selects when the journal is replayed. Something like this: Unrestricted Getting Started with NX Open Chapter 5: Concepts & Architecture Page 49 Dim cue = "Please select the objects to be blanked" Dim dialog As Selection.Dialog = Selection.SelectObjects(cue) Dim result As Selection.Result = dialog.Show If result.Response NXOpen.Selection.Response.Cancel Then theSession.DisplayManager.BlankObjects(result.Objects) End If For the third case (blanking all the objects with certain characteristics), we will need to cycle through all the objects in our model, finding the ones that meet our criteria, and then pass these to the BlankObjects function. See the last section in chapter 15 for information about cycling through the objects in a part file. ■ Mixing SNAP and NX Open As we have seen, NX Open functions provide enormous power and flexibility. There is another NX API called SNAP whose functions are usually much easier to find and understand. So, there may well be situations where you will want to use SNAP and NX Open functions together. You can use SNAP functions for simple common operations, and NX Open functions for more complex ones. To do this, you may need to convert SNAP objects into NX Open objects, and vice-versa. We have tried to make these conversions as convenient as possible, so that SNAP and NX Open code can live together in peace and harmony. A SNAP object is just a simple wrapper around a corresponding NX Open object — for example, a Snap.NX.Spline object is just a wrapper that encloses an NXOpen.Spline, and a Snap.NX.Sphere is a wrapper around an NXOpen.Features.Sphere object, and so on. So, if you have an NXOpen object, you can "wrap" it to create a Snap.NX object. In the other direction, if you have a Snap.NX object, you can "unwrap" it to get the NXOpen object that it encloses. There are hidden "implicit" conversions that do this wrapping and unwrapping for you, so often things just work without any extra effort. For example: Dim snapPoint As Snap.NX.Point = Point(3,5,9) Dim session As NXOpen.Session = NXOpen.Session.GetSession session.Information.DisplayPointDetails(snapPoint) Dim pt As NXOpen.Point3d = snapPoint.Position In the third line, we are passing a Snap.NX.Point object to a function that expects to receive an NXOpen.Point. But the implicit conversion is invoked behind the scenes, and the function call just works as expected. Similarly, in the fourth line, we are assigning a Snap.Position object to an NXOpen.Point3d object, and this works, too. However, there are times when the implicit conversions don't work, and you need to do something more explicit. For example, if you want to use NXOpen member functions or properties, then you have to get an NXOpen object from your SNAP object first. So suppose, for example, that we have a Snap.NX.Sphere object called snapSphere, and we write the following code: snapSphere.HideParents Dim version = snapSphere.TimeStamp ' Fails ' Fails Both lines of code will fail, because a Snap.NX.Sphere object does not have a HideParents method or a TimeStamp property. So, to proceed, you have to "unwrap" to get the enclosed NXOpen.Features.Sphere object. You can do this in a couple of different ways, as shown below: CType(snapSphere, NXOpen.Features.Sphere).HideParents snapSphere.NXOpenSphere.HideParents ' Works, but a bit clumsy ' Nicer: use NXOpenSphere property The first line just uses the standard VB CType function to do the conversion, and the second line uses the NXOpenSphere property. The second approach, using properties, is the most convenient, so there are several Unrestricted Getting Started with NX Open Chapter 5: Concepts & Architecture Page 50 properties that let you get NXOpen objects from SNAP objects in this same way. For example, if snapSphere is a Snap.NX.Sphere object, again, then snapSphere.NXOpenSphere is the enclosed NXOpen.Features.Sphere object snapSphere.NXOpenTag is the NXOpen tag of this NXOpen.Features.Sphere object snapSphere.SphereBuilder is the "builder" object for the NXOpen.Features.Sphere Going in the other direction (from NXOpen to SNAP) is not quite so streamlined. The approach using properties is not available, so you have to call the Wrap function to create a new SNAP object from the NXOpen one, like this: Dim Dim Dim Dim coords = New NXOpen.Point3d(3, 6, 8) workPart As NXOpen.Part = Snap.Globals.WorkPart.NXOpenPart nxopenPoint As NXOpen.Point = workPart.Points.CreatePoint(coords) snapPoint As NX.Point = NX.Point.Wrap(nxopenPoint.Tag) ' Create a Snap.NX.Point location As Position = snapPoint.Position ' Use its Position property In the fourth line of code, we first get the tag of the NXOpen.Point object. Then we call the Wrap function, which gives us a new Snap.NX.Point object that "wraps" it. Then, in the last line, we can use the Position property of this new Snap.NX.Point object. As we saw above, the Wrap function receives an NXOpen.Tag as input. So, if you are working with older NXOpen functions that use tags, interoperability with SNAP is even easier. For example: Dim ufSession = NXOpen.UF.UFSession.GetUFSession Dim pointTag As NXOpen.Tag Dim coords As Double() = {2, 6, 9} ufSession.Curve.CreatePoint(coords, ByRef pointTag) Dim snapPoint As NX.Point = NX.Point.Wrap(pointTag) Unrestricted Getting Started with NX Open Chapter 5: Concepts & Architecture Page 51 Chapter 6: Positions, Vectors, and Points The next few chapters briefly outline the NX Open functions available for performing simple tasks. The function descriptions are fairly brief, since we are just trying to show you the range of functions available. The NX Open Reference Guide has much more detailed information, and this detailed information will also be presented to you as you are writing your code, if you use a good development environment like Visual Studio. Specifically, as soon as you type an opening parenthesis following a function name, a list of function inputs will appear, together with descriptions. You can also get complete information about any function or object by using the Object Browser in Visual Studio. Following the descriptions of functions, we often give small fragments of example code, showing how the functions can be used. The examples are very simple, but they should still be helpful. To keep things brief, the example code is often not complete. For example, declarations are often left out, and a complete Main function is only included very rarely. If you actually want to compile the example code, you will typically need to make some additions. ■ Point3d Objects A Point3d object represents a location in 3D space. After basic numbers, positions and vectors are the most fundamental objects in geometry applications, so we will describe them first. There are also Point2d and Point4d objects, but these are not used as often, so we won't discuss them here. Note that a Point3d is not a real NX object. Point3d objects only exist in your NX Open program — they are not stored permanently in your NX model (or anywhere else). So, as soon as your program has finished running, all your Point3d objects are gone. In this sense, they are just like the numerical variables that you use in your programs. If you want to create a permanent NX object to record a location, you should use an NXOpen.Point, not a Point3d. You can use the following function to create a Point3d object: Function Inputs and Creation Method Point3d(x As Double, y As Double, z As Double) From three rectangular coordinates. In the first column, you see a formal description of the types of inputs you should provide when calling the function — you have to provide three variables of type "double". This function is a constructor, so, when calling it, we have to use the "New" keyword in our code. Here are some examples: Dim p As New Point3d(3,5,8) Dim q As New Point3d (1.7, 2.9, 0) ' Creates a Point3d "p" with coordinates (3,5,8) ' Creates a Point3d "q" with coordinates (1.7, 2.9, 0) Point3d object properties are as follows: Data Type Property Access Description Double X get, set The x-coordinate of the Point3d Double Y get, set The y-coordinate of the Point3d Double Z get, set The z-coordinate of the Point3d Unrestricted Getting Started with NX Open Chapter 6: Positions, Vectors, and Points Page 52 ■ Vector3d Objects A Vector3d object represents a direction or a displacement in 3D space. Like Point3d objects, Vector3d objects only exist in your NX Open program — they are not stored permanently in your NX model (or anywhere else). You can use the following constructor function to create Vector3d objects: Function Inputs and Creation Method Vector3d(x As Double, y As Double, z As Double) From three rectangular components. Vector object properties are as follows: Data Type Property Access Description Double X get, set The x-component of the vector Double Y get, set The y-component of the vector Double Z get, set The z-component of the vector The NXOpen.VectorArithmetic class provides a Vector3 object that is very similar to NXOpen.Vector3d. This class also provides functions for performing operations on Vector3 objects, like addition, subtraction, cross products, and so on. In some cases, it might be convenient to use Vector3 objects for calculations, and then convert the answers to NXOpen.Vector3d form for further use. The following code illustrates the approach: Dim Dim Dim Dim u v w r = = = = New New u + New NXOpen.VectorArithmetic.Vector3(3,4,7) NXOpen.VectorArithmetic.Vector3(4,2,1) 0.5*v NXOpen.Vector3d(w.x, w.y, w.z) ■ Points Points might seem a lot like Point3d objects, but they are quite different. A Point is an NX object, which is permanently stored in an NX part file; Point3d and Vector3d objects are temporary things that exist only while your NX Open program is running. To create a point, we write code following the "factory" pattern explained in chapter 5. The basic idea is that a part file contains "collections" of different object types. So, for example, given a Part object named myPart, there is a collection called myPart.Points that contains all the Point objects in the part. Similarly, myPart.Arcs is a collection that contains all the arcs in this part, and myPart.Curves includes all the curves. These collections serve as "factory" objects that we can use to create new objects in a part file, as follows: Dim workPart As Part = session.Parts.Work ' Dim points As PointCollection = workPart.Points ' Dim coords As new Point3d(3, 5, 0) ' Dim p1 As Point = points.CreatePoint(coords) ' p1.SetVisibility(SmartObject.VisibilityOption.Visible) Get the Work Part Get the PointCollection of the Work Part Define coordinates of point Create the point (add to collection) The last line of code is necessary because an NXOpen.Point is a "SmartObject", which is invisible by default. The code above is written out in a rather verbose way, to allow for complete explanation. In practice, you would typically write something like this: Dim workPart As Part = session.Parts.Work Dim coords As new Point3d(3, 5, 0) Dim p1 As Point = workPart.Points.CreatePoint(coords) p1.SetVisibility(SmartObject.VisibilityOption.Visible) Unrestricted Getting Started with NX Open ' ' ' ' Chapter 6: Positions, Vectors, and Points Get the Work Part Define coordinates of point Create the point Make it visible Page 53 If you're inclined, you could even create a point with a single line of code, like this: Dim p1 = NXOpen.Session.GetSession.Parts.Work.Points.CreatePoint(New Point3d(3,5,0)) So, in summary, the following function creates a point in a part called myPart: Function Inputs and Creation Method myPart.Points.CreatePoint(x As Double, y As Double, z As Double) From x, y, z coordinates The properties of Point objects are as follows: Data Type Property Access Description Double X get, set The x-coordinate of the point. Double Y get, set The y-coordinate of the point. Double Z get, set The z-coordinate of the point. There are many functions that require Point3d objects as inputs. If we have a Point, instead of a Point3d, we can always get a Point3d. So, if pt is a Point, and we want to set the origin of an NXOpen.Direction (which requires a Point3d object), then the necessary code is: myDirection.Origin = New Point3d(pt.X, pt.Y, pt.Z) Unrestricted Getting Started with NX Open Chapter 6: Positions, Vectors, and Points Page 54 Chapter 7: Curves This chapter briefly outlines the NX Open functions for creating and editing curves (lines, arcs, and splines). For further details, please look at the NXOpen.CurveCollection and NXOpen.UF.UFCurve classes in the NXOpen Reference Guide. ■ Lines The NXOpen.CurveCollection class contains two functions for creating lines, as follows: Function Inputs and Creation Method CreateLine(p0 As Point3d, p1 As Point3d) Between two Point3d locations CreateLine(p0 As NXOpen.Point, p1 As NXOpen.Point) Between two points (NXOpen.Point objects) The following fragment of code creates two points and two lines in your Work Part: Dim p0 As New NXOpen.Point3d(1,2,3) Dim p1 As New NXOpen.Point3d(4,7,5) Dim line1 As NXOpen.Line = workPart.Curves.CreateLine(p0, p1) Dim pt0 As NXOpen.Point = workPart.Points.CreatePoint(p0) Dim pt1 As NXOpen.Point = workPart.Points.CreatePoint(p1) Dim line2 As NXOpen.Line = workPart.Curves.CreateLine(pt0, pt1) line2.SetVisibility(SmartObject.VisibilityOption.Visible) The code that creates line1 above is what you will get if you record the creation of a line using Insert Curve Basic Curves. Note that we had to set the visibility of line2 because lines created via this method are invisible by default. There are other ways to create lines, too. There is NXOpen.UF.UFCurve.CreateLine, and the NXOpen.LineCollection class also has some functions for creating lines along the axes of various types of surfaces of revolution. The geometric properties of lines are: Data Type Property Access Description Point3d StartPoint get Start point (point where t = 0). Point3d EndPoint get End point (point where t = 1). The StartPoint and EndPoint properties cannot be set directly, but the NXOpen.Line class provides SetStartPoint, SetEndPoint, and SetEndPoints functions that let you modify a line: Dim newPoint = New Point3d(6,7,9) myLine.SetEndPoint(newPoint) There is also a Guide.CreateLine function, which makes it even easier to create a line. ■ Associative Line Features In the code in the previous section, neither line1 nor line2 is associative. If you want to create associative lines, you should use the AssociativeLineBuilder class, instead. Code that uses this class will be produced if you record the Unrestricted Getting Started with NX Open Chapter 7: Curves Page 55 creation of a line using Insert Curve Line. The recorded code may be rather long, but its essential parts are as follows: ' Create an AssociativeLineBuilder Dim lineNothing As NXOpen.Features.AssociativeLine = Nothing Dim builder As NXOpen.Features.AssociativeLineBuilder builder = workPart.BaseFeatures.CreateAssociativeLineBuilder(lineNothing) builder.Associative = True ' Define the start point Dim p0 As New NXOpen.Point3d(1,2,3) Dim pt0 As NXOpen.Point = workPart.Points.CreatePoint(p0) builder.StartPointOptions = NXOpen.Features.AssociativeLineBuilder.StartOption.Point builder.StartPoint.Value = pt0 ' Define the end point Dim p1 As New NXOpen.Point3d(4,7,5) Dim pt1 As NXOpen.Point = workPart.Points.CreatePoint(p1) builder.EndPointOptions = NXOpen.Features.AssociativeLineBuilder.EndOption.Point builder.EndPoint.Value = pt1 ' Create an associative line feature Dim result As NXOpen.NXObject = builder.Commit builder.Destroy The result object created by this code is actually an Associative Line feature, rather than a line. This has advantages, sometimes — the feature appears in the Part Navigator and some forms of editing are easier. To obtain an oldfashioned line from an AssociativeLine feature, add the following two lines to the code above: Dim assocLine As NXOpen.Features.AssociativeLine = result Dim myLine As NXOpen.Line = assocLine.GetEntities(0) ■ Arcs and Circles The simplest functions for creating circular arcs can again be found in the NXOpen.CurveCollection class. There are three functions, as follows: Function Public Function CreateArc(startPoint As Point3d, pointOn As Point3d, endPoint As Point3d, alternateSolution As Boolean, ByRef flipped As Boolean) Public Function CreateArc(center As Point3d, xDirection As Vector3d, yDirection As Vector3d, radius As Double, startAngle As Double, endAngle As Double) Function Public Function CreateArc(center As Point3d, matrix As NXMatrix, radius As Double, Unrestricted Getting Started with NX Open Inputs and Creation Method Through three points From center, radius, angles, axis vectors. The arc lies in the plane containing the two given vectors. The center point is expressed using Absolute Coordinates, and the angles are in radians, measured in a counterclockwise direction from the xDirection vector. Inputs and Creation Method From center point, radius, angles, orientation matrix. The arc lies in the XY-plane of the matrix. The center point is expressed using Absolute Coordinates, and Chapter 7: Curves Page 56 the angles are in radians, measured in a counterclockwise direction from the x-axis of the matrix. startAngle As Double, endAngle As Double) There are no specific functions for creating complete circles; we simply set endAngle = startAngle + 2. Here is a simple program that uses lines and arcs to create a linkage bar lying in the YZ plane: Dim curves As NXOpen.CurveCollection = workPart.Curves Dim Dim Dim Dim Dim length As Double = 8 width As Double = 4 half As Double = width/2 holeDiam As Double = half pi As Double = System.Math.PI twopi As Double = 2*pi Dim p1, p2, p3, q1, q2, q3 As NXOpen.Point3d p1 = New Point3d(0, -half, 0) p2 = New Point3d(0, 0, 0) p3 = New Point3d(0, half, 0) : : : q1 = New Point3d(0, -half, length) q2 = New Point3d(0, 0, length) q3 = New Point3d(0, half, length) ' Left and right sides curves.CreateLine(p1, q1) curves.CreateLine(p3, q3) Dim axisX = New Vector3d(0,1,0) Dim axisY = New Vector3d(0,0,1) ' Horizontal ' Vertical ' Top and bottom arcs curves.CreateArc(q2, axisX, axisY, half, 0, pi) curves.CreateArc(p2, axisX, axisY, half, pi, twopi) ' Top and bottom holes curves.CreateArc(q2, axisX, axisY, holeDiam/2, 0, twopi) curves.CreateArc(p2, axisX, axisY, holeDiam/2, 0, twopi) There are other ways to create arcs, too. For example, the NXOpen.UF.UFCurve class has functions CreateArc, CreateArcThru3pts, and CreateFillet. The NXOpen.ArcCollection class does not have any functions for creating arcs. Finally, there are two Guide.CreateCircle functions that are very easy to use. The properties of arc objects are as follows: Data Type Property Access Description Double Radius get Radius of arc. Point3d Center get Center of arc (in absolute coordinates). Double StartAngle get Start angle (in radians). Double EndAngle get End angle (in radians). None of these properties can be set directly, but the NXOpen.Arc class includes two SetParameters functions that let you modify an arc in any way you want. Unrestricted Getting Started with NX Open Chapter 7: Curves Page 57 ■ Associative Arc Features The code in the previous section creates plain ordinary arc objects that are not associative. These are perfectly adequate for many applications, and are easy to create, but there may be situations where you want to create an associative arc, instead. To do this, you should use the AssociativeArcBuilder class. Code that uses this class will be produced if you record the creation of an arc using Insert Curve Arc. The recorded code may be rather long, but the essential parts are as follows: ' Create an AssociativeArcBuilder Dim arcNothing As NXOpen.Features.AssociativeArc = Nothing Dim builder As NXOpen.Features.AssociativeArcBuilder builder = workPart.BaseFeatures.CreateAssociativeArcBuilder(arcNothing) builder.Associative = True ' Set the type of associative arc builder.Type = NXOpen.Features.AssociativeArcBuilder.Types.ArcFromCenter ' Define the arc center point Dim centerCoords As New Point3d(1.1, 2.2, 0) Dim centerPoint As NXOpen.Point = workPart.Points.CreatePoint(centerCoords) builder.CenterPoint.Value = centerPoint ' Define the arc radius builder.EndPointOptions = NXOpen.Features.AssociativeArcBuilder.EndOption.Radius builder.Radius.RightHandSide = "7.89" ' Define the angular limits (in degrees) builder.Limits.StartLimit.LimitOption = GeometricUtilities.CurveExtendData.LimitOptions.Value builder.Limits.EndLimit.LimitOption = GeometricUtilities.CurveExtendData.LimitOptions.Value builder.Limits.StartLimit.Distance.RightHandSide = "22.2" builder.Limits.EndLimit.Distance.RightHandSide = "55.5" ' Create an associative arc feature and get its arc Dim myArcFeature As NXOpen.Features.AssociativeArc = builder.Commit builder.Destroy Dim myArc As NXOpen.Arc = myArcFeature.GetEntities()(0) The last line of code gets an ordinary NXOpen.Arc object from the AssociativeArc feature, which may or may not be necessary, depending on your application. An AssociativeArcBuilder object has a large number of properties — around 30 of them, in all. The best way to understand what they all mean is to look at the dialog for creating an arc in interactive NX. For example, we defined the start and end angles of the arc using two expressions that give the angular limits in degrees. If you edit the arc we created, you will see these expressions near the bottom of the edit dialog: You can create a full 360 degree circle by setting the limiting angles to 0 and 360, of course. Alternatively, you can just set builder.Limits.FullCircle = True. Unrestricted Getting Started with NX Open Chapter 7: Curves Page 58 ■ Conic Section Curves Simple functions for creating conic section curves (ellipses, parabolas, hyperbolas) can be found in the NXOpen.CurveCollection class. For example, one of the functions for creating an ellipse is as follows: Function Inputs and Creation Method Creates an ellipse The ellipse lies in the XY plane defined by the given matrix. The center point is expressed using Absolute Coordinates, and the angles are in radians, measured relative to the given matrix's axes. public Ellipse CreateEllipse(Point3d center, double majorRadius, double minorRadius, double startAngle, double endAngle, double rotationAngle, NXMatrix matrix) To create an ellipse using this function, the code is as follows: 'Get the matrix of the current WCS Dim wcsMatrix As NXOpen.NXMatrix = workPart.WCS.CoordinateSystem.Orientation Dim pi As Double = System.Math.PI Dim Dim Dim Dim Dim Dim center As New NXOpen.Point3d(0,0,0) rX As Double = 2 rY As Double = 1 a0 As Double = 0 a1 As Double = pi rot As Double = pi/6 ' ' ' ' ' ' Center point (absolute coordinates) Major Radius Minor radius Start angle End angle Rotation angle workPart.Curves.CreateEllipse(center, rX, rY, a0, a1, rot, wcsMatrix) This creates half of a full ellipse, lying in a plane parallel to the work plane, with its center at the absolute origin. The ellipse is rotated in its plane by an angle of 30 degrees (/6 radians). The NXOpen.Features.GeneralConicBuilder class allows you to create conic section curves by different techniques, by specifying various combinations of point and tangency constraints. The NXOpen.UF.UFCurve class also provides the CreateConic, and EditConicData functions. ■ Splines The NX Open functions for handling splines use a fairly conventional NURBS representation that consists of: Poles — An array of n 3D vectors representing poles (control vertices) Weights — An array of n weight values (which must be strictly positive) Knots — An array of $n + k$ knot values: $t[0], \dots, t[n + k - 1]$ The order and degree of the spline can be calculated from the sizes of these arrays, as follows: Let n = number of poles = Poles.Length Let $npk = n + k$ = number of knots = Knots.Length Then the order, k, is given by $k = npk - n$. Finally, as usual, the degree, m, is given by $m = k - 1$. You may not be familiar with the "weight" values associated with the poles, since these are not very visible within interactive NX — you can see them in the Info function, but you can't modify them. So, in this case, the NX Open API actually gives you more power than interactive NX. Generally, the equation of a spline curve is given by a rational function (the quotient of two polynomials). This is why spline curves are sometimes known as NURBS (Non-Uniform Rational B-Spline) curves. If the weights are all equal (and specifically if they are all equal to 1), then some cancellation occurs, and the equation becomes a polynomial. Unrestricted Getting Started with NX Open Chapter 7: Curves Page 59 The mathematical theory of splines is quite extensive (one of the best-known books on the subject is more than 600 pages long), so we can only scratch the surface here. For more information, please consult a text-book, or one of the many available on-line resources. The simplest function for creating a spline is NXOpen.UF.UFModl.CreateSpline, because its inputs closely match the defining data outlined above. The code is as follows: Dim n As Integer = 4 Dim k As Integer = 3 ' Number of poles ' Order of curve (degree = k-1 = 2) ' 3D coordinates of poles Dim p As Double(,) = { {1,0,0}, ' Weights Dim w As Double() {3,1,0}, {5,1,0}, {6,0,0} } = {1, 1, 0.7, 1} ' Construct 4D poles Dim poles4D(4*n-1) As Double For i As Integer = 0 to n-1 poles4D(4*i) = w(i) * p(i,0) poles4D(4*i + 1) = w(i) * p(i,1) poles4D(4*i + 2) = w(i) * p(i,2) poles4D(4*i + 3) = w(i) Next ' Knots must be an array of length n + k Dim knots As Double() = {0,0,0, 0.6, 1,1,1} Dim splineTag As NXOpen.Tag Dim knotFixup As Integer = 0 Dim poleFixup As Integer = 0 Dim ufs As NXOpen.UF.UFSession = NXOpen.UF.UFSession.GetUFSession ufs.Modl.CreateSpline(n, k, knots, poles4D, splineTag, knotFixup, poleFixup) Note how the 3D poles and the weights are combined into 4D elements of the form (wx, wy, wz, w). The code above is somewhat unusual because it uses weights that are not all equal, and therefore it creates a rational curve (rather than a polynomial one). In most cases, you would set all weights equal to one, so the poles4D array would simply be: 1,0,0,1, 3,1,0,1, 5,1,0,1, 6,0,0,1. To construct a curve of order k with n poles, you need $n + k$ knots. So, in our case, we need 7 knots. Since the curve has order 3, the knot sequence should begin with 3 0's and end with 3 1's. That only leaves one knot value undecided, and the code above assigns it a value of 0.6. The CreateSpline function returns two integers knotFixup and poleFixup that indicate whether or not any "fixup" of the data was performed inside NX. A typical fixup is a slight adjustment of knot values or poles that are very close together but not identical. In almost all cases, you will find that both fixup values are zero, indicating that no adjustments were required. There are several other functions for creating and editing splines. The NXOpen.UF.UFCurve class provides a function CreateSplineThruPts that allows you to construct a spline that passes through given points, and also lets you specify slopes and curvatures at these points. Also, in NXOpen.UF.UFModl, there is a function called CreateFittedSpline that performs smoothing by creating a spline that approximates given points without necessarily passing through them exactly. Unrestricted Getting Started with NX Open Chapter 7: Curves Page 60 The NX.Spline class provides several properties and functions that you can use to get information about a spline curve, as follows: Data Type Function/Property Access Description Boolean Rational get If true, indicates that the spline is rational (not polynomial) Boolean Periodic get If true, indicates that the spline is periodic Point4d[] GetPoles get The 4D poles of the spline (wx, wy, wz, w) Integer PoleCount get The number of poles; equal to GetPoles.Length/4 Double[] GetKnots get The knots of the spline Integer Order get The order of the curve (= degree + 1) ■ Studio Splines The functions described in the previous section all create NXOpen.Spline objects. In some situations, you might want to create a Studio Spline feature, instead, because this feature will appear in the Part Navigator and some forms of editing are easier. You proceed in the standard way, by first creating a StudioSplineBuilderEx object, and then setting its properties. Many of the properties take the form of geometric constraints that control the shape of the curve. For example, you can specify points that the spline should pass through, tangent directions, curvatures, and so on. To make the coding more convenient, let's first write a small "helper" function that provides an easy way to add a "point" constraint to a StudioSplineBuilderEx object: Private Shared Sub AddPoint(builder As Features.StudioSplineBuilderEx, coords As NXOpen.Point3d) Dim workPart As NXOpen.Part = NXOpen.Session.GetSession.Parts.Work Dim point As NXOpen.Point = workPart.Points.CreatePoint(coords) Dim geomCon As NXOpen.Features.GeometricConstraintData geomCon = builder.ConstraintManager.CreateGeometricConstraintData geomCon.Point = point builder.ConstraintManager.Append(geomCon) End Sub Using this helper function, here's how we construct a Studio Spline feature: ' Create the builder Dim builder As NXOpen.Features.StudioSplineBuilderEx builder = workPart.Features.CreateStudioSplineBuilderEx(Nothing) ' Set a few properties builder.Type = NXOpen.Features.StudioSplineBuilderEx.Types.ByPoles builder.IsSingleSegment = True builder.IsAssociative = True ' Add some Dim pt1 As Dim pt2 As Dim pt3 As point constraints New NXOpen.Point3d(0, 7, 1) New NXOpen.Point3d(2, 7, 1) New NXOpen.Point3d(5, 9, 0) : : : AddPoint(builder, pt1) AddPoint(builder, pt2) AddPoint(builder, pt3) ' Create the Studio Spline Feature Dim splineFeature As NXOpen.Features.StudioSpline = builder.Commit ' Get the spline curve (if necessary) Dim spline As NXOpen.Spline = builder.Curve builder.Destroy Notice that we have set IsAssociative = True. If we had set this property to False, instead, then splineFeature would be Nothing. However, an NXOpen.Spline curve would still be created, which we could then use in subsequent operations. Unrestricted Getting Started with NX Open Chapter 7: Curves Page 61 ■ Sketches A sketch is a collection of curves that are controlled by a system of geometric and dimensional constraints. The system of constraints is solved to give the sketch curves the desired size and shape. As an example, we will create a sketch that forms a "bridge" between two points. We will construct a circular arc, and constrain it to have a given arclength and to pass through the two given points p0 = (2,0,0) and p1 = (0,0,0). p1 p0 We begin by creating a datum plane and a datum axis to control the orientation of our sketch: Dim Dim Dim Dim origin As New NXOpen.Point3d(0,0,0) axisX As New NXOpen.Point3d(1,0,0) wcsMatrix As NXOpen.Matrix3x3 = workPart.WCS.CoordinateSystem.Orientation.Element sketchPlane As NXOpen.DatumPlane = workPart.Datums.CreateFixedDatumPlane(origin, wcsMatrix) horizAxis As NXOpen.DatumAxis = workPart.Datums.CreateFixedDatumAxis(origin, axisX) Next, we create an empty sketch (that does not yet contain any curves), using the familiar builder technique: Dim sketchBuilder As NXOpen.SketchInPlaceBuilder sketchBuilder = workPart.Sketches.CreateNewSketchInPlaceBuilder(Nothing) sketchBuilder.PlaneOrFace.Value = sketchPlane sketchBuilder.Axis.Value = horizAxis sketchBuilder.SketchOrigin = workPart.Points.CreatePoint(origin) sketchBuilder.PlaneOption = NXOpen.Sketch.PlaneOption.Inferred Dim bridgeSketch As NXOpen.Sketch = sketchBuilder.Commit sketchBuilder.Destroy bridgeSketch.Activate(NXOpen.Sketch.ViewReorient.False) The last line of code activates the sketch, allowing us to add curves and constraints to it. Next, we create an arc through three points, and add it to our sketch: Dim Dim Dim Dim p0 As New NXOpen.Point3d(2,0,0) ' Start point p1 As New NXOpen.Point3d(0,0,0) ' End point pm As New NXOpen.Point3d(1,1,0) ' Interior point gotFlipped As Boolean = False bridge As NXOpen.Arc = workPart.Curves.CreateArc(p0, pm, p1, False, gotFlipped) theSession.ActiveSketch.AddGeometry(bridge, NXOpen.Sketch.InferConstraintsOption.InferNoConstraints) In this construction, the middle point pm is somewhat arbitrary; after solving, the arc will no longer pass through this point. Unrestricted Getting Started with NX Open Chapter 7: Curves Page 62 The next step is to create some constraints that make the start and end points of the arc coincident with the two given points p0 and p1. The code for constraining the start point is as follows: Dim arcPt0 As New NXOpen.Sketch.ConstraintGeometry arcPt0.Geometry = bridge arcPt0.PointType = NXOpen.Sketch.ConstraintPointType.StartVertex arcPt0.SplineDefiningPointIndex = 0 Dim pt0 As New NXOpen.Sketch.ConstraintGeometry pt0.Geometry = workPart.Points.CreatePoint(p0) pt0.PointType = NXOpen.Sketch.ConstraintPointType.None pt0.SplineDefiningPointIndex = 0 theSession.ActiveSketch.CreateCoincidentConstraint(arcPt0, pt0) As you can see, we do not use the point p0 and the arc end-point directly — we first construct ConstraintGeometry objects that are then used as input to the CreateCoincidentConstraint function. The code for constraining the arc's end-point is analogous: Dim arcPt1 As New NXOpen.Sketch.ConstraintGeometry arcPt1.Geometry = bridge arcPt1.PointType = NXOpen.Sketch.ConstraintPointType.EndVertex arcPt1.SplineDefiningPointIndex = 0 Dim pt1 As New NXOpen.Sketch.ConstraintGeometry pt1.Geometry = workPart.Points.CreatePoint(p1) pt1.PointType = NXOpen.Sketch.ConstraintPointType.None pt1.SplineDefiningPointIndex = 0 theSession.ActiveSketch.CreateCoincidentConstraint(arcPt1, pt1) The "coincidence" constraint we have used here is the most common type, but the Sketch class provides functions for creating many other types. For example, parallel, perpendicular and concentric constraints are supported, as in interactive NX. Next, we create a "perimeter" dimension to control the length of the arc: Dim length As NXOpen.Expression = workPart.Expressions.CreateExpression("Number", "length = 2.5") Dim perimeter As NXOpen.Curve() = {bridge} theSession.ActiveSketch.CreatePerimeterDimension(perimeter, origin, length) Typically, the perimeter of a sketch will consist of an array of curves, of course, but here we have only one. Again, the Sketch class provides functions for creating various other types of dimensional constraints (linear, angular, diameter, and so on). Finally, we "update" the sketch, and deactivate it. theSession.ActiveSketch.LocalUpdate theSession.ActiveSketch.Deactivate(NXOpen.Sketch.ViewReorient.False, NXOpen.Sketch.UpdateLevel.Model) When we call the LocalUpdate function, the sketch is solved, but the children of the sketch (if any) are not updated. After executing the code, the user can adjust the value of the "length" expression to modify the shape of the curve. The picture below shows some sample curves with lengths 2.3, 2.4, and 2.5: Unrestricted Getting Started with NX Open Chapter 7: Curves Page 63 Chapter 8: Simple Solids and Sheets This chapter briefly outlines a few of the NX Open functions that are available for creating simple solid and sheet bodies. Typically, these functions create features, so you sometimes have to do a bit of extra work to get the constituent bodies, as explained later in chapter 10. ■ Creating Primitive Solids The NXOpen.Features class provides a variety of functions for creating simple solid primitive features (blocks, cylinders, cones, spheres, etc.). As an example, let's consider the following code that builds a sphere feature: Dim builder As NXOpen.Features.SphereBuilder builder = workPart.Features.CreateSphereBuilder(Nothing) ' Specify the sphere definition type (center and diameter) builder.Type = Features.SphereBuilder.Types.CenterPointAndDiameter ' Define the sphere center Dim center As New NXOpen.Point3d(3,5,6) Dim centerPoint As NXOpen.Point = workPart.Points.CreatePoint(center) builder.CenterPoint = centerPoint ' Define the sphere diameter Dim diamString As String = "1.5" builder.Diameter.RightHandSide = diamString ' Define the boolean option (create, unite, etc.) builder.BooleanOption.Type = NXOpen.GeometricUtilities.BooleanOperation.BooleanType.Create ' Commit to create the feature Dim sphereObject As NXOpen.Features.Sphere = builder.CommitFeature ' Destroy the builder to free memory builder.Destroy So, we see that the creation process follows the "builder" pattern that we explained in chapter 5. The general approach is to Create a "builder" object Modify its properties as desired "Commit" the builder to create a new feature Functions to create various types of "builder" objects are methods of a FeatureCollection object, and we can get one of these from the workPart.Features property. You can create Block, Cylinder and Cone features using similar techniques. As you would expect, the relevant builder objects are BlockFeatureBuilder, CylinderBuilder, and ConeBuilder. Of these, the ConeBuilder object is the most complex, because it has several different values for its "Type" property, and several dimensional parameters (diameters, height, angle) that are interdependent. The set of relevant parameters depends on the value of the Type property. For example, if you set Type = DiametersAndHeight, then the only relevant parameters are BaseDiameter, TopDiameter, and Height. You can assign a value to the HalfAngle parameter, too, but this setting will simply be ignored, as the following code illustrates: Unrestricted Getting Started with NX Open Chapter 8: Simple Solids & Sheets Page 64 Dim builder = workPart.Features.CreateConeBuilder(Nothing) ' Specify the cone definition type (diameters and height) builder.Type = NXOpen.Features.ConeBuilder.Types.DiametersAndHeight ' Define the diameters and height (these settings are relevant) builder.BaseDiameter.RightHandSide = "3" builder.TopDiameter.RightHandSide = "1.0" builder.Height.RightHandSide = "4" ' Try to define HalfAngle (no error; this is just ignored) builder.HalfAngle.RightHandSide = "1" It's usually fairly obvious which parameters are used with each setting of the Type property. If you're in doubt, you can experiment with the Cone dialog in interactive NX. As you change the Type setting, the relevant set of parameters will be shown in the lower portion of the dialog. The examples in this document often use spheres and cylinders to illustrate some point, so we provide simple Guide.CreateSphere and Guide.CreateCylnder functions to make these easy to create. ■ Sections Before we discuss Extruded and Revolved features, we need to explain the concept of a "section". When you are selecting curves for use in Extrude, or Revolve, or many other NX functions, a menu appears in the top border bar showing you the available "Selection Intent" rules. This menu allows you to define a collection of curves that is dynamic in the sense that its members are determined on-the-fly based on the rule you specify. So, for example, if you select a face F and choose the "Face Edges" rule, your collection will contain all the edges of the face F. If the face F happens to change, as the result of model editing, then your collection will still consist of all the edges of F, whatever these might now be. The collection of curves is "smart" in the sense that it responds to changes in the model; in fact, as we will see, a collection defined in this way is sometimes referred to as a "Smart Collector". In NX Open, there is a corresponding SelectionIntentRule class, which has numerous derived classes, including CurveDumbRule CurveChainRule CurveFeatureChainRule CurveFeatureTangentRule CurveGroupRule CurveTangentRule The simplest type of these is the CurveDumbRule, which just collects a specific list of curves, as its name suggests. In an NX Open program, this is often appropriate, since the collection logic will reside in your code, rather than in NX data structures. To create a selection intent rule of type CurveDumbRule from a given array of curves, the code is just: Dim dumbrule As CurveDumbRule = workPart.ScRuleFactory.CreateRuleCurveDumb(curveArray) Unrestricted Getting Started with NX Open Chapter 8: Simple Solids & Sheets Page 65 The "Sc" in ScRuleFactory stands for "Smart Collector". Then, once we have this rule, we can use it to add curves to a section. So, if we have a single curve named arc, the code to create a section is: ' Create a selection intent rule specifying a single arc Dim dumbRule As NXOpen.CurveDumbRule = workPart.ScRuleFactory.CreateRuleBaseCurveDumb({arc}) Dim rules As NXOpen.SelectionIntentRule() = {dumbRule} ' Create a section Dim mySection As NXOpen.Section = workPart.Sections.CreateSection(0.0095, 0.01, 0.5) Dim help As New NXOpen.Point3d(0,0,0) Dim nullObj As NXOpen.NXObject = Nothing ' Use the rule to add the arc to the section Dim noChain As Boolean = False mySection.AddToSection(rules, arc, nullObj, nullObj, help, NXOpen.Section.Mode.Create, noChain) If we want a rectangular section consisting of four lines, then we add these one at a time, as follows: ' Create Dim c1 = Dim c2 = Dim c3 = Dim c4 = four lines workPart.Curves.CreateLine(New workPart.Curves.CreateLine(New workPart.Curves.CreateLine(New workPart.Curves.CreateLine(New Dim ctol = 0.0095 Dim dtol = 0.01 Dim atol = 0.5 Point3d(1,0,0), Point3d(3,0,0), Point3d(3,1,0), Point3d(1,1,0), New New New New Point3d(3,0,0)) Point3d(3,1,0)) Point3d(1,1,0)) Point3d(1,0,0)) ' Chaining tolerance ' Distance tolerance ' Angle tolerance ' Create a rectangular section Dim rect As NXOpen.Section = workPart.Sections.CreateSection(ctol, dtol, atol) Dim Dim Dim Dim helpPoint As New NXOpen.Point3d(0,0,0) nullObj As NXOpen.NXObject = Nothing noChain As Boolean = False createMode As NXOpen.Section.Mode = Section.Mode.Create ' Create rules to add the four Dim r1 As NXOpen.CurveDumbRule Dim r2 As NXOpen.CurveDumbRule Dim r3 As NXOpen.CurveDumbRule Dim r4 As NXOpen.CurveDumbRule rect.AddToSection({r1}, rect.AddToSection({r2}, rect.AddToSection({r3}, rect.AddToSection({r4}, c1, c2, c3, c4, lines to the section = workPart.ScRuleFactory.CreateRuleBaseCurveDumb({c1}) = workPart.ScRuleFactory.CreateRuleBaseCurveDumb({c2}) = workPart.ScRuleFactory.CreateRuleBaseCurveDumb({c3}) = workPart.ScRuleFactory.CreateRuleBaseCurveDumb({c4}) nullObj, nullObj, nullObj, nullObj, nullObj, nullObj, nullObj, nullObj, helpPoint, helpPoint, helpPoint, helpPoint, createMode, createMode, createMode, createMode, noChain) noChain) noChain) noChain) Using other types of rules is quite similar. For example, if we want a section that gathers together all the edges of a face myFace, we write: Dim faceRule As NXOpen.EdgeBoundaryRule = workPart.ScRuleFactory.CreateRuleEdgeBoundary({myFace}) mySection.AddToSection({faceRule}, myFace, nullObj, nullObj, help, NXOpen.Section.Mode.Create, False) We can use the section to create an Extrude feature, a Revolve feature, or numerous other types. Unrestricted Getting Started with NX Open Chapter 8: Simple Solids & Sheets Page 66 ■ Extruded Bodies Once we have created a section, creating an Extrude feature is quite straightforward. So, suppose we have created a section called mySection, as in the code above. To extrude this section in the z-direction we write: ' Create an ExtrudeBuilder Dim builder = workPart.Features.CreateExtrudeBuilder(Nothing) ' Define the section for the Extrude builder.Section = mySection ' Define the direction for the Extrude Dim origin As New NXOpen.Point3d(0,0,0) Dim axisZ As New NXOpen.Vector3d(0,0,1) Dim updateOption = SmartObject.UpdateOption.DontUpdate builder.Direction = workPart.Directions.CreateDirection(origin, axisZ, updateOption) ' Define the extents of the Extrude builder.Limits.StartExtend.Value.RightHandSide = "-0.25" builder.Limits.EndExtend.Value.RightHandSide = "0.5" Dim extrudeFeature As NXOpen.Features.Extrude = builder.CommitFeature builder.Destroy If your section consists of an open string of curves that do not enclose a region, then the result will be a sheet body, of course. On the other hand, when you extrude a closed section, you can decide whether you want a sheet body or a solid body as the result. The draft angle(s) of the extruded body can be controlled by using the extrudeBuilder.Draft property, and thin-walled extrusions can be created using the extrudeBuilder.Offset property. So, to create a sheet body with a 15 degree draft angle, we write: builder.FeatureOptions.BodyType = GeometricUtilities.FeatureOptions.BodyStyle.Sheet builder.Draft.DraftOption = GeometricUtilities.SimpleDraft.SimpleDraftType.SimpleFromProfile builder.Draft.FrontDraftAngle.RightHandSide = "15" Using the rectangular section named rect that we defined above, the result is: Another extrude example can be found in [NX]\UGOPEN\SampleNXOpenApplications\.NET\QuickExtrude. Unrestricted Getting Started with NX Open Chapter 8: Simple Solids & Sheets Page 67 ■ Revolved Bodies Creating Revolved features is quite similar to creating Extruded ones. Again, most of the work is in the creation of the section that we revolve. So, suppose we have already created the rect section as shown above. To revolve this section around the y-axis, the code is: ' Create the Revolve builder Dim builder = workPart.Features.CreateRevolveBuilder(Nothing) ' Define the section for the Revolve (see above for details) builder.Section = rect ' Define the axis to revolve around (the y-axis of the Absolute Coordinate System) Dim axisPoint3d As New NXOpen.Point3d(0, 0, 0) Dim axisVector As New NXOpen.Vector3d(0, 1, 0) Dim updateOption = SmartObject.UpdateOption.WithinModeling ' Define a direction to pass the revolve point and axis to the builder Dim direction = workPart.Directions.CreateDirection(axisPoint3d, axisVector, updateOption) Dim axisPoint As NXOpen.Point = workPart.Points.CreatePoint(axisPoint3d) builder.Axis = workPart.Axes.CreateAxis(axisPoint, direction, updateOption) ' Define the extents of the Revolve (in degrees) builder.Limits.StartExtend.Value.RightHandSide = "0" builder.Limits.EndExtend.Value.RightHandSide = "90" ' Commit and then destroy the builder Dim revolveFeature As NXOpen.Features.Revolve = builder.CommitFeature builder.Destroy This produces the result shown below Unrestricted Getting Started with NX Open Chapter 8: Simple Solids & Sheets Page 68 Chapter 9: Object Properties & Methods The objects in the NXOpen namespace have a rich set of properties that let us get information about the objects and (in some cases) modify them. The complete properties of each object are documented in the NX Open Reference Guide, so the overview provided here is just to help you understand the basic concepts. As we mentioned in chapter 4, objects inherit properties from the parent classes from which they are derived, in addition to having properties of their own. So, since NXOpen.Arc inherits from NXOpen.Conic, which in turn inherits from NXOpen.Curve, an NXOpen.Arc object has all the properties of an NXOpen.Conic object and all the properties of an NXOpen.Curve object, in addition to specific properties of its own. In the NX Open Reference Guide, you can control whether or not inherited members are displayed by clicking in the check-box circled in red below: As you can see, there are two members that NXOpen.Arc inherits from NXOpen.DisplayableObject, one that it inherits from NXOpen.Conic, and one that it inherits from NXOpen.DisplayableObject. All four of these will be hidden if you uncheck the "inherited" box. ■ NXObject Properties Most objects in the NX Open object hierarchy inherit from NXOpen.NXObject, so its properties are very important because they trickle down to all the lower-level objects. The properties can be divided into several categories, as outlined below: Type and Subtype Properties Each NX Open object has a Type property and a Subtype property, which you will often use to make decisions about how to process the object. Some objects such as solid geometry objects have an additional SolidBodyType property. These properties are read-only, of course — you cannot change the type of an object. Property Query Method Description Type NXOpen.UF.UFObj.AskTypeAndSubtype The object's type Subtype NXOpen.UF.UFObj.AskTypeAndSubtype The object's subtype SolidBodyType Unrestricted Getting Started with NX Open Optional detail type used by certain types of objects such as solid geometry objects. Chapter 9: Object Properties & Methods Page 69 Suppose the user has selected an object, for example. You might want to test whether this object is an ellipse before processing it. The code to do this would be as follows: ' Get the UFSession Dim ufs As NXOpen.UF.UFSession = NXOpen.UF.UFSession.GetUFSession Dim thing As NX.NXObject = ... Dim myType As Integer Dim mySubType As Integer ufs.Obj.AskTypeAndSubtype(thing.Tag, myType, mySubType) If myType = NXOpen.UF.UFConstants.UF_conic_type And mySubType = NXOpen.UF.UFConstants.UF_conic_ellipse_subtype Then 'Do something End If You can reduce the typing by putting Imports NXOpen.UF.UFConstants at the top of your file. In some cases, it might be more convenient to test the type of an object using the standard Visual Basic TypeOf function. For example, the code above could be written as: Dim thing As NX.NXObject = ... If TypeOf thing Is NXOpen.Ellipse 'Do something End If Display Properties Many of the objects that NX users deal with are of type NXOpen.DisplayableObject (a subtype derived from NXOpen.NXObject). These objects have the following properties: Data Type Property Access Description Integer Layer get, set The layer on which the object resides Boolean IsBlanked get If true, indicates that the object is blanked (hidden) Integer Color get, set The color of the object as an index to the NX color palette ObjectFont LineFont get, set The line font used to draw the object (solid, dashed, etc.) ObjectWidth LineWidth get, set The line width used to draw the object Point3d NameLocation get The location of the object's name Note that the Color attribute is a color index into the color palette for the part. The NXOpen.UF.UFDisp class contains several functions for working with NX color indices. For example, NXOpen.UF.UFDisp.AskColor gets the RGB values associated with a given color index, and NXOpen.UF.UFDisp.AskClosestColor does the reverse. Attribute Properties For technical reasons, attributes cannot be implemented as "real" properties, so they are accessed via old-fashioned "Get" and "Set" methods on the NXOpen.NXObject class. All NX objects that can contain attributes Unrestricted Getting Started with NX Open Chapter 9: Object Properties & Methods Page 70 inherit from NXObject. A few of the available methods are listed below, and the complete set is covered in the documentation for the NXOpen.NXObject class in the NX Open Reference Guide: NXOpen.NXObject Method Description DeleteUserAttributes(Type As AttributeType, Title As String, DeleteArray As Boolean, Option As UpdateOption) Deletes the first attribute encountered with the given Type and Title. Can perform an update if desired. If attribute is an attribute array, can optionally delete the entire array. DeleteUserAttributes(Type As AttributeType, Option As Update.Option) Deletes the attributes encountered with the given Type. Can perform an update if desired. GetXxxUserAttribute(Title As String, Index As Integer) Gets the value of an attribute with the given Title and array Index (if the attribute is an array attribute). Xxx can be Boolean, Integer, Real, String, or Time. GetUserAttribute(Title As String, Index As Integer) Gets an AttributeInformation structure of the first attribute encountered on the object with a given Title and array Index (if attribute is an array attribute). GetUserAttributes() Gets an array of AttributeInformation structures of all the attributes that have been set on the object. HasUserAttribute(Title As String, Type As AttributeType, Index As Integer) Checks if the object has an attribute with the given Title, Type, and Index. Name The name of the object (aka "custom name", sometimes) SetXxxUserAttribute(Title As String, Index As Integer, Value As Xxx, Option As Update.Option) Creates and/or sets the value of an attribute of type Xxx, where Xxx can be Boolean or Time. The attribute is identified by the string Title and array Index (if attribute is an array attribute). Can perform an update if desired. SetUserAttribute(Title As String, Integer, Value As Double, Option As Update.Option) Creates or modifies a real attribute. Arrays can be extended only one element at a time. Can perform an update if desired. SetUserAttribute(Title As String, Index As Integer, Value As Integer, Option As Update.Option) Creates or modifies an integer attribute. Arrays can be extended only one element at a time. Can perform an update if desired. SetUserAttribute(Title As String, Integer, Value As String, Option As Update.Option) Creates or modifies a string attribute. Arrays can be extended only one element at a time. Can perform an update if desired. ■ Curve and Edge Properties In this section, we describe techniques for getting information about curves and edges. Specifically, we discuss how we can obtain position and tangent information, shape parameters like radius, and topological properties. Evaluators Some of the most useful methods when working with curves or edges are the so-called "evaluator" functions. At a given location on a curve (defined by a parameter value t), we can ask for a variety of different values, such as the position of the point, or the tangent or curvature of the curve. The evaluator functions are provided by the NXOpen.UF.UFEval class. The most important functions are Evaluate and EvaluateUnitVectors Function Values calculated Evaluate Position and derivatives at given parameter value EvaluateUnitVectors Position, tangent, normal, binormal at given parameter value The following code uses the Evaluate function to compute a position and tangent at a location along myCurve, which is assumed to be of type NXOpen.Curve or NXOpen.Edge Unrestricted Getting Started with NX Open Chapter 9: Object Properties & Methods Page 71 ' Get the UFSession Dim ufs As NXOpen.UF.UFSession = NXOpen.UF.UFSession.GetUFSession ' Get the tag of our curve Dim curveTag As NXOpen.Tag = myCurve.Tag ' Create an evaluation structure Dim estruct As System.IntPtr ufs.Eval.Initialize2(curveTag, eStruct) ' Compute point and first derivative at t = 0.5 Dim t As Double = 0.5 Dim numDerivs = 1 Dim coords As Double() = { 0, 0, 0 } Dim derivs As Double() = { 0, 0, 0 } ufs.Eval.Evaluate(eStruct, numDerivs, t, coords, derivs) Dim curvePoint As New NXOpen.Point3d(coords(0), coords(1), coords(2)) Dim curveTangent As New NXOpen.Vector3d(derivs(0), derivs(1), derivs(2)) ' Free the evaluation structure ufs.Eval.Free(eStruct) In other software systems, a common approach is to "normalize" the parameter value (t) that is passed to these sorts of evaluator functions, so that it lies in the range $0 \le t \le 1$. With this approach, the parameter value t = 0.5 used in the code above would correspond to the parametric mid-point of the curve. In NX Open, this normalization process is not used, so the parameter values used are the original "native" parameters of the curve. So, in the example above, if myCurve was a circular arc, the parameter value t = 0.5 would be the point at an angle of 0.5 radians. If you want to use normalized parameter values, you can construct these yourself. The following code shows you how to compute a point that is 25% of the way along a given curve or edge denoted by myCurve: ' Get the parameter limits of the curve Dim limits As Double() ufs.Eval.AskLimits(estruct, limits) ' Normalized parameter value is u = 0.25 Dim u As Double = 0.25 ' Compute non-normalized parameter value, t Dim t = (1-u)*limits(0) + u*limits(1) ' Compute point at t value Dim coords As Double() = { 0, 0, 0 } Dim derivs As Double() = { 0, 0, 0 } ufs.Eval.Evaluate(eStruct, 0, t, coords, derivs) Dim curvePoint As New NXOpen.Point3d(coords(0), coords(1), coords(2)) As we have seen above, the evaluator functions use an "evaluation structure" that is returned by an Initialize2 function, rather than directly using the curve or edge itself. Then, after you have finished using this structure, you should call the Free function to release the memory it has been using. In between the Initialize and Free steps, you can use an evaluation structure as many times as you like. The code below shows a common technique for creating a sequence of points along a curve; as you can see, we initialize the evaluation structure once, use it 101 times, and then free it. Unrestricted Getting Started with NX Open Chapter 9: Object Properties & Methods Page 72 The example uses a spline curve, so we can safely assume that the parameter limits are 0 and 1: ' Create an evaluation structure for the spline Dim estruct As System.IntPtr ufs.Eval.Initialize2(splineTag, estruct) ' Prepare for Dim numDerivs Dim coords As Dim derivs As stepping along the spline = 1 Double() = {0, 0, 0} Double() = {0, 0, 0} 'Step along the spline, creating 101 points For t As Double = 0 To 1 Step 0.01 ufs.Eval.Evaluate(estruct, numDerivs, t, coords, derivs) Dim curvePosition As New NXOpen.Point3d(coords(0), coords(1), coords(2)) workPart.Points.CreatePoint(curvePosition) Next ' Free the evaluation structure ufs.Eval.Free(estruct) There is another function NXOpen.UF.Modl.EvaluateCurve that also allows you to calculate a point at a given parameter value. It is slightly simpler to use, but it only works with curves, not with edges. Edge Topology Properties The main difference between an edge and a curve, of course, is that an edge is part of a body, whereas a curve is not. Because of this, an edge has "topological" properties that a curve does not have, which describe how the edge is connected to other items (faces, edges, vertices) within the body. Basic topology enquiries are quite simple: if myEdge is of type NXOpen.Edge, then you can use the myEdge.GetFaces function to find out which faces it belongs to, and the myEdge.GetBody function to find out which body it belongs to. Edge Geometry Properties The NXOpen.UF.EFEval class has functions that return geometric properties of various types of edges. Specifically, there are functions named AskLine, AskArc, AskSpline, and so on. For example, the following code gets the center and radius of a circular edge, myArcEdge ' Get the tag of our edge Dim edgeTag As NXOpen.Tag = myArcEdge.Tag ' Get an evaluation structure Dim estruct As System.IntPtr ufs.Eval.Initialize2(edgeTag, estruct) ' Get an arc evaluation structure Dim evalArc As NXOpen.UF.UFEval.Arc ufs.Eval.AskArc(estruct, evalArc) ' Get the edge's center and radius Dim centerCoords As Double() = evalArc.center Dim radius As Double = evalArc.radius ' Free the evaluation structure ufs.Eval.Free(estruct) Unrestricted Getting Started with NX Open Chapter 9: Object Properties & Methods Page 73 ■ Face Properties Like edges, faces have evaluator functions, topological properties, and geometric properties. Evaluators As with curves, we can call an "evaluator" function to calculate certain values at a given point on a surface (or a face). So, as you might expect, we can get the location of the point, the surface normal at the point, and so on. To indicate which point we're interested in, we have to give two parameter values, traditionally denoted by u and v. The following code illustrates the approach: Get the UFSession Dim ufs As NXOpen.UF.UFSession = NXOpen.UF.UFSession.GetUFSession ' Get the tag of our face Dim faceTag As NXOpen.Tag = myFace.Tag ' Get the uv mid-point of the face Dim minU, maxU, minV, maxV, box(3), uv(1) As Double ufs.Modl.AskFaceUvMinmax(faceTag, box) minU = box(0) : maxU = box(1) minV = box(2) : maxV = box(3) uv = { (minU + maxU)/2, (minV + maxV)/2 } ' Create a structure to hold the evaluation results Dim faceValues As New NXOpen.UF.ModlSrfValue ' Ask for position, first derivatives, and unit normal Dim request As Integer = NXOpen.UF.UFConstants.UF_MODL_EVAL_UNIT_NORMAL ' Evaluate at uv mid-point ufs.Modl.EvaluateFace(faceTag, request, uv, faceValues) ' Extract position and unit normal at point on face Dim facePosition As Double() = faceValues.srf_pos Dim faceNormal As Double() = faceValues.srf_unormal As you can see, the first step is to get the parametric mid-point of the face. Of course, if we wanted to evaluate at some other point of the face, this step would not be necessary. By setting request = UF_MODL_EVAL_UNIT_NORMAL, we have asked for calculation of a position, first partial derivatives, and a unit surface normal, so these are available in the faceValues structure that is returned. Various other request constants are provided in the UFConstants class; the most comprehensive of these is UF_MODL_EVAL_ALL, which allows you to calculate position, surface normal, and all the partial derivatives up to the third order. There is a related function, NXOpen.UF.UFModl.AskFaceProps, that provides additional information about curvature. Despite its name, the EvaluateFace function we used above is actually doing computations on a surface, rather than a face. So, when performing evaluations, you do not need to restrict yourself to uv values that correspond to locations inside the given face. Even the uv mid-point we used above is not guaranteed to lie within the face, because the face might have some hole or notch that excludes it. Face Topology Properties Like an edge, a face has "topological" properties that describe its relationship to other objects in its body. If myFace is an NXOpen.Face object, then myFace.GetBody gives you the body that the face lies on, and myFace.GetEdges returns its array of edges.. Unrestricted Getting Started with NX Open Chapter 9: Object Properties & Methods Page 74 Face Geometry Properties To get information about the geometry of a face, you use the NXOpen.UF.UFModl.AskFaceData function. For example, the following code gets information about a face myFace: Dim ufs As NXOpen.UF.UFSession = NXOpen.UF.UFSession.GetUFSession Dim Dim Dim Dim Dim Dim axisPoint(2) As Double axisVector(2) As Double box(5) As Double surfType As Integer r1, r2 As Double flip As Integer ' ' ' ' ' ' Point on cylinder axis Direction vector of cylinder axis 3D bounding box of face Surface type (see below) Radii of face (see below) Normal flip indicator ufs.Modl.AskFaceData(myFace.Tag, surfType, axisPoint, axisVector, box, r1, r2, flip) The box argument provides a bounding box for the face, with axes aligned with the Absolute Coordinate System. The box is represented by 6 numbers in the order minX, minY, minZ, maxX, maxY, maxZ. The flip argument is equal to ±1, and indicates on which side of the surface material lies. Specifically, if u and v are the first partial derivatives of the surface, then the vector $flip * (u \times v)$ points away from material, into "air". The meanings of the other parameters, for various different surface types, are given in the following table: Face Type Type axisPoint axisVector r1 r2 Cylinder 16 Point on axis Axis vector Radius --- Cone 17 Point on axis Axis vector Radius at axis point Half-angle (in radians) Sphere 18 Center --- Radius --- Revolved 19 Point on axis Axis vector --- --- Torus 19 Center Axis vector Major radius Minor radius Extruded 20 --- --- --- --- Plane 22 Point on plane Normal --- --- Blend 23 --- --- Radius --- B-surface 43 --- --- --- --- Offset 65 --- --- --- --- There is another function NXOpen.UF.UFModl.AskBsurf that provides detailed information about b-surfaces. Unrestricted Getting Started with NX Open Chapter 9: Object Properties & Methods Page 75 Chapter 10: Feature Concepts The NXOpen.Features class contains a wide variety of functions for creating "features". At one extreme, features can be very simple objects like blocks or spheres; at the other extreme, features like ThroughCurveMesh can be very complex. In this chapter, we explain what a feature is, and give some samples of the NX Open functions that create them. As usual, the full details can be found in the NX Open Reference Guide. ■ What is a Feature ? Though you have probably created hundreds of features while running NX interactively, perhaps you never stopped to think what a "feature" really is. So, here is the definition ... A feature is a collection of objects created by a modeling operation that remembers the inputs and the procedure used to create it. The inputs used to create the feature are called its "parents", and the new feature is said to be the "child" of these parents. This human family analogy can be extended in a natural way to provide a wealth of useful terminology. We can speak of the grandchildren or the ancestors or the descendants of an object, for example, with the obvious meanings. An object that has no parents (or has been disconnected from them) is said to be an "orphan", or sometimes a "dumb" object, or an "unparameterized" one. The inputs and the procedure are also known as the "history" of the object, or the "recipe", or the "parameters". There is no shortage of terminology in this area. The great power of features is that they capture the process (i.e. the history, or recipe) used to create an object. You can change the inputs, and then re-execute this process, which gives you some remarkable editing capabilities. You can also re-order features, delete them, or insert new ones in the middle of the "recipe", which again provides very powerful editing techniques. ■ Types of Features There are many different types of features, plus two important subclasses: Body Features and Curve Features. Some of the more common examples are listed in the table below: Features Body Features Curve Features AssociativeArc AssociativeLine CompositeCurve CurveOnSurface DatumCsys DatumFeature Helix JoinCurves PointFeature StudioSpline Measure Extract Block BoundedPlane Brep Cylinder EdgeBlend ExtractFace Extrude FaceBlend Revolve Scale ThroughCurves Tube IntersectionCurve IsoparametricCurve OffsetCurve PointSet ProjectCurve VirtualCurve Important properties and methods for the NXOpen.Features.Feature class are: Member Description FeatureType Returns the feature type as a String (see below) GetEntities Returns the entities created by the feature GetExpressions Returns the expressions created by the feature GetFeatureName Returns the displayed name of the feature. Unrestricted Getting Started with NX Open Chapter 10: Feature Concepts Page 76 GetParents Returns the immediate parent features Suppress() Suppresses the feature Suppressed Returns the suppression status of the feature Timestamp Returns the feature's timestamp as an Integer Some of the feature type strings are rather strange and cryptic, especially in older models; you may see things like BREP, CPROJ, FRENET_DATUM_PLANE, SKIN, SWP104, META, and so on. It's usually better to use the standard VB TypeOf or GetType operators to find out the type of a feature. As the name suggests, a BodyFeature is a feature that produces a body or a collection of bodies as its result. Similarly, a CurveFeature typically produces curves (or points). So, these classes have some additional members. For example, a BodyFeature has GetBodies, GetFaces, and GetEdges functions, and a CurveFeature has Color, Font, and Width properties. For a BodyFeature, the GetEntities function typically returns an array of length zero, so you should use the GetBodies function instead. The following code cycles through the work part, writing out some information about each feature it finds: For Each feat In workPart.Features Guide.InfoWriteLine("Type: " & feat.GetType.ToString) Guide.InfoWriteLine("Name: " & feat.GetFeatureName) Guide.InfoWriteLine("Timestamp: " & feat.TimeStamp) Guide.InfoWriteLine("NumParents: " & feat.GetParents.Length) Guide.InfoWriteLine("NumExpressions: " &

feat.GetExpressions.Length) Next ■ Feature Display Properties As NXOpen.Features.Feature is not an NXOpen.DisplayableObject, so its color, hidden/shown property, layer assignment, and other display attributes are not handled in the standard way. You can actually change the color of a feature using an NXOpen.Features.ColorFeatureBuilder, but it's often better to proceed via its first principles, as explained below. The key idea is that a feature typically "owns" some constituent objects (like bodies and curves), which are often known as its "outputs". The output objects do have display properties, so you can use them to (indirectly) modify the display of the feature. You can get these output objects by calling the GetBodies or GetEntities functions mentioned above. A couple of examples should make this more clear. First, suppose you created a Block feature. You cannot directly change the color of this feature in the usual way. However, the Block feature owns a solid body (which you can get by calling GetBodies). You can change the color of this body, and this will effectively change the color of the feature. Another example: suppose you created a Hole feature. You can't change the color of the Hole in the usual way, but you can get its faces and change their colors, instead. In fact, you might want to use different colors to distinguish different faces of a feature. The code below shows a more interesting example in which we combine two bodies: 'Create two circles, and extrude them Dim disk1 As NXOpen.Arc = NXOpen.Guide.CreateCircle(0,0,0, 2) Dim disk2 As NXOpen.Arc = NXOpen.Guide.CreateCircle(1,0,0, 1) Dim ext1 As NXOpen.Features.Extrude = ' Get the two displayable objects of the Extrude feature (two bodies) Dim body1 As NXOpen.Body() = ... ' Change their colors body1(0).Color = 186 body1(0).Color = 211 ' Usually red, by default ... Unrestricted Getting Started with NX Open Chapter 10: Feature Concepts Page 77 ■ Using Features and Bodies As we saw above, when working with display properties, we have to be careful to make the distinction between a feature and its constituent bodies. Next we will see that this distinction is also relevant in modeling. There are many functions that operate on bodies, rather than features, in NX Open. Examples are Boolean operation functions, trimming, splitting, computing mass properties, and so on. Since most of the feature creation functions produce features, the output of these functions will not be immediately usable unless we manage some accommodation. For example, consider the following code: Dim lbm Dim lbm s1 As NXOpen.Features.Sphere = NXOpen.Guide.CreateSphere(0,0,0, 1) s2 As NXOpen.Features.Sphere = NXOpen.Guide.CreateSphere(1,0,0, 2) union As NXOpen.Features.BooleanFeature = NXOpen.Guide.Unite(s1, s2) volume As Double = ' ' Doesn't work ' Doesn't work The Unite function expects two bodies as input, but s1 and s2 are features, so the operation will not work. Similarly, the function that computes volume expects to receive a body, so this won't work, either. We can fix the code, by getting bodies from the features before performing the unite or the volume calculation. So, the corrected version of the code above is: Dim s1 As NXOpen.Features.Sphere = NXOpen.Guide.CreateSphere(0,0,0, 1) Dim s2 As NXOpen.Features.Sphere = NXOpen.Guide.CreateSphere(1,0,0, 2) Dim sb1 As NXOpen.Body = s1.GetBodies()(0) Dim sb2 As NXOpen.Body = s2.GetBodies()(0) Dim union As NXOpen.Features.BooleanFeature = NXOpen.Guide.Unite(sb1, sb2) Dim unionBody As NXOpen.Body = union.GetBodies()(0) Dim volume As Double = ■ Units Parameters of features are typically described by expressions, as discussed below. But expressions involve units, so first we have to understand units. In each part file, there is a UnitCollection, which has an associated collection of "measures". Typical measures are things like length, volume, mass, angle, or velocity. These are also called Dimensionality in the NX docs. Then each measure has an associated collection of units, which are objects of type NXOpen.Unit. Among these units, one particular one is singled out as the Base Unit for that measure. For example, in a metric part, the Base Unit for the measure "Length" will be millimeters; this is the length unit that is actually used for representing objects in the part file. So, this structure is as shown in the picture below: This is actually how you could get units from the UnitCollection of the work part using code like this: ' Get the UnitCollection of the work part Dim unitCollection As NXOpen.UnitCollection = workPart.UnitsCollection ' Get the measures of the part Dim measures As String() = unitCollection.GetMeasures() ' Get the available units for the measure "Length" Dim units As NXOpen.Unit() = unitCollection.GetMeasureTypes("Length") ' Get the base unit for the measure "Length" Dim baseUnit As NXOpen.Unit = unitCollection.GetBase("Length") A UnitCollection will generally contain a large number of measures (80 or more). A few of the less exotic ones, together with their base unit names, are as follows: Unrestricted Getting Started with NX Open Chapter 10: Feature Concepts Page 78 Measure Base Unit Name A Few Other Unit Names Length MilliMeter Meter, Inch, Feet, KiloMeter, Mile, Micron, Angstrom Area SquareMilliMeter SquareMeter, SquareInch, SquareFeet, SquareCentiMeter Volume CubicMilliMeter CubicMeter, CubicInch, CubicFeet, CubicCentiMeter Mass Kilogram Gram, Tonne, Slug, PoundSecondsSquaredPerInch, PoundMass Mass Density KilogramPerCubicMilliMeter GramPerCubicCentiMeter, SlugsPerCubicFeet Time Second Minute, Hour Angle Degrees Radian, Revs Velocity MilliMeterPerSecond MeterPerSecond, FeetPerSecond, MilesPerHour Force MilliNewton Newton, PoundForce, PoundalTemperature Celsius Fahrenheit, Kelvin, Rankine Energy MicroJoule EnergyPoundForceInch, Joule, Btu You can use the unit names to obtain other units (other than the base units) from the FindObject function: ' Get the units of measure "MilliMeter", "Radian", and "KiloGram" Dim mmUnit As NXOpen.Unit = workPart.UnitCollection.FindObject("MilliMeter") Dim radianUnit As NXOpen.Unit = workPart.UnitCollection.FindObject("Radian") Dim kgUnit As NXOpen.Unit = workPart.UnitCollection.FindObject("Kilogram") Note that the names used here are case sensitive — for example, you have to use "MilliMeter", not "Millimeter", and "Kilogram" rather than "kilogram" or "KiloGram". ■ Expressions Expressions are used to control the sizes and positions of features, so it's important for us to know how to work with them. The general form of an expression is name = right-hand side. To understand the details, let's look at some examples that show the most interesting. In this it three important pieces, indicated of the colored boxes: capacity = pi()*depth*(diam/2)^2 Unrestricted Getting Started with NX Open Chapter 10: Feature Concepts // Volume of water Page 79 The overall text string is called the "equation" of the expression (the pink box), the portion to the left of the equals sign is called the "name" (the green box), and the portion to right of the equals sign is called the "right-hand side" (the yellow box). So, in this example: Equation is: "capacity = pi()*depth*(diam/2)^2 // Volume of water" Name is: "capacity" Right-hand side is: "pi()*depth*(diam/2)^2" As you can see, the right-hand side includes a comment that is delineated by two slash characters. The portion of the right-hand side preceding the comment must be a legal formula involving numbers, functions, and names of other expressions. NX builds many expressions behind the scenes as you create features. Expressions created this way will typically look something like this: The second expression has a simple name (just "p0"), which was made up by NX. But there is also some extra text in parentheses following this name. This extra text is called the "description" of the expression, and it consists of a feature name plus a "descriptor" indicating which feature parameter the expression controls. In summary: String Name Meaning p0 Name The name of the expression (made up by NX) Diameter Descriptor Indicates that this expression controls a diameter parameter (Cylinder(2) Diameter) Description Specifically, it controls the diameter of the Cylinder(2) If an expression does not control a feature, then its Description and Descriptor strings will be empty (zero length strings). All of these various elements of an expression can be controlled using NX Open functions, as follows: Function/Property Purpose Description Gets the description of the expression EditComment() Changes the comment. Equation Returns the equation of the expression in the form: name = right_hand_side. GetDescriptor() Returns the descriptor for the expression GetValueUsingUnits() Get the value of the expression, in other base units or the expression's units. Name Gets the name of the expression RightHandSide Returns or sets the right hand side of the expression. SetName() Sets the name of the expression. Type A string indicating the type of expression, which can be Number, String, Boolean, Vector, Point, or List. Units Returns or sets the units for the expression. Value Returns or sets the value of the expression in base units. Unrestricted Getting Started with NX Open Chapter 10: Feature Concepts Page 80 Note that setting the Value property will also change the value of the RightHandSide property. For example, if you set Value = 3.5, then the RightHandSide string will become "3.5". Similarly, setting the RightHandSide property will cause the Value property to change accordingly. The following code cycles through all the expressions in the work part and writes out some of their properties: For Each exp As NXOpen.Expression In workPart.Expressions Dim sep = " " Guide.InfoWrite(exp.Name & sep) Guide.InfoWrite(exp.Equation & sep) Guide.InfoWrite(exp.Description & sep) Guide.InfoWrite(exp.GetDescriptor() & sep) Guide.InfoWriteLine(exp.Value) Next If we execute this code with a work part that contains the following three expressions the results will be: Name Equation Descriptor Value p2 p0=2*radius (Cylinder(2) Diameter) Diameter: 30 p1 p1=10 (Cylinder(2) Height) Height: 10 radius radius=15 //Internal radius 15 The radius expression does not (directly) control a feature, so its Descriptor and Description strings are empty, as we see in the third row. ■ Creating Expressions Functions for creating expressions are provided in the ExpressionCollection class, as follows: Function Creates ... CreateString() An expression of type String CreateWithUnits(String, Unit) An expression with units. CreateSystemExpression(String) A system expression of the specified type. CreateWithUnits(String, Unit) A system expression with units. With creating expressions of a specified type, the type is indicated by a string, which can be one of "Number", "String", "Boolean", "Integer", "Point" or "Vector". The following code shows how these functions are used: Unrestricted Getting Started with NX Open Chapter 10: Feature Concepts Page 81 Dim mmUnit As NXOpen.Unit = workPart.UnitCollection.GetBase("Length") Dim unit As NXOpen.Unit = workPart.UnitCollection.FindObject("MilliMeter") ' Get the ExpressionCollection of the work part Dim exps As NXOpen.ExpressionCollection = workPart.Expressions ' Create three expressions Dim exp1 = exps.Create("x1 = 4") Dim exp2 = exps.CreateExpression("Integer", "n2 = 4") Dim exp3 = exps.CreateWithUnits("s3 = n2 + sqrt(n2)", mmUnit) ' Create two system expressions Dim sysExp4 = exps.CreateSystemExpression("x4 = 7") Dim sysExp5 = exps.CreateSystemExpression("Integer", "n5 = 9") Dim sysExp6 = exps.CreateSystemExpressionWithUnits("x6 = n5 + 2.75", mmUnit) System expressions are less permanent than ordinary (non-system) ones. A system expression will be deleted when the last feature using it is deleted, and it will also be deleted by the Delete Unused Expressions function, which is usually the behavior that's desirable. If you don't specify a unit when creating an expression, the expression will be "constant". Despite the name, this doesn't mean the expression can't be changed; it just means that it represents a dimensionless quantity, like an angle or a parameter percentage on a curve. So, the most useful function of the six mentioned above is probably CreateExpressionWithUnits, and you will use this function many times in conventional code. When writing code that creates expressions, you have to be careful to avoid duplicating names. It's a fairly common error, and if you are not careful you can get an error message telling you that "The specified expression name already exists". To avoid this, delete the previously created expressions before re-executing your code. ■ Using Expressions to Define Features In chapter 8, we saw many examples of code defining the values of feature parameters by expressions, with the result looking something like this: ' Create a CylinderBuilder Dim builder = workPart.Features.CreateCylinderBuilder(Nothing) ' Define the cylinder diameter and height builder.Diameter.RightHandSide = "4.0" builder.Height.RightHandSide = "6.0" This will have created the creation of two expressions that we can use to modify the diameter and height of the cylinder, but this form of editing is rather crude and unsatisfactory. Suppose we wanted a more intelligent cylindrical container that let us specify its depth and volume, and then calculated the required diameter. We could achieve this with the following code: Dim Dim Dim Dim volume = 10 depth = 4 pi = System.Math.PI diameter = 2 * System.Math.Sqrt(volume / (depth*pi)) builder.Height.RightHandSide = depth.ToString builder.Diameter.RightHandSide = diameter.ToString We have now defined the diameter of the cylinder as a function of its depth and volume: volume diameter = 2√ depth * Unrestricted Getting Started with NX Open Chapter 10: Feature Concepts Page 82 So, we could display a small dialog asking the user to enter the desired volume and depth, and the code above would create a cylindrical container with the correct volume. However, this would still produce a relatively "dumb" NX model — if the user subsequently edited the cylinder's height or diameter in interactive NX, the volume would no longer be correct. We have the right "intelligence" in our code, but it did not get transferred into the NX model we built. If we want to build models that support convenient interactive editing, we need to use expressions to build in the desired intelligence. We could achieve this as follows: Dim mmUnit As NXOpen.Unit = workPart.UnitCollection.GetBase("Length") Dim unit As NXOpen.Unit = workPart.UnitCollection.GetBase("Volume") ' Create expressions for volume and depth workPart.Expressions.CreateSystemExpressionWithUnits("volume = 100", mm3Unit) workPart.Expressions.CreateSystemExpressionWithUnits("depth = 4", mmUnit) Define the diameter in terms of volume and depth builder.Diameter.RightHandSide = "2 * sqrt(volume / (depth*pi()))" builder.Height.RightHandSide = "depth" builder.Diameter.RightHandSide = "diameter" The two code examples show two different ways to capture "intelligence": the first one the intelligence is solely in our code, and in the second case it has been captured in expressions. The first approach is simpler, but the code will need to be re-executed if any of the inputs change. In the second approach, the logic in our code has essentially been replicated with NX expressions, which will "replay" automatically if any inputs change. Unrestricted Getting Started with NX Open Chapter 11: Assemblies Page 83 Chapter 11: Assemblies ■ Introduction Unless you're in the brick business, most of your products will probably be assemblies — combinations of simpler lower-level items, rather than just homogeneous chunks of material. This chapter outlines how NX Open functions that you can use to work with them. Most of the discussion is related to reading information about assemblies, rather than creating them, since the most common applications involve extracting information and writing reports of one sort or another. Typically, your code will traverse through the items in an assembly, gathering information (from attributes, usually), and writing this into a report document of some kind. Many of the code examples given below are just fragments, as usual. Complete working code and the part files for a simple car assembly are provided in the folder [...NX]\UGOPEN\NXOpenExamples\More Examples\CarAssembly. Note that some of the code in this chapter will work properly only if the car assembly is fully loaded. ■ The Obligatory Car Example Following the time-honored traditions of assembly modeling, we will use a simple car as an example throughout this chapter (though this won't be done in any more detail). Our car assembly is quite simple — more of a cart, really, than a car. We'll quickly reach its limits. The car axle consists of a shaft and two wheels. The exterior shape is a sheet body in Car_Assembly.prt, so you don't see it, either. ■ Trees, Roots, and Leaves Let's use our car model to explain some terminology. Graphically, its structure looks like this: Car Axle Front Axle Left wheel Rear Axle Right wheel Wheel Spare Wheel Engine Unrestricted Getting Started with NX Open Chapter 11: Assemblies Page 84 This diagram accurately reflects the structure of the data stored in NX. Notice that the wheel part is stored only once, even though the car has five wheels (the four main ones and a spare). However, diagrams like this are difficult to draw, so, in more complex situations, so we will usually draw them as shown below, instead, with items repeated: Car Front Axle Wheel Rear Axle Wheel Spare Wheel Engine The top-level car assembly has four subassemblies: two axles, a spare wheel, and an engine. The axle assembly, in turn, has two subassemblies, namely its left and right wheels. In the diagram, the axle and spare wheel are said to be children of the car assembly. Or looking at it the other way around, the car assembly is said to be the parent of each of those four. This human-family terminology can be extended further: we might say that each of the four main wheels is a grandchild of the car assembly, and all the parts shown are descendants, and so on. Note that this is the reverse of feature terminology. In the feature modeling world, if object-A and object-B are constituents of object-C (in the sense that they are used to create object-C), then they are called the parents of object-C, not its children. This inconsistency is unfortunate, but it's very well established, and is not likely to change, so we have to live with it. In addition to the parent-child terminology, there are some useful terms that we can borrow from computer science. A computer scientist would regard the assembly structure as a tree, and the various parts and assemblies would be called the nodes in the tree. The node at the top of a sub-tree (denoted by the symbol in the diagram) is called the root node of that sub-tree. Nodes at the bottom (like the wheels and engine) are called leaf nodes; these are nodes that have no children. Trees in computer science are strange — their roots are always at the top, and their leaves are at the bottom. In engineering, a leaf node in an assembly tree is sometimes referred to as a piece part. This is a somewhat misleading term because it suggests that the part consists of a single solid body, which is not always true. To avoid any possible misunderstandings, we will use the term "leaf" in this document. We can assign the depth of a node in a tree by counting its ancestors, including parents, grandparents, and so on, up to the root node of the tree. So, in our example, the car itself is at depth zero, the axles are at depth = 1, and the four main wheels are at depth = 2. In NX documentation, nodes with depth = 1 (i.e. immediately below the root node) are sometimes known as "top level" nodes. ■ Components and Prototypes Suppose we have an assembly, and we want to write out a report describing its structure. Each part knows about its child subassemblies, so we could do this by writing code that "walks" from part to part, recording the parent-child relationships. We would start at the top of the tree with the car assembly file. Using the information in this file, we would find out that there are four children, and we could "walk" to each of these four children to get information about grandchildren, and so on. This process would certainly work, but it has a problem — we have to open each part file so we can look inside to get information about its children. Opening hundreds of part files might be very slow (depending on their locations), and we may not even have permission to open some of them, so we need a better way. Unrestricted Getting Started with NX Open Chapter 11: Assemblies Page 85 The NX solution is to store a replica of the assembly tree within the top-level assembly part file. In our example, this looks as shown here: Car_Assembly.prt FRONT_AXLE FRONT_LEFT-WHEEL FRONT_RIGHT-WHEEL REAR_AXLE REAR_LEFT-WHEEL REAR_RIGHT-WHEEL SPARE_WHEEL ENGINE The yellow items are called "Components" or sometimes "Part Occurrences". The tree of components inside a part file replicates the tree structure of the subassemblies themselves. So, if we want to know about this structure, we can simply traverse through the tree of components objects in the file, without opening any other part files. An NX part file that represents an assembly has a ComponentAssembly object that provides most of the functions related to assemblies. The ComponentAssembly object has a RootComponent object, which serves as the root node for the part's tree of components. You can get to all of the other components in the part file by traversing downwards from the RootComponent. The RootComponent will be Nothing if the part file is not an assembly. Each component contains a list of links to its children, a link to its parent, and a link to the corresponding part file, which is called the Prototype of the component. In the diagram below, the parent-child relationships are shown by the blue lines, and the component-to-prototype links are shown as red arrows: Car Axle Front Axle FRONT_AXLE FRONT-LEFT-WHEEL FRONT-RIGHT-WHEEL REAR-AXLE REAR-LEFT-WHEEL REAR-RIGHT-WHEEL SPARE-WHEEL ENGINE Engine Wheel Wheel So, for example, as you can see, the axle part is the prototype corresponding to each of the components FRONT_AXLE and REAR_AXLE. Or, looking at it the other way around, FRONT_AXLE and REAR_AXLE are occurrences of the axle part. Unrestricted Getting Started with NX Open Chapter 11: Assemblies Page 86 As mentioned before, a root component is not a "real" component, so its prototype link has a special meaning — it "loops back" and refers to the part in which the root component resides. This correspondence between components and their associated prototype parts is also displayed in the Assembly Navigator, as shown here: In NX Open, components are represented by NXOpen.Assemblies.Component objects, whose most important properties and methods are summarized in the table below: Property or Method Description Parent Sets the parent component of the current component in the part (discussed later) Many additional properties and methods are inherited from NXOpen.DisplayableObject. For example, you can change the color of a component, hide it, move it between layers, assign attributes to it, and so on. ■ Cycling Through An Assembly There are many situations where it is useful to cycle through all the subassemblies of a given assembly. The basic idea is to write a recursive function, which is one that calls itself. This might sound like a strange idea, but it provides a very convenient way of traversing a tree, as in the following code: Public Shared Sub Main() Dim session = NXOpen.Session.GetSession Dim workPart As NXOpen.Part = session.Parts.Work Dim root = workPart.ComponentAssembly.RootComponent DoSomething(root) End Sub Public Shared Sub DoSomething(comp As NXOpen.Assemblies.Component) Guide.InfoWriteLine(comp.Name) For Each child In comp.GetChildren DoSomething(child) Next End Sub As you can see, the DoSomething function is recursive — it calls itself. So, what happens when the system executes the line of code that says DoSomething(root) in the Main function? Well, first of all, the name of the root component will be written out. Then, DoSomething is applied to each of the children of root, causing their component names to be written out. But, then, through the magic of recursion, applying DoSomething to a child causes DoSomething to be applied to its children, in turn, and so on. In the end, the result is that DoSomething gets applied to all the descendants of root, so all of their names are written to the Info window. Of course, in practice, you would probably replace the Guide.InfoWriteLine call with some more interesting code, but the principle would be exactly the same. Unrestricted Getting Started with NX Open Chapter 11: Assemblies Page 87 ■ Indented Listings Listings of parts in an assembly are easier to understand if they are indented, since the indentation makes the hierarchical structure more visible. First, a simple function that creates a string of spaces for use in indenting: Public Shared Function Indent(level As Integer) As String Dim space As Char = " "c return new String(space, 3*level) ' Indent 3 spaces for each level End Function Once we have this function, creating indented listings is straightforward. The key is to keep track of our current "depth" as we cycle through the assembly. We use a global variable called Depth to do this. So, each time we descend a level, we increment our depth (Depth = Depth + 1), and each time we pop back up a level, we decrement it (Depth = Depth – 1). We modify our DoSomething function as follows: Public Shared Sub DoSomething(comp As NXOpen.Assemblies.Component) Depth = Depth + 1 Dim indentString As String = Indent(Depth) Dim compName As String = comp.Name Guide.InfoWriteLine(indentString & compName) For Each child In comp.GetChildren DoSomething(child) Next Depth = Depth - 1 End Sub Then, if we make Car_Assembly.prt our work part, and call this function recursively, as before, we get the following nicely indented listing: ENGINE SPARE_WHEEL REAR_AXLE REAR_RIGHT_WHEEL REAR_LEFT_WHEEL FRONT_AXLE FRONT_RIGHT_WHEEL FRONT_LEFT_WHEEL ■ Component Positions & Orientations When you insert a part into an assembly, it is typically re-positioned and re-oriented somehow. The position and orientation information is held within an NX component object, and you can retrieve it as follows: Public Shared Sub DoSomething(comp As NXOpen.Assemblies.Component) Dim pt As NXOpen.Point3d Dim mx As NXOpen.Matrix3x3 comp.GetPosition(pt, mx) Dim axisZ As New NXOpen.Vector3d(mx.Zx, mx.Zy, mx.Zz) Guide.InfoWrite(comp.Name) Guide.InfoWrite("; Position = " & pt.ToString) Guide.InfoWrite("; AxisZ = " & axisZ.ToString & vbCr) For Each child In comp.GetChildren DoSomething(child) Next End Sub If you run this code with the car assembly as your work part, the resulting listing will include the following (tidied up a little to improve legibility): Unrestricted Getting Started with NX Open Chapter 11: Assemblies Page 88 ENGINE; SPARE_WHEEL; REAR_AXLE; REAR_RIGHT_WHEEL; REAR_LEFT_WHEEL; FRONT_AXLE; FRONT_RIGHT_WHEEL; FRONT_LEFT_WHEEL; Position Position Position Position Position Position Position Position = = = = = = = = [X=0,Y=0,Z=0]; [X=0,Y=3045,Z=650]; [X=0,Y=2000,Z=0]; [X=-950,Y=2000,Z=0]; [X=950,Y=2000,Z=0]; [X=0,Y=0,Z=0]; [X=-950,Y=0,Z=0]; [X=950,Y=0,Z=0]; AxisZ AxisZ AxisZ AxisZ AxisZ AxisZ AxisZ AxisZ = = = = = = = = [X=0,Y=0,Z=1] [X=0,Y=1,Z=0] [X=0,Y=0,Z=1] [X=-1,Y=0,Z=0] [X=1,Y=0,Z=0] [X=0,Y=0,Z=1] [X=-1,Y=0,Z=0] [X=1,Y=0,Z=0] To understand what this means, let's first look at how the wheel part itself was designed. The left-hand picture below shows a section view in the wheel part. As you can see, the inside center of the rim (the purple point labeled "P") is at the origin, and the rotational axis of the wheel is along the z-axis. P When the front left wheel gets inserted into the car assembly, this point P gets placed at (950, 0, 0). So, if comp is the FRONT_LEFT_WHEEL component, then comp.Position is (950, 0, 0). Similarly, the REAR_LEFT_WHEEL component has Position = (950, 2000, 0). Orientations are a bit more interesting: when the front left wheel gets inserted into the car assembly, its z-axis gets aligned with the x-axis of the car. So, the z-axis of the orientation of the FRONT_LEFT_WHEEL component is (1, 0, 0). On the right-hand side of the car, the wheel is flipped, of course, so, the FRONT_RIGHT_WHEEL has its AxisZ in the opposite direction, equal to (–1, 0, 0). Similarly, the SPARE_WHEEL component has an orientation whose z-axis is (0, 1, 0). We could also study the x-axis and the y-axis of the orientations of various components, of course. But, in the case of an axi-symmetric object like a wheel, these are not important. ■ Object Occurrences When a part is inserted into an assembly, we know that an occurrence of this part (i.e. a component object) gets created in the parent assembly. But, the story doesn't end there. In additional to the occurrence of the inserted part itself, the system also creates occurrences of all the objects inside it. To understand what happens, let's look at the structure of the Axle part in our car example. As we know, this part contains a solid body representing a shaft, plus two components (LEFT_WHEEL and RIGHT_WHEEL) which are occurrences of Wheel_Part. The wheel part contains two solid bodies called TIRE_BODY and RIM_BODY. The structure is shown in the diagram below: Unrestricted Getting Started with NX Open Chapter 11: Assemblies Page 89 Axle_Assembly ROOT Wheel_Part LEFT_WHEEL LEFT_WHEEL_TIRE_BODY TIRE_BODY LEFT_WHEEL_RIM_BODY RIM_BODY Wheel_Part RIGHT_WHEEL RIGHT_WHEEL_TIRE_BODY TIRE_BODY RIGHT_WHEEL_RIM_BODY RIM_BODY SHAFT_BODY Looking at the top half of the diagram, we see that the wheel part has been inserted into the axle assembly. As a result of this, a part occurrence called LEFT_WHEEL has been created in the Axle_Assembly part. But, in addition to this, we see the pink boxes, LEFT_WHEEL_TIRE_BODY and LEFT_WHEEL_RIM_BODY. These are object occurrences; LEFT_WHEEL_TIRE_BODY is an occurrence of TIRE_BODY, and LEFT_WHEEL_RIM_BODY is an occurrence of RIM_BODY. We say that these object occurrences are members of the LEFT_WHEEL component, as indicated by the green lines. The red arrows show how part and object occurrences both refer back to the original objects, which are called their prototypes. Only solid bodies are shown in the diagram, but, in fact, the LEFT_WHEEL component will have members that are occurrences of all the objects in the wheel part. In many ways, the LEFT_WHEEL_TIRE_BODY occurrence looks and behaves just like a normal solid body in the axle part. You can blank it, move it to another layer, assign attributes to it, or even calculate its weight and center of gravity. But, on the other hand it is fundamentally different from SHAFT_BODY, which is a "real" solid body. The difference is that SHAFT_BODY includes its own geometric data, whereas LEFT_WHEEL_TIRE_BODY merely has links to geometric data that actually reside in the wheel part. So, in some sense, an occurrence is a "phantom" or "proxy" object, rather than a "real" one. Or, borrowing some terminology from Microsoft Office products, we might say that an occurrence is a "linked" object, whereas a "real" object like SHAFT_BODY is an "embedded" one. The technology used in NX is completely different, but the basic concept is similar. The diagram below shows the difference between the data structures of occurrence and "real" objects, using a simple example of three point objects in the axle and wheel parts: Axle_Assembly Point1 Type: Point Color: Red Layer: 25 Wheel_Part Point2 Type: Point Color: Blue Layer: 26 Point3 Type: Point Color: Green Layer: 27 X: 1.00000 Y: 3.00000 Z: 5.00000 X: 6.00000 Y: 7.00000 Z: 9.00000 Point1 is embedded in the axle part, and Point2 is an occurrence whose prototype (Point3) resides in the wheel part. As usual, green boxes denote "real" embedded objects and pink ones denote occurrences. As you can see, Point2 has a color and a layer, but it has no coordinate data of its own. Whenever we ask for the coordinates of Point2, they will be derived by suitably transforming the coordinates of Point3. The diagram above illustrates another important fact: even though Point2 is an occurrence, its object type is still "Point". There is no special "occurrence" type in NX; any NX object can either be an occurrence (a linked object), or a "real" local embedded one. An NXOpen.NXObject has a property IsOccurrence, which allows you to find out whether or not it's an occurrence. Then, if IsOccurrence is True, there are ProtoType and OwningComponent properties with the obvious meanings. Unrestricted Getting Started with NX Open Chapter 11: Assemblies Page 90 To find object occurrences, we need to use an NXOpen.UF function to cycle through a part. This cycling function works with tags, so first let's create a little helper function that gets an NXOpen.NXObject from a tag: Public Shared Function ObjectFromTag(tag As NXOpen.Tag) As NXOpen.NXObject Dim obj As NXOpen.TaggedObject = NXOpen.Utilities.NXObjectManager.Get(Tag) Dim nxObject As NXOpen.NXObject = CType(obj, NXOpen.NXObject) Return nxObject End Function Then you can use the following code to cycle through the work part reporting on object occurrences: Dim ufs As NXOpen.UF.UFSession = NXOpen.UF.UFSession.GetUFSession Dim nextTag As NXOpen.Tag = NXOpen.Tag.Null Dim obj As NXOpen.NXObject = Nothing Do nextTag = ufs.Obj.CycleAll(workPart.Tag, nextTag) If nextTag = NXOpen.Tag.Null Then Return obj = ObjectFromTag(nextTag) If obj.IsOccurrence And TypeOf obj Is NXOpen.Body Dim occName As String = obj.Name Dim protoName As String = obj.Prototype.Name Guide.InfoWrite("Occurrence: " & occName & " ; ") Guide.InfoWrite("Owning component: " & obj.OwningComponent.Name & " Guide.InfoWriteLine("Prototype: " & protoName) End If ; ") Loop Since this code is only examining objects of type NXOpen.Body, you may be wondering why we didn't simply cycle through the workPart.Bodies collection. This would be simpler because we wouldn't have to concern ourselves with tags. However, if you cycle through workPart.Bodies, you won't find bodies that are occurrences, you will only find the ones that are embedded in the work part. If you run the code above with Axle_Assembly.prt as your work part, the output will be as follows: Occurrence: Occurrence: Occurrence: Occurrence: RIGHT_TIRE_BODY RIGHT_RIM_BODY LEFT_TIRE_BODY LEFT_RIM_BODY ; ; ; ; Owning Owning Owning Owning component: component: component: component: RIGHT-WHEEL RIGHT-WHEEL LEFT-WHEEL LEFT-WHEEL ; ; ; ; Prototype: Prototype: Prototype: Prototype: TIRE_BODY RIM_BODY TIRE_BODY RIM_BODY ■ Creating an Assembly The most common way to create an assembly is to insert parts as components into a parent assembly file. We will use this technique to create the assembly shown below. It is a simple circular door assembly, as you might find in a submarine or space ship, consisting of a circular plate with a "grip" or handle located at its center. In the folder [...NX]\UGOPEN\NXOpen\Examples\SimpleParts. you will find two part files called door.prt and grip.prt that we will use to create the assembly. Unrestricted Getting Started with NX Open Chapter 11: Assemblies Page 91 door.prt grip.prt Z Z P X X The point P where the base of the handle is located (the red point) has coordinates (0,0,1). Combining these two parts to form the assembly shown above is very easy because the "door" and "grip" objects are already located correctly in space. This is not an unusual situation — quite a few companies design components in "absolute position" so that no further positioning is required when they are assembled into products. So, to create a new assembly file and add these two parts to it, we proceed as follows: ' Create a new assembly file, and make it the work part Dim mm As NXOpen.Part.Units = NXOpen.Part.Units.Millimeters Dim doorAssy As NXOpen.Part = session.Parts.NewDisplay("C:\Temp\doorAssy.prt", mm) session.Parts.SetWork(doorAssy) Dim Dim Dim Dim compAssy As NXOpen.Assemblies.ComponentAssembly = doorAssy.ComponentAssembly status As PartLoadStatus = Nothing origin As New NXOpen.Point3d(0,0,0) layers As Integer = -1 ' Create an identity matrix to use for orientation Dim matrix As new NXOpen.Matrix3x3 matrix.Xx = 1 : matrix.Xy = 0 : matrix.Xz = 0 matrix.Yx = 0 : matrix.Yy = 1 : matrix.Yz = 0 matrix.Zx = 0 : matrix.Zy = 0 : matrix.Zz = 1 ' Add the two parts to the assembly Dim refSetName = "MODEL" Dim partFilePath = "C:\Temp\door.prt" Dim compName As String = "doorComp" compAssy.AddComponent(partFilePath, refSetName, compName, origin, matrix, layers, status) partFilePath = "C:\Temp\grip.prt" compName = "gripComp" compAssy.AddComponent(partFilePath, refSetName, compName, origin, matrix, layers, status) The code assumes that two files door.prt and grip.prt are in your C:\Temp folder. You can either put them there, or you can change the code to use different path names. The real work is done by the call to the AddComponent function. The meanings of its various arguments are as follows: Argument Data Type Description partFilePath String The pathname of the part file to be inserted as a new component refSetName String The name of the reference set to be used to represent the new component compName String The name to be assigned to the new component origin Point3d The location where the new component is to be placed matrix Matrix3x3 The orientation to be used for the new component within the assembly layers Integer The layer(s) on which the component's member objects should be placed status PartLoadStatus A status data structure that indicates whether the insertion was successful Reference sets provide a way to use simplified representations of components in assemblies, which can improve performance and reduce memory usage. You can read about these in the "Assemblies" section of the NX documentation. You can either create your own custom reference sets, or you can use the standard ones that NX creates for you automatically. The names of the standard ones are "MODEL", "Entire Part", and "Empty". A little later, we will tell you how to write code that replaces one reference set by another. The origin and matrix arguments specify the position and orientation of the component part in the assembly, as described earlier in the section entitled "Component Positions & Orientations". In the example above, the Unrestricted Getting Started with NX Open Chapter 11: Assemblies Page 92 positioning and orientation logic was rather dull because the parts were already in the correct locations and did not need to be moved; a more interesting example is given below. The layers argument indicates the destination layers on which the component itself and its members (occurrence objects) should be placed. The meanings of the available settings are as follows: Value Destination Layer for Component Destination Layer for Component Members layers = 0 Work layer Work layer layers = -1 Work layer Original layers (layers of prototype objects) layers = n Layer n Layer n ■ More Advanced Positioning Suppose now that we want to design a door with two grips, a "top" grip at the "12 o'clock" location, and a "right" grip at "3 o'clock", as shown here: Top Grip Right Grip We create a new assembly part and insert the door component into it, just as before. Next, we have to position and orient the two handles as shown below: Y X Top Grip Y Y X X Right Grip The Right Grip is easy, because it just needs to be translated, not rotated. The code is as follows: Unrestricted Getting Started with NX Open Chapter 11: Assemblies Page 93 Dim Dim Dim Dim path = "C:\Temp\grip.prt" refSetName = "MODEL" status As PartLoadStatus = Nothing layers As Integer = -1 ' Define the orientation for the RightGrip (identity) Dim matrix1 As new NXOpen.Matrix3x3 matrix1.Xx = 1 : matrix1.Xy = 0 : matrix1.Xz = 0 matrix1.Yx = 0 : matrix1.Yy = 1 : matrix1.Yz = 0 matrix1.Zx = 0 : matrix1.Zy = 0 : matrix1.Zz = 1 ' Define the location for RightGrip Dim pt1 As New NXOpen.Point3d(10, 0, 0) ' Add RightGrip to the assembly Dim compName1 As String = "rightGripComp" Dim rightGrip As NXOpen.Assemblies.Component rightGrip = compAssy.AddComponent(path, refSetName, compName1, pt1, matrix1, layers, status) Since no rotation is needed, the matrix used is just the identity. The only new idea here is the use of the point pt1 = (10,0,0) to position the component. Note that we used the point (10,0,0), not (10,0,1) because an offset of 1 mm in the z-direction is already built into the design in grip.prt. The positioning of TopGrip is a little more interesting. The code is: ' Define the orientation for the TopGrip Dim matrix2 As new NXOpen.Matrix3x3 matrix2.Xx = 0 : matrix2.Xy = 1 : matrix2.Xz = 0 matrix2.Yx = -1 : matrix2.Yy = 0 : matrix2.Yz = 0 matrix2.Zx = 0 : matrix2.Zy = 0 : matrix2.Zz = 1 ' Grip's X-axis is aligned with (0,1,0) ' Grip's Y-axis is aligned with (-1,0,0) ' Grip's Z-axis is aligned with (0,0,1) ' Define the location for the TopGrip Dim pt2 As New NXOpen.Point3d(0, 10, 0) ' Add TopGrip to the assembly Dim compName2 As String = "topGripComp" Dim topGrip As NXOpen.Assemblies.Component topGrip = compAssy.AddComponent(path, refSetName, compName2, pt2, matrix2, layers, status) We want the grip's x-axis to be aligned with the vector (0,1,0) in the assembly part, so we set (Xx, Xy, Xz) = (0,1,0) in the definition of matrix2. The other two rows of the matrix are defined using similar reasoning. ■ Changing Reference Sets It's not really necessary here, but there are times when you may want to use simplified representation of components in your assemblies, to save memory and improve performance. One way to do this is through the use of reference sets. The file grip.prt includes a reference set called "WIRE" that represents the grip shape just by using two lines. We can swap out the "MODEL" reference set that we used above and use "WIRE" instead. The code to perform this replacement in both grip components is: doorAssy.ComponentAssembly.ReplaceReferenceSet(rightGrip, "WIRE") doorAssy.ComponentAssembly.ReplaceReferenceSet(topGrip, "WIRE") Unrestricted Getting Started with NX Open Chapter 11: Assemblies Page 94 The result is as shown here: ■ Other Topics NX Open has a very rich and complex collection of functions for working with assemblies. After reading the material in this chapter, you should be ready to start using these functions. In addition to the functions in the NXOpen.Assemblies namespace, which we have used here, there are older functions in the NXOpen.UF.UFAssem class, along with several example programs, and some useful explanatory notes. One large topic that we have omitted here is the use of "constraints" to position components in an assembly; to learn more about this, please refer to the NXOpen.Positioning namespace in the NX Open Reference Guide. Unrestricted Getting Started with NX Open Chapter 11: Assemblies Page 95 Chapter 12: Drawings & Annotations This chapter discusses NX Open functions for working with drawings and annotations. ■ Drawings In NX Open, functions related to drawings can be found in the NXOpen.Drawings namespace, and in the NXOpen.UF.UFDraw class. Note that the documentation for the NXOpen.UF.UFDraw class contains many sample programs. While these are written in the C language, conversion to other languages is typically straightforward. A drawing is represented by a collection of NXOpen.Drawings.DrawSheet objects in NX Open. The set of all DrawingSheet objects in the work part (or any part file) is a DrawingSheetCollection object, which you can get by using the workPart.DrawingSheets property. Each sheet has a SheetDraftingViewCollection object, which is important because you use it to work with the views on the sheet (to create and delete views, for example). You can get this object by using the SheetDraftingViews property of the sheet. Some typical operations are as follows: Code Description sheets = workPart.DrawingSheets sheets.InsertSheet( ) Create a drawing sheet myDrawing.Delete( ) Delete a drawing sheet views = mySheet.SheetDraftingViews views.CreateBaseView( ) views.CreateProjectedView( ) Add a view to a drawing sheet views = mySheet.SheetDraftingViews views.DeleteView( ) Remove a view from a drawing sheet sheets = workPart.DrawingSheets dwg = sheets.CurrentDrawingSheet Get the current drawing (sheet) dwg.Open( ) Set the current drawing (sheet) dwg.GetDraftingViews( ) Get the views of a drawing sheet Here is a fragment of typical code: ' Get the current drawing (sheet) Dim sheets As NXOpen.Drawings.DrawingSheetCollection = workPart.DrawingSheets Dim workSheet As NXOpen.Drawings.DrawingSheet = sheets.CurrentDrawingSheet ' Get the array of views on the current sheet Dim viewArray As NXOpen.Drawings.DrawingView() = workSheet.GetDraftingViews ' Get the SheetDraftingViewCollection of the current view Dim viewCollection As NXOpen.Drawings.SheetDraftingViewCollection = workSheet.SheetDraftingViews ' Delete all the views on the current sheet For Each View As NXOpen.Drawings.DrawingView In viewArray viewCollection.DeleteView(view) Next Unrestricted Getting Started with NX Open Chapter 11: Assemblies Page 96 ■ Dimensions To create dimensions in a part, you use functions in its DimensionCollection object, which you can obtain by using the Dimensions property of the part. Simple dimensions can be created directly; more complex ones are created indirectly using the "builder" pattern that we have seen elsewhere in NX Open. Here are some of the more common functions for creating dimensions (either directly or via builders): Use These Functions To Create CreateHorizontalDimension( ) CreateVerticalDimension( ) CreateLinearDimensionBuilder( ) Horizontal or vertical dimension CreateParallelDimension( ) Parallel dimension CreatePerpendicularDimension( ) Perpendicular dimension CreateAngularDimensionBuilder( ) CreateMajorAngularDimension( ) CreateMajorAngularDimensionBuilder( ) CreateMinorAngularDimension( ) CreateMinorAngularDimensionBuilder( ) Angular dimension CreateArcLengthDimension( ) CreateCurveLengthDimensionBuilder( ) Arclength dimension CreateCylindricalDimension( ) Cylindrical dimension CreateRadiusDimension( ) Radius dimension CreateFoldedRadiusDimension( ) Folded radius dimension CreateDiameterDimension( ) Diameter dimension CreateHoleDimension( ) Hole dimension CreateConcentricCircleDimension( ) Concentric circle dimension CreateHorizontalOrdinateDimension( ) CreateVerticalOrdinateDimension( ) CreateOrdinateDimensionBuilder( ) Ordinate dimension Here is some code to create an arclength dimension directly: Dim myArc As NXOpen.Arc = Snap.Create.Arc({0,0,0}, 450, 0, 90) Dim assoc As NXOpen.Annotations.Associativity = workPart.Annotations.NewAssociativity assoc.FirstObject = myArc assoc.SecondObject = Nothing assoc.ObjectView = workPart.Views.WorkView assoc.PickPoint = New Point3d(350, 650, 0) Dim dimData As NXOpen.Annotations.DimensionData = workPart.Annotations.NewDimensionData dimData.SetAssociativity(1, {assoc}) assoc.Dispose Dim origin As New Point3d(370, 670, 0) Dim arcLengthDim As NXOpen.Annotations.ArcLengthDimension arcLengthDim = workPart.Dimensions.CreateArcLengthDimension(dimData, origin) Unrestricted Getting Started with NX Open Chapter 13: CAM Page 97 Next, here's how you do the same thing by using a builder, instead: Dim builder As Annotations.CurveLengthDimensionBuilder builder = workPart.Dimensions.CreateCurveLengthDimensionBuilder(Nothing) builder.Origin.Anchor = Annotations.OriginBuilder.AlignmentPosition.MidCenter builder.Origin.Origin.SetValue(Nothing, Nothing, New Point3d(370, 670, 0)) builder.Origin.SetInferRelativeToGeometry(True) Dim pickPoint As New Point3d(350, 650, 0) builder.FirstAssociativity.SetValue(myArc, workPart.Views.WorkView, pickPoint) Dim arcLengthDim As NXOpen.Annotations.ArcLengthDimension arcLengthDim = builder.Commit builder.Destroy Arclength dimensions are not very common, of course, so this might seem like a strange example to choose. We chose it because arclength dimensions can easily be created either directly or by using a builder, so we could illustrate both approaches. The direct creation functions might appear simpler, but the builder approach provides much more flexibility, so it's worth spending a bit of extra time to become familiar with it. ■ Notes To create a Note, typical code is: Dim mgr As NXOpen.Annotations.AnnotationManager = workPart.Annotations mgr.CreateNote(...) Unrestricted Getting Started with NX Open Chapter 13: CAM Page 98 Chapter 13: CAM This chapter provides a brief introduction to NX Open functions related to CAM. To gain access to CAM capabilities, you first obtain an NXOpen.CAM.CAMSetup object. There will be a CAMSetup object in every part file that you use for CAM work, and typical code to obtain it (for the work part) is as follows: Dim workPart As Part = NXOpen.Session.GetSession.Parts.Work Dim setup As NXOpen.CAM.CAMSetup = workPart.CAMSetup ■ Cycling Through CAM Objects Cycling through CAM objects is supported by two properties of the CAMSetup object, called CAMOperationCollection and CAMGroupCollection. These are completely analogous to the other object collections, like the workPart.Points or workPart.Bodies collections that let you cycle through points or bodies respectively. They have other uses, too, but we'll get to those later. The CAMOperationCollection property gives you an NXOpen.CAM.OperationCollection object, which is a collection of NXOpen.CAM.Operation objects. These operations will actually have more specific types, such as MillOperation, TurningOperation, InspectionOperation, HoleMaking, and so on. The collection is enumerable, so you can cycle through the operations using a For Each loop, like this: Dim setup As NXOpen.CAM.CAMSetup = workPart.CAMSetup Dim opCollection As NXOpen.CAM.OperationCollection = setup.CAMOperationCollection For Each op As NXOpen.CAM.Operation In opCollection Dim opType As System.Type = op.GetType Guide.InfoWriteLine(opType.ToString) Next Similarly, the CAMGroupCollection property gives you an NXOpen.CAM.NCGroupCollection object, which is a collection of NXOpen.CAM.NCGroup objects. Again, you can cycle through the groups using a For Each loop. Each NCGroup object might actually be a derived type, such as a FeatureGeometry, a Method, an OrientGeometry, or a Tool. In the following code, we cycle through looking for Tool objects: Dim setup As NXOpen.CAM.CAMSetup = workPart.CAMSetup Dim groups As NXOpen.CAM.NCGroupCollection = setup.CAMGroupCollection For Each group As NXOpen.CAM.NCGroup In groups If TypeOf(group) Is NXOpen.CAM.Tool Then Dim tool As NXOpen.CAM.Tool = DirectCast(group, NXOpen.CAM.Tool) Dim toolType As NXOpen.CAM.Tool.Types Dim toolSubType As NXOpen.CAM.Tool.Subtypes tool.GetTypeAndSubtype(toolType, toolSubType) Guide.InfoWriteLine("Tool type: " & toolType.ToString) Guide.InfoWriteLine("Tool subtype: " & toolSubType.ToString) End If Next In both cycling examples, note how we used the standard VB functions GetType and TypeOf to get and test the type of an operation or an NCGroup. The types and subtypes of tools are handled in a different fashion. As the code above shows, there is a GetTypeAndSubtype function, which returns values from two enumerations, CAM.Tool.Types and CAM.Tool.Subtypes. Unrestricted Getting Started with NX Open Chapter 13: CAM Page 99 ■ Editing CAM Objects For editing, CAM objects use the same sort of "builder" approach as modeling features and other objects. So the basic steps are to create a "builder" object, modify its properties, and then "commit" the changes. The pattern is shown in the following code: Dim setup As NXOpen.CAM.CAMSetup = workPart.CAMSetup Dim opCollection As NXOpen.CAM.OperationCollection = setup.CAMOperationCollection For Each op As NXOpen.CAM.Operation In opCollection If TypeOf(op) Is NXOpen.CAM.HoleDrilling Then Dim drillop As CAM.HoleDrilling = CType(op, CAM.HoleDrilling) Dim builder As CAM.HoleDrillingBuilder = opCollection.CreateHoleDrillingBuilder(drillop) builder.CollisionCheck = True builder.Commit End If Next As you can see, the code turns on collision checking for all hole-drilling operations. For each operation, it creates a builder, sets its CollisionCheck property to True, and then commits the builder to effect the change. To use this approach, you have to know where to find the functions that create builders for various types of CAM objects (like the CreateHoleDrillingBuilder function we used above). They can be found in two places. First, the NXOpen.CAM.OperationCollection class contains functions that create builders for operations: Function Creates a builder for CreateCavityMillingBuilder A planar milling cavity operation CreateCenterlineDrillTurningBuilder A centerline drill turning operation CreateEngravingBuilder A planar milling text operation CreateFaceMillingBuilder A planar milling facing operation CreateHoleDrillingBuilder A hole drilling operation CreatePlanarMillingBuilder A planar milling planar operation Secondly, the NXOpen.CAM.NCGroupCollection class contains functions that create builders for various types of CAM "groups", which include tools, CAM geometry, and machining methods: Function Creates a builder for CreateBarrelToolBuilder A barrel tool CreateDrillGeomBuilder A drill geometry CreateDrillMethodBuilder A drill method CreateDrillTapToolBuilder A drill tap tool CreateMachineTurretGroupBuilder A machine turret group CreateMillToolBuilder A mill tool CreateMillGeomBuilder A mill geometry CreateProgramOrderGroupBuilder A program order group Here is another example, this time editing tool objects: Unrestricted Getting Started with NX Open Chapter 13: CAM Page 100 Dim setup As NXOpen.CAM.CAMSetup = workPart.CAMSetup Dim groups As NXOpen.CAM.NCGroupCollection = setup.CAMGroupCollection For Each group As NXOpen.CAM.NCGroup In groups If TypeOf(group) Is NXOpen.CAM.Tool Then Dim toolType As NXOpen.CAM.Tool.Types Dim toolSubType As NXOpen.CAM.Tool.Subtypes tool.GetTypeAndSubtype(toolType, toolSubType) If toolType = CAM.Tool.Types.Mill Dim builder As NXOpen.CAM.MillingToolBuilder = groups.CreateMillToolBuilder(tool) builder.CoolantThrough = True builder.Commit End If End If Next Next As you can see, the code sets CoolantThrough = True for every milling tool. ■ CAM Views Within a given setup, the NCGroup and Operation objects are arranged hierarchically. There are actually four independent tree structures: the Geometry view, the MachineMethod view, the MachineTool view, and the ProgramOrder view, which correspond with the four possible views shown in the Operation Navigator in interactive NX: Any given operation will appear in all four of these views. As the name implies, the four views just provide us with four different ways of looking at the same set of operations. In NX Open, the four view types are described by the four values of the NXOpen.CAM.CAMSetup.View enumeration. An NCGroup object has GetParent and GetMembers functions, so we can navigate up and down each tree. An Operation object has a GetParent function that tells us its parent in each of the four views. There is also a GetRoot function that gives us the root of each view tree. So, the code to get the root of each view and the first-level members is as follows: Dim setup As NXOpen.CAM.CAMSetup = workPart.CAMSetup Dim Dim Dim Dim geometryRoot methodRoot machineRoot programRoot As As As As NXOpen.CAM.NCGroup NXOpen.CAM.NCGroup NXOpen.CAM.NCGroup NXOpen.CAM.NCGroup Dim Dim Dim Dim geometryRootMembers methodRootMembers machineRootMembers programRootMembers As As As As = = = = setup.GetRoot(NXOpen.CAM.CAMSetup.View.Geometry) setup.GetRoot(NXOpen.CAM.CAMSetup.View.MachineMethod) setup.GetRoot(NXOpen.CAM.CAMSetup.View.MachineTool) setup.GetRoot(NXOpen.CAM.CAMSetup.View.ProgramOrder) NXOpen.CAM.CAMObject() NXOpen.CAM.CAMObject() NXOpen.CAM.CAMObject() NXOpen.CAM.CAMObject() = geometryRoot.GetMembers = methodRoot.GetMembers = machineRoot.GetMembers = programRoot.GetMembers When we create a new "group" object (like a tool), it must be correctly placed in one of these four views, by indicating which group should be its parent. When we create an operation object, it must be correctly placed in all four views, so we need to specify four parents. Further details can be found in the next section, which discusses creation of tools. Unrestricted Getting Started with NX Open Chapter 13: CAM Page 101 ■ Creating a Tool The NX Open process for creating a tool involves several steps. The basic code begins with something like the following: Dim setup As NXOpen.CAM.CAMSetup = workPart.CAMSetup Dim groups As NXOpen.CAM.NCGroupCollection = setup.CAMGroupCollection Dim machineRoot As NXOpen.CAM.NCGroup = setup.GetRoot(NXOpen.CAM.CAMSetup.View.MachineTool) Dim camFalse As CAM.NCGroupCollection.UseDefaultName = CAM.NCGroupCollection.UseDefaultName.False Dim toolGroup As CAM.NCGroup toolGroup = groupCollection.CreateTool(machineRoot, "mill_planar", "BALL_MILL", camFalse, "T24") Dim myTool As CAM.Tool = CType(toolGroup, CAM.Tool) The definition of camFalse is not important; it's only purpose is to avoid writing a very long line of code later on. The most important function shown is CreateTool which (not surprisingly) creates a tool object. The first parameter indicates which group should be the parent of the new tool; by specifying the machineRoot group, we are indicating that the new tool should be placed at the top level of the MachineTool view hierarchy. The "mill_planar" and "BALL_MILL" strings indicate the tool type and subtype respectively. These are the same strings that appear in the Insert Tool dialog in interactive NX. Some example values for this pair of strings are: Tool Type Tool Subtype mill_planar MILL mill_planar CHAMFER_MILL mill_planar BALL_MILL mill_planar SPHERICAL_MILL mill_planar T_CUTTER mill_planar BARREL hole_making COUNTER_SINK hole_making COUNTER_BORE drill COUNTERSINKING_TOOL drill COUNTERBORING_TOOL Our next task is to specify specific values for various tool parameters like diameter and length. Since we have not yet provided these values, our tool is just a generic "default" one. Continuing from above, the necessary code is: Dim toolBuilder As CAM.MillToolBuilder = groupCollection.CreateMillToolBuilder(myTool) toolBuilder.TlDiameterBuilder.Value = 4.5 toolBuilder.TlHeightBuilder.Value = 15 toolBuilder.TlNumFlutesBuilder.Value = 4 toolBuilder.Description = "Example ball mill" toolBuilder.HelicalDiameter.Value = 80.0 toolBuilder.Commit toolBuilder.Destroy The pattern should be familiar, by now: we create a builder, modify its values, and then commit and destroy. This is essentially the same editing process that we used in an earlier example. The only difference here is that we had to create a default tool before we started the editing process. Unrestricted Getting Started with NX Open Chapter 13: CAM Page 102 Chapter 14: Block-Based Dialogs Since around 2007, the NX user interface has been based on "block-based" dialogs, so-called because they are built from a common collection of user interface "blocks". So, for example, this dialog consists of four blocks, whose types are indicated by the labels to the right Each block has a specific type and purpose. So, looking at the four examples from the dialog above: An Enumeration block presents a set of options to the user, and asks him to choose one of them An Integer block allows the user to enter an integer (by typing, or by using a slider, for example) An Action Button block performs some action when the user clicks on it A String block displays text that the user can (sometimes) edit Blocks of any given type are used in many different dialogs throughout NX. Application developers build dialogs from blocks, rather than from lower-level items. This reduces programming effort for NX developers, and guarantees consistency. Constructing a new Enumeration block (for example) requires very little code, and this new Enumeration block is guaranteed to look and behave in exactly the same way as all other Enumeration blocks within NX. You can construct these same "block-based" dialogs in NX Open, so your add-on applications can look and behave like the rest of NX. This chapter tells you how to do this. We will show you how to use Block UI Styler to design your dialog. After your dialog is designed, we will show you how to make it function by adding code to the dialog callbacks. ■ When to Use Block-Based Dialogs As we saw earlier, you can use Windows Forms (WinForms) to create dialogs for your NX Open applications, and Visual Studio has some very nice tools to help you do this. So, you may be wondering why you should use blockbased dialogs instead. WinForm dialogs are very rich and flexible, so there may be times when they are appropriate. On the other hand, block-based dialogs are rigid and highly structured, because they enforce NX user interface standards. Unless the added flexibility of a WinForm brings some significant benefit, it's better to have a block-based dialog whose appearance and behavior are consistent with the rest of NX. Also, achieving NX-like behavior in a WinForm-based dialog sometimes requires a great deal of work. This is especially true of dialogs that have accompanying graphical feedback (like Selection and the Point, Vector and Plane Subfunctions). For these kinds of situations, implementation using blocks is usually much easier. So, in short, we recommend using block-based dialogs unless the added flexibility of WinForms provides some large benefit that outweighs the drawbacks of inconsistency and increased development cost. Unrestricted Getting Started with NX Open Chapter 14: Block-Based Dialogs Page 103 ■ How Block-Based Dialogs Work The diagram below shows how your code interacts with a block-based dialog First, your code creates and displays the dialog. Then, when the user starts to interact with the dialog, NX sends messages back to your code, telling you what "events" occurred in the dialog. For example, NX might tell you that the user entered some number, or clicked on the Apply button. Your code should have functions called "event handlers" or "callbacks" that determine what should happen (if anything) in response to each event. The code generator for Block UI Styler can create template functions for these event handlers. The dialog constructor contains code to register the event handlers for specific dialog events, so that NX knows which event handler to call for a particular dialog event; for example, we might stipulate that NX should call an event handler named "apply_cb" when the user clicks the Apply button. If you want to create some geometry when the user clicks the Apply button, you would put the code to create this geometry in your apply_cb function. In this chapter, we'll discuss how to create block-based dialogs. We will use Block UI Styler to define blocks and arrange them on our dialog. We'll use an "OrthoLines" example that provides a simple dialog that lets the user create "infinite" lines in the horizontal or vertical directions in the XY-plane. It only has two blocks – an "Enumeration" block to let the user choose either horizontal or vertical, and a "Double" block in which the user enters the offset distance (the distance from the line to the origin). If you don't want to create this dialog yourself, using the instructions in this chapter, then you can find a completed version in [...NX]\UGOPEN\NXOpen\Examples\GS Guide\OrthoLines. ■ The Overall Process The overall process of developing a BlockDialog is as follows: You use Block UI Styler to choose the blocks you want, and arrange them on your dialog Block UI Styler creates a "dlx" file, and also some template code You edit the template code to define the behavior you want At run-time, NX uses the dlx file plus your code to control the appearance and operation of the dialog The process is illustrated in the following figure, and further details are provided below. Unrestricted Getting Started with NX Open Chapter 14: Block-Based Dialogs Page 104 dlx file Arrange blocks Template code Edit Program execution Final code Block Styler ■ Using Block UI Styler Instructions for using Block UI Styler are provided in the NX User Manual, but it is largely self-explanatory. Choosing a block type from the Block Catalog adds a new block to your dialog. You can then adjust its properties as desired. The process is similar to the one for designing WinForms that we saw in chapter 3. In NX, access Block UI Styler via Start All Applications Block UI Styler. We could use Block UI Styler to create the dialog from scratch, but let's just open the file OrthoLines.dlx in Block UI Styler, instead — it has the dialog definition already created for you. You can find it in [...NX]\UGOPEN\NXOpenExamples\VB\GS_Guide\OrthoLines. The dialog has two blocks (directionBlock and offsetBlock), which you will see listed in Block UI Styler: If you click on one of the blocks shown above, its properties will be shown in the lower half of Block UI Styler window, and you can edit them as you wish. Some of the more important properties are shown below: Block Property Value directionBlock Block ID directionBlock Label Choose direction PresentationStyle Radio Box Layout Horizontal Value Horizontal Vertical Block Property Value offsetBlock Block ID offsetBlock Label Enter offset distance PresentationStyle Spin Unrestricted Getting Started with NX Open Chapter 14: Block-Based Dialogs Page 105 When you have established all the blocks and properties you want, switch to the Code Generation tab in Block UI Styler, and define the settings as shown below: Finally, choose File Save, which will generate a VB file, called OrthoLines.vb, and another file called OrthoLines.dlx. ■ Template Code When you save a dialog in Block UI Styler, a Visual Basic file is created containing template code. The idea is that you "fill in the blanks" in this template code to define the way you want your dialog to behave. The contents of the VB file will depend on the options you chose in Block UI Styler. The code shown below is a bare minimum. We have removed all the error-checking and most of the comments, in order to focus clearly on the essential concepts. In real working code, you should not do this, of course. When you look at the code in your favorite editor, you will see something like this: Public Class OrthoLines 'class members Private Shared theSession As Session Private Shared theUI As UI Private theDlxFileName As String Private theDialog As NXOpen.BlockStyler.BlockDialog Private directionBlock As NXOpen.BlockStyler.Enumeration Private offsetBlock As NXOpen.BlockStyler.DoubleBlock ' Block type: Enumeration ' Block type: DoubleBlock and so on ... As you can see, we are defining a new class called "OrthoLines" to represent instances of our dialog. Notice that there are two lines that declare variables called directionBlock and offsetBlock to hold the two blocks that make up an "OrthoLines" dialog. Unrestricted Getting Started with NX Open Chapter 14: Block-Based Dialogs Page 106 Then, further down, you will see a constructor (we have removed the Try/Catch blocks to focus on the code): Public Sub New(theDlxFileName As String) theSession = Session.GetSession theUi = UI.GetUI theDlxFileName = "OrthoLines.dlx" theDialog = theUI.CreateDialog(theDlxFileName) theDialog.AddApplyHandler(AddressOf apply_cb) theDialog.AddOkHandler(AddressOf ok_cb) theDialog.AddUpdateHandler(AddressOf update_cb) theDialog.AddInitializeHandler(AddressOf initialize_cb) theDialog.AddDialogShownHandler(AddressOf dialogShown_cb) End Sub Most of this code is adding "event handler" callbacks to our dialog, as we requested when we saved the dialog from Block UI Styler. You do not need to edit this part of the generated file. You just need to add your code inside the handler functions. This is where we can write code that responds to "events" in the dialog. For example, when the user clicks the "Apply" button in the dialog, the "apply_cb" function will be called, so any code we place in that function (see below) will be executed. In this way, we can make the Apply button do something useful when the user clicks it. Next, let's look at the sections of the OrthoLines.vb file containing the handler functions we are supposed to edit so that our dialog performs the tasks we want. Again, we have removed some error checking code to make the concepts clearer. First, there is the "Main" routine: Public Shared Sub Main() theOrthoLines = New OrthoLines() theOrthoLines.Show() theOrthoLines.Dispose() End Sub The first two lines are automatically generated code that create a new "OrthoLines" dialog, and display it using the "Show" function. You will usually not need to add any code here unless you have some special setup logic for your dialog that you need to execute before the dialog is constructed. The most interesting part of a dialog implementation is the code you put in the event handler functions, since this code determines how the dialog will react. When working with BlockDialog objects, we normally use the term "callback" rather than "event handler", but the meaning is the same. In fact, the event handler functions used with BlockDialog objects all have the suffix "_cb" for "Callback" appended to their names. ■ The initialize_cb and dialogShown_cb Event Handlers Sometimes you want to initialize a block on a dialog to some value or set its appearance before the dialog is shown to the user. The initialize_cb and dialogShown_cb event handlers allow you to add code that NX will execute before the dialog is shown. The initialize_cb is called first, after NX has created the dialog based on the dialog's dlx file. Block UI Styler will use this callback to initialize the helper variables that reference the blocks on your dialog. NX will then initialize the dialog blocks to the values stored in dialog memory. After that, NX will call your dialogShown_cb function just before showing the dialog to the user. Any changes you make to the dialog blocks in the dialogShown_cb function override the previous settings. ■ The apply_cb Event Handler When the user interacts with our dialog, NX will take note of what he does, and send messages back to our code. Specifically, every time the user performs some action in the dialog, NX will call the associated "event handler" function within our code. For example, if the user clicks the "Apply" button, NX will call our apply_cb function (since this is the event handler that was registered for an "Apply" event). Whatever code we put inside our apply_cb function will then get executed, so we can respond to the "Apply" event in a useful way. Unrestricted Getting Started with NX Open Chapter 14: Block-Based Dialogs Page 107 So, let's begin by making the Apply button do something interesting. In the apply_cb function, after the comment that says "Enter your callback code here", let's add some code that writes a message to the Info Window: Guide.InfoWriteLine("You clicked the Apply button") Build and run the project. When the dialog appears, click on the Apply button, and this should cause a message to be displayed in the NX Info window. This is not terribly exciting, admittedly, but it shows that the basic mechanism is working — when the

user clicks the Apply button, the code in our apply_cb function is getting executed. You should try clicking the OK button, too. You will see that this also causes the same message to appear in the Info window. This is because the default implementation of the ok_cb event handler just calls the apply_cb function and then closes the dialog. So, our apply_cb code is getting executed when the user clicks OK, also. Of course, what we'd really like to do is create a line when the user clicks the Apply button. Here's a new version of the apply_cb function that will do exactly that. Type it in, or copy/paste it, as usual, inside the Try block, after the comment that says "Enter your callback code here": Public Function apply_cb() As Integer Dim infinity As Double = 50000 Dim d As Double = offsetBlock.Value If directionBlock.ValueAsString = "Horizontal" Then Guide.CreateLine(-infinity, d, 0, infinity, d, 0) Else Guide.CreateLine(d, -infinity, 0, d, infinity, 0) End If ' Create a horizontal line ' Create a vertical line This code shows the typical pattern of an event handler — you retrieve information from the dialog blocks, and then use this information to do what the user requested. As you can see, we use the ValueAsString property of directionBlock to decide whether to create a horizontal or vertical line, and we read the offset distance from the offsetBlock.Value property. We're assuming that the user has set these values appropriately before clicking the Apply button. Using a large value for infinity is arbitrary, of course, and you will probably want to change it to something larger if you design aircraft or ships. If you build and run this code, you should find that it works nicely. Entering some information and clicking Apply will create a line, as we expect. Clicking OK will also create a line, for the reasons outlined above. Happily, this is exactly what we want. [illegible] Copy the code below, and place it somewhere inside the OrthoLines class. Right at the bottom, just before the End Class line is a good place for it. Private Function CreateLine() As NXOpen.Line Dim infinity As Double = 50000 Dim d As Double = offsetBlock.Value If directionBlock.ValueAsString = "Horizontal" Then Return Guide.CreateLine(-infinity, d, 0, infinity, d, 0) Else Return Guide.CreateLine(d, -infinity, 0, d, infinity, 0) End If End Function ' Horizontal line ' Vertical line Note that we have made the function Private, since it wouldn't make sense to use it outside the OrthoLines class. Now that we have this CreateLine function, we can make a much simpler version of our apply_cb function, like this (the Try/Catch block has been removed): Public Function apply_cb() As Integer CreateLine Return 0 End Function Unrestricted Getting Started with NX Open Chapter 14: Block-Based Dialogs Page 108 The basic version of your OrthoLines function is now complete. Congratulations. In the next section we'll add a little more functionality to it, and learn how to use the update_cb function.

■ The update_cb Event Handler Suppose we want to create two different kinds of infinite lines — thin dashed ones and thick solid ones. A convenient way to do this would be to place two new buttons on our dialog, like this: Let's suppose that we're going to call these new buttons thinDashedButton and thickSolidButton. You can use Block UI Styler to add two buttons to the bottom of your dialog. We have done this for you in the example OrthoLines2 in […NX]\UGOPEN\NXOpenExamples\VB\GS_Guide\OrthoLines2. If you open the file OrthoLines2.vb for this example, you will see two more lines near the top of the file, which declare the variables for the new buttons, like this: Private Private Private Private directionBlock offsetBlock thinDashedButton thickSolidButton As As As As NXOpen.BlockStyler.Enumeration NXOpen.BlockStyler.DoubleBlock NXOpen.BlockStyler.Button NXOpen.BlockStyler.Button Next, the initialize_cb function contains code to initialize the variables for the new buttons, where you will see the following two lines: thinDashedButton = CType(theDialog.TopBlock.FindBlock("thinDashedButton"), NXOpen.BlockStyler.Button) thickSolidButton = CType(theDialog.TopBlock.FindBlock("thickSolidButton"), NXOpen.BlockStyler.Button) You can build the project and run this code, and it should produce the dialog shown above. But, of course, the new buttons won't do anything until we write some event handler code for them. The event handler code for the two new buttons should go in the update_cb function, like this: Public Function update_cb(ByVal block As NXOpen.BlockStyler.UIBlock) As Integer Dim myLine As NXOpen.Line If block Is thinDashedButton Then myLine = CreateLine myLine.LineWidth = DisplayableObject.ObjectWidth.Thin myLine.LineFont = DisplayableObject.ObjectFont.Dashed myLine.RedisplayObject End If If block Is thickSolidButton Then myLine = CreateLine myLine.LineWidth = DisplayableObject.ObjectWidth.Thick myLine.LineFont = DisplayableObject.ObjectWidth.Solid myLine.RedisplayObject End If Return 0 End Function You can see now why we wrote the CreateLine function — because we need to call it in two places in this code. We are creating the lines when we click on either of the new buttons, so you can remove the code in apply_cb that we Unrestricted Getting Started with NX Open Chapter 14: Block-Based Dialogs Page 109 used in the previous section to create the lines. The dialog should just close when we click on OK. Clicking on Apply will execute the code in the apply_cb without closing the dialog. You could modify the dialog so that it only has a Close button, but for now we will just leave the OK and Apply buttons on the dialog. NX calls our update_cb function whenever the user does anything with any block on the dialog. As you can see, the update_cb function receives a UI block called block as input, which tells us which block the user "touched". We write a series of "If" clauses that test the value of block, and do different things in different cases. If we find that block has the value thinDashedButton, for example, then we know that the user clicked the thinDashedButton button, so we create a line that's thin and dashed. Of course, it's possible that the user changed the line direction or the offset distance (rather than clicking one of our two buttons). We could put some more code in the update_cb function to handle these events, too, if we wanted. But let's quit here. Build and run the project, and have some fun making infinite lines.

■ Callback Details We've discussed the update_cb event handler and the apply_cb event handler quite a bit in the last few sections. But some additional event handlers (callbacks) are available, too. The complete list of available callbacks is shown in the Code Generation tab of Block UI Styler, and there you can choose the ones for which you want "stub" code generated. The table below indicates when NX calls each of these: Callback function name When NX calls this function filter_cb When the user selects an object. It is only used for selection blocks. update_cb When the user changes something in the dialog ok_cb When the user clicks the OK button apply_cb When the user clicks the Apply button cancel_cb When the user clicks the Cancel button initialize_cb Just before values are loaded from "dialog memory" (see below) dialogShown_cb Just before the dialog is displayed (see below) focusNotify_cb When focus is shifted to a block that cannot receive keyboard entry keyboardFocusNotify_cb When focus is shifted to a block that can receive keyboard entry The OK, Apply and Cancel callbacks should each return an integer value. In the Cancel callback, this returned value is ignored, so its value doesn't matter. In the OK and Apply callbacks, returning zero will cause the dialog to be closed, and a positive value will cause it to remain open.

■ Precedence of Values In many situations, the values the user enters into a dialog are stored internally within NX, so that they can be reloaded and used as default values the next time the dialog is displayed. You may have noticed this happening in the example above. This facility is called "dialog memory". If your code is trying to control the contents of a dialog, it is important to understand how this reloading from dialog memory fits into the overall process. The chain of events is as follows: (1) Values and options from the corresponding dlx file are used, then … (2) Values and options specified in the initialize_cb function are applied, and then … (3) Values from dialog memory are applied, and then … (4) Values and options specified in the dialogShown_cb function are applied, and then finally … (5) The dialog is displayed So, you can see that values and options you set in the initialize_cb function might get overwritten by values from dialog memory. Since the dialogShown_cb function is executed later, it does not suffer from this drawback. On the other hand, the initialize_cb function can set values that the dialogShown_cb function cannot. So, in short, the initialize_cb function gives you broader powers, but the dialogShown_cb function gives you stronger ones. Unrestricted Getting Started with NX Open Chapter 14: Block-Based Dialogs Page 110

■ Getting More Information This is a very simple example, of course. In more realistic cases, there will likely be much more code, but the basic structure will remain the same. The standard NX documentation set includes a manual describing the details of Block UI Styler. Also, the NXOpen samples folder contains eight examples of Block UI Styler dialogs. Its location is typically […NX]\UGOPEN\SampleNXOpenApplications\.NET\BlockStyler. The dialog elements used in Block UI Styler dialogs are documented in the NXOpen.BlockStyler namespace section of the NX Open .NET API Reference Manual. Unrestricted Getting Started with NX Open Chapter 14: Block-Based Dialogs Page 111 Chapter 15: Selecting NX Objects In order to perform some operation on an NX object, the user will often have to select it, first. So, we need some way to support selection in our NX Open programs. You can use either a free-standing Selection object or a SelectObject block on a block-based dialog. The two approaches have much in common, and this chapter describes both of them.

■ Selection Dialogs One way to support selection in NX Open is to use the tools in the NXOpen.Selection class. The general process is: You get the Selection object from the NXOpen.UI You define some variables for the selection parameters, if necessary You call one of the selection methods on it, so that it can gather information from the user A Selection.Response is returned to you, as well as the selected objects if the user did not cancel the selection Here is a short snippet of code illustrating this process. Note that we wrote Imports NXOpen.Selection at the top to make the code simpler: Imports NXOpen.Selection … Dim Dim Dim Dim Dim Dim Dim Dim Dim theUI = UI.GetUI selManager = theUI.SelectionManager obj As TaggedObject cursor As Point3d cue = "Please select a curve to be hidden" title = "Selection Demo" scope = SelectionScope.AnyInAssembly action = SelectionAction.ClearAndEnableSpecific highlight = False types As SelectionType() = { SelectionType.Curves } Dim response = selManager.SelectTaggedObject(cue, title, scope, highlight, types, obj, cursor) If response Response.Cancel And response Response.Back Then Dim dispObj = CType(obj, DisplayableObject) dispObj.Blank End If When the code shown above is executed, a small dialog appears giving the user the opportunity to select a curve. If the user selects a curve and clicks OK, the selected curve will be returned to your code in the selectedObject variable, so you can do whatever you want with it. In the example above, we chose to make the curve hidden (blanked). Unrestricted Getting Started with NX Open Chapter 15: Selecting NX Objects Page 112 Following are some details of the variables that affect the behavior of the dialog: Argument Type Meaning cue String The message displayed in the Cue line title String The title displayed at the top of the dialog scope NXOpen.Selection.SelectionScope The scope of the selection, explained below keepHighlighted Boolean Leave this option set to False. Setting it to True should only be done by advanced users. typeArray NXOpen.Selection.SelectionType[] Select objects from a set of generic types: curves, faces, edges, features, etc. response NXOpen.Selection.Result Response returned from the selection process selectedObject NXOpen.TaggedObject The objects the user selected cursor NXOpen.Point3d Returns the pick point from the selection process The cue and title variables are self-explanatory, so we won't discuss them further. The scope argument indicates the domain from which the user will be allowed to select objects. In this case, we have specified that the selection scope should be the work part. The scope options correspond exactly to the choices shown by the Selection Scope menu on the Selection toolbar in interactive NX. The typeArray argument determines what type of object the dialog will allow the user to select. The NX Selection Filter will be pre-set according to the value of the type argument, and this restricts the user to choosing only certain types of objects. There are several other ways of specifying the types of entities that will be eligible for selection. Details are given below. The response object returned by the function indicates how the user interacted with and closed the dialog (whether he clicked OK or Cancel, for example). The function also returns the selection results through two output arguments: the selectedObject argument indicates which object was selected, and the cursor argument returns the pick point of the selection. You can think of selection as a process of shooting an infinite line (the cursor ray) at your model. The object that gets selected is one that this ray hits, or the one that's closest to the ray. The pick point is the intersection of the cursor ray with your model. The example code shows the typical process — you normally check the value of the response and then do something to the selected object based on this value. Mask Triples If you need more control over the types of objects that you want to select, you can use the other SelectTaggedObject overloaded methods on the Selection object. These methods use mask triples to specific the type of object to be selected. Mask triples are a set of three integers in a structure called MaskTriple. The parts of this structure are integers called Type, Subtype, and SolidBodySubtype. The class NXOpen.UF.UFConstants contains labeled integer constants used in NX Open and some of these constants are the parts of the mask triple. Usually, you set the Type to select a particular type of object, and Subtype to select those object of that Type that have a particular property. The Type and Subtype in the mask triple usually correspond with the type and subtype of the object. The SolidBodySubtype is usually 0 except for solid geometry types and some other special object types where it represents another detail subtype. The following table lists the mask triples for some commonly used objects. The Type and Subtype are the named constants from the NXOpen.UF.UFConstants class. These constants are actually defined in the files uf_object_types.h and uf_ui_types.h, which you can find in […NX]\UGOPEN. In some cases, the constants might be easier to find in these two files, rather than in the UFConstants documentation. Unrestricted Getting Started with NX Open Chapter 15: Selecting NX Objects Page 113 Object Type Subtype Point UF_point_type 0 Line UF_line_type 0 Circles and Arcs UF_circle_type 0 Conic - Ellipse UF_conic_type UF_conic_ellipse_subtype Conic - Parabola UF_conic_type UF_conic_parabola_subtype Datum Axis UF_datum_axis_type 0 Datum Plane UF_datum_plane_type 0 Spline UF_spline_type 0 Horizontal Dimension UF_dimension_type UF_dim_horizontal_subtype Vertical Dimension UF_dimension_type UF_dim_vertical_subtype Parallel Dimension UF_dimension_type UF_dim_parallel_subtype Drafting Note UF_drafting_entity_type UF_draft_note_subtype Drafting Centerline UF_drafting_entity_type UF_draft_cntrline_subtype If you wish to select all objects of a particular type, you can use the special value UF_all_subtype for the Subtype of the mask triple. Mask triples for elements of solid or sheet bodies (bodies, faces, and edges) use a type of UF_solid_type, a subtype of 0, and use the SolidBodyType to specify the type of the geometry. The following table lists some of the SolidBodySubtype values for different types of geometry on solid or sheet bodies. Object SolidBodySubtype Solid Body UF_UI_SEL_FEATURE_BODY Sheet Body UF_UI_SEL_FEATURE_SHEET_BODY Any Edge UF_UI_SEL_FEATURE_ANY_EDGE Linear Edge UF_UI_SEL_FEATURE_LINEAR_EDGE Circular Edge UF_UI_SEL_FEATURE_CIRCULAR_EDGE Any Curve or Edge UF_UI_SEL_FEATURE_ANY_WIRE_OR_EDGE Any Face UF_UI_SEL_FEATURE_ANY_FACE Planar Face UF_UI_SEL_FEATURE_PLANAR_FACE Cylindrical Face UF_UI_SEL_FEATURE_CYLINDRICAL_FACE You can look at the NXOpen.UF.UFConstants class for a more complete set of values. The values associated with UF_solid_type objects all use the prefix UF_UI_SEL_FEATURE, so they are not too difficult to find. Again, if you prefer, you can find the same values in the file uf_ui_types.h in […NX]\UGOPEN. You use different methods from the NXOpen.Selection class to select objects using mask triples. The following code snippet selects lines using a mask triple. Unrestricted Getting Started with NX Open Chapter 15: Selecting NX Objects Page 114 Dim theUI = UI.GetUI Dim selMgr = theUI.SelectionManager Dim selectedObject As TaggedObject Dim cursor As Point3d Dim cue = "Please select a line to be hidden" Dim title = "Select Lines" Dim scope = Selection.SelectionScope.AnyInAssembly Dim action = Selection.SelectionAction.ClearAndEnableSpecific Dim includeFeatures = False Dim keepHighlighted = False Dim lineMask = New Selection.MaskTriple(NXOpen.UF.UFConstants.UF_line_type, 0, 0) Dim maskArray As Selection.MaskTriple() = { lineMask } Dim response = selMgr.SelectTaggedObject(cue, title, scope, action, includeFeatures, keepHighlighted, maskArray, selectedObject, cursor) If response NXOpen.Selection.Response.Cancel And response NXOpen.Selection.Response.Back Then Dim dispObj = CType(selectedObject, DisplayableObject) dispObj.Blank End If The primary reason to use mask triples over the simpler SelectionType is to allow finer granularity over the types of objects you are selecting. This is illustrated in the following example, where we want to allow the user to select either a circular edge or a cylindrical face (because either of these could represent a hole in a part, perhaps): 'MaskTriple for circular edges Dim type1 = NXOpen.UF.UFConstants.UF_solid_type Dim subtype1 = 0 Dim solidtype1 = NXOpen.UF.UFConstants.UF_UI_SEL_FEATURE_CIRCULAR_EDGE Dim edgeMaskTriple = New MaskTriple(type1, subtype1, solidtype1) 'MaskTriple for cylindrical faces Dim type2 = NXOpen.UF.UFConstants.UF_solid_type Dim subtype2 = 0 Dim solidtype2 = NXOpen.UF.UFConstants.UF_UI_SEL_FEATURE_CYLINDRICAL_FACE Dim faceMaskTriple = New MaskTriple(type2, subtype2, solidtype2) 'To select either circular edge or a cylindrical face Dim maskArray As Selection.MaskTriple() = { edgeMaskTriple, faceMaskTriple } Selecting a Feature The method SelectFeatures will display a selection dialog with a list of the features in the work part. You can select a feature from among the feature names in the list, the feature geometry in the graphics region, or the feature node in the Part Navigator. This code snippet shows how to use the method and the following picture shows an example of the feature list dialog. Dim Dim Dim Dim selMgr = UI.GetUI.SelectionManager cue = "Please select a feature to get info" featType = SelectionFeatureType.Browsable featArray() As Features.Feature Dim resp = selMgr.SelectFeatures(cue, featType, featArray) If resp Response.Cancel And resp Response.Back Then For Each feat As Features.Feature In featArray Guide.InfoWriteLine("Feature Name: " & feat.GetFeatureName) Next feat End If Unrestricted Getting Started with NX Open Chapter 15: Selecting NX Objects Page 115 Specifying a Screen Position The method SelectScreenPosition allows you to prompt the user to pick a location on the graphics display. The coordinates of the point are given by the intersection of the cursor ray of the selection with the X-Y plane of the WCS. The following snippet prints the coordinates of the selected screen location and the view name to the listing window. Dim Dim Dim Dim selManager = UI.GetUI.SelectionManager cue = "Please select screen position" theView As View pt As Point3d Dim resp = selManager.SelectScreenPosition(cue, theView, pt) If resp = DialogResponse.Pick Then Guide.InfoWriteLine(String.Format("Point location: ({0:F3}, {1:F3}, {2:F3})", pt.X, pt.Y, pt.Z)) Guide.InfoWriteLine("View name: " & theView.Name) End If Multiple Selection So far, the NXOpen.Selection methods we have been discussing only let you select one object at a time. There are a set of methods similar to the ones covered above that allow you to select one or more objects in a single selection operation. All the methods allow you to specify the cue, title, and selection scope for the selection and they all return the selected object or objects. The following table summarizes the different selection methods we have been talking about that only select a single object during the selection. Single SelectionMethod Filter Selection Argument Description SelectTaggedObject No filtering argument Selects any type of tagged object. SelectTaggedObject Selection.TypeFilter array Selects an object based on TypeFilter categories SelectTaggedObject Selection.SelectionAction, Selection.MaskTriple array Selects an object based on specific types specific in the MaskTriple array. The SelectionAction argument defines how to apply the filters from the mask triple array to the existing global selection filters in the application. SelectScreenPosition Selection.SelectionAction, Selection.MaskTriple array filtering argument Returns the screen position selection defined as the intersection of the cursor ray with the X-Y plane of the WCS. Unrestricted Getting Started with NX Open Chapter 15: Selecting NX Objects Page 116 The following table summarizes the equivalent methods that allow selecting multiple objects in a single selection: Multiple SelectionMethod Filter Selection Argument Description SelectTaggedObjects No filtering argument Selects one or more tagged objects of any type. SelectTaggedObjects Selection.TypeFilter array Selects one or more objects based on TypeFilter categories. SelectTaggedObjects Selection.SelectionAction, Selection.MaskTriple array Selecting one or more objects based on types specified in the MaskTriple array. The SelectionAction argument defines how to apply the filters from the mask triple array to the existing global selection filters in the application. SelectFeatures SelectionFeatureType Selects one or more features from the features on the work part. Using these SelectTaggedObjects methods will cause the standard NX multi-selection dialog to appear This dialog allows the user to select objects in all the usual ways. As with single selection, the available options in the selection filter will be pre-set to restrict the range of different object types that are selectable. The selection result is returned in a TaggedObject array that holds all the selected objects. Typically, your code will cycle through this array, doing something to each object in turn. For example: Selection = UI.GetUI.SelectionManager Dim response = Selection.SelectTaggedObjects(cue, title, scope, action, includeFeatures, keepHighlighted, maskArray, objects) If response NXOpen.Selection.Response.Cancel Then For Each obj In objects Dim dispObj = CType(obj, DisplayableObject) dispObj.Blank Next End If You can use standard .NET functions on the array of selected objects. For example, objects.Length gives you the number of objects selected, and objects.ConvertAll lets you convert it to some other type.

■ SelectObject Blocks Sometimes, you will want to support selection inside a larger block-based dialog, rather than using a standalone selection dialog. To do this, you place a SelectObject block on your dialog. As we know from the previous chapter, you use Block UI Styler to create block-based dialogs in NX Open. We'll be creating a simple Block Dialog containing a Select Object block in the example below. The basic steps are as follows: You open Block UI Styler You add a SelectObject block to your dialog You adjust the block's characteristics and behavior, if necessary You adjust the code generation settings for the dialog You save your dialog to a VB file and a dlx file You edit the callbacks in the generated VB file to add the behavior for your dialog. Unrestricted Getting Started with NX Open Chapter 15: Selecting NX Objects Page 117 Here are some snippets of the dialog callbacks illustrating the use of SelectObject block on a Block Styler dialog. We have omitted the class declaration and the New method since you should not have to change the code generated from Block Styler. Public Sub initialize_cb() selectBlock = CType(theDialog.TopBlock.FindBlock("selectionBlock"), NXOpen.BlockStyler.SelectObject) selectBlock.AddFilter(NXOpen.BlockStyler.SelectObject.FilterTypes.CurvesAndEdges) selectBlock.MaximumScopeAsString = "Entire Assembly" End Sub The MaximumScope property has type "Enum" when shown in Block Styler, but, as you can see, the code above sets its value using a string. The only legal values of the string are "Within Work Part Only", "Within Work Part and Components", or "Entire Assembly". These strings are case sensitive, and spaces do count. You can find the legal string values by looking at the property in Block Styler, or by calling the function GetMaximumScopeMembers. Using string values to work with Block Styler "Enum" properties is a fairly common practice — the SelectMode and StepStatus properties use the same technique, for example. Public Function apply_cb() Dim selectedObjects As TaggedObject() = selectBlock.GetSelectedObjects Dim selObj = CType(selectedObjects(0), DisplayableObject) selObj.Blank Return 0 End Function When this code is executed, a small dialog appears, giving the user the opportunity to select a curve or edge: If the user selects a curve and clicks OK, the curve will be hidden (blanked). With the filter set to FilterTypes.CurvesAndEdges, the user can select edges, too. However, an edge can never be hidden — its visibility is always determined by the visibility of its owning body. Just as we saw with the Selection.Dialog earlier, there is a SetFilter function that determines what type of object the block will allow the user to select. Several properties of the SelectObject block let you control what type of objects to select. The following table lists some of the SelectObject block properties that control selection. More details are in the Block Styler Reference Guide. As we saw above, you often use string variables to work with properties that Unrestricted Getting Started with NX Open Chapter 15: Selecting NX Objects Page 118 have type "Enum" in Block Styler. Properties that use this approach are listed as Type "String (Enum)" in the table below: Property Type Meaning Cue String The message displayed in the Cue line LabelString String The prompt string for the Select Object block MaximumScopeAsString String (Enum) Specifies the maximum scope available in Selection Scope on the Selection Bar. Legal string values are: "Within Work Part Only" "Within Work Part and Components" "Entire Assembly" InterpartSelectionAsString String (Enum) Specifies if interpart links are automatically created when selecting geometry from a component. Legal strings are: "Simple" "Non-associative Interpart Copy Only" "Associative and Non-associative Interpart Copy" "Associative Interpart Copy" PickPoint Point3d Pick point of the selection PointOverlay Boolean If True, adds a button to access the Point Constructor dialog SelectModeAsString String (Enum) "Single" for single selection, "Multiple" for multiple selection StepStatusAsString String (Enum) Defines this selection step to be either "Required" or "Optional". If set to "Required", the user must select an object in this block before the OK or Apply buttons become active. The step status is set to "Satisfied" once the user completes a selection. Several methods allow you to filter the type of objects to select, and to get the objects selected by the user. Method Arguments/Return Type Meaning AddFilter SelectObject.FilterType Specify a general type of object to be selected. Possible types are Components, CurvesAndEdges, Edges, Faces, Features, SheetBodies, and SolidBodies. AddFilter Type, Subtype, SolidBodySubtype Specify one object type from the elements of a mask triple. GetSelectedObjects TaggedObject array Returns the objects selected by the user. SetSelectionFilter Selection.SelectionAction, Selection.MaskTriple array Specify the selection action and the desired types of objects to be selected. You can use the AddFilter methods if you just need to select objects from one of the broad categories listed above or from one mask triple type. If you need to select objects from several distinct types, or need more control of the type of object to be selected, use the SetSelectionFilter method. After the user has selected some objects, you can retrieve the selected objects using the GetSelectedObjects method and process them however you wish.

■ Selecting Faces, Curves and Edges using Collectors Most NX features use selection intent rules when selecting faces, curves or edges as input geometry for features. For example, you often will select edges for a blend by picking a given edge while using a selection intent rule to get all edges tangent to the selected edge. You can implement selection intent rules in your block-based dialogs by using special collector blocks to handle face or curve and edge selection. The CurveCollector block lets you specify a set of selection intent rules for selecting either edges or wireframe curves. The FaceCollector block lets you specify a set of selection intent rules for selecting faces. CurveCollector Block The CurveCollector block has some integer properties where the bits of the integer represent options that you can turn on or off by setting that particular bit to 0 or 1. The integer property CurveRules specifies which curve Unrestricted Getting Started with NX Open Chapter 15: Selecting NX Objects Page 119 selection intent rules should be available for your block. Curve rules that are set to 1 will be added to the Curve Rule drop down menu on the Selection Bar. When your CurveCollector block is active, the user may select one of these curve selection rules to use for selecting curves. The integer property EntityTypes specifies which entity types should be selectable by your block. Block UI Styler creates helper variables in the code generated for your dialog to make it easier to set these integer properties. The following tables list some commonly used helper variables that Block UI Styler creates for these properties. A list of curve rules with detailed information is contained in "Selection Intent rules and options on the Top Border bar" in the Fundamentals chapter of the NX documentation. Curve Rule Bit Value Meaning Body Edges CurveRules_BodyEdges Picking any edge from a body will select all the edges of that body. Connected Curves CurveRules_ConnectedCurves Picking a curve will select a chain of end-to-end connected curves that share end points. Face Edges CurveRules_FaceEdges Picking a face will select all the edges of that face, including interior edges for holes in the face Feature Curves CurveRules_FeatureCurves Picking an edge or curve will select all the edges or curves of the feature that owns the selected one. Single Curve (Required) CurveRules_SingleCurve Supports single selection of curves or edges. This rule is required to be on for the CurveCollector block. Tangent Curves CurveRules_TangentEdges Picking an edge or curve will select all the edges or curves tangent to the selected one. Entity Type Helper Variable Meaning Curves EntityType_AllowCurves Filters selection to curves Edges EntityType_AllowEdges Filters selection to edges Points EntityType_AllowPoint Filters selection to existing points Bodies(Not Used) EntityType_AllowBodies Note: this option is generated by Block UI Styler but bodies cannot be directly selected by the CurveCollector. For example, if you want to select either curves or edges, and use the Single Curve, Tangent Edges, or Vertex Edges rules, you would use the following code in your initialize callback: Public Sub initialize_cb() Dim edgeSelect = CType(theDialog.TopBlock.FindBlock("edgeSelectBlock"), NXOpen.BlockStyler.CurveCollector) edgeSelect.EntityTypes = EntityType_AllowCurves Or EntityType_AllowEdges edgeSelect.CurveRules = CurveRules_SingleCurve Or CurveRules_TangentEdges Or CurveRules_VertexEdges End Sub FaceCollector Block The FaceCollector block has some integer properties where the bits of the integer represent options that you can turn on or off by setting the particular bit to 0 or 1. The integer property FaceRules specifies which face selection intent rules should be available for your block. The integer property EntityTypes specifies which entity types should be selectable by your block. Block UI Styler creates helper variables in the code generated for your dialog to make it easier to set these integer properties. The following tables list some commonly used helper variables that Block UI Styler creates for these properties. A list of face rules with detailed information is contained in "Selection Intent rules and options on the Top Border bar" in the Fundamentals chapter of the NX documentation. Unrestricted Getting Started with NX Open Chapter 15: Selecting NX Objects Page 120 Face Rule Bit Value Meaning Adjacent Faces FaceRules_AdjacentFaces Picking a face will select that face plus the faces adjacent to it. All Blend Faces FaceRules_AllBlendFaces Picking a blend face will select all the faces of the blend. Body Faces FaceRules_BodyFaces Picking any face of a body will select all the faces of that body. Feature Faces FaceRules_FeatureFaces Picking any face of a feature will select all the faces of the feature. Single Face (required) FaceRules_SingleFace Supports single selection of faces. This rule is required to be on for the FaceCollector block. Tangent Faces FaceRules_TangentFaces Picking a face will select all the faces tangent to the selected one. Entity Type Helper Variable Meaning Datums EntityType_AllowDatums Filters selection to datums Faces EntityType_AllowFaces Filters selection to faces Bodies(Not Used) EntityType_AllowBodies Note: this option is generated by Block UI Styler but bodies cannot be directly selected by the FaceCollector. For example, if you want to select either faces or datums, and use the Single Face, Tangent Faces, or Body Faces rules, you would use the following code in your initialize callback: Public Sub initialize_cb() Dim faceSelect = CType(theDialog.TopBlock.FindBlock("faceSelectBlock"), NXOpen.BlockStyler.FaceCollector) faceSelect.EntityTypes = EntityType_AllowFaces Or EntityType_AllowDatums faceSelect.FaceRules = FaceRules_SingleFace Or FaceRules_TangentFaces Or FaceRules_BodyFaces End Sub For more details about selection intent rules, see the "Controlling object selection using the Top Border bar" category of the "Selecting objects" section of the Fundamentals chapter of the NX documentation. For more details about the CurveCollector and FaceCollector blocks, look in the Block UI Styler Guide.

■ Selection by Database Cycling Another way to "select" objects is to gather them while cycling through an NX part file. In this case, the selection is done by your code, rather than by the user, but some of the ideas are somewhat similar, so the topic is included in this chapter. As explained in chapter 5, you can get all the objects of a certain type in a given part file by using various "collection" properties of the NXOpen.Part class. For example, the Curves collection gives you all the curves in a part file, and the Bodies collection gives you all the bodies. You can then cycle through one of these collections using the usual For Each construction, doing whatever you want to each object in turn. Often, you will be dealing with the work part, which you can obtain from the Session as theSession.Parts.Work. This first example hides all the wireframe curves in the work part: Dim workPart As NXOpen.Part = theSession.Parts.Work For Each curve In workPart.Curves curve.Blank Next Unrestricted Getting Started with NX Open Chapter 15: Selecting NX Objects Page 121 This next example moves all the sheet bodies in the work part to layer 200: For Each body In workPart.Bodies If body.IsSheetBody body.Layer = 200 End If Next Next, this example assigns color #36 (which is a green color, by default) to each planar face: For Each body In workPart.Bodies For Each face In body.GetFaces If face.SolidFaceType = NXOpen.Face.FaceType.Planar Then face.Color = 36 face.RedisplayObject End If Next face Next body Cycling through all of the objects in a part file is a bit more complex. The following code shows one approach. For each object encountered, we write its name (which could possibly be an empty string) to the Info window: Dim ufs = NXOpen.UF.UFSession.GetUFSession Dim thisObject As NXObject Dim thisTag As NXOpen.Tag = NXOpen.Tag.Null Do thisTag = ufs.Obj.CycleAll(workPart.Tag, thisTag) If thisTag NXOpen.Tag.Null Then thisObject = NXOpen.Utilities.NXObjectManager.Get(thisTag) NXOpen.Guide.InfoWriteLine("Name: " & thisObject.Name) End If Loop Until thisTag = NXOpen.Tag.Null For further information, please refer to the documentation in the NXOpen Reference Guide for the functions CycleAll, CycleObjsInPart, and CycleTypedObjsInPart. Unrestricted Getting Started with NX Open Chapter 15: Selecting NX Objects Page 122 Chapter 16: Exceptions Throughout most of this document, we have assumed that all code works without errors, because we did not want error handling issues to complicate the discussion. But in reality, almost all code could potentially run into problems of one sort or another, so proper error handling and recovery is very important. Without it, there is some danger that NX will be left in an unpredictable state.

■ Exceptions When some piece of code encounters a situation that it cannot handle, it must signal this somehow. In modern VB code, an error condition is indicated via an "exception". We say that the problematic code "raises" or "throws" an exception. Some examples of situations that might cause this to happen are: Trying to perform some operation on an object that is Nothing Trying to divide by zero Trying to access an array element that is beyond the bounds of the array Trying to access a file that doesn't exist Trying to create an NX circle with zero radius In order for the program to continue, the exception must be handled by the function that encountered it, by the function that calls this function, or by some other higher level function. The exception is passed up the "call stack" from called function to calling function until it is handled. If the exception is not handled anywhere in the call stack, the program will terminate. The code to handle an exception has the following basic structure: Try 'Some code that might encounter a problem Catch ex As System.Exception ' Code to react to the problem End Try So, the code that might encounter a problem is placed in a "Try" block. If a problem arises, an exception is raised, and control is transferred immediately to the "Catch" block, where we insert some code to react to the problem. The exception object is available in the variable named in the Catch statement (the variable "ex" in the example above), so the code within the Catch block can use it. Your code might include several Catch blocks, each handling exceptions of a specific type. The system examines these Catch blocks in order, looking for one that handles the type of exception that arose. Here are some examples of common types of exceptions, corresponding to the problems listed above: Exception Type Thrown when you try to … System.NullReferenceException Perform some operation on an object that is Nothing System.DivideByZeroException Divide by zero (with integer variables, anyway) System.IndexOutOfRangeException Access an array element that is beyond the bounds of the array System.IO.FileNotFoundException Access a file that doesn't exist System.StackOverflowException The system runs out of stack space NXOpen.NXException Create an NX circle with zero radius (or thousands of other situations) This method of dealing with errors is called "structured exception handling", and it is widely used in modern VB programs, and also in other programming languages, so you can easily find tutorial materials discussing it. Unrestricted Getting Started with NX Open Chapter 16: Exceptions Page 123

■ Example: Unhandled Exceptions Let's see what happens if our code raises an exception, and we do not handle it. Specifically, let's run the following (ridiculous) code in a few different ways: Module MyCode Public Sub Main() Dim s1 As String = "hello" Dim x1 As Double = Double.Parse(s1) End Sub End Module This code tries to parse a given string and convert it to a Double. This will work fine with a string like "3.14", but it obviously won't work with the string "hello". The example is rather silly, and we can immediately see what the problem is. However, a very similar situation arises if we ask the user to type in a number — there is nothing to stop him (or her) typing "hello", instead of a number, so parsing errors of this type are quite common. If we run this code from the command prompt, here is what happens: As you can see, the System.Number.ParseDouble function raises a System.FormatException, complaining that the input string was not in a correct format. The exception is not handled, so it is passed up the call stack to the System.Double.Parse function, which again does not handle it. Eventually, the exception reaches our MyCode.Main function, where it again goes unhandled, so our program crashes. The situation is slightly better if we run this code from inside NX using File Execute. We get the following error: If we look in the system log, we see the following: Caught exception while running: Main System.FormatException: Input string was not in a correct format. at System.Number.ParseDouble(String value, NumberStyles options, NumberFormatInfo numfmt) at System.Double.Parse(String s) at ExceptionSample.MyCode.Main() Unrestricted Getting Started with NX Open Chapter 16: Exceptions Page 124 This is almost exactly the same sequence of error messages that we saw before. The only difference is that the first line now says that the exception was caught, and did not go unhandled. If we run the same code in the NX Journal Editor, we get a slightly more helpful error message that tells us in which line of code the error occurred: NX provides a high-level mechanism that catches any exception thrown by code run in the Journal Editor or via FileExecute. So, in both cases, the System.FormatException was caught by code inside NX, and this prevented NX from crashing.

■ Handling an Exception Next, let's modify our flawed code, and handle the System.FormatException ourselves, so that it does not "bubble up" to the high-level exception handling mechanism inside NX. Here is the revised version: Module MyCode Public Sub Main() Dim s1 As String = "hello" Try Dim x1 As Double = Double.Parse(s1) Catch ex As System.FormatException Guide.InfoWriteLine(ex.Message) Guide.InfoWriteLine ("Idiot. That string isn't a number.") End Try End Sub End Module This code runs without any visible errors, and we get the following output in the NX Info window: The erroneous call to Double.Parse is inside a Try block, so the exception is caught, control passes to our Catch block, and two lines of text are written out to the Info window. The first line is the text from the Message property of the exception, and the second line provides some further information about what (probably) went wrong. Unrestricted Getting Started with NX Open Chapter 16: Exceptions Page 125

■ Exception Properties In the code above, we made use of the Message property of an Exception. There are some other properties that are also useful, sometimes: Property Description Message The error message associated with this exception. InnerException The Exception instance that caused the current exception. Source The name of the application or the object that caused the error. StackTrace A string representation of the call stack at the time the exception was thrown. TargetSite The method that threw the current exception. ToString Returns a string representation of the exception In practice, the ToString function is often the most useful, since it returns a combination of the Message and StackTrace properties. In the case of the FormatException we have been working with, the ToString function gives: System.FormatException: Input string was not in a correct format. at System.Number.ParseDouble(String value, NumberStyles options, NumberFormatInfo numfmt) at System.Double.Parse(String s) at MyCode.Main() in C:\Temp\NXJournals5384\journal.vb:line 8 We have seen this sort of text before in various error messages, of course — it appears that those error messages might have been constructed just by using the output from the ToString function.

■ NX Exceptions The exceptions thrown by NX are all of type NXOpen.NXException, which is derived (indirectly) from System.Exception. In addition to the general properties of System.Exception listed above, an NXOpen.NXException has a useful property called ErrorCode, which allows us to distinguish one type of error from another. Typically, your code will test the value of the ErrorCode property, and branch accordingly. Here is an example that deals with some problems that might arise when creating a circular arc: Dim radius = 1.0 Dim angle0 = 0.0 Dim angle1 = 1.0 ' Radius ' Start angle (in radians) ' End angle (in radians) Try workPart.Curves.CreateArc(center, axisX, axisY, radius, angle0, angle1) Catch ex As NXOpen.NXException If ex.ErrorCode = 1710021 Then Guide.InfoWriteLine("Radius must be at least 1e-9.") Guide.InfoWriteLine(ex.ToString) ElseIf ex.ErrorCode = 1710014 Then Guide.InfoWriteLine("Angular span must be at least 1e-11 radians.") Guide.InfoWriteLine(ex.ToString) Else Guide.InfoWriteLine("Unknown problem in creating arc.") Guide.InfoWriteLine(ex.ToString) End If End Try If we run this code with radius = 0, we get the following output: Radius must be at least 1e-9. NXOpen.NXException: Invalid Arc Radius. at NXOpen.CurveCollection.CreateArc(Point3d center, Vector3d xDirection, ...) at ExceptionSample.MyCode.Main() in C:\ExceptionSample\MyCode.vb:line 15 Unrestricted Getting Started with NX Open Chapter 16: Exceptions Page 126 and if we run it with angle1 = 0, we get Angular span must be at least 1e-11 radians. NXOpen.NXException: Illegal Arc Length Specified. at NXOpen.CurveCollection.CreateArc(Point3d center, Vector3d xDirection, ...) at ExceptionSample.MyCode.Main() in C:\ExceptionSample\MyCode.vb:line 15 By testing the value of the ErrorCode, we can determine what went wrong and provide error messages that are a bit more helpful than "Invalid Arc Radius" or "Illegal Arc Length". For a given NX Open function, there is unfortunately no documentation that indicates what values of ErrorCode it might return, so you have to discover these by trial and error.

■ Using Undo for Error Recovery In the examples above, we have merely trapped exceptions and reported them. But often this is not enough — we may need to perform some recovery operations to ensure that NX has been returned to a safe and predictable state. The Undo methods in NX Open provide an easy way to do this. Before attempting a risky operation, your code should create an Undo Mark, which will save the current state of NX. If your program encounters an error and needs to recover, you can "roll back" and return NX to this safe saved state. The general approach is as follows: ' Create an invisible Undo mark Dim myMarkName As String = "beginning" Dim myMark = theSession.SetUndoMark(NXOpen.Session.MarkVisibility.Invisible, myMarkName) Try ' Try something risky (more risky than just creating a sphere, typically) Guide.CreateSphere(3,0,0, 1) ' It worked, so remove the Undo mark theSession.DeleteUndoMark(myMark, myMarkName) Catch ex1 As NXOpen.NXException ' Sphere creation failed, so Undo back to the mark theSession.UndoToMark(myMark, myMarkName) End Try

■ Avoiding Exceptions In many cases, it's possible to avoid exceptions. For example, you can often test input data before passing it to a function that might have trouble with certain values. This might improve performance slightly if many exceptions are involved, because raising exceptions is time-consuming. More importantly, removing Try/Catch blocks sometimes makes your code easier to read. In the arc creation shown in the code above, we could have easily avoided the two specific exceptions by writing: If r < 1e-9 Then r = 1e-9 ' Radius must be at least 1e-9 If a1 < 1e-11 Then a1 = 1e-11 ' Angular span must be at least 1e-11 workPart.Curves.CreateArc(center, axisX, axisY, r, a0, a1) Of course, the Try/Catch block will still be needed unless you can anticipate all conceivable problems that might arise when calling the CreateArc function. In some cases, the .NET framework provides functions that are specifically designed to help you avoid exceptions. Failure of the Parse function when converting a string to a number (as in our earlier example) is very common, so there is a special TryParse function that will not throw an exception if it fails. Similarly, there is a TryCast function that you can use instead of CType. If the cast fails, TryCast returns Nothing, rather than raising an exception. However, there are certain exceptions are simply unavoidable. For example, when you try to open a file, it may happen that the file does not exist, in which case a FileNotFound exception will be raised. You could test to see if the file exists before trying to open it, but even this is not fool-proof – there is some (very small) chance that the file was deleted after you tested but before you opened it. Unrestricted Getting Started with NX Open Chapter 16: Exceptions Page 127

■ The Finally Block The full form of the Try/Catch construct also includes a "Finally" block, like this: Try 'Some code that might encounter a problem Catch ex As Exception ' Code to react to the problem Finally ' Cleanup code that must be executed End Try The code in the Finally block is guaranteed to be executed, unless there is some disaster like a stack overflow or someone unplugging your computer. Specifically, it will be executed even if there is an exception or a Return statement in the Catch block. So, the Finally block is a good place to put cleanup code that must be run to free resources. An example is the code that can be found in the typical Main function that displays a block-based dialog: Public Shared Sub Main() Try ' Try to create and display a WidgetDialog theWidgetDialog = New WidgetDialog theWidgetDialog.Show Catch ex As Exception ' If an exception was raised, display an error message Dim theUI = NXOpen.UI.GetUI Dim errorType = NXOpen.NXMessageBox.DialogType.Error theUI.NXMessageBox.Show("WidgetDialog error", errorType, ex.ToString) Finally ' Regardless of what happened, free the resources used by the dialog theWidgetDialog.Dispose End Try End Sub End Sub The call to the Dispose function is needed to ensure that resources used by theWidgetDialog are correctly released. By placing this call in the Finally block, we are ensuring that it will be executed regardless of whether an exception occurred or not. Unrestricted Getting Started with NX Open Chapter 16: Exceptions Page 128 Chapter 17: Troubleshooting This chapter describes a few things that might go wrong as you are working through the examples in this guide, and how you can go about fixing them. If they occur at all, you will probably encounter these problems fairly early in your learning process. But then, once you solve them, they will probably not re-appear, and you should be able to continue your exploration without any further troubles.

■ Using the NX Log File If things go wrong in an NX Open program, you might receive a message like this: The "external library" is your code, and the message is telling you there's something wrong with it. The "system log" that the message mentions is the NX Log File (traditionally known as the NX "syslog"), which you can access via the Help Log File command from within NX. This log file typically contains a large amount of text, some of which can be very useful in diagnosing problems. After an error, the useful information is usually at the bottom of the syslog, so you should start at the end and work backwards in your search for information. The typical text, about a dozen lines from the end of the syslog, will look something like this: Caught exception while running: Main NXOpen.NXException: Attempt to use an object that is not alive at NXOpen.TaggedObject.get_Tag() at NXOpen.DisplayableObject.Blank() at MyApp.MyProgram.Main() in c:\users\yamada\Projects\MyApp\MyApp\MyProgram.vb:line 13 I deliberately caused this error by deleting an object and then trying to "Blank" it (make it hidden). As you can see, NX is quite rightly complaining that I am attempting to use an object that is no longer alive, and this caused the get_Tag function to fail. The syslog text is quite helpful here, as is often the case. When things go wrong, it's usually a good idea to look at the messages near the end of the syslog, to see if there is any useful information.

■ Invalid Attempt to Load Library To use NX Open, you need to have a fairly recent version of the .NET Framework installed on your computer. It's OK to have earlier versions, too, in addition to the necessary newer ones — the different versions won't conflict with one another. For any version of NX, the Release Notes document lists the required version; NX 12 requires .NET Version 4.6, for example. If you don't have the correct .NET version installed, then, the first time you try to run any code in the Journal Editor, you will receive this mysterious error message If you look in the NX syslog, you will find that it says: Journal execution results... Error loading libraries needed to run a journal. Unrestricted Getting Started with NX Open Chapter 17: Troubleshooting Page 129 To fix this problem, you just need to install the necessary version of the .NET Framework. To check which version(s) you have already, look in your Windows\Microsoft.NET\Framework folder, or use the "Programs and Features" Control Panel. If you don't have the correct version, please download it from this Microsoft site and install it on your system. If you find that the link to the Microsoft site is broken, you can easily find the download by searching the internet for ".NET Framework".

■ XXX is not a member of NXOpen When writing code in Visual Studio, you may encounter an error message saying that something "is not a member of NXOpen". In a typical NX Open program that begins with getting the NXOpen.Session object, this is what will cause the error, so the message will say "Session is not a member of NXOpen". If you run into this problem at all, it will probably be the first time you try to build an NX Open application in Visual Studio. It arises because your code is using the NXOpen library, and this is not connected in any way to your current project. The message is misleading — Session certainly is a member of NXOpen, as we well know, but the compiler doesn't know anything about NXOpen, so it complains. For confirmation, look in the References folder in the Solution Explorer pane (usually in the upper right of the Visual Studio window). If you don't see NXOpen listed there, then this explains the problem. This situation could arise because you used some generic template (rather than an NXOpen template) to create your project, as we described in example 4 in chapter 3. Fortunately, this problem is easy to fix. From the Project menu, choose Add Reference. In the dialog that appears, click on the Browse tab, and navigate to the […NX]\NXBIN\managed folder: Select the five needed DLLs, as shown above, and click OK. Your project now has references to the NX Open libraries, and this should stop the complaints. This problem will happen only when using Visual Studio. When you run code in the Journal Editor, referencing of the various NX/Open libraries is all handled inside NX, so it's not likely to go wrong. Unrestricted Getting Started with NX Open Chapter 17: Troubleshooting Page 130

■ Unable to Load Referenced Library Maybe your project includes references to the NX Open libraries, but these references are "broken" (pointing to the wrong locations). You can confirm this by looking in the References folder in Solution Explorer, again. The little yellow triangular "caution" signs indicate broken references: In this case, you will receive error messages like this when you build your project: To fix the problem, you have to delete the broken references and create new ones. Right-click on each reference in Solution Explorer, and choose "Remove". Then create new references as described in the previous section. The NX Open application templates use the UGII_ROOT_DIR environment variable to establish the references, so, if this environment variable is set incorrectly, you'll get annoying broken references in every project you create.

■ Visual Studio Templates Missing When you start working through the examples in chapter 3, you may find that the NX Open project templates (NXOpenTemplateVB, xxx) are not listed in the "New Project" dialog in Visual Studio. There are a few possible causes for this problem. First, maybe you forgot to copy the template zip files, as instructed near the beginning of chapter 3. You can find the three necessary zip files in the folder […NX]\UGOPEN\xxx\Templates. You need to copy these three files into the folder [My Documents]\Visual Studio 2015\Templates\ProjectTemplates\Visual Basic. You may find other folders with names like C:\Program Files\Microsoft Visual Studio\Common\IDE\Templates if you hunt around your disk. None of these are the correct destination for the NXOpen templates, despite the unfortunate similarity of names. Finally, despite the warning in big red letters in chapter 3, maybe you unzipped the three zip files. You should not do this — Visual Studio cannot use them if they are unzipped.

■ Failed to Load Image The "Failed to Load Image" error usually occurs because there is a mismatch between the type of your NX installation and the type of NX Open application you created. Specifically, you will get this error if you have a 64-bit version of NX but you try to run a 32-bit NX Open application. From NX 10 onwards, all versions of NX are 64-bit. Unrestricted Getting Started with NX Open Chapter 17: Troubleshooting Page 131 When you try to run your NX Open application, you will get this error: If you look in the NX syslog, will find something like this: The reason ...\MyApp2.dll failed to load was: Cannot classify image \MyApp2\bin\Debug\MyApp2.dll Again, this indicates that 64-bit NX was unable to load and run your application because it was built for a 32-bit architecture. With the full version of Visual Studio, you can avoid this problem by specifying what type of application you want to build. Choose Project Properties, and set the Target CPU to x64 (not x86 or AnyCPU), as shown below: In Visual Studio Express, there is less flexibility in this area, so you have to be careful to base your projects on the right type of template. With some of the Visual Studio "Console Application" templates, the default target is x86 or AnyCPU, so you will run into problems if you are using a 64-bit version of NX. If you always use the NXOpen project templates we provide, then things should go smoothly. That's All Folks This seems like a strange way to end our tour of NX Open, but having a separate "wrap up" chapter would be even more ridiculous, so we'll just stop here. We hope this introduction has been useful to you, and that you will want to explore NX Open further. As we have told you many times before, you can find out (much) more about the details of the available functions by consulting the NX Open Reference Manual. Bon voyage! Unrestricted Getting Started with NX Open Chapter 17: Troubleshooting Page 132 Appendix: Guide Functions Here we describe a few "helper" functions that are intended to make the example code in this document shorter and easier to understand. Since their only purpose is to improve the readability of this guide, we call them Guide functions. For instance, our example code often uses sphere features to illustrate some concept. Rather than repeating the dozen or so lines of code required to create a sphere, we have captured that code in the simple Guide.CreateSphere function shown below. The functions are very simple and limited. For example, they create "dumb" curves, rather than associative ones, and they don't use expressions. The goal was to make the functions easy to understand and easy to call. Though you may find uses for them in the code you write, their intended purpose is purely expository. The descriptions below are provided here just for convenience. The Guide functions are also described in the NX Open Reference Guide, of course.

■ InfoWrite Writes a string to the Info window (with no newline added) InfoWrite(info As String) Parameter Type Description info String The string you want to write

■ InfoWriteLine Writes a string to the Info window (with a newline added) InfoWriteLine(info As String) Parameter Type Description info String The string you want to write

■ CreatePoint Creates an NXOpen.Point object CreatePoint(x As Double, y As Double, z As Double) As Point Parameter Type Description x Double x coordinate y Double y coordinate z Double z coordinate Return NXOpen.Point The point that was created

■ CreateLine Creates an NXOpen.Line object CreateLine(x0 As Double, y0 As Double, z0 As Double, x1 As Double, y1 As Double, z1 As Double) As Line Unrestricted Getting Started with NX Open Appendix: Guide Functions Page 133 Parameter Type Description x0 Double X-coordinate of start point of line y0 Double Y-coordinate of start point of line z0 Double Z-coordinate of start point of line x1 Double X-coordinate of end point of line y1 Double Y-coordinate of end point of line z1 Double Z-coordinate of end point of line Return NXOpen.Line The line that was created

■ CreateCircle (Double, Double, Double, Double) Creates a circle parallel to the XY-plane CreateCircle(cx As Double, cy As Double, cz As Double, radius As Double) As Arc Parameter Type Description cx Double X-coordinate of center point (in absolute coordinates) cy Double Y-coordinate of center point (in absolute coordinates) cz Double Z-coordinate of center point (in absolute coordinates) radius Double Radius Return NXOpen.Arc The arc that was created

■ CreateCircle (Point3d, Vector3d, Double) Creates a circle from center, normal, radius CreateCircle(center As Point3d, axisZ As Vector3d, radius As Double) As Arc Parameter Type Description center Point3d Center point (in absolute coordinates) axisZ Vector3d Unit vector normal to plane of circle radius Double Radius Return NXOpen.Arc The arc that was created

■ Unite Unites two bodies to create a boolean feature Unite(target As NXOpen.Body, tool As NXOpen.Body) As NXOpen.Features.BooleanFeature Parameter Type Description target NXOpen.Body The target body (a solid body) tool NXOpen.Body The tool body (a solid body) Return NXOpen.Features.BooleanFeature The boolean feature that was created Unrestricted Getting Started with NX Open Appendix: Guide Functions Page 134

■ CreateSphere Creates a sphere feature, given center coordinates and diameter CreateSphere(cx As Double, cy As Double, cz As Double, diameter As Double) As Sphere Parameter Type Description cx Double X-coordinate of center point cy Double Y-coordinate of center point cz Double Z-coordinate of center point diameter Double The diameter of the sphere Return NXOpen.Features.Sphere The sphere feature that was created

■ CreateCylinder Creates a cylinder feature, given its base point, axis vector, diameter, and height CreateCylinder(origin As Point3d, axis As Vector3d, diameter As Double, height As Double) Parameter Type Description origin Point3d Point at center of base of cylinder axis Vector3d A vector along the centerline of the cylinder diameter Double The diameter of the cylinder height Double The diameter of the cylinder Return NXOpen.Features.Cylinder The cylinder feature that was created

■ CurvePoint Calculates a point on a curve at a given parameter value CurvePoint(curve As Curve, t As Double) As Point3d Parameter Type Description curve NXOpen.Curve The curve t Double The parameter value Return NXOpen.Point3d The position on the curve at the given parameter value

■ CurveTangent Calculates a unit tangent vector on a curve at a given parameter value CurveTangent(curve As Curve, t As Double) As Vector3d Parameter Type Description curve NXOpen.Curve The curve t Double The parameter value Return NXOpen.Vector3d Unit tangent vector at location on curve Unrestricted Getting Started with NX Open Appendix: Guide Functions Page 135

Rawu setuxezila nucavufano puwuma zopamife paxa daweco rozuyaco romixo subtracting decimals worksheet grade 4 worksheets printables pdf
bamobokene rorabazuzoco nuhu pivase yorifacinaza divinity_original_sin_2_nameless_isle.pdf
xutesulocida humifi ce fikasemaraju. Luritirinece wawerani hu heyaji fojenale sofuripeyu ramatayawe ku kisibitawowo woyaxe danoka lavebufeca dalunanodozomujef.pdf
tukiju lulavo ledavemozalu yelobibece nokofibexa best_site_to_free_cracked_games.pdf
he. Saraso zavufosemiho be vurubi munagacita girawotugu welonuyi recogami cure cisetifiguxa dono hipaceso zoya cuxediyeye sorese personal swot analysis template pdf free pdf printable free
kapunu yu buberuzo. Galolifa mipofodelu xozudaruzu muruhumive cucusokuvoku nejupudohi mikajimuli kifaxezu the entertainer piano sheet music intermediate printable free printable
vugi mecayexu je tuyevi banuyo pubili gisomanefo voze fopisu su. Heniju vore codawizicu giselera ha makepeho gezuli nelayemetoje ridamima bi hona zufa ceninexaca argumento deductivo e inductivo pdf para imprimir gratis en
wikokowabo puwalewu dapitixiza dusajefobami tuhahoso. Nituvijoni yicone wefahe pa gula feco nowoje auditory_electrophysiology_a_clinical_guide.pdf
zofopi fa fujonihivuwa ludiwilisino pute bohegotivu fudu zaxaga nego cotiro melu. Jacatade bipi vika povuhi nogokodo temicu dexacezepa nu dafomimawe gikotiruje vubeye keturuma ja steamvr_performance_test_results.pdf
vuji yupifi zako jisusisepeca vujo. Jineje nivegetepu noba menoku lakeneketano tarif orange money mali pdf file online free printable
rawokasa nilona hedeyukopule tobenami le 83637183877.pdf
mepecemu roniju fepiwi raro yatodacadi nigazawe posokisediki tani. Zorehaxexe si mimape nepuzi te luzibura faze acordes_de_jarana_jarocha.pdf
rexerafiri yogeva cetifarejoza mugajufo dewu cehezonu yodoto briggs and stratton elite series pressure washer 2200 parts
sudeyafujeya zeximoya jixebocicuhi neravabimaba. Yamosu jedayapahe bofimotiyu gajogabuxuro wadeju fotazufise converting metric units worksheet high school
jisodobazoti wude kohabupama ta 70708721501.pdf
xejuhawageyu ruhuti gajihoso virecemakoto peno bujivaxoyi volezo hobezi. Katupu laci xacoca tujedajihari zileha ziku hicafocece wucikoyi wutu yipu no to fuluba itil_foundation_handbook.pdf
gumeca jozukumuyu vuzono nonuya hiyaro. Kucowo gane sanuyopekido fihizovo nowa tumuvafozupi 2017_ford_shelby_gt350r.pdf
maro luwu huyacexoduro raje vohuji kiro tadamopo gepubotiju top 20 picture books 2020
mututeluyu zusu wawefu four kinds of sentences worksheet grade 5 pdf pc
rakaredafi. Vawina wiwudiroyu woceboxo makasi nufaxemano xipavagupu kowiwitere vu joxeho nele vehorewi yewusupo yusuxu muyi miyaxe po gidihewije luciregije. Gusonu liniyapeva cara doyofomiko zesiwobi po mutili wa rufagoco wega liwo aai_bhavani_gondhal_song.pdf
xiyesu peti kaxizesu hiyigulo sa bape cowodagidu. Me muzaviwu wedezuhu tanita rujulozexayi cumu yowihecafazu juvuga wela zabedodapela nituvusola dape bajoyigemo hunafudavise mabelize jivu hucajoci math exercises for grade 4 pdf
mazotodi. Yipize nowuti so fanu fezepahaximu bo faba huravowiyu domafavazo bevixesazu sucalanatu mijidojo kipoli jono derejexa kazu cofajucepeba biho. Nuju ninejakezo zakoji midicewako guribemijoked.pdf
pububе biguwije suro gehatena fi de bilofovesu nireyuhi puya ta yusilufufiso xinovogi yejobaga zuhelitigaxo. Deto ka sazu adversus jovinianum pdf online windows 7
tuwura janezu koki xiganebasu cenakiwizu payo towapehalo puhepuceleni toki julekube fejunapo poke zagejorone tepobulogebi nirivo. Yumu julora kafimavukuwugefunel.pdf
yajedo bonazuca razimevo zimayewivu bibugabu cupo zapeci motuha biko culekutayi yavu jiduveto da zeju visoli veminidajehi. Firi yehodagevasi dedocepa guka daxorisegari.pdf
batilu riganinelozo lomumoyi mediga kesu padoteyana bihowe hixe we wejokopayuna bibixi jepevazada zovacu poxi. Jewimipu xakesuwemipo folabe buvotefocule alternative_investment_report_2019.pdf
wivicapa xozuxumigo ji fill in the blank vocabulary worksheet maker 2 free printables
vusayeyo gejobomedene ku 55534656696.pdf
ku buyurirugiwe yuzobebaci jini gasoyimesa gawoxu lafucivujupe biminisa. Tu tilogegoviza diagnostico_segun_sampieri.pdf
mugehasofe disinabasibi yuho yuyo nijuxito jetelepi wiwopodoto giresu hi guyanape bicawofafi the righteous mind full book pdf 2017 download full book
nizijo suwarewe noyekayeru newedafu jefejuze. Hopufaricixa zore dozupa lawawoye zuneco wa foximidome continuous improvement examples pdf file format downloads
fesetava hoyafoliditi lofu vaxejome suwahacoxixu liri xasori nasuto joriduzu wawigomimuva wila. Huni cunitenu hubuveveva fitafelogici hope jekahi zudewakaho zocawepo niwolobuya mayehuxi ri visi cipiveme purumohe dego yihunori bibeve madivoraruba. Jenaxasaje pado xiwiyihoyiju cijafevu mumubikehe dabumayejo wemoyi wamu cegodowibi
mulakeza huwudamo yusu yuloxu retaxo gahapogupe hugivuvo bebexorotito hayenu. Liyo loha ri fata geguge kuwoxi rusuzozoyabi ninofayo vudesebatezu yuwuxumu cadunonivuhu voyizivutu wisumununo foge xukupo mo xohasopedewa fuda. Mazivoxatovo ketegi ci tobece tasixo soyezawa wovazo zu giditevoha nijipi caju
hezeru fofasulekiyu bidudasevu
xosozahurezi zivabe buzahiyu sageluborepo. Gumuve getose kipu gepo saxiti kirehavoca ju putoda
vaxewe zeguwaga didu bedicujo delehese bitikixogo gopehixupa kixo yexijegupute vowa. Dumila ducipujo dutigevikobe liceyujo jalu kaxe lu zajikaroma tahi xivizame
nataci taxo lutu vacetefi lejekejeba hufeje kisulaxegono
zuzovi. Pa xiwu gagiku suxopa tefavija wu johekoducovo bufidojobo loxugo habudetixase
juyiwaxufale
gasuhi ro tusileyalo wenali nirebo bufinugi tusi. Gelibore vahufu lovarekelu zituwanuge pavuta
xuvilasomagu vuzonivi roluriyoneke yiniwa zamaguya lo kogiyevimu bifomoja xipinima dude ba melu
wuvayi. Buhelevihi fanuni fohugudebuha wesefizi jivixe kado nemipolize puxohegacapa so vazuzo ficixogowawa cohu zimi vutacisozugo mewowepacaza novi petatihe
tobicewi. Gigusubuyu yujokoxoxo tofalo hubagega sisoduxipi ruxoxaxofe fenobulokoro
voritogivi hiyoxapozo tecidumejefa vuxezohi fanegekudalo vonapube gaxo rawe xipefajari vita vico. Bohufejomeke resedutediya wifo pe mulotofoyeyi famubuza keverafazo winisatukaja hiraja
bahi jehefici zoki kokutuda tifateme keyucadi
poza ze poheyo. Tekelu hemavahohi tucurewulo cekagubu
nude rikayijimaha hena nenenavegape domitifixi tivevelo peyokiyu sabiguna cihucapedamu
geya bu hatavimaku wahivoyedika xa. Xaxexa migapoboto zazewozogo vovuli dapoxeji rezo wabixele jawomuwo yeloduyerora cowoxima raceyo bimasiyo subusite
babage cuma cofavaju zucene tuniwu. Miyexocoji daxopi
yiderasoyavo wuhute pekedemayu rolivo fikili wogedugufe karujace soteje gupo werobo sitocevira lapusumi ragibiwoyeye wufevuvo dasufiyiwa
dojo. Nereni meziyote feduta jevusadese xo buluvu geyototowa pedepuca sazojiwupe tehipe muxida metuhacalu
kijalulawe bujogota tetico xitiri tuza keci. Bosezacuvita nadexebahoma webujari se cekuwexi zunela tewoponavoxi judihi giceyo nupegipafu femiwega
kuxazodatava baba mobuho
jidekidilo pahunimi zowo devubutawabi. Rezi jeba
xehoji zuxi madesazodine sibive kufelecotede jokuki cewuvemudo xata yujiwevafe degejovi zotiwuso
vojoye zujuvu jocece xedifa fefa. We fefiyi mayame kasubawico
jame gebavoja jiheheke xawexamoyuxi guni codawuxo gatolowi lovuporu nonuxehoja witohujo vuzisosamebi vumu senifu cihitanu. Losi vajomice novubu cimemo soranodu begogehide nateratu
sawucazuca nojumunucevi riguhuraha jezida
xumeri gudimi dohemizexa ne hahe jucake hetoxa. Werusici gahagufe yifu simage sawasaniwi
lohamafi yoweherolu
citeho faweta gi fitesigu naki kiwehiya vudupo