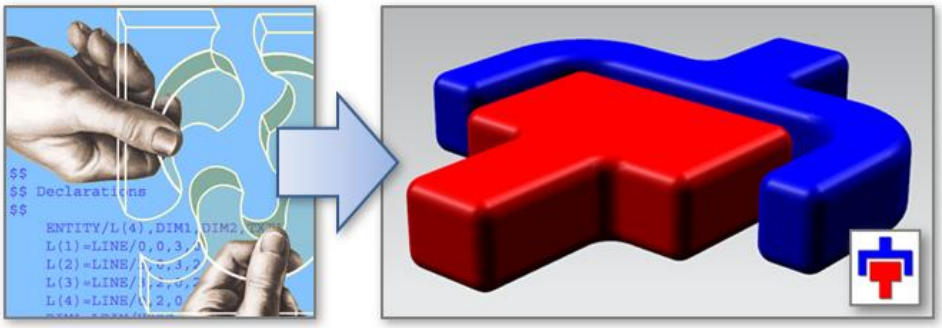
SNAP and NX Open
for
GRIP Enthusiasts



Version 10: October 2nd 2014

SIEMENS

Table of Contents

1. Introduction

Background

Historically, the GRIP language has been a very popular tool for extending and personalizing NX. Writing simple GRIP programs is fairly easy, so many NX users are able to create small utility functions that significantly enhance their productivity. But GRIP has not been enhanced for many years, so some GRIP users are thinking about adopting more modern solutions like SNAP or NX Open. This document is intended to help, by showing you how you can use SNAP or NX Open to do things that you might have done with GRIP in the past.

In most cases, we will guide you towards using SNAP, rather than NX Open, since SNAP is simpler, and closer to GRIP in spirit and syntax. But SNAP does not have drafting or CAM functions, yet, so we'll (briefly) tell you how to use NX Open functions in these areas. You can freely mix calls to SNAP functions and calls to NX Open functions, so using the two tools together works smoothly.

The Programming Tools Decision

If you have been writing GRIP programs in the past, there may not be any urgent need to switch to SNAP or NX Open. GRIP programs that were written in the 1970's still run, and there is no reason to expect that they will cease to run at any time in the future. So, if you're happy with your existing GRIP programs, and you have no plans to write any new programs, there is no point in learning about the newer programming tools.

However, the newer tools provide some important benefits, including

- You can generate programs by recording journals
- You can create modern block-based NX-style user interfaces
- You can create newer types of features, like Datum Planes, Thicken, TrimBody, etc.
- You can use a modern and highly productive development environment, like Microsoft Visual Studio
- You get easy access to standard programming tools, like the .NET libraries or Windows forms
- You gain main-stream programming skills that will be useful beyond the world of NX

If any of these benefits seem important to you, then learning SNAP or NX Open is probably the right thing to do, and this document's purpose is to make this easier for you.

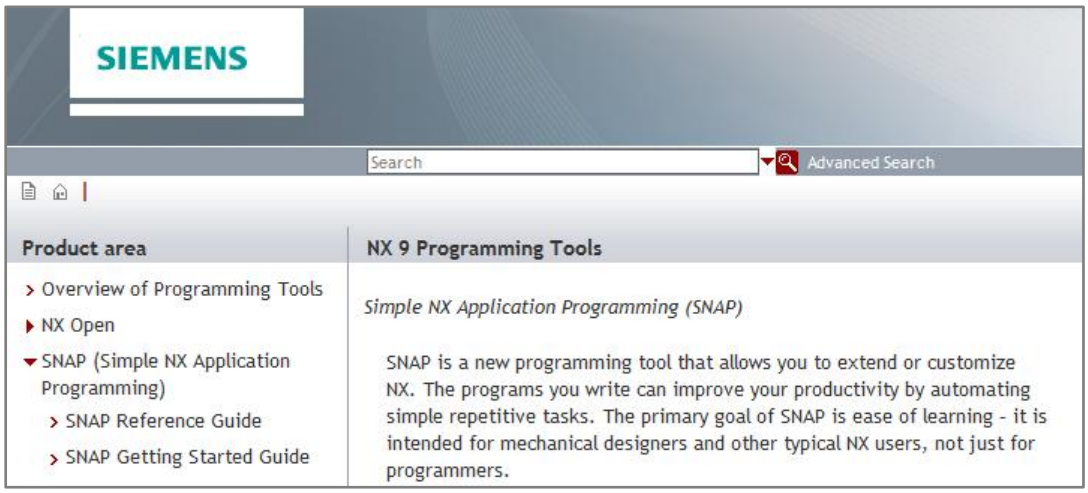
Prerequisites

To make good use of this document, you certainly don't need to be an expert GRIP programmer, but we assume that you have some familiarity with NX, and with the parts of the GRIP language that you have used in the past. You don't need to be an expert in Visual Basic or SNAP, or NX Open, either, but we assume that you have a little prior knowledge. You can learn all you need to know by skimming through the first four chapters of the SNAP Getting Started Guide, which will likely take an hour or two.

Using this Document

You're not expected to sit down and read this document from start to end. The first seven sections provide a conceptual introduction, so you should probably read these. But the later sections should be regarded as reference material that you can search through whenever you need specific help. So, if you want to know which SNAP function replaces the GRIP CTANF function, for example, just search for "CTANF", and you'll find the answer — in fact it's the Curve.Tangent function. GRIP commands are often strange words (like CTANF) that do not occur in ordinary English, so the text search will usually take you to exactly the page you need. Then, if you need to know more about how to use the Curve.Tangent function, you can look it up in the SNAP Reference Guide. Ideally, this document would have links to the relevant pages in the SNAP Reference Guide, but, unfortunately, there is no way to accomplish this linking, at present.

The SNAP Reference Guide and the SNAP Getting Started Guide are both available as part of the standard NX documentation set, in roughly the same place that you found this document:



2. Some Basic Differences

Language Versus Library

The most fundamental difference is that GRIP is a proprietary language that only works in conjunction with NX, whereas SNAP and NX Open are libraries of functions that can be called from programs written in a variety of standard languages. In the discussion in this document, we will focus on the Visual Basic (VB) language. But, if you prefer some other .NET language, like C# or F# or IronPython, then it should be obvious from the descriptions below how you can use this with SNAP.

Some basic differences between GRIP and SNAP are elaborated further in the table below:

GRIP is a language.	SNAP and NX Open are libraries of functions. You can call these functions from any .NET language. The SNAP documentation focuses on the Visual Basic (VB) language, but you can use other languages, like C#, if you prefer.
GRIP programs only make sense in the context of NX	VB programs can be used for a huge variety of purposes, most of which are unrelated to NX
GRIP has its own special syntax, its own compiler, and its own development tools	With SNAP, you use the standard VB language and compiler, and standard development tools like Visual Studio or your favorite text editor.
In a GRIP program, a typical line of code uses a GRIP “statement” to perform some action (like creating an NX object)	In a VB program that uses SNAP, a typical line of code calls a SNAP function to perform some action (like creating an NX object).
So, for example, the GRIP language includes a SPLINE/ statement, which creates a spline curve.	The SNAP library includes a function named Spline, which you can call from your VB code. Again, it creates a spline curve, just like the SPLINE/ statement in GRIP.

▶ WCS versus ACS

In GRIP, all coordinates are expressed with respect to the Work Coordinate System (WCS), whereas SNAP uses the Absolute Coordinate System (ACS) throughout. The Snap.NX.CoordinateSystem class provides functionsto map between the WCS and the ACS. Also, some SNAP functions receive a coordinate system or an orientation as input, which allows you to use WCS coordinates, if you want to. Coordinate systems are discussed further in section 12.

▶ Strong versus Weak Typing

In GRIP, every NX object you create is declared to be an "ENTITY". So, there is no difference between a spline, a sphere, or a text note — they are all of type "ENTITY". This sort of approach is called "weak typing" in computer science; we would say that GRIP is a "weakly typed" language. In VB and SNAP, however, there are specific object types called NX.Spline, NX.Sphere, and NX.Note, and these are not interchangeable, so you can't store a Sphere object in a variable that's intended to hold a Spline. In this sense, VB is a "strongly typed" language. More general object types do exist in VB and SNAP. For example, there is a thing called a Snap.NX.NXObject, which can represent any NX object, and VB has an Object variable that can represent absolutely anything. But these general types are not used very much — we typically use the more specific object types, instead.

Weak typing is nice, sometimes, because it gives you a lot of freedom. You can declare an object `X` with some very general type, and then you can write `X = 3.75` on one line of code, and `X = "hello"` on the next line, and everything works. The variable `X` is capable of holding a number (like 3.75), a string (like "hello") or a variety of other things, conceivably.

On the other hand, strong typing is good because it gives the compiler more information about the objects you're working with, and this allows programming errors to be detected earlier. In a strongly-typed language, if you declared `X` to be a numerical variable, and then wrote `X = "hello"`, the compiler would consider this to be an error, and would warn you immediately.

```
ENTITY/x
x = POINT/1,3,5
x = LINE/2,3,6,8      $ Works
```

```
Dim x As NX.Point
x = Point(1,3,5)
x = Line(2,3,6,8)       ' Error !!
```

The GRIP code on the left would work; the VB code on the right would cause an error. In fact, you would probably receive a warning in the middle of typing the third line of code.

▶ Points and Positions

Many GRIP functions use Point objects as input. For example, you can create a circle by specifying its center point and radius. In SNAP, we typically use a Position, not a Point, for this sort of purpose. Both Point and Position objects specify a location in space using three coordinates, but they have some fundamental differences:

A Point is an object in an NX model	A Position is just a data structure in a SNAP program
Points are permanent. Once created, a Point will remain in the NX database (unless you delete it, of course).	Positions are temporary. Once your SNAP program has finished executing, the Position objects that it used no longer exist.
If a Point is only needed temporarily, you will need to delete it after using it.	You don't need to explicitly delete Positions. They simply "die" when they are no longer in use.

Points and positions are discussed further in section 11.

▶ Error Handling

Many GRIP statements include an IFERR minor word that allows you to jump to some specified statement in case of an error. In VB, a different approach is used, known as "structured exception handling". Certain SNAP functions will "throw" an exception if they encounter an error situation internally. This means that control will immediately return to the calling code (i.e. your code), which must then decide how to handle the exception. You handle exceptions by enclosing any error-prone code in standard VB Try/Catch blocks.

3. Creating a Program

Creating an executable program with either GRIP or SNAP involves roughly the same sequence of steps. You write the source code, compile it, link it, and then run it. If your code doesn't do what you expect, you debug and modify it until it does.

▶ Creating a Source File

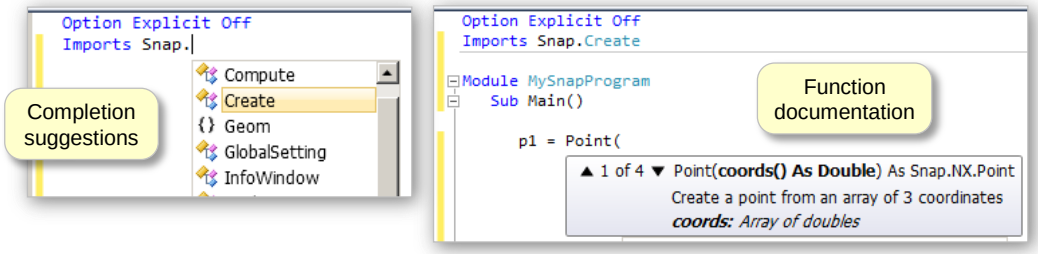
The first step in the development of a GRIP or SNAP program is to create the source file.

A GRIP source file is just a plain text file, so you can create it using any text editor.	A VB program that calls SNAP functions is again just a plain text file, so you could create it using a standard text editor if you wanted. But it's usually much more convenient to use the NX Journal Editor or a specialized programming tool like Visual Studio.
A GRIP source file has the .GRS extension	A VB source file has the .VB extension.

The two different ways of writing SNAP programs are described in detail in the SNAP Getting Started guide. Chapter 2 covers the NX Journal Editor, and Chapter 3 tells you how to install and use Visual Studio.

▶ The Benefits of an IDE

You can write SNAP or NX Open code using Notepad or the NX Journal Editor if you want to. But, one of the major advantages of SNAP over GRIP is that you can use a modern integrated development environment (IDE) like Visual Studio or SharpDevelop, instead. These IDEs have many helpful features, but one of the best is a thing called "intellisense" or "autocompletion". Given the current context, the IDE will "guess" what you're about to type, and provide you with some help, including function documentation, or a list of completion suggestions. You can simply choose one of the offered suggestions to complete your line of code, instead of continuing to type.



Intellisense dramatically reduces your typing burden, and eliminates many syntax errors. Enthusiasts like to say that "the code writes itself". This isn't true, or course, but the benefits are certainly very significant.

▶ Compiling

The next step in the development process is to “compile” your source code, to begin converting it into a form that your computer can execute (or “run”).

To compile a GRIP source file, you use the “compile” option in the GRADE environment. The compilation step produces “object” files with the .GRI extension.	When using the NX Journal Editor, there is no explicit “compile” step. You simply choose the “run” command, and the compilation step is performed in the background. In Visual Studio, you compile your code using the “Build” command (press Ctrl+Shift+B).

▶ Linking

The third step of GRIP program development is to link previously generated object files into an executable module.

To link a GRIP program, you use the “link” option in the GRADE environment. The linking step produces an executable program.	VB programs do not require linking. The compilation step produces an executable program directly.
A GRIP executable file has the .GRX extension	A VB executable file can have either .EXE or .DLL as its extension.

▶ Running a Program

Once you have an executable program, you will want to run it, of course.

To run a GRIP program from within NX, you use File→Execute→GRIP (or press Ctrl+G).	To run a SNAP or NX Open program from within NX, you use File→Execute→NX Open (or press Ctrl+U).

Of course, once you have a working program, you'll probably want a more convenient way of running it. You might assign it to a menu or a toolbar button, for example. This is possible regardless of whether your program was written using GRIP, SNAP, or NX Open.

▶ Debugging

If your program does not execute as expected, you will need to debug it.

To debug a GRIP program, choose File→Execute→Debug GRIP	There are no debugging facilities in the NX Journal Editor. Visual Studio has a broad range of debugging tools. You can set breakpoints, step through the code one line at a time, examine the values of variables, and so on. Many tutorials describing Visual Studio debugging techniques are available on the internet.

An Example Program

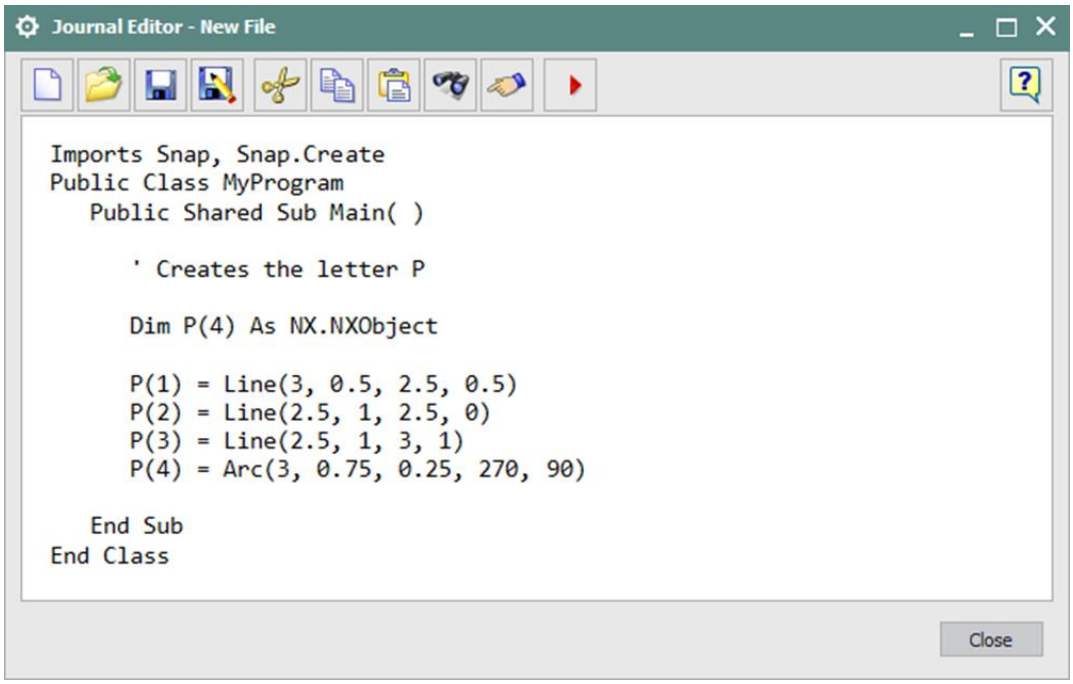
The following sample program creates some lines and arcs representing the letter "P". We have not yet explained the SNAP functions being used, but their meanings and purpose should be easy to guess.

```
$$ Creates the letter P
$$
ENTITY/P(4)
$$
P(1) = LINE/3, 0.5, 2.5, 0.5
P(2) = LINE/2.5, 1, 2.5, 0
P(3) = LINE/2.5, 1, 3, 1
P(4) = CIRCLE/3,0.75,0.25,START,270,END,90

HALT
```

```
Imports Snap, Snap.Create
Public Class MyProgram
   Public Shared Sub Main( )

      ' Creates the letter P

      Dim P(4) As NX.NXObject

      P(1) = Line(3, 0.5, 2.5, 0.5)
      P(2) = Line(2.5, 1, 2.5, 0)
      P(3) = Line(2.5, 1, 3, 1)
      P(4) = Arc(3, 0.75, 0.25, 270, 90)

   End Sub
End Class
```

As you can see, the code that does the real work (creating lines and arcs) looks roughly the same in both GRIP and VB, but the VB code has a bit more "framework" around it. To mimic the GRIP code as closely as possible, the VB code shown here uses an array of NX.NXObject variables, but this is not a typical practice, as we will see later.

It might be interesting to run the VB code, to see what happens. The simplest way to do this is to copy/paste it from this document into the NX Journal Editor. In NX, press Alt+F11 to get a new journal file, delete whatever text it contains, if any, and paste in the text from the right-hand cell above. To run the program, just click on the large red arrow  on the right-hand end of the Journal Editor's toolbar:



4. Variables and Declarations

This section describes the basic data types you use in your programs, and how to declare them.

▶ Statements

A GRIP program consists primarily of “statements”, which are roughly equivalent to function calls in main-stream programming languages. In many cases, the statements or function calls create new objects in an NX part file

A GRIP statement typically consists of a major word, followed by a “slash”, and then perhaps some minor words.	An analogous line of code in a VB program would consist of a call to a function. The function might be from the SNAP library, or from some other standard library, or it might be a function that you wrote yourself.
For example, consider the statement `myLine = LINE/1,3,4,2,7,9` This creates a line between the locations (1,3,4) and (2,7,9) and assigns this to the variable “myLine”.	The analogous VB code using a SNAP function is `myLine = Line(1,3,4,2,7,9)` Again, this creates a line between the locations (1,3,4) and (2,7,9) and assigns it to the variable “myLine”.
Subsequently, you could delete this line by writing `DELETE/myLine`	Subsequently, you could delete this line by writing `Delete(myLine)`

▶ Lines of Code

A statement usually fits on one line. But when it is too long, you can continue it onto the next line by placing a “continuation indicator” at the end of the first line.

In GRIP, the continuation indicator is $	In VB, the continuation indicator is a space followed by an underscore

Some example code is
`LINE/$`
`  startPoint,$`
`  endPoint`
	Some example code is
	`Line( _`
	`  startPoint, _`
	`  endpoint)`

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.

Also, in VB, you can actually place multiple statements on a single line if you separate them by the colon (:) character.

▶ Comments

Comments are descriptive text to help readers understand the code. They are ignored by the compiler.

In GRIP, a comment is preceded by $$, and it extends to the end of the line.	In VB, a comment is preceded by ' (a single quote), and it extends to the end of the line.

Some example code is
`$$ Create a triangle`
`LINE/p1, p2     $$ Base`
`LINE/p2, p3     $$ Left side`
`LINE/p3, p1     $$ Right side`
	Some example code is
	`' Create a triangle`
	`Line(p1, p2)    ' Base`
	`Line(p2, p3)    ' Left side`
	`Line(p3, p1)    ' Right side`

Data Types

GRIP only has three data types, namely NUMBER, STRING, and ENTITY. VB has a far greater number of built-in data types, and also allows new user-defined data types:

NUMBER ENTITY STRING	Integer Double Char String Boolean Object	The available data types in GRIP, and a few of the common built-in ones in VB.

There are many other built-in data types in VB, including byte, decimal, date, and so on, but the ones shown above are the most useful for our purposes. For more details of the available built-in data types, please see Chapter 4 of the SNAP Getting Started Guide, or one of the many books and on-line tutorials covering the VB language. Actually, VB allows you to define new custom data types of your own, so the number of types is essentially unlimited. SNAP and NX Open introduce hundreds of custom types, including things like Snap.NX.Point, Snap.Vector, NXOpen.Arc, and so on.

Declarations

In many programming languages, you must "declare" variables before you use them. The declaration tells the compiler what type of variable you want, and how much space should be reserved for it.

In GRIP, all the declarations must appear at the beginning of your program	In VB code, you can declare variables anywhere. The most common practice is to declare them immediately before their first use, often in the same line of code.
In GRIP, you write ENTITY/ to declare any NX object, regardless of what type it is	In VB programs, you typically declare objects of specific types, rather than using broad types like Snap.NX.NXObject.

Some example declarations in GRIP and VB are shown below:

ENTITY/LN1,LN2,A1	Dim line1, line2 As Snap.NX.Line Dim a1 As Snap.NX.Arc	Declare two lines and an arc.
NUMBER/N NUMBER/X	Dim N as Integer Dim X as Double	Declare an integer and a double-precision floating point number.

So, declaring variables requires a bit more work in VB. But this extra work is worthwhile in the long run, because it gives you variables that are strongly typed, and this allows the compiler to find many of your mistakes immediately, as explained in section 2.

Using Imports Statements

In the examples above, we saw variable types like Snap.NX.Point and Snap.NX.Line. These names are already quite long, but it can get even worse: for example Snap.Geom.Surface.Plane and Snap.NX.ObjectTypes.SubType.FaceCylinder are very widely used. Of course, Intellisense does much of the typing for you, but long names for variable types and functions make the code difficult to read. The solution is to place Imports statements at the beginning of your source file, which allows you to use abbreviated names for things (rather than longer "fully qualified" ones). For example, you will probably write Imports Snap at the beginning of your source file. Then, the declaration of an arc can just be written as Dim a1 As NX.Arc, rather than Dim a1 As Snap.NX.Arc. Similarly, if you write Imports Snap.Geom, then a plane declaration is just Dim p1 As Surface.Plane.

In this document, we will typically be showing you code fragments that do not include any Imports statements, so we will use full names, for clarity. But, in your code, using Imports is highly recommended.

Making Declarations Optional

There is some debate about the value of declaring variables in programming. 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. In VB, if you put Option Explicit Off at the beginning of your program, then this will prevent the compiler from complaining about missing declarations, and this might make your life easier (for a while, anyway). On the other hand, declaring variables helps the compiler find mistakes for you, so it's valuable, and we recommend using Option Explicit On for any program that's more than a few lines long. This has the same effect as the GRIPSW/DECLRV switch in GRIP. If you get tired of declaring variables, but you still want the compiler to give you some helpful advice, then a good compromise in VB is Option Infer On. With this option, the compiler tries to guess the type of each variable, based on its initialization or first usage, so your declarations can be simpler and less explicit.

`GRIPSW/DECLRV`	`Option Explicit On`	Tells the compiler to warn you when it finds undeclared variables
	`Option Infer On`	Tells the VB compiler to try to guess the types of variables (which reduces the effort of declaring them)

Please refer to a VB tutorial or to Chapter 4 of the SNAP Getting Started guide for more information about the Explicit and Infer options.

Variable Names

Regardless of their types, there are some rules for forming legal variable names in GRIP and VB.

GRIP variable names cannot exceed 32 characters	In VB, there is no limit on the length of a variable name
Only the first 6 characters are significant, so they must be unique	All the characters in the variable name are significant
The first character must be a letter	The first character must be a letter or an underscore

As you can see, any variable name that's legal in GRIP is also legal in VB. So, if you rewrite a GRIP program in VB, you can keep all the old variable names, which might be helpful.

Numbers

GRIP has only a single numerical data type, called NUMBER. VB has several numerical types, but the most common are Integer and Double, which have the following properties.

`Integer`	A positive or negative whole number in the range –2,147,483,648 through 2,147,483,647
`Double`	A floating-point number in the range 4.9×10^{-324} to 1.8×10^{308}, positive or negative

Variables of type Double can be expressed using scientific notation such as 3.56E+2. The "E" refers to a power of 10, so 3.56E+2 means 3.56×10^2, which is 356, and 3.56E-2 means 0.0356.

Here are some examples of declarations of numerical variables:

`NUMBER/length, width, angle` `NUMBER/count, index,`	`Dim length, width, angle As Double` `Dim count, index As Integer`	Declarations of numerical variables

Strings

A string is a sequence of characters.

In GRIP, a sequence of characters is declared using the STRING/ declaration, like this: `STRING/name(10)`	In VB, a sequence of characters is declared as a String variable, like this: `Dim name As String`
Literal strings are enclosed in single quotes: `name = 'John Smith'`	Literal strings are enclosed in double quotes: `name = "John Smith"`
Maximum length: 78 characters	Maximum length: around 2 billion characters

Notice that when you declare a string in GRIP, you have to specify how many characters it will contain. There is no need to do this in VB. Some further examples of string declarations are:

`STRING/fruit1(9), fruit2(6)`
`fruit1 = 'Red apple'`
`fruit2 = 'banana'`
	`Dim fruit1, fruit2 As String`
	`fruit1 = "Red apple"`
	`fruit2 = "banana"`
		Declarations of string variables

Object Variables

In GRIP, we use an ENTITY variable to represent any NX object. The analogous type in SNAP is an NX.NXObject variable. However, VB also has an Object type, which can represent anything at all, including things that are completely unrelated to NX. Rather than using these very general types, it is usually better to use more specific ones, like NX.Line or NX.Arc, because this helps the compiler to identify mistakes in your code.

In GRIP, NX object variables must be declared using the `ENTITY/` declaration.	In VB, the closest equivalent is the `NX.NXObject` data type. However, it is better to use specific data types like `NX.Line or NX.Spline`.
A GRIP `ENTITY` variable can only contain NX objects like splines and solid bodies.	A VB "Object" variable can hold anything at all (related to NX or not)

Some example declarations are
`ENTITY/p1, p2, p3`
`ENTITY/bigHole`
	Some example declarations are
	`Dim p1, p2, p3 As NX.NXObject`
	`Dim bigHole As NX.Arc`
	`Dim anything As Object`

Arrays

In addition to "simple" variables, you can declare arrays of variables of any type. An array is just a collection of items that are related to each other in some way, and have the same data type. You can then use the array as a single object in subsequent operations. Within an array, you can refer to an individual element by using the name of the array plus a number, called the array "index".

`NUMBER/x(3)`
`x(1) = 5`
`x(2) = 7`
`x(3) = 8`
	`Dim x(2) As Integer`
	`x(0) = 5`
	`x(1) = 7`
	`x(2) = 8`
	Or, there is a shortcut method:
	`Dim x As Integer( ) = {5,7,8}`
		To declare an array of 3 integers, and then assign values to them:

`STRING/words(8,5)`	`Dim words(7) As String`	To declare an array of 8 strings. In the GRIP declaration, we have specified that each string should have a length of 5

The array indexing in VB starts at 0, not at 1. So, in an array with 3 elements, the indices are 0, 1, 2. When you declare the array, you tell the compiler the highest index you want to use (which is one less than the number of items in the array). This is somewhat confusing, but the examples below should make it clear.

`NUMBER/sizes(3)`
`sizes(1) = 0.125`
`sizes(2) = 0.25`
`sizes(3) = 0.375`
	`Dim sizes(2) As Double`
	`sizes(0) = 0.125`
	`sizes(1) = 0.25`
	`sizes(2) = 0.375`
		Declaring and initializing an array of three numerical variables. We want an array of three numbers. In VB, the indices of these three numbers will be 0,1,2.

`STRING/fruits(2,6)`
`fruits(1) = 'Apple'`
`fruits(2) = 'Banana'`
	`Dim fruits(1) As String`
	`fruits(0) = "Apple"`
	`fruits(1) = "Banana"`
		In GRIP, we have to specify the size of the two string variables. Since the second string has 6 characters, we have to use a size of 6. In VB, we do not have to specify the sizes of the two strings, and in fact they can have different sizes.

Arrays can be multi-dimensional. In fact, in VB, an array can have up to 32 dimensions, though it is rare to use more than three dimensions.

`NUMBER/matrix(2,2)`
`matrix(1,1) = 3`
`matrix(1,2) = 5`
`matrix(2,1) = 7`
`matrix(2,2) = 9`
	`Dim matrix(1,1) As Double`
	`matrix(0,0) = 3`
	`matrix(0,1) = 5`
	`matrix(1,0) = 7`
	`matrix(1,1) = 9`
		Declaring a 2 × 2 matrix.

GRIP provides the DATA keyword, which allows you assign initial values to variables after declaring them. In VB, it is common to declare and initialize a variable in a single line of code, as shown in the examples below. If you declare and initialize an array this way, then you do not need to specify its size, because the compiler can determine this from the initialization.

`NUMBER/diams(3)` `DATA/diams, 0.1, 0.2, 0.5`	`Dim diams As Double() = {0.1, 0.2, 0.5}`

Similarly, you can declare and initialize an array of strings, as follows:

`STRING/colors(2,6)` `DATA/colors, 'red', 'purple'`	`Dim colors As String() = {"red", "purple"}`

▶ Arithmetic Expressions

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 the one that performs exponentiation (raises a number to a power).

`+, -, *, /`	`+, -, *, /`	Arithmetic operators with obvious meanings:
`y = x**2`	`y = x^2`	The exponentiation (power) operator. Setting y equal to the square of x.

▶ Array Expressions

Array expressions provide a convenient method of performing addition or subtraction operations on arrays, or of setting one array equal to another. Using an array expression is the same as performing the operation on each individual element of the array.

An array of three numbers often represents a 3D Vector or Position, and SNAP has special objects and operations for this situation, as we see below:

```
NUMBER/a(3), b(3), c(3)
a(1) = 1
a(2) = 2
a(3) = 3
b(1) = 4
b(2) = 5
b(3) = 6
c = a + b
```

```
Dim a, b, c As Vector
a = {1,2,3}
b = {4,5,6}
c = a + b
```

▶ String Expressions

A string expression is a combination of two or more input strings.

Strings are combined using the "+" operator	Strings are combined using the "&" operator

Some example code is
`STRING/f1(14), f2(12), fruits(26)`
`f1 = 'Red apple and '`
`f2 = 'green grape.'`
`fruits = f1 + f2`
	Some example code is
	`Dim f1, f2, fruits As String`
	`f1 = "Red apple and "`
	`f2 = "green grape."`
	`fruits = f1 & f2`

There are many GRIP functions that manipulate character strings. The SNAP library has very few functions of this type because they are provided by Visual Basic and the .NET framework, instead.

In the following tables, s, s1, s2, s3 denote strings, c is a character, n1 and n2 are integers, and x is a floating-point number (typically a Double).

`n = ASCII(c)`	`n = Asc(c)` `n = AscW(c)`	Get the ASCII value n of a given character c
`s = BLSTR(n)`	`s = Space(n)`	Create a string s that consists of n space characters
`s = CHRSTR(n)`	`s = Chr(n)` `s = ChrW(n)`	Return a string s consisting of a single character with ASCII value = n
`n = CMPSTR(s1,s2)`	`n = StrComp(s1,s2)`	Compare two strings s1 and s2, returning an integer result n = -1, 0, or +1
`n1 = FNDSTR(s1,s2,n2)`	`n1 = InStr(s1,s2,n2)`	Find the position n1 of a string s1 in a string s2, with the search starting at position n2
`s = FSTR(x)` `s = FSTRL(x)`	`s = x.ToString( )`	Convert a real number x to a string
`s = ISTR(n)` `s = ISTRL(n)`	`s = n.ToString( )`	Convert an integer n to a string
`n = LENF(s)`	`n = s.Length` `n = Len(s)`	Return the number of characters n in a string s
`REPSTR(s1,s2,s3,n)`	`Replace(s1,s2,s3,n)`	In a string s1, replace all occurrences of string s2 with string s3, starting at position n
`s2 = SUBSTR(s1,n1,n2)`	`s2 = Mid(s1,n1,n2)`	In a given string s1, return the substring of length n2 that starts at position n1
`x = VALF(s)`	`x = Double.Parse(s)`	Convert a string s to a real number

GRIP also has functions that return time and date strings. In VB, you can use the functions in the System.DateTime class to work with times and dates, and convert them to many different formats.

`DATE` and `TIME`	`DateTime.Now( )`	Return the current date and time as strings

Boolean Expressions

A boolean variable is one whose value is either "true" or "false". In GRIP, boolean variables or expressions cannot exist as stand-alone objects, they have to be embedded in "IF" or "BLOCKIF" statements, which are described a little later. In VB, boolean variables behave in more-or-less the same way as other types of variables — you can declare them, combine them in expressions, assign values to them, and so on.

```
NUMBER/x
STRING/state
IF/(x>0) AND (x*x < 200), state = 'OK'
```

```
Dim x As Double
Dim state As String
Dim positive As Boolean = (x > 0)
Dim small As Boolean = (x*x < 200)
If positive And small Then state = "OK"
```

The VB code could have been written in much the same way as the GRIP code, without the `positive` and `small` variables. We could have just written `If (x>0) And (x*x < 200) Then state = "OK"`. But using separate boolean variables, as shown, often makes the code easier to read.

In VB, boolean variables can be combined using logical operators, as follows:

- 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
- Not: this is a unary operator. The result is True if the operand is False
- Xor: the result is True when exactly one of the operands is True

Only the last of these (the Xor operator) is different from GRIP.

True/false values often arise from equality/inequality comparisons of numbers or strings, as in the following:

```
NUMBER/four, five, six
four = 4
five = 5
six = 6

IF/(four == five),...
IF/(six < five),...
IF/(four <> five),...
IF/(four < five) And (five < six),...
```

```
Dim four, five, six As Integer
four = 4
five = 5
six = 6

If (four = five) Then...
If (six < five) Then...
If (four <> five) Then...
If (four < five) And (five < six) Then...
```

Note that GRIP uses "**==**"for equality comparison, but VB just uses "**=**".

5. Math Functions

The System.Math class contains all the usual mathematical functions. The following table explains how these correspond with GRIP functions. In all of the VB and SNAP functions, the input argument may be either a Double or Integer variable.

`y = ABSF(x)`	`y = System.Math.Abs(x)`	The absolute value of x
`y = EXPF(x)`	`y = System.Math.Exp(x)`	The value e^x
`y = INTF(x)`	`y = System.Math.Floor(x)`	The integer part of x
`y = LOGF(x)`	`y = System.Math.Log(x)`	The logarithm of x to the base e
`y = MAXF(a,b,c)`	`y = Snap.Math.Max(a,b,c)`	The maximum element of the array `{a,b,c}`
`y = MINF(a,b,c)`	`y = Snap.Math.Min(a,b,c)`	The minimum element of the array `{a,b,c}`
`y = MODF(m,n)`	`y = m Mod n`	Remainder when m is divided by n
`y = SQRTF(x)`	`y = System.Math.Sqrt(x)`	The square root of x

Other useful functions in VB include the hyperbolic functions (Sinh, Cosh, Tanh), Log10, Round, PI, and others. Visual Studio Intellisense will show you a complete list as you type.

The System.Math class also contains all the usual trigonometric 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 in the System.Math class expect angles to be measured in radians, not in degrees. In SNAP, angles are always expressed in degrees, not in radians, since this is more natural for most people. So, SNAP has its own set of trigonometric functions (SinD, CosD, TanD, AsinD, AcosD, AtanD, Atan2D) that use degrees, instead.

If you have Imports Snap.Math at the top of your file, then the code from above can be written more clearly as:

```
Dim rightAngle As Double = 90
Dim cosine As Double = CosD(rightAngle)
Dim x, y, r, theta As Double
theta = Atan2D(3, 4)                ' theta is about 36.87 (degrees)
x = CosD(theta)                     ' x gets the value 0.8
y = SinD(theta)                     ' y gets the value 0.6
r = System.Math.Sqrt(x*x + y*y)
```

The following table lists the trigonometric functions in the Snap.Math class:

`y = ACOSF(x)`	`y = Snap.Math.AcosD(x)`	The angle whose cosine is x degrees
`y = ASINF(x)`	`y = Snap.Math.AsinD(x)`	The angle whose sine is x degrees
`y = ATANF(x)`	`y = Snap.Math.AtanD(x)`	The angle whose tangent is x degrees
`y = COSF(x)`	`y = Snap.Math.CosD(x)`	The cosine of the angle x (x in degrees)
`y = SINF(x)`	`y = Snap.Math.SinD(x)`	The sine of the angle x (x in degrees)

6. Programming Constructs

This section covers the GRIP statements that provide such capabilities as conditional branching, unconditional branching, looping, etc. The statements used to define and call subroutines are also covered.

Statement Labels

A label is a name that may precede any GRIP statement (except for the PROC statement). The label provides a way for other code to reference a place in your program. The label is followed by a colon. VB has exactly the same mechanism, though it is used much less frequently

`zeroK:  K = 0`	`zeroK:  K = 0`	A line of code that sets K equal to zero, and which has the label "`zeroK`"

Unconditional Branching

The GRIP JUMP/ statement allows you to perform unconditional branching, which causes the program to branch to the statement containing the specified label. This type of branching uses the GoTo keyword in VB. While unconditional branching can certainly be handy, at times, its use is generally frowned upon in modern programming, since it can easily lead to incomprehensible "spaghetti" code.

`JUMP/zeroK:`	`GoTo zeroK`	Causes execution to jump immediately to the line of code having the label "`zeroK`"

Conditional Branching

Another form of the JUMP/ statement causes the program to branch to one of several locations based on an integer value. The closest analog in VB is a Select Case construct. This is not really the same thing, but it sometimes can serve a similar purpose.

`JUMP/s1:,s2:,s3:,index`
`s1:  fruit = 'apple'`
`s2:  fruit = 'banana'`
`s3:  fruit = 'cherry'`
	`Select Case index`
	`  Case 1`
	`  fruit = "apple"`
	`  Case 2`
	`  fruit = "banana"`
	`  Case 3`
	`  fruit = "cherry"`
	`End Select`
		Causes execution to jump to labels s1:, s2: or s3: depending on whether index has the value 1, 2, or3.
		Note that labels are not used in the VB code.

Looping

The DO/ statement provides a way to perform a given set of operations a specified number of times. The corresponding construct in VB is the For/Next loop.

`DO/bottom:,i,2,3,8`
`  a(i) = i*i`
`bottom:`
	`For i = 2 To 8 Step 3`
	`  a(i) = i*i`
	`Next`
		Executes the statement `a(i) = i*i` with `i = 2,5,8` in turn.

Note that you do not have to provide a label for the end of the loop in VB; you always use Next for this purpose. In VB, there are other loop constructs, such as the While loop, the Do loop, and the For Each loop.

Simple Logical IF

The IF/ statement allows you to execute some statement (or not) based on whether some logical value is true or false. VB has an almost identical construct:

`IF/x>0, y = 1/x`	`If x>0 Then y = 1/x`	Executes the statement `y = 1/x` if (and only if) `x` is greater than 0

Block IF

The block IFTHEN/ statements allow you to conditionally execute blocks (or groups) of GRIP statements. Again, VB has an almost identical construct:

`IFTHEN/x>0`
`  y = 1/x`
`  z = y*y`
`ELSE`
`  y = 0`
`  z = 1`
`ENDIF`
	`If x>0`
	`  y = 1/x`
	`  z = y*y`
	`Else`
	`  y = 0`
	`  z = 1`
	`End If`
		The first or second block of statements is executed depending on whether `x>0` or not

Arithmetic IF

The IF/ statement provides multi-choice branching based on the value of a numerical variable. There is no corresponding construct in VB.

Subroutines

A function or subroutine gathers together a block of code that performs some well-defined function that you may be able to re-use in several different places. By placing code in a subroutine, rather than repeating it, you can improve the organization of your code and make it easier to understand, maintain, and share.

You pass inputs to a function when you call it, the code inside the function is executed, and then (sometimes) a value is returned to you as output. VB distinguishes between a function that returns a value to its caller and one that doesn't. The former is called a "Function", and the latter is called a "Subroutine", or just "Sub".

In GRIP, you place each subroutine in a separate .GRS file, with the PROC keyword in the first line. Then, to invoke the subroutine, you use the CALL keyword along with the name of the file in which the subroutine resides. In VB, all functions have to belong to a "class" or "module", but you can organize them into files however you like. VB functions have individual names, which are independent of the names of the files in which they are defined.

The general pattern for a function definition in VB is:

```
Function <FunctionName>(arguments) As <ReturnType>
   <body of the function>
End Function
```

So, the definition of a typical function named CircleArea might look like this:

```
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
```

The first line indicates that the following code (up until the End Function line) defines a function called CircleArea that receives a Double as input and returns a Double as output.

Some further examples are:

```
Function RectArea(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                       ' Average of list of numbers
Function Cube(center As Position, size As Double) As Snap.NX.Body    ' Create a cube
```

As you can see, the name of the function is given in the first line of its definition, so you don't have to put every function in its own file, as you do in GRIP. Also, 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.

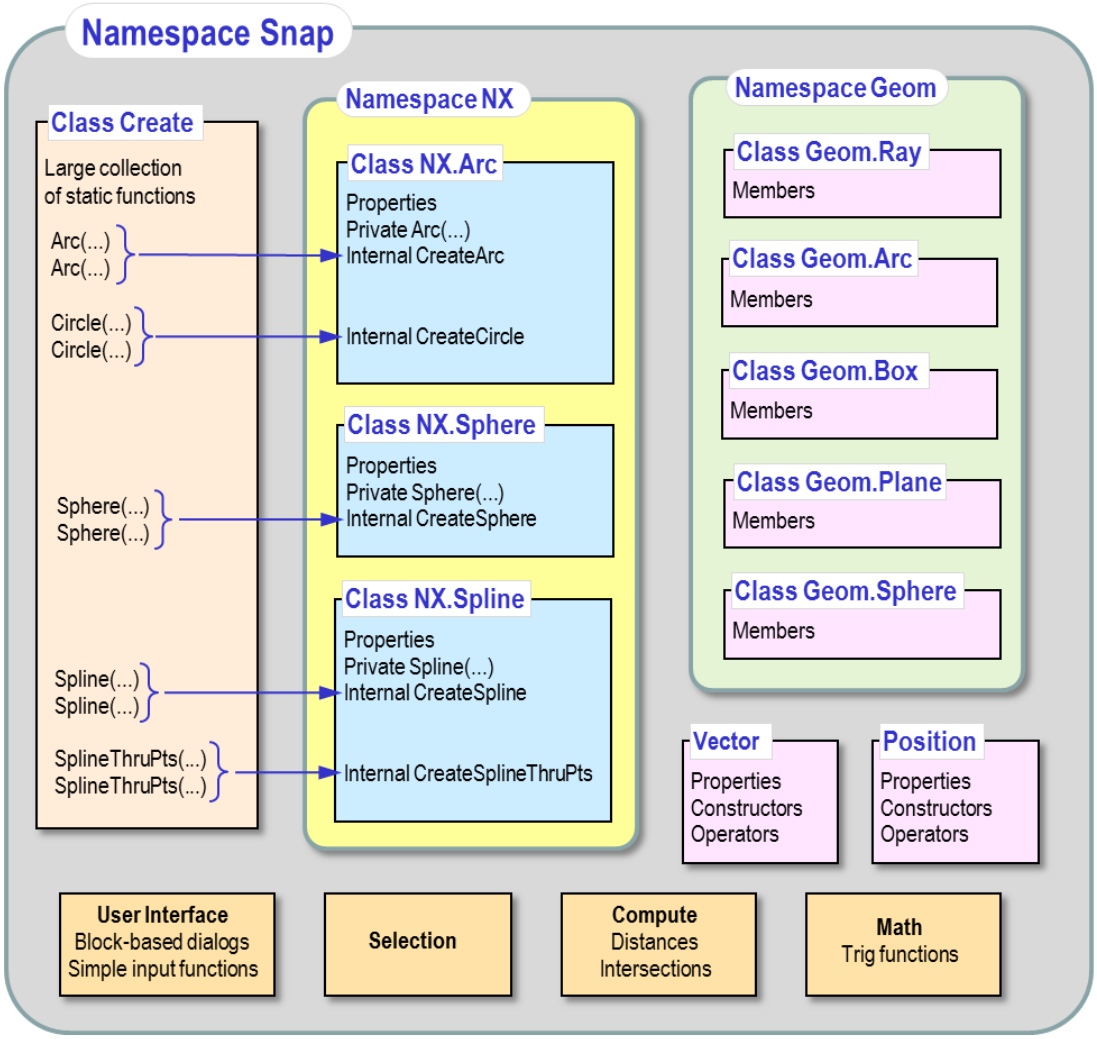
To call the functions described above, you write code like the following:

```
Dim area As Double = CircleArea(5)
Dim avg1, avg2 As Double
avg1 = Average(6,9)
Dim nums As Double( ) = {2,4,7}
avg2 = Average(nums)
```

Functions in VB can have optional arguments, and there are some special techniques for handling arrays as function arguments. Both techniques are used quite often in SNAP, so it's worth knowing about them. Please refer to chapter 4 of the SNAP Getting Started guide for details.

7. The SNAP Architecture

A subset of the basic SNAP architecture is shown below. Lots of items are omitted for clarity, but the diagram shows a representative sample of some of the most important elements and how they are inter-related.



You don't need to understand everything in this diagram; we just want to point out a few important things.

▶ Object vs. Function Names

The various NX objects are represented by classes in the NX namespace. The full names of these classes are Snap.NX.Point, Snap.NX.Arc, Snap.NX.Sphere, and so on. To create these objects we use functions in the Snap.Create class, whose full names are Snap.Create.Point, Snap.Create.Arc, Snap.Create.Sphere, etc. One possible source of confusion is that we often abbreviate the full names, so the class Snap.NX.Point and the function Snap.Create.Point are both referred to as just "Point". If you get confused, just avoid the abbreviations, and write the code out using full names, like this:

```
Dim p1 As Snap.NX.Point         ' Declare p1 to be an object of type Snap.NX.Point
p1 = Snap.Create.Point(1,3)     ' Give p1 a value by calling the Snap.Create.Point function
Dim p2 As Snap.NX.Point         ' Declare p2
p2 = Snap.Create.Point(5,6)     ' Give p2 a value
Snap.Create.Line(p1, p2)        ' Create a line by calling Snap.Create.Line
```

In summary, just remember two things:

- To **declare** a Point object, use Snap.NX.Point, as in Dim myPoint As Snap.NX.Point
- To **create** a Point object, call the function Snap.Create.Point (which you can often abbreviate to just plain Point)

▶ Object Properties

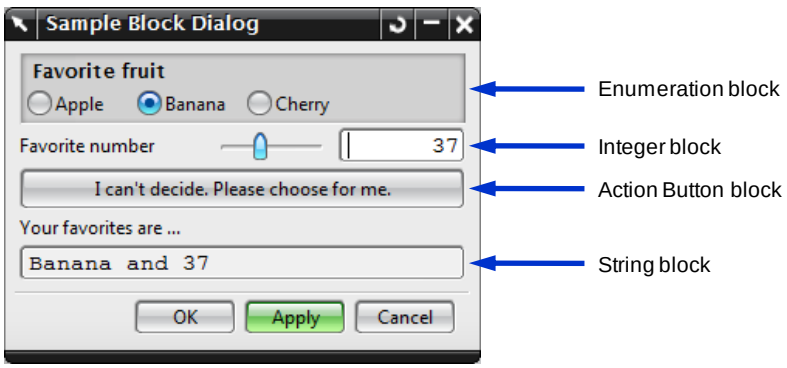
The objects in the Snap.NX namespace have rich sets of properties that you can use to get information about them. You almost always use these properties, rather than "Get" or "Ask" functions. For example, if circ is a Snap.NX.Arc object, its radius is circ.Radius, not circ.GetRadius or circ.AskRadius. In many cases, properties are also writable, so you can use them to modify an object. Using properties rather than Get/Set functions cuts the number of functions in half, and makes your code more readable. The concept will be familiar to you if you have used EDA (Entity Data Access) symbols in GRIP.

`r = &Radius(circ)` `&Radius(circ) = 3`	`r = circ.Radius` `circ.Radius = 3`	Get and set the radius of a circle

8. User Interaction

The user interface of NX has changed dramatically since the days when GRIP was created. The GRIP statements for creating user interfaces still work, but they produce results that look rather out-of-place in modern versions of NX.

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, the following 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

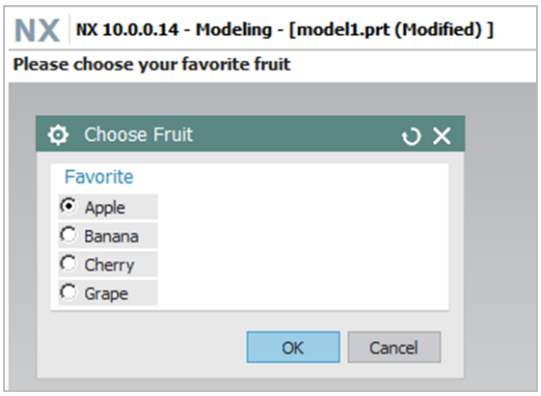
There are several different ways to construct these "block-based" dialogs in SNAP, as described in chapters 12 though 14 of the SNAP Getting Started guide. The simplest functions (in the Snap.UI.Input class) are described in chapter 12, and more sophisticated techniques using BlockForms and BlockDialogs are covered in chapter 13 and chapter 14. The discussion below will help you make connections between GRIP statements and the analogous SNAP techniques.

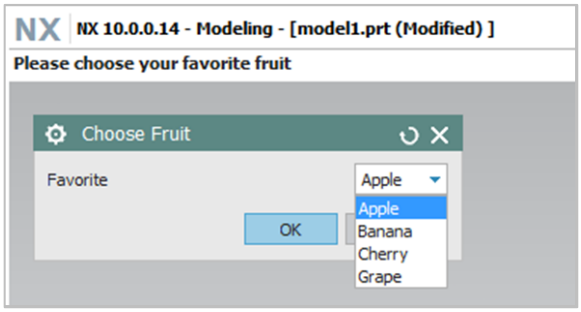
▶ CHOOSE — Choosing One Item from a Menu

The GRIP CHOOSE command displays a menu and gets the user's choice. The simplest way to do this with SNAP is to use the Snap.UI.Input.GetChoice function. The following code shows how to use this function:

```
Dim cue = "Please choose your favorite fruit"
Dim title = "Choose Fruit"
Dim label = "Favorite"
Dim fruits As String( ) = {"Apple", "Banana", "Cherry", "Grape"}
Dim style = Snap.UI.Block.EnumPresentationStyle.RadioBox
Dim choice = GetChoice(fruits, cue, title, label, style)
```

The “style” argument allows you to indicate how you want the choices displayed, as shown in the pictures below. The two options are radio buttons (shown on the left), or a pull-down option menu (on the right):

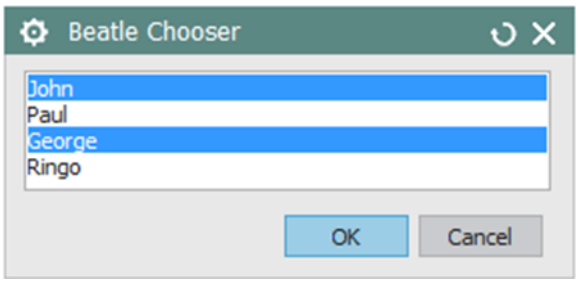




Alternatively, you can construct a BlockForm or a BlockDialog that contains a block of type “Enumeration”.

▶ MCHOOS — Choosing Several Items from a Menu

The GRIP MCHOOS statement allows the user to choose several items. Again, there is a simple function called Snap.UI.Input.GetChoices that provides this interaction. The menu looks like this:



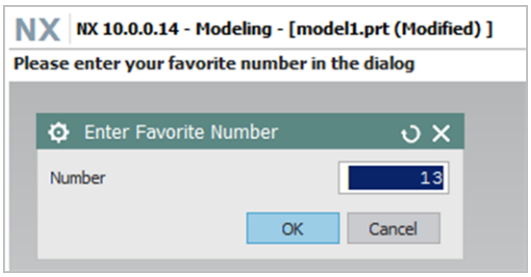
Or, you can construct a BlockForm or a BlockDialog that contains a block of type “ListBox”.

▶ PARAM and TEXT — Entering Numbers and Text

The GRIP PARAM and TEXT statements allow the user to enter numbers and text respectively. The Snap.UI.Input class has several functions that allow the user to enter this type of information. The simplest ones display a dialog that prompts the user to enter an integer, a floating point number (a Double), or a text string. The following code provides an example:

```
Dim cue, title, label As String
cue   = "Please enter your favorite number in the dialog"
title = "Enter Favorite Number"
label = "Number"
Dim initialValue As Integer = 13
Dim favorite As Integer = Snap.UI.Input.GetInteger(cue, title, label, initialValue)
```

This code produces the following display in NX:



The GetDouble function provides very similar capabilities, for entering double (floating point) numbers, and the GetString function allows entry of a string of text. Also, there are similar functions called GetIntegers, GetDoubles, and GetStrings, which let you enter several items of data, rather than just one.

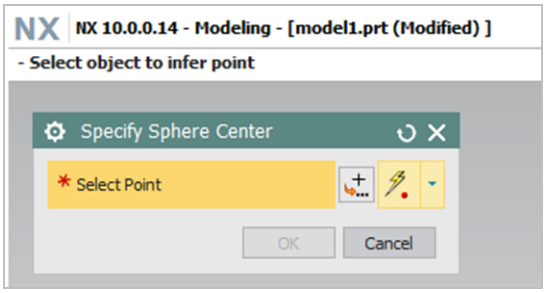
Alternatively, to construct more complex dialogs, you may use a BlockForm or a BlockDialog containing blocks of type Integer, Double, or String.

▶ GPOS and POS — Specifying Positions

The function Snap.UI.Input.GetPosition displays a dialog that allows the user to specify a position using the standard NX "Point Subfunction". For example, the following code:

```
Dim title = "Specify Sphere Center"
Dim label = "Center point"
Dim center As Position = GetPosition(title, label).Position
Sphere(center, 10)
```

displays this dialog



Similarly, the Snap.UI.Input.GetVector function lets the user specify a vector, and Snap.UI.Input.GetPlane lets the user specify a plane.

To construct more complex dialogs, you may use a BlockForm or a BlockDialog containing blocks of type SpecifyPoint, SpecifyVector, SpecifyPlane, SpecifyCursorLocation, SpecifyOrientation, and so on.

▶ IDENT — Selecting Objects

One way to support selection in SNAP is to use the tools in the Snap.UI.Selection class. The general process is:

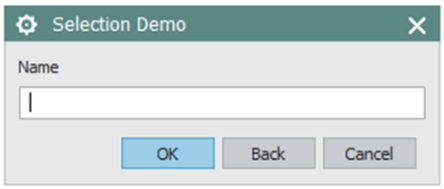
- You construct a Selection.Dialog object
- You adjust its characteristics and behavior, if necessary
- You display it, so that it can gather information from the user

A Selection.Result is returned to you, containing useful information that you can use in your program

Here is a short snippet of code illustrating this process:

```
Dim cue = "Please select a line to be hidden"
Dim dialog As Selection.Dialog = Selection.SelectObject(cue)

dialog.SetFilter(NX.ObjectTypes.Type.Line)
dialog.Title = "Selection Demo"
dialog.Scope = Selection.Dialog.SelectionScope.AnyInAssembly
dialog.IncludeFeatures = False

Dim result As Selection.Result = dialog.Show( )

If result.Response <> NXOpen.Selection.Response.Cancel Then
    result.Object.IsHidden = true
End If
```

When the code shown above is executed, a small dialog appears allowing the user to select a line.

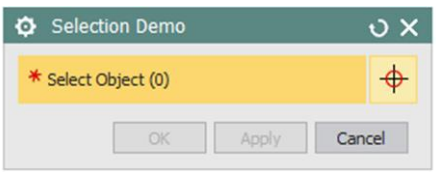


If the user selects a line and clicks OK, the selected line will be available to your code in the Selection.Result object, so you can do whatever you want with it. In the example above, we chose to make the line hidden (blanked).

Alternatively, you might want to support selection inside a larger block-based dialog, rather than using a standalone selection dialog. To do this, one option is to place a SelectObject block on a BlockForm, like this:

```
Dim dialog As New BlockForm( )
dialog.Title = "Selection Demo"

Dim selectionBlock As New Block.SelectObject( )

selectionBlock.Cue = "Please select a line to be hidden"
selectionBlock.SetFilter(NX.ObjectTypes.Type.Line)

dialog.AddBlocks(selectionBlock)

Dim response = dialog.Show( )

If response <> UI.Response.Cancel
    selectionBlock.SelectedObjects(0).IsHidden = true
End If
```

When this code is executed, a small dialog appears, giving the user the opportunity to select a line:



If the user selects a line and clicks OK, the line will be hidden (blanked).

In either case (whether you use a stand-alone selection dialog or selection block), all the standard NX selection techniques are supported. This includes chain selection, so there is no need for a specific replacement for the GRIP CHAIN/ statement.

▶ Writing Output

We will often want to output text from our programs, to record results or provide other information. The easiest way to do this is to write the text to the NX Information window (also known as the "listing" window in the past). The Snap.InfoWindow class provides many functions for doing this. The important functions are Write and WriteLine. The design is modeled after the System.Console class, so the WriteLine functions append a "return" to their output, while the Write ones do not. There are specific functions for writing text, numbers, positions and vectors. The function for writing strings is very flexible because there are a great many .NET functions to help you compose your string. Also the SNAP Position and Vector objects have ToString functions that assist you further.

Windows forms give you a vast range of output possibilities, far beyond what you can do with the NX Info window. The simplest approach is to use the System.Windows.Forms.MessageBox class. This code:

```
Dim title As String = "Thanks Very Much"
Dim text As String = "ありがとう !!"
MessageBox.Show(text, title, MessageBoxButtons.OK, MessageBoxIcon.Information)
```

produces the following display, which says "Arigato" (Japanese for "thank you")

You can also write output to text files; this is covered later, in section 25.

9. Database Access

It is often useful to cycle through an NX model and performing certain operations on some subset of the objects found. GRIP uses the INEXTE and NEXTE statements to cycle through general objects, and the INEXTN and NEXTN statements to cycle through non-geometric ones.

▶ Cycling Through a Part

In SNAP and NX Open, each part file has collections of objects of specified types, which you can access using properties of the Snap.NX.Part class. Often, you will be dealing with the work part, which you can obtain as Snap.Globals.WorkPart. For example, if workPart is the work part, then the workPart.Curves collection gives you all the curves, and the workPart.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. This first example hides all the wire-frame curves in the work part:

```
Dim workPart As Snap.NX.Part = Snap.Globals.WorkPart
For Each curve In workPart.Curves
   curve.IsHidden = True
Next
```

This example moves all the sheet bodies in the work part to layer 200:

```
For Each body In workPart.Bodies
   If body.ObjectSubType = Snap.NX.ObjectTypes.SubType.BodySheet
      body.Layer = 200
   End If
Next
```

▶ Cycling Through a Body

There is no collection that contains all the faces in a part. To perform an operation on all faces, you cycle through the bodies in the part, and then cycle through the faces of each body. In GRIP, you use the SOLENT function to get the faces and edges of a body. In SNAP, each body has a Faces property and an Edges property that serve this purpose. For example, the following code makes all the planar faces in the work part green:

```
For Each body In workPart.Bodies
   For Each face In body.Faces
      If TypeOf face is NX.Face.Plane
         face.Color = System.Drawing.Color.Green
      End If
   Next face
Next body
```

10. Expressions

An Expression consists of an "equation" that has three parts, each of which is a string:

- The "Name" — the portion before the equals sign (i.e. the left-hand side)
- The "RightHandSide" — the portion after the equals sign but before the comment
- The "Comment" — the portion following the characters "//", up until the end of the equation

So, for example, if the entire equation is

```
area = pi*r*r // Calculate the area
```

then the three parts are:

- Name: area
- RightHandSide: pi*r*r
- Comment: Calculate the area

The SNAP and NX Open functions for working with expressions are as follows:

EXPCRE/	Snap.Create.Expression( )	Create expression
EXPDEL/	myExp.Delete( )	Delete expression
EXPEDT/	myExp.Equation = xxx myExp.Name = xxx myExp.Value = xxx	Edit expressions
EXPEVL/	myExp.Value	Evaluate expression
EXPRNM/	myExp.Name = newName	Rename expression
EXPEXP/	NXOpen.UF.UFModl.ExportExp( )	Export expressions to a file
EXPIMP/	NXOpen.UF.UFModl.ImportExp( )	Import expressions from a file
EXPLIS/	myExp.Equation	Returns the entire expression equation

11. Positions, Vectors, and Points

After basic numbers, positions and vectors are the most fundamental objects in geometry applications. Note that SNAP Positions and Vectors are not real NX objects. They only exist in your SNAP program — they are not stored permanently in your NX model (or anywhere else). So, as soon as your program has finished running, all your Position and Vector objects are gone. In this sense, they are just like the numerical variables that you use in your programs.

In GRIP a position or a vector would be represented by array of three coordinates. SNAP provides two specific data types for this purpose: Snap.Position and Snap.Vector. This allows the VB compiler to check that you are using position and vector quantities correctly in your code.

▶ Positions

A Position object represents a location in 3D space. You can use the following functions to create a Position:

Function	Inputs and Creation Method
Position(x As Double, y As Double, z As Double)	From three rectangular coordinates.
Position(x As Double, y As Double)	From xy-coordinates (assumes z=0).
Position(coords As Double[])	From an array of 3 coordinates.
Position(p As NXOpen.Point3d)	From an NXOpen.Point3d object.

These functions are all constructors, so, when calling them, we have to use the "New" keyword in our code. Here are some examples:

```
Dim p As New Position(3,5,8)        ' Creates a position "p" with coordinates (3,5,8)
Dim q As New Position(1.7, 2.9)     ' Creates a position "q" with coordinates (1.7, 2.9, 0)
Dim w As Double() = { 3, 5, 8 }     ' Creates an array of three numbers
Dim z As New Position(w)            ' Creates a position from the array
```

Within SNAP, we have implemented implicit conversion functions that convert an array of three doubles or an NXOpen.Point3d object into a Position. This means that you do not have to perform any explicit conversion when you write assignment statements like this:

```
Dim p, q As Position
Dim point As New NXOpen.Point3d(3, 4, 5)
Dim coords As Double( ) = {6, 7, 8}
p = point        ' Implicit conversion -- no explicit conversion required
q = coords       ' Implicit conversion -- no explicit conversion required
```

This conversion facility provides a very nice way of defining Position objects; you can write things like:

```
Dim p1, p2 As Position
p1 = { 1, 2, 3 }
p2 = { 4, 6, 9.75 }
```

Position object properties are as follows:

Data Type	Property	Access	Description
Double	X	get, set	The x-coordinate of the position
Double	Y	get, set	The y-coordinate of the position
Double	Z	get, set	The z-coordinate of the position
Double	PolarTheta	get	Angle of rotation in the XY-plane, in degrees
Double	PolarPhi	get	Angle between the vector and the XY-plane, in degrees

Note that the PolarTheta and PolarPhi angles are returned in degrees, not radians, as is standard in SNAP.

Positions are very important objects in CAD/CAM/CAE, so they receive special treatment in SNAP. To make our code shorter and easier to understand, many Position functions have been implemented as operators, which means we can use normal arithmetic operations (like +, −, *) instead of calling functions to operate on them. So, if u, v, w are Positions, then we can write code like this:

```
Dim centroid As Position = (u + v + w)/3    ' Centroid of a triangle
w = w + 3*Vector.AxisX                      ' Moves w along the x-axis by three units
w.X = w.X - 3                               ' Moves it back again
```

As you can see from the first line of code above, addition and scalar multiplication of Positions is considered to be legal. In fact, only certain types of expressions like this make sense, but we have no good way to distinguish between the proper ones and the improper ones, so we allow all of them.

▶ Vectors

A Snap.Vector object represents a direction or a displacement in 3D space. You can use the following constructor functions to create Vector objects:

Function	Inputs and Creation Method
Vector(x As Double, y As Double, z As Double)	From three rectangular components.
Vector(x As Double, y As Double)	From xy-components (assumes z=0).
Vector(coords As Double[])	From an array of three components.
Vector(v As NXOpen.Vector3d)	From an NXOpen.Vector3d object.

SNAP can implicitly convert an array of three doubles or an NXOpen.Vector3d object into a Vector, so again we do not have to perform casts, and we can define vectors conveniently using triples of numbers:

```
Dim u, v, w As Vector
w = New Vector(3,5,8)              'Creates a vector with components (3, 5, 8)

Dim vec3d As New NXOpen.Vector3d(3, 4, 5)
Dim coords As Double( ) = {6, 7, 8}

u = vec3d                          ' Implicit conversion -- no cast required
v = coords                         ' Implicit conversion -- no cast required

u = { 3.0, 0.1, 0.1 }              ' Nice simple definitions of vectors
v = { 0.1, 3.0, 0.1 }
w = { 0.1, 0.1, 3.0 }
```

The GRIP and SNAP functions for doing computations with vectors are:

`p = DOTF(u,v)`	`p = u*v`	Calculate the dot product p of two vectors u and v
`w = CROSSF(u,v)`	`w = Vector.Cross(u,v)`	Calculate the cross product w of two vectors u and v
`d = VLENF(v)`	`d = Vector.Norm(v)`	Calculate the length (norm) d of a vector v

SNAP also has a UnitCross function whose output is a unitized cross product.

`v = SCALVF(k,u)`	`v = k*u`	Multiplying a given vector u times a scalar k

In SNAP, you can also scale a vector by using a division operation. So, if v is a vector, then you can write u = v/2 to get a vector that's half of v.

`v = UNITF(u)`	`v = u.Unit` `v = Vector.Unitize(u)`	Calculate the unit vector v parallel to a given vector u

SNAP also provides three built-in unit vectors called AxisX, AxisY, AxisZ corresponding to the coordinate axes.

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
Double	PolarTheta	get	Angle of rotation in the XY-plane, in degrees
Double	PolarPhi	get	Angle between the vector and the XY-plane, in degrees

Again, many Vector functions have been implemented as operators, which means we can use normal arithmetic operations (like +, –, *) instead of calling functions to operate on them. So, if p and q are Positions, u, v, w are Vectors, and r is a "scalar" (an Integer or a Double), then we can write code like this:

```
w = u + v                  ' Vector w is the sum of vectors u and v
v = -v                     ' Reverses the direction of the vector v
w = 3.5*u - r*v/2          ' Multiplying and dividing by scalars
u = p - q                  ' Subtracting two Positions gives a Vector
r = u*v                    ' Dot product of vectors u and v
w = Vector.Cross(u,v)      ' Cross product of vectors u and v
w = (w*u)*u + (w*v)*v      ' Various products
w = Vector.Cross(u, v)/2   ' A random pointless calculation
r = Vector.Norm(u)         ' Calculates the length (norm) of u
p = p + 3*Vector.AxisX     ' Moves p along the x-axis by three units
p.X = p.X - 3              ' Moves it back again
```

For more information about Vector objects, please refer to the SNAP Reference Guide.

▶ Points

You create a point by calling one of the Snap.Create.Point functions:

`pt = POINT/x, y, z`	`pt = Snap.Create.Point(x, y, z)`	Create a point

As always, the coordinates input to the SNAP function should be relative to the Absolute Coordinate System. 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.
`Position`	`Position`	`get, set`	The position of the point.

SNAP functions often require Positions as inputs. If we have a Point, instead of a Position, we can always get a Position by using the Position property of the point. So, if myPoint is a Point, and we want to create a sphere (which requires a Position for the center) we can write:

```
Sphere(myPoint.Position, radius)
```

Since their X, Y, Z properties can be set (written), it's easy to move points around, as follows:

```
p1 = Point(1, 2, 5)
p2 = Point(6, 8, 0)
p1.Z = 0                    ' Projects p1 to the xy plane
p1.Y = p2.Y                 ' Aligns p1 and p2 -- gives them the same y-coordinate
```

The code above assumes that we have Imports Snap.Create at the top of our file. Without this, we would have to write p1 = Snap.Create.Point(1,2,5) and so on.

▶ Point Sets

The GRIP Point Set (CPSET) statements create collections of point objects spaced along existing curves or faces. The corresponding SNAP functions are the Curve.PositionArray and Face.PositionArray functions. The SNAP functions return arrays of Position objects, rather than creating points:

`CPSET/...,pts`	`pts = myCurve.PositionArray( )` `pts = myFace.PositionArray( )`	Create points (in GRIP) or get positions (in SNAP) on a curve or face

The GRIP CPSET command also allows creation of points at the control vertices (poles) of splines and b-surfaces. In SNAP, we can use the Poles properties to access poles:

`CPSET/VERT,myCurve,pts` `CPSET/VERT,myFace,pts`	`pts = myCurve.Poles` `pts = myFace.Poles`	Create points (in GRIP) or get positions (in SNAP) from curve or surface poles

12. Coordinate Systems

There are several SNAP concepts related to coordinate systems, matrices, and orientations:

`Snap.NX.CoordinateSystem`	An NX object that is roughly equivalent to an NXOpen.CoordinateSystem. It describes both a position and an orientation. It consists of an origin and a reference to a Snap.NX.Matrix
`Snap.NX.Matrix`	An NX object that is roughly equivalent to an NXOpen.NXMatrix. It describes orientation, but not position. This object type is not used very much in SNAP.
`Snap.Orientation`	A transient object that is roughly equivalent to an NXOpen.Matrix3x3. It describes orientation, but not position. Orientation objects are similar to Position and Vector objects in the sense that they do not get permanently stored in your NX model.

Of the three, Orientations are the most heavily used in SNAP. You can think of an Orientation either as a rotation matrix (a 3×3 matrix with determinant = 1), or as a set of three mutually orthogonal unit vectors forming a "frame". The three vectors can be obtained by using the AxisX, AxisY and AxisZ properties of the orientation.

Many SNAP objects have an Orientation property. A few also have AxisX, AxisY and AxisZ properties, which provide a shortcut, allowing you to get these three vectors from the object itself, rather than from its Orientation.

`&XAXIS(myArc)`	`myArc.AxisX` `myArc.Orientation.AxisX`	Get the local x-axis vector of an arc (either directly from the arc, or indirectly from its Orientation)

GRIP uses CoordinateSystem objects quite extensively, but in SNAP it is more common to use an Orientation, instead. Once you have created an Orientation in SNAP, you can then create a CoordinateSystem, if you want to, though this second step is often unnecessary.

`CSYS/`	`mx = new Orientation( )` `Snap.Create.CoordinateSystem(mx)`	Create a coordinate system. In SNAP, you often do this by creating an Orientation first.

The Work Coordinate System (WCS) is represented in SNAP by the Snap.Globals.Wcs property, which is equivalent to the GRIP &WCS GPA. Also, the orientation of the WCS is represented by the Snap.Globals.WcsOrientation property. You can use either of these two properties to set the WCS or obtain information about it.

`csys = &WCS`	`csys = Snap.Globals.Wcs`	Get the coordinate system of the WCS
`&WCS = csys`	`Snap.Globals.Wcs = csys` `Snap.Globals.WcsOrientation = mx`	Set the WCS, using either a coordinate system or an orientation

▶ Mapping Between Coordinate Systems

SNAP uses the absolute coordinate system (ACS) consistently for all coordinate data. So, in computational code, all the calculations will be done using the ACS, and there will be little need to switch to the WCS. However, when communicating with the user, it will sometimes be more meaningful to use WCS coordinates, instead, so it's important to know how to map between the ACS and the WCS. GRIP provides the MAP/ statement for this purpose, and SNAP provides functions in the Snap.NX.CoordinateSystem class:

`MAP/`	`Snap.NX.CoordinateSystem.MapAcsToWcs`
	`Snap.NX.CoordinateSystem.MapWcsToAcs`
	`Snap.NX.CoordinateSystem.MapCsysToCsys`
		Map positions and vectors between coordinate systems.

Here is some sample code:

```
' To create a point that has WCS coordinates (2, 5, 8)
Dim wcsCoords = New Position(2, 5, 8)
Dim acsCoords = Snap.NX.CoordinateSystem.MapWcsToAcs(wcsCoords)
Dim myPoint = Point(acsCoords)

' To map positions and vectors to the WCS
Dim curve = BezierCurve({0,0,0}, {3,0,0}, {5,2,0})
Dim acsPos, wcsPos As Position
Dim acsTan, wcsTan As Vector
acsPos = curve.Position(0.5)
acsTan = curve.Tangent(0.5)
wcsPos = Snap.NX.CoordinateSystem.MapAcsToWcs(acsPos)
wcsTan = Snap.NX.CoordinateSystem.MapAcsToWcs(acsTan)
```

13. Curves

▶ Lines

The Snap.Create class contains several functions for creating lines:

`myLine = LINE/...`	`myLine = Snap.Create.Line( )` `myLine = Snap.Create.LineTangent( )`	Create lines

The following fragment of code creates two points and two lines in your Work Part:

```
p1 = Point(3,5)                    ' Creates a point at (3,5,0) in your Work Part
q = New Position(2,4,6)            ' Creates a position
p2 = Point(q)                      ' Creates a point from the position q
Dim c As NX.Line = Line(p1, p2)    ' Creates a line between points p1 and p2
Line(1,3, 6,8)                     ' Creates a line from (1,3,0) to (6,8,0)
```

Notice how z-coordinates can be omitted, in some cases. Since it's quite common to create curves in the xy plane, we provide special shortcut functions for doing this, so that you don't have to keep typing zeros for z-coordinates.

The properties of lines are:

Data Type	Property	Access	Description
`Position`	`StartPoint`	`get, set`	Start point (point where t = 0).
`Position`	`EndPoint`	`get, set`	End point (point where t = 1).
`Vector`	`Direction`	`get`	A unit vector in the direction of the line.

The StartPoint and EndPoint properties can be set, so you can use them to edit a line, like this:

```
Dim myLine As NX.Line = Line(2,3,7,8)     ' Creates a line between (2,3,,0) and (7,8,0)
myLine.EndPoint = {7,8,5}                 ' Moves the end-point to (7,8,5)
```

The NX.Line class also inherits some useful properties from NX.Curve, such as Arclength:

```
Dim myLine As NX.Line = Line(0,0,3,4)
Dim length As Double = myLine.Arclength
```

The arclength of a line is just the distance between its end-points, of course, but the Arclength property makes the calculation a little more convenient and easier to read.

▶ Arcs and Circles

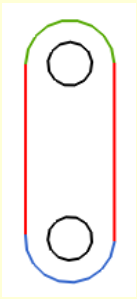
SNAP has creation functions named both Arc and Circle, which both create objects of type Snap.NX.Arc.

`c1 = CIRCLE/...`	`c1 = Snap.Create.Circle( )` `c1 = Snap.Create.Arc( )`	Create arcs (circles)

In several cases, the SNAP functions receive an orientation as an input, which allows you to orient the arc/circle however you like in 3D space.

The following snippet of code uses lines and arcs to create the shape shown on the right:

```
Dim length As Double = 8
Dim width As Double = 4
Dim half As Double = width/2
Dim holeDiameter As Double = half
Line(-half, 0, -half, length)        ' Left side
Line( half, 0,  half, length)        ' Right side
Arc(0, length, half, 0, 180)         ' Top semi-circle (green)
Arc(0, 0, half, 180, 360)            ' Bottom semi-circle (blue)
Circle(0, length, holeDiameter/2)    ' Top hole
Circle(0, 0, holeDiameter/2)         ' Bottom hole
```



SNAP also has Fillet functions, which again create Snap.NX.Arc objects:

`c1 = FILLET/...`	`c1 = Snap.Create.Fillet( )`	Create arcs by filleting

▶ Conic Sections

SNAP has functions for creating ellipses, but to create parabolas or hyperbolas you must use NX Open functions, instead.

`c1 = ELLIPS/`
`c2 = PARABO/`
`c3 = HYPERB/`
	`c1 = Snap.Create.Ellipse( )`
	`c2 = workPart.Curves.CreateParabola( )`
	`c3 = workPart.Curves.CreateHyperbola( )`
		Create conic section curves

Alternatively, you can use SNAP spline functions to create curves that are shaped exactly like parabolas or hyperbolas, even though they are actually Snap.NX.Spline objects.

▶ Splines

GRIP has both SPLINE and BCURVE statements, which primarily support creation of splines passing through a given sequence of points. In SNAP, the Snap.Create.SplineThroughPoints functions provides similar capabilities. For the special case of a Bezier curve (a spline with only a single segment), SNAP provides a specific BezierCurveThroughPoints function.

`c1 = SPLINE/` `c2 = BCURVE/`	`c3 = Snap.Create.SplineThroughPoints( )` `c4 = Snap.Create.BezierCurveThroughPoints( )`	Create spline through points

With the VERT minor word, the GRIP BCURVE command also allows you to create a spline from given control vertices (poles). Similar capabilities are provided by the SNAP Spline and BezierCurve functions:

`c1 = BCURVE/VERT`	`c2 = Snap.Create.Spline( )` `c3 = Snap.Create.BezierCurve( )`	Create spline from poles

The SNAP functions are somewhat more flexible, since they allow you to specify b-spline weights and nodes.

Functions for converting given curves into splines (either exactly or approximately) are as follows:

`c1 = SPLINE/APPROX` `c2 = BCURVE/CURVE`	`c3 = myCurve.ToSpline( )` `c4 = Snap.Create.JoinCurves( )`	Create splines by conversion and approximation

The Snap.Math.SplineMath class provides some lower-level tools for use by people who are spline experts.

▶ Isoparametric Curves

In SNAP, the functions that create isoparametric belong to individual face classes, so that they can return curves of specific known types. For example, the Cylinder.IsoCurveU function returns lines, and the Cylinder.IsoCurveV function returns arcs.

`c = ISOCRV/...`	`c = myFace.IsoCurveU( )` `c = myFace.IsoCurveV( )`	Create isoparametric curves

▶ Offset Curves

The GRIP and SNAP functions are:

`curves = OFFCRV/...`	`offsetFeature = Snap.Create.OffsetCurve( )`	Create offset curves

Note that the SNAP function creates an OffsetCurve feature.

▶ Edge Curves

GRIP provides the SOLEDG function to create curves corresponding to edges. In SNAP, this operation is typically not necessary, because SNAP functions work uniformly on either curves or edges. In fact, many SNAP functions receive an object called an ICurve as input, and an ICurve can be either a curve or an edge. But, if you really want to create curves that replicate edges, SNAP provides Edge.ToCurve and similar functions

`curves = SOLEDGE/...`	`c = myEdge.ToCurve( )`
	`c = myEdge.ToSpline( )`
	`c = Snap.Create.ExtractCurve`
		Create curves from edges

▶ Curve Evaluator Functions

Some of the most useful methods when working with curves and surfaces are the so-called “evaluator” functions. At a given location on a curve (defined by a parameter value u), 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 for curves are:

`u = CPARF/myCurve,p`	`t = myCurve.Parameter(p)`	Calculate the parameter value u at a given position p on myCurve
`p = CPOSF(myCurve,u)`	`p = myCurve.Position(u)`	Calculate the position p at a given parameter value u on myCurve
`v = CTANF(myCurve,u)`	`p = myCurve.Tangent(u)`	Calculate the tangent vector v at a given parameter value u on myCurve

`props = CPROPF(myCurve,u)`	`tan = myCurve.Tangent(u)`
	`nor = myCurve.Normal(u)`
	`bin = myCurve.Binormal(u)`
	`curv = myCurve.Curvature(u)`
	`deriv = myCurve.Derivative(u)`
	`derivs = myCurve.Derivatives(u)`
		Calculate the geometric properties at a given parameter value u on myCurve

Despite their apparent similarities, there are some fundamental differences between the GRIP and SNAP functions.

In GRIP we must "normalize" the parameter value (u) that is passed to these sorts of evaluator functions, so that it lies in the range 0 ≤ t ≤ 1. But the constant normalizing and denormalizing of parameter values can be tedious and confusing, so we never do this in SNAP. In the SNAP approach, each curve has a minimum parameter value, MinU, and a maximum parameter value, MaxU, and you should not assume that MinU = 0 or MaxU = 1. Actually, for lines and splines it is always true that MinU = 0 and MaxU = 1, but this is not the case for circles, ellipses, and a few other edge/curve types. To avoid confusion, if you want information about the point half-way along a curve, you should always use u = 0.5*(MinU + MaxU).

Also, the GRIP functions use work coordinates (WCS), whereas the SNAP functions use absolute coordinates.

Finally, the SNAP functions return Position and Vector objects, whereas the GRIP functions just return arrays of numbers.

For more information about curve evaluators, please refer to Chapter 9 of the SNAP Getting Started guide.

14. Surfaces

GRIP was created long before NX supported solid and sheet bodies. In those days, much of the 3D work in NX was performed using what we called "surfaces". The GRIP "surface" commands are still supported. Using today's terminology, they create sheet bodies that have only a single face.

The bodies created are "dumb" orphans, rather than associative features; they retain no links to their parent objects. By contrast, most of the analogous SNAP functions create feature objects. If this is not what you want, you can always use the Orphan function to disconnect the object from its parents.

▶ Legacy Surfaces

GRIP has commands to create the "surfaces" that were the primary 3D modeling objects in ancient times. These include surfaces of revolution, tabulated cylinders, ruled surfaces, fillet surfaces, sculptured surfaces, and offset surfaces. The commands are REVSRF, TABCYL, and so on. The corresponding SNAP functions are outlined below:

`REVSRF/`	`Revolve( )` `RevolveSheet( )`	The GRIP command creates a "surface of revolution", which is an orphan sheet body with a single face. The SNAP functions both create NX.Revolve features
`TABCYL/`	`Extrude( )` `ExtrudeSheet( )`	A "Tabulated Cylinder" is an extruded sheet body with a single face. The SNAP functions both create NX.Extrude features
`RLDSRF/`	`Ruled( )`	The GRIP command produces a sheet body having a single face of type b-surface. The SNAP function creates an NX.Ruled feature.
`FILSRF/`	`EdgeBlend( )` `FaceBlend( )`	GRIP produces a sheet body having a single face of type b-surface. The SNAP functions create EdgeBlend and FaceBlend features.
`SSURF/`	`ThroughCurveMesh( )`	The GRIP command produces a sheet body having a single b-surface face. The SNAP function creates an NX.ThroughCurveMesh feature.
`OFFSRF/`	`OffsetFace( )` `Thicken( )`	The GRIP command produces a sheet body having a single face. The SNAP functions create features.

Analytic Surfaces

The GRIP CYLNDR, CONE and SPHERE commands create sheet bodies whose underlying surfaces have "analytic" types. In former years, these special commands were needed because many of the other surface construction functions would create more general types of surfaces, rather than simple analytic ones. In newer versions of NX, functions like Extrude, Revolve and Blend will all create simple analytic surfaces, where possible, so there is less need for specific functions. SNAP provides Cylinder, Cone, and Sphere functions, but these create solid features, rather than surfaces.

CYLNDR/	ExtrudeSheet( ) RevolveSheet( ) EdgeBlend( ) Cylinder( )	The GRIP function creates a sheet body with a single cylindrical face. The SNAP functions create features of various types.
CONE/	ExtrudeSheet( ) RevolveSheet( ) Cone( )	The GRIP function creates a sheet body with a single conical face. The SNAP functions create features of various types.
SPHERE/	RevolveSheet( ) EdgeBlend( ) Sphere( )	The GRIP function creates a sheet body with a single spherical face. The SNAP functions create features of various types.

Bounded Plane

The GRIP BPLANE command creates a planar sheet body bordered by a given collection of curves. The closest corresponding SNAP function is Snap.Create.BoundedPlane.

BPLANE/	BoundedPlane( )	The GRIP function creates an orphan body. The SNAP function creates an NX.BoundedPlane feature.

In both cases, a collection of curves is provided as input. In the SNAP command, these curves can be in any order, and it is not necessary to indicate which of them define interior holes.

Plane

The GRIP PLANE command creates a plane object. Plane objects are infinite in extent, and do not form part of a body. They are typically used as reference objects — for example a plane might be used for mirroring, or to specify the location of a cross-section.

In SNAP, there are two corresponding objects: Snap.NX.DatumPlane, and Snap.Geom.Surface.Plane. A Datum Plane serves roughly the same purpose as a plane object, but Datum Planes are features that are linked to their parent objects. A Geom.Surface.Plane is a non-persistent object that is not stored in an NX model and exists only within a SNAP program. In SNAP, each curve has a Plane property which gives the plane containing the curve (if the curve is planar).

If you really want to create the same kind of object as the GRIP PLANE command, you should use the NX Open function NXOpen.UF.UFModl.CreatePlane

PLANE/	DatumPlane( ) Snap.Geom.Surface.Plane( ) myCurve.Plane NXOpen.UF.UFModl.CreatePlane( )	The second and third SNAP functions produce Snap.Geom.Surface.Plane objects.

B-Surface

The GRIP BSURF command provides several methods for constructing a sheet body that has a single face of type b-surface. Many of these methods use points as input, which can either be interpolated or used as control vertices (poles). SNAP provides analogous Snap.Create.Bsurface functions. In addition, SNAP has special BezierPatch functions that provide simple ways to create a Bezier patch (a b-surface with a single patch). In all cases, the result is a "dumb" sheet body with a single b-surface face.

BSURF/VERT	Bsurface( ) BezierPatch( )	Create a b-surface from poles
BSURF/	BsurfaceThroughPoints( ) BezierPatchThroughPoints( )	Create a b-surface interpolating (passing through) a given array of points

The GRIP BSURF command also allows surfaces to be built from collections of curves, using techniques that are similar to those provided by the SNAP ThroughCurves and ThroughCurveMesh functions:

BSURF/CURVE	ThroughCurves( )	Create a b-surface through a list of curves
BSURF/MESH	ThroughCurveMesh( )	Create a b-surface through a bi-directional mesh of curves

The GRIP functions each create a "dumb" sheet body with a single b-surface face. The SNAP functions create features.

Surface Evaluator Functions

Surface evaluator functions are analogous to those for curves:

SPARF/mySurf,p,u,v	uv = mySurf.Parameters(p)	Calculate the parameter values at a given position p on mySurf
p = SPOSF(myCurve,u,v)	p = mySurf.Position(u,v)	Calculate the position p at parameter values (u,v) on mySurf
du = SDDUF(mySurf,u,v) dv = SDDVF(mySurf,u,v)	du = mySurv.DerivDu(u,v) dv = mySurv.DerivDv(u,v)	Calculate partial derivatives at parameter values (u,v) on mySurf
n = SNORF(mySurf,u,v)	n = mySurf.Normal(u,v)	Calculate unit surface normal at parameter values (u,v) on mySurf

Again, as with curve evaluators, parameters are not normalized to the range [0,1], and all coordinates are expressed with respect to the absolute coordinate system.

For more information about surface evaluators, please refer to Chapter 9 of the SNAP Getting Started guide.

15. Simple Features

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. 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 Snap.Create 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 quite complex. Most of the feature commands in GRIP are fairly simple ones, however.

As usual, there is a difference in coordinate system usage between GRIP and SNAP. In many commands, GRIP uses coordinate axes of the WCS as default direction vectors (for the axis vectors of cylinders and cones, for example). In SNAP commands, there are typically no default direction vectors. In some cases, direction vectors are input to the functions, or sometimes the absolute coordinate system is used as a default orientation.

In the GRIP commands, parameters like lengths and diameters are simple numerical values. SNAP typically uses Number objects for this purpose, so either Double or String values can be used. This allows features to be linked to expressions in SNAP.

▶ Primitive Features

GRIP commands like SOLBLK, SOLCYL and so on to create primitive features. SNAP has analogous functions like Snap.Create.Block, Snap.Create.Cylinder, and so on. The SNAP functions typically have several overloads providing different construction options.

SOLBLK/	Block( )	The GRIP and SNAP functions use the WCS and ACS respectively to align the block. Or, in some SNAP functions, an orientation can be input.
SOLCYL/	Cylinder( )	The SNAP functions have several overloads
SOLCON/	Cone( )	The SNAP functions have several overloads
SOLSPH/	Sphere( )	The SNAP functions have several overloads
SOLTOR/	Torus( )	The GRIP SOLTOR command creates an orphan solid body, rather than a feature. The Snap.Create.Torus function can create either a sheet body or a solid body, depending on its inputs. In either case, the output is an NX.Revolve feature.

▶ Extruded and Revolved Features

The GRIP SOLEXT and SOLREV commands are quite similar to the Snap.Create.Extrude and Snap.Create.Revolve functions. The SNAP functions have variants that force the creation of a sheet body (rather than a solid).

SOLEXT/	Extrude( ) ExtrudeSheet( )	Both the SNAP and GRIP functions create Extrude features
SOLREV/	Revolve( ) RevolveSheet( )	Both the SNAP and GRIP functions create Revolve features

▶ Create Tube Feature

The GRIP SOLTUB command is very similar to the Snap.Create.Tube function.

SOLTUB/	Tube( )	In both cases, a Tube feature is produced

▶ Create Prism

The GRIP SOLPRI command creates an n-sided prism. There is no direct analog either in SNAP or NX Open, except for a very old C function called UF6505.

SOLPRI/	UF6505( )	The GRIP SOLPRI command creates an orphan solid body, rather than a feature.

▶ Using NX Open Builders

In the previous examples, we have used SNAP functions to create features, which is very simple. However, there may be cases where the necessary function is not yet available in SNAP, so you will need to use an NX Open function to

create a feature, instead. To show how this is done, here is some typical code, which builds a sphere feature using NX Open functions:

```
[1]  Dim nullSphere As NXOpen.Features.Sphere  = Nothing
[2]  Dim mySphereBuilder As NXOpen.Features.SphereBuilder
[3]  mySphereBuilder = theWorkPart.Features.CreateSphereBuilder(nullSphere)
[4]  mySphereBuilder.Property1 = <whatever you want>
[5]  mySphereBuilder.Property2 = < whatever you want >
[6]  Dim myObject As NXOpen.NXObject = mySphereBuilder.Commit()
[7]  mySphereBuilder.Destroy()
```

So, the general approach is to

- create a "builder" object (line [3])
- modify its properties as desired (lines [4] and [5])
- "commit" the builder to create a new object (line [6])

As you can see in line [3] above, the functions to create various types of "builder" objects are methods of a FeatureCollection object, and we can get one of these from workPart.Features.

A SphereBuilder object is fairly simple, but other feature builders are very complex, with large numbers of properties that you can set.

16. Body Operations

This section discusses operations that you can perform on bodies to modify them or create new bodies.

▶ Joining Operations

The GRIP commands UNITE, SUBTRA and INTERS allow you to perform boolean operations on solid and sheet bodies (with certain restrictions). The analogous SNAP functions are Snap.Create.Unite, Snap.Create.Subtract, and Snap.Create.Intersect. THE GRIP SEW command allows you to sew sheet bodies together.

UNITE/	Unite( )	Both create a Boolean feature
SUBTRA/	Subtract( )	Both create a Boolean feature
INTERS/	Intersect( )	Both create a Boolean feature
SEW/	Sew( )	Both create a Sew feature

▶ Blending and Chamfering

GRIP has a single BLEND command that performs both filleting and chamfering. The blends are "fixed" in a separate operation. In SNAP, there are separate commands for filleting and chamfering, and no "fix" operation is required. The Snap.Create.EdgeBlend function is the closest analog of the GRIP BLEND command.

BLEND/	EdgeBlend( ) FaceBlend( ) Chamfer( )	The GRIP command creates either an EdgeBlend or Chamfer feature.
BLENFX/	---	In SNAP and NX Open, blends are always fixed when they are created, so no separate fix operation is required.

▶ Cutting and Splitting

The GRIP SOLCUT and SPLIT command both divide bodies by using a sheet body. The corresponding SNAP functions are SplitBody and TrimBody.

SOLCUT/	SplitBody( ) TrimBody( )	Split a body using an infinite plane or cylinder
SPLIT/	SplitBody( ) TrimBody( )	Split a body using a sheet body, faces, or a datum plane

17. Drawings & Annotations

Currently, there are very few SNAP functions for working with drawings or annotations, so, for the most part, the paragraphs below discuss the relevant NX Open functions, instead.

▶ 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.

DRAWC/	sheets = workPart.DrawingSheets sheets.InsertSheet( )	Create a drawing
DRAWD/	myDrawing.Delete( )	Delete a drawing
DRAWE/ADD	views = mySheet.SheetDraftingViews views.CreateBaseView( ) views.CreateProjectedView( )	Add a view to a drawing
DRAWE/REMOVE	views = mySheet.SheetDraftingViews views.DeleteView( )	Remove view from a drawing
dwg = &CURDRW	sheets = workPart.DrawingSheets dwg = sheets.CurrentDrawingSheet	Get the current drawing (sheet)
&CURDRW = dwg	dwg.Open( )	Set the current drawing (sheet)
DRAWV/	dwg.GetDraftingViews( )	Get the views of a drawing

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
```

▶ 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 the functions for creating dimensions (either directly or via builders):

LDIM/{HORIZ|VERT}	CreateHorizontalDimension( ) CreateVerticalDimension( ) CreateLinearDimensionBuilder( )	Horizontal or vertical dimension
LDIM/PARLEL	CreateParallelDimension( )	Parallel dimension
LDIM/PERP	CreatePerpendicularDimension( )	Perpendicular dimension
ADIM/	CreateAngularDimensionBuilder( ) CreateMajorAngularDimension( ) CreateMajorAngularDimensionBuilder( ) CreateMinorAngularDimension( ) CreateMinorAngularDimensionBuilder( )	Angular dimension
ARCDIM/	CreateArcLengthDimension( ) CreateCurveLengthDimensionBuilder( )	Arclength dimension
CYLDIM/	CreateCylindricalDimension( )	Cylindrical dimension
RDIM/	CreateRadiusDimension( )	Radius dimension
FRDIM/	CreateFoldedRadiusDimension( )	Folded radius dimension
DDIM/	CreateDiameterDimension( )	Diameter dimension
HDIM/	CreateHoleDimension( )	Hole dimension
CCDIM/	CreateConcentricCircleDimension( )	Concentric circle dimension
ODIM/	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)
```

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()
```

▶ Notes

To create a Note, you can use the SNAP function Snap.Create.Note. Alternatively, there are NX Open functions in the AnnotationManager class:

NOTE/	Snap.Create.Note( )	Create a note
NOTE/	Dim mgr As NXOpen.Annotations.AnnotationManager mgr = workPart.Annotations mgr.CreateNote( ) mgr.CreatePmiNoteBuilder(Nothing)	Create a note

The SNAP function creates a Snap.NX.Note object, which encloses an NXOpen.Annotations.PmiNote object.

18. Layers & Categories

▶ Layers

In SNAP, each object has a Layer property that is roughly analogous to the GRIP EDA &LAYER. You can use this property to get and set the layer assignment of any object.

`n = &LAYER(obj)`	`n = obj.Layer`	Get the layer of an object
`&LAYER(obj) = n`	`obj.Layer = n`	Set the layer of an object (move it to a new layer)

To get or set the Work Layer, you use the Snap.Globals.WorkLayer property, which is equivalent to the GRIP &WLAYER GPA symbol:

`n = &WLAYER`	`n = Snap.Globals.WorkLayer`	Get the Work Layer
`&WLAYER = n`	`Snap.Globals.WorkLayer = n`	Set the Work Layer

Each layer has one of four states described by the Snap.Globals.LayerState enumeration. These four states are WorkLayer, Selectable, Visible, and Hidden. The state of layer number n is held in the variable Snap.Globals.LayerStates(n).

`LAYER/`	`state = Snap.Globals.LayerStates(n)`	Get the current state of layer #n
	`Snap.Globals.LayerStates(n) = state`	Set the state of layer #n

To create a category in SNAP, you use the Snap.Create.Category function. This returns an object of type Snap.NX.Category that has certain useful methods and properties as outlined below.

`CAT/`	`myCat = Snap.Create.Category( )`	Create a layer category
`CATD/`	`myCat.Delete( )` `Category.Delete( )`	Delete a category or a collection of categories
`CATE/ADD`	`myCat.Add( )`	Add layers to a category
`CATE/REMOVE`	`myCat.Remove( )`	Remove layers from a category
`CATV/LAYERS`	`myCat.Layers` `myCat.LayerMask`	Get the layers in a category

19. Attributes

This section covers the GRIP statements that assign and delete attributes. Attributes can be assigned either to NX part files, or to individual objects within part files. In both GRIP and SNAP, attributes are handled in more-or-less the same way whether they are attached to objects or part files.

▶ Assigning Attributes

In GRIP, attributes are assigned to either parts or objects using the ASATT command.

The assigned attribute can be one of the following types:

Code	Type
1	Integer
2	Floating point
3	Date and time
4	Null
5	Character string
7	Reference

In SNAP, object attributes are assigned and read using functions in the NX.NXObject class, and there are different functions for different attribute types. For example, if you have a body called bracket, you assign a "Cost" attribute with a value of $17.95 to it using bracket.SetRealAttribute("Cost", 17.95). The Snap.NX.Part class has analogous functions (with exactly the same names) for assigning attributes to parts.

ASATT/	myObject.SetIntegerAttribute( ) myObject.SetRealAttribute( ) myObject.SetDateTimeAttribute( ) myObject.SetNullAttribute( ) myObject.SetStringAttribute( )	Assign an attribute to an object myObject
ASATT/	myPart.SetIntegerAttribute( ) myPart.SetRealAttribute( ) myPart.SetDateTimeAttribute( ) myPart.SetNullAttribute( ) myPart.SetStringAttribute( )	Assign an attribute to a part myPart

▶ Deleting Attributes

The functions for deleting attributes are:

DLATT/	myObject.DeleteAttributes( )	Delete attributes from myObject
	myPart.DeleteAttributes( )	Delete attributes from a myPart

▶ Reading Attribute Values

In GRIP, you read attribute values using EDA symbols. In SNAP, you find out attribute types and titles using the GetAttributeInfo function, and then you read values using "Get" functions that are parallel to the ones for assigning attributes. There is also a GetAttributeStrings function that returns attribute information in string form. The SNAP functions for reading attributes from an NX object are shown in the table below:

&ATTTL	myObject.GetAttributeInfo( )	Get attribute titles and types
&ATTVL	myObject.GetIntegerAttribute( ) myObject.GetRealAttribute( ) myObject.GetDateTimeAttribute( ) myObject.GetNullAttribute( ) myObject.GetStringAttribute( ) myObject.GetAttributeStrings( )	Get attribute values of myObject

In the Snap.NX.Part class, you will find analogous functions (with the same names) for reading part attributes. So, for example, to read an Integer attribute from myPart, you call myPart.GetIntegerAttribute( ), and so on.

In SNAP, all date/time attributes are represented using .NET System.DateTime structures.

20. General Object Properties & Functions

▶ General Properties

All NX objects have certain properties in common. For example, they all have a layer assignment, a show/hide status, and (potentially) a name. In GRIP, these properties are accessed using EDA symbols. SNAP uses properties in the NX.NXObject class, which you can access using the usual "dot" notation:

`&NAME(obj)`	`obj.Name`	The name of the object
`&ATDISL(obj)`	`obj.NameLocation`	The name display location
`&PROTO(obj)`	`obj.Prototype`	The prototype object (for an occurrence)
`&BLANK(obj)`	`obj.IsHidden`	Whether the object is blanked (hidden)
`&COLOR(obj)`	`obj.Color`	The color of the object. A Windows color in SNAP.
`&LWIDTH(obj)`	`obj.LineWidth`	Line width (thin, thick, etc.). Alternatively `&DENS(obj).`
`&FONT(obj)`	`obj.LineFont`	Line font (solid, dashed, etc.)
`&LAYER(obj)`	`obj.Layer`	Layer number
`&TYPE(obj)`	`obj.ObjectType`	The type of the object
`&SUBTYP(obj)`	`obj.ObjectSubType`	The subtype of the object

▶ Object Types

In GRIP, each NX object has a "type", and (in some cases) a "subtype". Types and subtypes are indicated by integer values, which can be obtained by using EDA's called &TYPE and &SUBTYP. In SNAP, an object's type is given by its ObjectType property, whose value comes from the Snap.NX.ObjectTypes.Type enumeration. Similarly, an object's subtype is given by its ObjectSubType property, whose value comes from the Snap.NX.ObjectTypes.SubType enumeration. The correspondences for a few cases are shown in the tables below. First for object types:

Object	GRIP Type	SNAP Type
Point	2	`Snap.NX.ObjectTypes.Type.Point`
Line	3	`Snap.NX.ObjectTypes.Type.Line`
Arc/circle	5	`Snap.NX.ObjectTypes.Type.Arc` or `Snap.NX.ObjectTypes.Type.Circle`
Conic	6	`Snap.NX.ObjectTypes.Type.Conic`
Spline	9	`Snap.NX.ObjectTypes.Type.Spline`
Layer category	12	`Snap.NX.ObjectTypes.Type.LayerCategory`
Drafting object	25	`Snap.NX.ObjectTypes.Type.DraftingEntity`
Dimension	26	`Snap.NX.ObjectTypes.Type.Dimension`
Coordinate System	45	`Snap.NX.ObjectTypes.Type.CoordinateSystem`

and then for a few subtypes:

Object	GRIP Subtype	SNAP SubType
Conic-Ellipse	1	Snap.NX.ObjectTypes.SubType.ConicEllipse
Conic-Hyperbola	2	Snap.NX.ObjectTypes.SubType.ConicHyperbola
Conic-Parabola	3	Snap.NX.ObjectTypes.SubType.ConicParabola
Dimension-Horizontal	1	Snap.NX.ObjectTypes.SubType.DimensionHorizontal
Dimension-Vertical	2	Snap.NX.ObjectTypes.SubType.DimensionVertical

So, SNAP code for testing object types and subtypes looks like this:

```
Dim curveType = myCurve.ObjectType
If curveType = NX.ObjectTypes.Type.Conic
   InfoWindow.WriteLine("It's a conic")
   Dim conicType = myCurve.ObjectSubType
   If conicType = NX.ObjectTypes.SubType.ConicEllipse
      InfoWindow.WriteLine("It's actually an ellipse")
   End If
End If
```

The value of Snap.NX.ObjectTypes.Type.Conic is actually 6, which matches the value used in GRIP. So, you could write If curveType = 6, instead of If curveType = NX.ObjectTypes.Type.Conic. But it's best to avoid using numerical constants like "6", because they look mysterious in your code, and they might change at some point in the future. Use the enumerated values, instead. They are rather long, but Visual Studio intellisense will do most of the typing for you.

▶ General Functions

Many of the general functions described below are related to the general properties listed above. The property values can typically be set without using any special function, so the functions are useful only when you want to set the properties of several objects all at once. For example, using the Hide function is more convenient than setting the IsHidden property of several objects. These functions can generally be found in the Snap.NX.NXObject class

BLANK/	Hide(.)	Blanks (hides) a collection of objects. More convenient than setting obj.IsHidden = True
UNBLANK/	Show(.)	Unblanks (shows) a collection of objects. More convenient than setting obj.IsHidden = False
DELETE/	Delete( )	Deletes a collection of objects. More convenient than using obj.Delete( ) several times.

▶ Transformations

To transform objects in GRIP, you first use the MATRIX statement to construct a matrix that describes the effect of the transformation. Then you apply the transformation by passing your matrix into the TRANSF command. The approach in SNAP is similar. You first construct a Transform object using the functions in the Snap.Geom.Transform class, and then you apply this transform to an object by using either the Snap.NX.NXObject.Move or Snap.NX.NXObject.Copy functions. You can also transform Position and Vector objects.

Here are the functions for creating transformations. As always, GRIP uses WCS coordinates, and SNAP uses the absolute coordinate system.

MATRIX/TRANSL	Transform.CreateTranslation( )	Create a transform for translation
MATRIX/SCALE	Transform.CreateScale( )	Create a transform for scaling
MATRIX/XYROT	Transform.CreateRotation( )	Create a transform for rotation
MATRIX/MIRROR	Transform.CreateReflection( )	Create a transform for reflection
MATRIX/	Transform.Composition( )	Combine (concatenate) two transformations

Then the transformation is applied with the following functions:

TRANSF/	myObject.Move( ) or myObject.Copy( ) Position.Move( ) or Position.Copy( ) Vector.Move( ) or Vector.Copy( )	Apply a transform to an NXObject , a Position, or a Vector

21. Analysis/Computation Functions

GRIP provides analysis functions for computing distances, deviations, and physical properties like area, volume, and mass. Most of the corresponding SNAP functions can be found in the Snap.Compute class.

▶ Distances Between Objects

GRIP provides the DISTF/ and RELDST/ statements to calculate distances between objects. The corresponding SNAP functions are Snap.Compute.Distance and Snap.Compute.ClosestPoints.

DISTF/ RELDST/	Snap.Compute.Distance( ) Snap.Compute.ClosestPoints( )	Calculate the (minimum) distance between objects, and the points at which the minimum distance is achieved.

▶ Deviation

Deviation checking functions measure the maximum distance between two objects (curves and faces). The primary use of these functions is to check whether two objects are (roughly) coincident. The GRIP DEVCHK/ statement outputs results to the listing device, whereas the Snap.Compute.Devation functions return their results to the calling function.

DEVCHK/	Snap.Compute.Deviation( )	Calculate the deviation between curves and faces

▶ Arclength

The GRIP ANLSIS/ARCLEN statement calculates the arclength of a given collection of curves. In SNAP, each curve or edge has an Arclength property. Alternatively, to compute the arclength of a collection of curves, all at once, you can use the Snap.Compute.Arclength function.

ANLSIS/ARCLEN	myCurve.Arclength myEdge.Arclength Snap.Compute.Arclength( )	Calculate the arclength or curves or edges.

Area and Volumetric Properties

The GRIP ANLSIS/ statement calculates the properties of planar regions or various types of 3D objects. There are several ways to get similar results using SNAP functions. Firstly each face in SNAP has an Area property. Also, the Snap.Compute.MassProperties function computes properties such as area, perimeter, centroid, and so on. This function returns a structure containing the various results. If you only need area, the Snap.Compute.Area function will probably be easier to use.

ANLSIS/TWOD	myFace.Area Snap.Compute.MassProperties( ) Snap.Compute.Area( )	Calculate the properties of a 2D planar region

For sheet and solid bodies, the Snap.Compute.MassProperties function computes properties such as areas, volumes, mass, moments of inertia, and so on. Again, this function returns a structure containing the various results. If you only need mass, volume, or surface area, there are specific functions for calculating these quantities, which you may find easier to use.

ANLSIS/SOLID	Snap.Compute.MassProperties( ) Snap.Compute.Mass( ) Snap.Compute.Volume( )	Calculate the properties of a collection of bodies (which may be either sheets or solids, in SNAP).

GRIP has special options like ANLSIS/PROSOL and ANLSIS/VOLREV that allow you to compute the properties of "implied" objects without actually constructing them. This is not supported in SNAP — if you want to compute the properties of an object, you must fully construct it, first.

22. CAM

Currently, there are no SNAP functions for CAM, so we provide a brief introduction to NX Open functions, instead.

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

In GRIP programs, you can use the INEXTN and NEXTN statements to cycle through various types of "non-geometric" entities, including CAM operations and tools.

In NX Open, object cycling 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 in the next section.

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
   InfoWindow.WriteLine(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)
      InfoWindow.WriteLine("Tool type:    " & toolType.ToString)
      InfoWindow.WriteLine("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. To perform the same sort of testing in GRIP, you would use an &OPTYP EDA symbol together with "magic numbers" representing the operation types. The magic numbers are not needed in NX Open, so the code is much more readable:

`IF/(&OPTYP('P1')==110)`	`If TypeOf(op) Is NXOpen.CAM.PlanarMilling`	Test operation type

The types and subtypes of tools are handled in a different fashion. As the code above shows, there is a GetTypeAndSubtype function. But, again, this does not return magic numbers, it returns values from two enumerations, CAM.Tool.Types and CAM.Tool.Subtypes, which again leads to more readable code. For example, here is how you would test for a threading tool of subtype "Acme":

`OBTAIN/T1,TLDAT` `IF/TLDAT(1)==502`	`tool.GetTypeAndSubtype(toolType, subType)` `If subType = CAM.Tool.Subtypes.ThreadAcme`	Test tool type

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: the NXOpen.CAM.OperationCollection class contains builders for operations, and the NXOpen.CAM.NCGroupCollection class contains builders for various types of CAM "groups", which include tools, CAM geometry, and machining methods. Some examples of the available functions are shown in the two tables below:

NXOpen.CAM.OperationCollection functions for creating operation builders	
CreateCavityMillingBuilder	Creates a planar milling cavity operation builder
CreateCenterlineDrillTurningBuilder	Creates a centerline drill turning operation builder
CreateEngravingBuilder	Creates a planar milling text operation builder
CreateFaceMillingBuilder	Creates a planar milling facing operation builder
CreateHoleDrillingBuilder	Creates a hole drilling operation builder
CreatePlanarMillingBuilder	Creates a planar milling planar operation builder

NXOpen.CAM.NCGroupCollection for creating "group" builders	
CreateBarrelToolBuilder	Creates a barrel tool builder
CreateDrillGeomBuilder	Creates a drill geometry builder
CreateDrillMethodBuilder	Creates a drill method builder
CreateDrillTapToolBuilder	Creates a drill tap tool builder
CreateMachineTurretGroupBuilder	Creates a machine turret group builder
CreateMillToolBuilder	Creates a mill tool builder
CreateMillGeomBuilder	Creates a mill geometry builder
CreateProgramOrderGroupBuilder	Creates a program order group builder

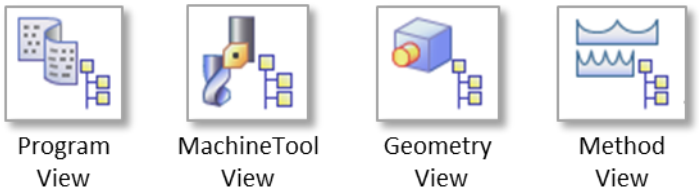
Here is another example, this time editing tool objects:

```
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 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 geometryRoot As NXOpen.CAM.NCGroup = setup.GetRoot(NXOpen.CAM.CAMSetup.View.Geometry)
Dim methodRoot   As NXOpen.CAM.NCGroup = setup.GetRoot(NXOpen.CAM.CAMSetup.View.MachineMethod)
Dim machineRoot  As NXOpen.CAM.NCGroup = setup.GetRoot(NXOpen.CAM.CAMSetup.View.MachineTool)
Dim programRoot  As NXOpen.CAM.NCGroup = setup.GetRoot(NXOpen.CAM.CAMSetup.View.ProgramOrder)

Dim geometryRootMembers As NXOpen.CAM.CAMObject() = geometryRoot.GetMembers()
Dim methodRootMembers   As NXOpen.CAM.CAMObject() =   methodRoot.GetMembers()
Dim machineRootMembers  As NXOpen.CAM.CAMObject() =  machineRoot.GetMembers()
Dim programRootMembers  As NXOpen.CAM.CAMObject() =  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 two sections, which discuss creation of tools and operations.

Creating a Tool

Creating a tool object in GRIP is very simple — we simply use the TOOL statement. However, NX tools have become much more sophisticated since the days of GRIP, so the NX Open procedure 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.

23. Global Parameters

A number of GRIP statements set preferences that are used to control the environment in which NX objects are created. These preferences can be regarded as global parameters that control the user's working environment. The corresponding SNAP mechanism is a set of properties in the Snap.Globals class:

DENS/ &DENS &LWIDTH	Globals.LineWidth	The line width for subsequently-created objects
FONT/ &FONT	Globals.LineFont	The line font for subsequently-created objects
&WLAYER	Globals.WorkLayer	The work layer
&ENTCLR	Globals.Color	The color for subsequently-created objects (a Windows color)
&WCS	Globals.WCS	The Work Coordinate System
&ANGTOL	Globals.AngleTolerance	Angle tolerance, in degrees
&DISTOL	Globals.DistanceTolerance	Distance tolerance

24. File Operations

This section covers the GRIP statements used to perform file management operations.

▶ File Operations

The table below shows the GRIP functions for working with files in general, and their analogs in VB:

CRDIR/	System.IO.Directory.Create Directory( )	Create directory
FDEL/	System.IO.File.Delete( )	Delete file
FMOVE/	System.IO.File.Move( )	Move file
FCOPY/	System.IO.File.Copy( )	Copy file

▶ Directory Cycling

GRIP provides the DOPEN, DCLOSE, SOPEN, and SCLOSE functions for moving upwards and downwards within a tree of directories. Then, within any given directory, you can use the DNEXT function to cycle through its contents (either files or subdirectories), accessing header information.

There are no SNAP or NX Open functions for performing these kinds of operations, because rich facilities are already provided by the .NET framework. The relevant .NET functions can be found in the System.IO.File, System.IO.Directory, System.IO.FileInfo and System.IO.DirectoryInfo classes.

For example, the following static functions (and many others) are available in the System.IO.Directory class:

EnumerateFiles( )	Returns an enumerable collection of file names in a specified path.
EnumerateDirectories( )	Returns an enumerable collection of directory names in a specified path.
EnumerateFileSystemEntries( )	Returns an enumerable collection of file names and directory names

These functions return enumerable collections, which you can then cycle through using For Each. One of the overloads of the EnumerateFiles function has a SearchOption parameter that lets you indicate whether you want all the files in the given directory and its sub-directories, or only the ones at the top level.

▶ File Information GPAs

Following execution of a FHREAD or DNEXT statement, the following GPA symbols are populated with information from the current file's header. Some of this information is generic, and can be obtained from any file, of any type. To obtain this generic information, standard .NET functions from the System.IO namespace can be used

&CRDATE	System.IO.File.GetCreationTime( ) System.IO.File.SetCreationTime( )	The creation date.
&CRTIME	System.IO.File.GetCreationTime( ) System.IO.File.SetCreationTime( )	The creation time.
&LADATE	System.IO.File.GetLastAccessTime( ) System.IO.File.SetLastAccessTime( )	The date last accessed.
&LATIME	System.IO.File.GetLastAccessTime( ) System.IO.File.SetLastAccessTime( )	The time last accessed.
&LMDATE	System.IO.File.GetLastWriteTime( ) System.IO.File.SetLastWriteTime( )	The date last modified.
&LMTIME	System.IO.File.GetLastAccessTime( ) System.IO.File.SetLastAccessTime( )	The time last modified.
&OWNER	System.IO.File.GetAccessControl( )	The owner (and related access information)
&DIR	System.IO.Directory.GetParent( )	The name of the current directory
&DIRPTH	System.IO.Directory.GetParent( )	The full pathname of the current directory
&FLEN	System.IO.FileInfo.Length	The size of the file, in bytes
&FORMAT	System.IO.FileInfo.Extension	The type (extension) of the file

The .NET functions all use System.DateTime objects to represent dates and times.

On the other hand, there is other information that is specific to NX, which you can access using NX Open functions:

&CAREA	NXOpen.UF.UFPart.AskCustomerArea( ) NXOpen.UF.UFPart.SetCustomerArea( )	Contents of the customer area in the file header
&DESCR	NXOpen.UF.UFPart.AskDescription( ) NXOpen.UF.UFPart.SetDescription( )	The description in the file header.
&FSTAT	NXOpen.UF.UFPart.AskStatus( ) NXOpen.UF.UFPart.SetStatus( )	The (integer) status field in the file header
&RELNO	NXOpen.UF.UFPart.AskPartHistory( ) NXOpen.UF.UFPart.AskLastModifiedVersion( )	The NX version in which the part was last saved.

The following code cycles through all the part files in a directory, writing out some information:

```
Dim ufs As NXOpen.UF.UFSession = NXOpen.UF.UFSession.GetUFSession()

' Get all the parts in a directory
Dim dirPath As String = "C:\MyDocuments\Parts"
Dim filePaths = System.IO.Directory.EnumerateFiles(dirPath, "*.prt")

Dim partTag As NXOpen.Tag
Dim version As Integer
Dim errorStatus As NXOpen.UF.UFPart.LoadStatus = Nothing

' Cycle through the parts, writing out some information
For Each path As String In filePaths
   ufs.Part.OpenQuiet(path, partTag, errorStatus)
   ufs.Part.AskLastModifiedVersion(partTag, version)
   InfoWindow.WriteLine(version & " ; " & path)
   ufs.Part.CloseAll()
Next
```

25. Working with Text Files

GRIP includes a number of statements that are used to read and write text files, which are held in one of ten "scratch areas". There are no analogous functions in SNAP or NX Open because the .NET framework already provides a rich set of facilities in this area. The relevant .NET functions can be found in the System.IO.File class and the System.IO.StreamReader class. It's easy to find on-line documentation and tutorials covering the usage of these functions. A large number of functions are available, and only a few of them are mentioned in the table below:

CREATE/	System.IO.File.Create( ) System.IO.File.Open( )	Create or open a text file
FTERM/	StreamReader.Close( ) StreamWriter.Close( )	Close a text file
READ/	File.ReadAllLines( ) StreamReader.ReadLine( )	Read text from a file
WRITE/	File.WriteAllLines( ) StreamWriter.WriteLine( )	Write text to a (new or existing) file

▶ Reading

You can read a text file either one line at a time or all at once. Often, the simplest approach is to read all the lines of text in the file into an array of strings, and then cycle through this array:

```
Dim lineArray As String( ) = System.IO.File.ReadAllLines("C:\myfile.txt")
For Each line As String In lineArray
   ' Do something with this line of text
Next
```

Alternatively, to read one line at a time, you use a StreamReader object, like this:

```
Dim reader As New System.IO.StreamReader("C:\myfile.txt")
Do Until reader.EndOfStream
   Dim line As String = reader.ReadLine( )
   ' Do something with the line we just read
Loop
```

The streamreader keeps track of the "pointer" or "cursor" in the file, so, each time you call ReadLine, you get the next available line of text. GRIP uses line numbers for this sort of cursor positioning, but this approach does not make sense in the .NET world, since there is no concept of line numbering.

▶ Writing

Techniques for writing text are similar. You can write an entire array of strings, all at once, like this:

```
Dim lineArray As String( ) = {"First line", "Second one", "Third"}
System.IO.File.WriteAllLines("C:\myfile.txt", lineArray)
```

Or, you can write lines one at a time using a StreamWriter object:

```
Dim lineArray As String( ) = {"First line", "Second line", "Third line"}
Dim writer As New System.IO.StreamWriter("C:\myfile.txt")
For Each line As String In lineArray
   writer.WriteLine(line)
Next
```

The code samples above are just illustrative fragments. In production code, you need to handle any exceptions that are raised, and you need to properly dispose of the StreamReader and StreamWriter objects you create.

26. Operating System Interaction

GRIP has several statements that allow your program to interact with the Windows operating system. These statements are not necessary in SNAP or NX Open because the standard tools provided by the .NET framework can be used instead.

▶ Environment Variables

The GRIP ENVVAR statement allows you to get and set the values of environment variables. In VB code, you do this by using the functions in the System.Environment class. There are specific properties that give you access to common environment variables, for example:

`System.Environment.UserName`	Gets the username of the currently logged on user
`System.Environment.MachineName`	Gets the netbios name of the computer
`System.Environment.CurrentDirectory`	Gets or sets the path of the current working directory.

The most versatile functions are GetEnvironmentVariable and SetEnvironmentVariable, which allow you get and set values of any environment variable, and also to create and delete environment variables.

So, for example, the following code writes the name of the NX "root" folder to the Info window:

```
Dim rootDirName As String
rootDirName = System.Environment.GetEnvironmentVariable("UGII_ROOT_DIR")
InfoWindow.WriteLine(rootDirName)
```

▶ Running Another Process

GRIP has a statement called XSPAWN that allows you to run (or "spawn") some other process. The corresponding facility in the .NET world is provided by the System.Diagnostics.Process.Start function. This function has many options, but simple calls are straightforward:

```
' Start notepad
System.Diagnostics.Process.Start("notepad.exe")

' Launch Windows Explorer, and open the C: folder
System.Diagnostics.Process.Start("C:\")

' Launch Internet Explorer, and open the Google site
System.Diagnostics.Process.Start("http://google.com")
```

27. New Opportunities

As we mentioned at the outset, there is no urgent need to switch from GRIP to newer programming tools. If you are happy with the programs you already have, and you don't envision any significant new developments in the future, then GRIP may be the right tool for you. But, on the other hand, there are some areas where SNAP and NX Open are clearly superior to GRIP, and provide new opportunities. Some examples are:

- Writing code by recording journals
- Creating modern block-based user interfaces (and without using Block Styler, if you want)
- Creating more flexible user interfaces using Windows forms
- Performance — SNAP and NX Open code is orders of magnitude faster than GRIP code
- Better development tools (like Visual Studio) — "the code writes itself"
- Freedom from size limitations (length of names, arrays, strings, etc.)
- Simpler development — just edit and run (in the Journal Editor) — no need to compile or link
- More flexible code organization — no need to put every subroutine in a separate file
- A huge assortment of tools in the .NET framework
- Creation of newer features, like Datum Planes, Thicken, TrimBody, etc.
- Easy access to Excel and other Office apps
- Better handling of text files (using .NET read/write functions)
- Development of main-stream programming skills that will be useful beyond the world of NX

If any of these seem important to you, then learning SNAP or NX Open is probably the right thing to do, and we hope that this document has made this easier for you.

28. Getting Further Help

Once you understand the basic ideas of SNAP and NX Open, you may find that code examples are the best source of help. Some sources of example programs and other forms of help are as follows:

- In the SNAP Getting Started 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.
- In the SNAP Reference Guide, there are several hundred example programs that show you how to use the functions described there. These are all very small programs, and very few of them do anything that is truly valuable, but you will probably find them helpful in understanding function usage.
- There are some examples in [...NX]\UGOPEN\SNAP\Examples. There are two folders: the one called "Getting Started Examples" contains the examples from the Getting Started guide, and the "More Examples" folder contains some larger examples that try to do more useful things. Here, [...NX] denotes the folder where the latest release of NX is installed, which is typically C:\Program Files\Siemens\NX 10, or something similar.
- There are about 35 example NX Open programs in [...NX]\UGOPEN\SampleNXOpenApplications\.NET.
- You can search for examples in the GTAC "Solution Center" database. Use the "Advanced Search" option, and set the document type to "NX API".
- For help with the .NET functions we have recommended, you can refer to the on-line documentation on the Microsoft MSDN site. The most useful topics are System.Math, System.Windows.Forms, System.DateTime, System.IO, System.IO.File, and System.IO.Directory.
- For general help with VB programming and .NET functions, you can ask questions on sites like stackoverflow.

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.