The GSystems FunctioNater Control

Copyright/Disclaimer

Contents

About the GSystems FunctioNater Control

The GSystems family of components is a suite of powerful tools with which software developers can produce superior quality systems.

This particular component grew out of the development of the GSystems Graphics component as something was needed to generate data for that component.

In the context of this control, an expression is some mathematical function that produces numbers given any number of independent variables; aka dimensions. An independent variable is itself just a number (though it can use an expression as in the case of a parameterized variable) that ranges from some minimum to maximum value in a certain number of steps.

This control is used in situations where you have a need for an arbitrary, or user defined, expression somewhere in an application or other control. Using this control, your end-users can enter some function that describes a particular behavior about whatever system they are working on and have the numbers representing that expression produced from it.

Part of the philosophy behind every component in the GSystems family is that there is no knowledge, nor should there be, of the context in which each component is used. Other than that the general need is one that the component is designed to fulfill. Specifically, the goal is that these components will not impact you in any way with respect to your programming model or habits nor will you find that they are "in the way" as you grow and maintain your system.

As with all of the components in the GSystems toolset, there is no use of any Microsoft MFC software or component. No '.dlls, other than the system '.dlls, are needed for implementation of any of the GSystems components. Indeed, all of the GSystems sub-systems are stand-alone '.ocx files that will have no reliance on *any* other '.dll *except* amongst themselves and then only through COM. The latter statement means that there will never be any reliance by GSystems binaries on anything other than the standard system level '.dlls (kernl32.dll, user32.dll, common dialogs, etc.), and amongst themselves, for example, when one GSystems component creates and uses another GSystems component.

Various “modes” of implementation

The GSystems FunctioNater Control can be used in different modes of operation. The control has a fully featured built-in interface that allows you to simply plug the control into your application with no further user interface programming necessary.

Alternatively, you can specify that any of the features in this interface be invisible at run time and you can provide your own mechanism to interact with the GSystems FunctioNater control.

As an example of this latter mode, suppose you had a series of expressions describing different types of investment performance measures. You could have these expressions in some database and have your application provide a choice of which method to use at run time. You would then simply set the expression for the FunctioNater control as the user exercises this choice. Other parts of your interface may contain the values of the variables you may need to define to the FunctioNater control. No part of the control’s interface even need be visible.

Alternatively, you can use the built-in interface in the control and allow the user to interact with this full featured interface. You may need to implement nothing more than some event handlers to capture the data produced by the control and save and/or display that data as appropriate to your application.

About the Company

InnoVisioNate Inc. is focused on the development of high quality and reliable software components that provide customers solid power, flexibility, and extensibility.

We know that the systems in which our components are used are built by professional developers with a keen desire to produce the best software they possibly can.

It is therefore an honor for us to be a part of these development teams.

Our goal is to continually earn our customer's trust by producing software components of the utmost quality and utility and by providing strong support for our products.

Do you have an idea for a great component? InnoVisioNate Inc. is more than willing to partner with other interests to see that the best quality components get developed and placed on the market.

About this Document

This document will describe how to effectively use the GSystems FunctioNater Control in your development efforts.

The style of this document is intended to get you started as quickly as possible while providing efficient access to the information once you understand the general concept of using the component. We feel that typical software documents are either at such a low level that they are insulting for professionals to use, or, that they are so unconnected with the actual *process* of using the tool that they are not effective in every day, real world, situations.

Part 0 - What is the GSystems FunctioNater Control?

The GSystems FunctioNater Control is used to place an arbitrary or user defined expression into an application or other component without the developed system having had to know what the needs of the end user might be.

We have encountered many times when generalized functionality is needed in an application. For example, you create a system based on some mathematical phenomenon and you have certain users who feel that one representation of the theory is preferable over another. Using the GSystems FunctioNater control, *all* your users can get what they want out of your application.

The control can be thought of as having two closely related, but separable, parts. First is a numerical "engine" that provides the number crunching and expression parsing mechanism of the control. Secondly, there is a fully featured user interface which can be presented to the users so that they can define a particular expression and all of its related variable definitions.

Part I - Instantiating a GSystems FunctioNater Control

Because the GSystems FunctioNater Control is an ActiveX Control, you can create and use the object within your environment just as you would any other control. In Visual Basic, for example, you can double click the icon in the Palette of controls, and an instance of the control will be placed on your form.

If you have already built the user interface aspects of your function definition, you can simply hide all of the interface parts of the FunctioNater control. By setting the visibility properties of the control in your development environment, you set what part of the control your user can interact with.

If all you need is the FunctioNater Control's numerical engine capabilities, you can even instantiate the control as a simple, non-visual, run time object. The control does not *need* to be activated on some window in "container" fashion.

An example of run-time instantiation in C/C++ environment is:

```
IGSFunctioNater *iFunction;
CoCreateInstance(
           CLSID_GSystemFunctioNater,
           NULL,
           CLSCTX_INPROC_SERVER,
           IID_IGSFunctioNater,
           (void**)&iFunction);
.
.
.
iFunction -> Release();
```

In the example, the vertical ellipsis is used to mean "sometime later". In practice, you simply release the interface when you are done with it, perhaps when your user closes the application. Note that the above C code needs to have the file FunctioNater_i.h "included"[1] for it to compile. Note also that the system that uses this code also needs to have the file FunctioNater_i.c file "included" in one and only one source file in the system. Both of these files are provided with the GSystems FunctioNater Control, you may simply need to manage the location of these files such that your development environment can properly find them at compile time.

Setting the properties of the GSystems FunctioNater Control

The control has a set of properties, available at both design and run-times, that set one of two categories of attributes. First, certain properties deal with the way the control is presented to the user. That is, you set what parts of the interface you want to use by making these parts visible or not. Secondly, you can set the expression and the required variables that the control will use when first instantiated.

These properties are shown in the following table.

Table 1 GSystems FunctioNater Control Properties

Property Name	Type	Notes
AllowControlVisibility Settings	True/False	Specifies whether the user (at run-time only) can tell set the visibility of parts of the control

[1] Of course, every development environment has multiple ways to "import" the definition of external objects, such as through type libraries. Therefore, your preferred mechanism may vary.

Property Name	Type	Notes
		visibility of parts of the control.
AllowPropertySettings	True/False	Specifies whether the "Properties" button is available at run-time.
DefaultMinValue	String expression	This value will serve as the default minimum value of any new independent variable found. This is a string expression that resolves to some numerical value.
DefaultMaxValue	String expression	The maximum value used in the creation of new independent variables.
DefaultStepCount	long	The default number of steps to exercise a new independent variable over.
Expression	String expression	The equation representing the function.
ExpressionLabel	String	The label over the expression entry field. Specify "" to prevent the label field from showing and using space.
ResultsLabel	String	The label over the results display field. Specify "" to prevent the field from showing.
ShowControls	True/False	Turn On/Off the visibility of all control buttons; Start, Pause, Resume, and Stop.
ShowExpression	True/False	Display the expression entry field (or not)
ShowPause	True/False	Show or not show the Pause button

Property Name	Type	Notes
		button
ShowResults	True/False	Show or not show the results display field (also affects the Results label)
ShowResume	True/False	Show or not show the Resume button
ShowStart	True/False	Show or not show the Start button
ShowStop	True/False	Show or not show the Stop button
ShowVariables	True/False	Turn off or on the tab control holding the Help, Errors, and variables tabs.

In an application development environment, such as Visual Basic, clicking on the visual representation of the FunctioNater Control typically provides an opportunity to set the above properties.

In an environment or situation where the GSystems FunctioNater Control is created at run time, you set or retrieve these properties at run-time. For example, in a VB application

```
GSystemFunctioNater1.expression =
                    "z = cos(x * y + sin(y^2/a))"
```

or, in a C environment,

```
iFunction -> put_Expression(
                  L" z = cos(x * y + sin(y^2/a))");
```

where iFunction is a variable such as that retrieved on page 2. Note the use of the L"" macro in the C example, which causes the argument to be passed as an OLE safe string.

Even if the GSystems FunctioNater Control is placed on a form at design time, you can still override these properties with calls as in the above example.

Part II – Providing access to the FunctioNater Control

There are two distinct ways to interact with the FunctioNater control from your own software. The first is to simply leave the control as it is pasted on the form and let the built-in interface be presented to the user. The other is to hide parts of this interface and to programmatically supply information to the control where needed. These two methods are kept distinct in the following discussion.

Even though the latter method involves programmatically controlling the FunctioNater, there is no reason why you cannot use the techniques described for that mode even if you have the full interface enabled for the user. Achieving some combination of the two modes as best befits the needs of your application.

In both of these modes of operation, however, it is important to note that the control's communication with your code does not change.

Specifically, the FunctioNater control uses an event mechanism to communicate its activity to your software to allow you to take any appropriate action. Further, this event mechanism doesn't change based on the method you choose to present the control to the user. See page 19 for more information.

Using the built-in user interface mechanism

This section describes the "default" user interface of the GSystems FunctioNater Control. If you simply drop the control onto a form in your application and run the application, you will be able to see all of the details of this discussion.

When you instantiate the GSystems FunctioNater Control, you should see an object as depicted in the following figure.

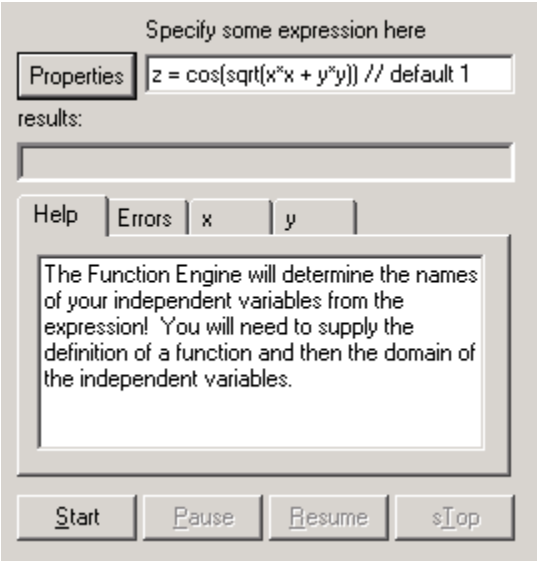


Figure 1 The standard user interface

If you make no changes to the "visibility" properties for the control, the user of your application will see the representation shown. Of course, the background color and the font from the container will be used. For this reason, the control blends correctly into your container's background window and all of the elements on the control will appear as natural members of the application.

The user defines the expression's domain

To clarify the dynamic nature of this interface, we describe what happens as the expression in the topmost entry field is changed.

As the user (or software) changes the value of the expression in the expression entry field, the FunctioNater control will parse that expression and will learn of any "undefined" variables in the expression. These variables are considered to be independent variables that will be used to define the domain of the expression.

For each of the independent variables defined while parsing the expression, there will become 1 tab for that variable on the tab control as shown in Figure 1. Note the figure is showing the tab for each of the variables "x" and "y". If you look at the expression you will see that these two variables

are the only ones in the expression. If the user selects one of the variables' tab in the interface, the properties for that variable can be set.

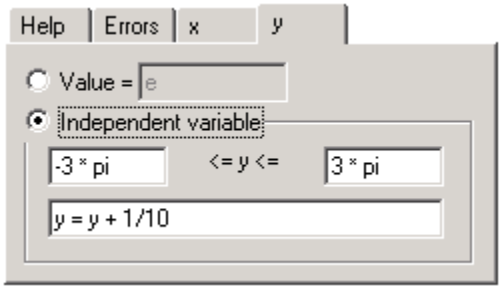


Figure 2 Variable property settings

Figure 2 shows what the user would see by selecting the "y" variable tab on the default user interface. Note that all of the necessary values for the variable would have already been set by the GSystems FunctioNater control when it first created the control[2]

The user will be defining the domain over which the independent variable is valid. Specifying the minimum value, the maximum value, and some appropriate expression that is used by the evaluator to "transition" the variable from one value to the next.

It is also possible to specify the variable as a single valued variable by clicking the "Value" radio button and specifying its value in the associated entry field.

Any of the numbers or values specified in these fields may be expressions and variables defined to the FunctioNater Control. For example, the variables "pi" and "e" are intrinsic values to the control and my be used as values just as any other number would be.

Note also there can be any number of independent variables in the expression; these variables can have as many characters in the name as you'd like; and the names *are* case sensitive.

[2] Or, when your code created the variable if you were programmatically interacting with the control.

Manipulating the FunctioNater Control programmatically

When you want to provide your own interface components for the FunctioNater control, you simply set the properties of the control that dictate the visibility of its interface components.

Then, when you want to operate the control at run time, you can call methods on the control that operate it. There are only a few methods and each of them performs some action that would have been available to the end user in exactly the same manner as if an end user caused the action, by, for example, pressing the start button.

Table 2 Methods used with FunctioNater control

Method	Description
Start	Causes the FunctioNater control to parse the expression, fire the Clear() event, fire the Started() event, and begin processing the expression over the defined domain.
Pause	Causes the FunctioNater control to halt execution in a resumable fashion.
Resume	Resumes a Paused control
Stop	Causes the control to stop execution and fire the Finished() event.
DefineIndependentVariable	Defines a variable by name, minimum value, maximum value, and # of steps to traverse in the domain.
DefineSimpleVariable	Defines a variable by name and value. This variable is not an independent variable and does not have a range.
UndefineVariable	Removes knowledge of the named variable from the Control – possibly causing the evaluator to fire and undefinedVariable event in the future.

You can simply call these methods from your own code to operate the FunctioNater control on behalf of the user's needs.

Don't forget, however, that you must set the expression that the FunctioNater should evaluate. If you don't set this expression, the evaluation will produce no results.

Part III –Evaluating the expression

Parsing the expression

There is a lot of processing within the FunctioNater Control while it is looking over the provided expression. Either when the user types a new expression into the entry field or if your code programmatically sends this expression to the control.

The point of all of this is to determine from the expression the names of any independent variables that appear in the expression.

While the expression is parsed, the FunctioNater control will send events to your code (if you implement the event handler) to help in the definition of these variables. If your code does not define the variable when it receives the `undefinedVariable()` event, then the FunctioNater control will define a default variable using the range specified in the control's property settings. To see these properties, reference Table 1 on page 2.

If you do want to provide the variable definitions before the expression is parsed, you should call the `DefineIndependentVariable()` (or `DefineSimpleVariable()`) methods on the control before providing the expression. Then, when the expression is parsed, the control will not need to send the previously mentioned event.

However, if the expression does change, specifically, if any new variables are used in the expression, you may still need to respond to the event unless the default range used is satisfactory at that point.

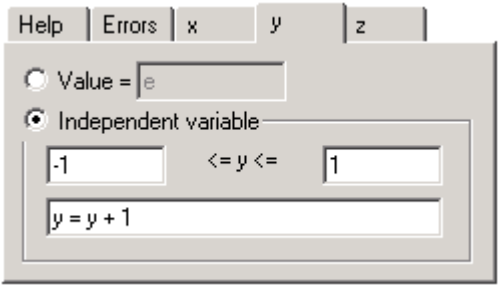
Evaluating the expression

When the FunctioNater evaluates the expression, it will be setting all of the independent variables' values in an orderly way such that the function will be evaluated for every value of every variable.

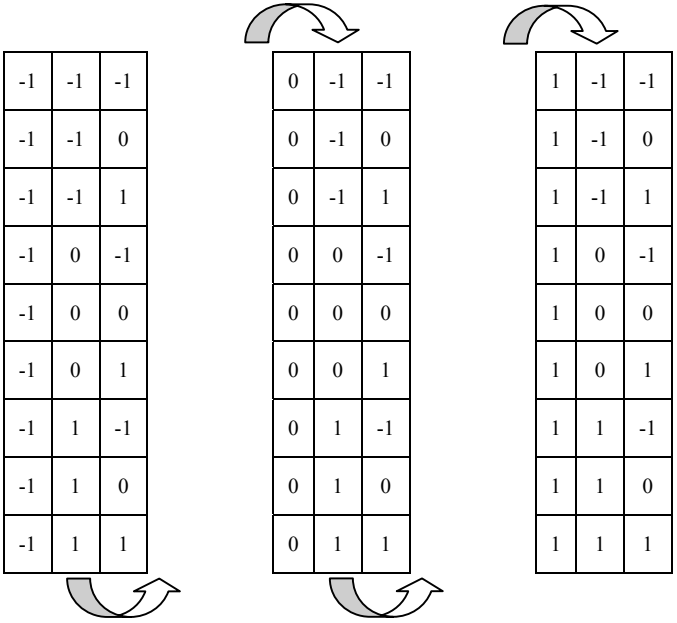
As an example, if the expression were:

```
F = sin(x * cos(x^y) + z)
```

And each of the three independent variables were defined as follows:



then the evaluator will be producing the following table of independent variable values:



-1	-1	-1
-1	-1	0
-1	-1	1
-1	0	-1
-1	0	0
-1	0	1
-1	1	-1
-1	1	0
-1	1	1

0	-1	-1
0	-1	0
0	-1	1
0	0	-1
0	0	0
0	0	1
0	1	-1
0	1	0
0	1	1

1	-1	-1
1	-1	0
1	-1	1
1	0	-1
1	0	0
1	0	1
1	1	-1
1	1	0
1	1	1

In the illustration, the rightmost variable, in this case “z”, moves the fastest. In this way, the entire domain is traversed by having every independent variable traverse its entire range. At each step of every variable, every step of every other variable will occur.

For each cell in the table, one value of the expression will be created and sent, along with the independent variable values, to your application in an event.

The evaluation process

The detailed process that will occur when the FunctioNater processes your expression is as follows.

1. The expression is parsed and each independent variable in the expression will cause an event to be generated which will arrive in your application – if you defined an event handler for the undefinedVariable event.

   The event will pass the name of the unknown variable. Here is an example in Visual Basic:

   ```
   Private Sub GSystemFunctioNater1_UndefinedVariable( _
                  ByVal variableName As String)

   GsystemFunctioNater1.DefineIndependentVariable _
                  variableName, "-3", "3", stepCount.Text

   End Sub
   ```

   This is the `UndefinedVariable()` implementation on the instance of the FunctioNater control called GSystemFunctioNater1 on the form whose code contains the above subroutine. The form also contains a text entry field called stepCount into which the user has specified the desired # of iterations per variable.

   If you did not supply this event handler, or if you did not call the `DefineIndependentVariable()` method, then the system will create a suitable default variable for use in the evaluation of the expression. Note you can also call the `DefineSimpleVariable()method.`

2. The FunctioNater sends the `Clear()` event. This tells your code that processing is about to start and that now might be a good time to, for example, clear the current storage of the function values from the previous evaluation.

3. The `Started()` event will be sent. This event passes the predicted number of values that will be sent during the processing.

4. Processing starts and your code will receive a stream of `TakeValues()` and `TakeResults()` events. These events provide your application with the results of the expression at each of the points in the domain of the expression.

   The `TakeValues()` event passes the iteration number, the number of values contained in each of two arrays (also passed), the Names() string array and the Values() double array.

   The `TakeResults()` event passes the iteration number and a formatted string containing variable/value pairs separated by a space. The variable/value pairs consist of "variableName = value".

5. During this time, the FunctioNater control might send a `Paused()`, `Resumed()`, or `Stopped()` event. These events occur when either the user presses Pause, Resume, or Stop on the interface, or code calls the `Pause()`, `Resume()`, or `Stop()` method on the FunctioNater control.

6. Also, during the execution of the expression, your code may receive one or more of the `DivideByZero()` event and the `InvalidArgument()` event. Both of these occur as the indicated invalid mathematical operation is encountered. The `InvalidArgument()` event provides you the name of the function and the value of the argument that caused the error.

7. Finally, the `Finished()` event will occur, this will happen as the expression naturally reaches the end of its domain.

If you need more information on any of the events mentioned in this process, please refer to page 19.

Part IV – Interface reference information

When you need to communicate with the GSystems FunctioNater Control, you use the interface described in this section[3], the IGSFunctioNater interface. This interface is a dual automation interface which means you can

[3] The events interface IGSFunctioNaterEvents is described in this section. However, you don't communicate with the control using this interface – you implement this interface and the control communicates with your code through that implementation.

use it in one of two ways, as dictated by the needs of your software and/or environment.

You can access the GSystems FunctioNater Control using early or late binding. In particular, you can have your development environment learn of, and use, the capabilities of the control when your application or object is being compiled and linked; this is early binding. Alternatively, for environments that don't utilize a concept of "compiling" and linking, you may need to use "late" binding which allows your application or object to discover and use the capabilities of the GSystems FunctioNater Control at run time.

In some development environments, the use of some of the methods described herein is very much like the syntax if the method were an actual property, or variable, on the interface. For example, in Visual Basic you may have the two expressions:

```
dim minValue as String
minValue = fc.DefaultMinValue
fc.DefaultMinValue = minValue
```

Where fc is a reference to an instance of the FunctioNater Control.

In reality, however, there are two distinct "methods" on the IGSFunctioNater interface. In environments that don't support the translation, you need to know the actual method name and calling syntax. Therefore, when you see the description of the `get_` and `set_` methods in the following discussion, you will know that you may be able to use the method as outlined above, i.e., as a variable on the "object" containing the interface.

The convention used in this document to represent these methods is, by example:

```
{put | get}_DefaultMinValue({BSTR | BSTR *} strValue)
```

In the above, there are two methods represented, one is:

```
put_DefaultMinValue(BSTR strValue)
```

The other is:

```
get_DefaultMinValue(BSTR* pstrValue)
```

And, in Visual Basic and other enviroments, which understand how to translate the calls into a more natural look, the syntax is:

```
fc.DefaultMinValue = strValue
```

or,

```
strValue = fc.DefaultMinValue
```

In this case, “fc” will be an instance of the FunctioNater Control. Also, the BSTR data type is an OLE Safe string; the same type which is a String in Visual Basic.

Our syntax will describe both methods at once. In particular, the “{ }“s with the “|” inside denote that one or the other of the items to either side of the “|” is to be included (of course, spaces would be removed where they cause syntax errors). Also, that if more than one of these appear in a single construct, then the same numerical order of selection that was made in the first group must be made in every other one as well.

When you see the argument is passing a pointer, you are generally expecting that the object will “return” a value to you (the caller) through this method[4]. Visual Basic will deal with this appropriately if you use the standard assignment syntax as described above.

Note that if no `put_` method is documented, then that “variable” may not appear on the left side of an ‘=’ statement. Assignment is not possible with such an item.

The IGSFunctioNater Interface

This is the only interface available with the GSystems FunctioNater Control and is used by your software to control every aspect of the control’s operation.

This discussion will be grouped in two areas:

1. Properties – descriptions of the properties available for the control.

2. Methods – descriptions of the methods used to exercise the control, for example, to make it start evaluating the expression.

The Properties of the FunctioNater Control

Again, these properties may be set at either design time, or run-time, or both. Even if you set the values of the properties in design mode, if your

[4] With some exceptions, usually where passing an “object” would be more efficient by passing a pointer to an object instead.

Form load method (in VB) calls these methods, the properties will have the values supplied by the Form load method.

```
{put | get}_AllowControlVisibilitySettings(
                {BOOL | BOOL*} isAllowed)
```

This property is typically used to prevent the user from setting the visibility of the interface components on the FunctioNater Control. This is because setting these visibilities will resize the control and by so doing might obscure other controls on your application.

```
{put | get}_AllowPropertySettings(
                        {BOOL | BOOL*} isAllowed)
```

This prevents the user from pressing the "Properties" button (by hiding the button) at run-time. You might typically want to turn this off if your application is going to fully manage the definition of the expression and all of the variables. Setting this value to TRUE will mean the user can manipulate the expression and variables at run-time. Even if the variables display has been disabled (page 17), if the user edits the properties of the control, the variables will be visible there.

```
{put | get}_DefaultMaxValue(
                {BSTR | BSTR*} strMaxValue)
```

This property specifies the default expression to use when the system creates a variable. For example, when finding an undefined variable in an expression and the host software does not provide an event handler for `UndefinedVariable()`.

Remember that the value provided this property is a string expression. Any number can be represented by an expression as long as the expression (which will be evaluated by the FunctioNater Control) will result in a number when evaluated. For example, an expression can use the names of previously defined variables.

```
{put | get}_DefaultMinValue(
          {BSTR | BSTR*} strMinValue)

{put | get}_DefaultStepCount(
          {long | long*} stepCount)
```

These two properties are for the same purpose as in the previously property. Note that the `StepCount()` property is a long value indicating the number of iterations to do between the minimum and maximum value of any auto-created variables.

```
{put | get}_Expression(
          {BSTR | BSTR*} expression)
```

This property specifies the expression the FunctioNater Control should evaluate. Note that the control will parse this expression as soon as it is supplied. This means that if the control is in run-time mode, that your application may receive the `UndefinedVariable()` event (page 19). At design time, the FunctioNater Control will be providing default variable definitions as appropriate.

```
{put | get}_ExpressionLabel(
          {BSTR | BSTR*} expressionLabel)
```

A label is displayed over the expression entry field in the user interface. If you want to customize the contents of this label, you would use this property. If the property's value is "" (an empty string), the label is not shown and the height of the FunctioNater Control is reduced accordingly.

This label is also not shown if the expression is not visible.

```
{put | get}_ResultsLabel(
          {BSTR | BSTR*} resultsLabel)
```

A label is displayed over the results display field in the user interface. If you want to display different text, use this property. If the property's value is "" (an empty string), the label is not shown and the height of the FunctioNater Control is reduced accordingly.

This label is also not shown if the results field is not visible.

```
{put | get}_ShowControls (
          {BOOL | BOOL*} isShowing)
```

There are 4 controls that the user can use to operate the FunctioNater Control; i.e., the Start button, etc.

If you want to remove this capability from the user, set this property to FALSE. Note that this applies to *all* controls rather than to a single one of them, as in the following properties.

```
{put | get}_ShowPause(
          {BOOL | BOOL*} isShowing)

{put | get}_ShowResume(
          {BOOL | BOOL*} isShowing)

{put | get}_ShowStart(
          {BOOL | BOOL*} isShowing)

{put | get}_ShowStop(
          {BOOL | BOOL*} isShowing)
```

These 4 properties control each of the control command buttons on the built-in interface. Specifically, you can show or hide any combination of these buttons. As you do so, the buttons will realign themselves left to right to prevent "gaps" in the interface.

The associated buttons will not be showing if the `ShowControls` property is FALSE.

```
{put | get}_ShowResults(
          {BOOL | BOOL*} isShowing)
```

The results window is a simple (disabled) entry field that is populated with the value of the results each time the FunctioNater Control calculates a new one. You can turn the display of this field off if it is of no interest to your end users.

```
{put | get}_ShowExpression(
          {BOOL | BOOL*} isShowing)
```

This property can turn off the expression display altogether. If the expression field is not visible, then your application or code would be responsible for providing the function definition. Also, when this property is FALSE, there is no chance the user would enter an arbitrary expression which may not be desirable in the context of your current application.

```
{put | get}_ShowVariables(
          {BOOL | BOOL*} isShowing)
```

This property provides the ability to turn off the variables interface component. This component is the tabbed control containing one tab for each variable plus a help and errors tab. If this property has the value FALSE, the FunctioNater Control will resize in order to take up considerably less vertical space.

The Methods of the FunctioNater Control

These methods are intended for programmatic manipulation of the FunctioNater Control.

```
Start()
Pause()
Resume()
Stop()
```

Their function is obvious from their names. Note that the control buttons on the built-in interface provide the user level access to these methods.

Also, either when the control buttons are used, or these methods are called by software, the enabled state of the buttons on the interface reflects the running state of the FunctioNater Control.

Specifically, until `Start()` is called, all buttons except Start are disabled. When `Start()` is called, Pause and Stop are enabled and Start is disabled. When `Pause()` is called, Resume is enabled and Stop is disabled. Finally, when `Resume()` is called, Pause and Stop are disabled.

```
DefineIndependentVariable(
                BSTR variableName,
                BSTR minValue,
                BSTR maxValue,
                long stepCount)
```

This method defines the named variable to the system. The variables domain will range from the `minValue` to the `maxValue` and will include `stepCount` iterations.

Keeping in mind that the minimum and maximum values are string expressions and can therefore contain any mathematical expression.

```
DefineSimpleVariable(
                BSTR variableName,
                VARIANT value)
```

A “simple” variable contains just a single valued. In this call, the `value` is a VARIANT which means it may be any kind of number. It can also be a string value which will get evaluated as in the description of the previous method.

```
PersistTo(BSTR fileName)
PersistFrom(BSTR fileName)
```

This methods allow the FunctioNater Control to save (or restore) it's internal data to the named file in case the user has modified any of the properties of the control. You may also use any of the standard COM persistence mechanism or use the GSystems Properties Control for extensive persistence support in this and other applications.

The FunctioNater Control's Events interface

The events interface is implemented by the client of the FunctioNater control.

Though it is not required that your code implement all of this interface, you will need to implement at least the `TakeValues()` method if you want to receive any data from the FunctioNater Control.

`Clear()`

This event will occur before evaluation of the expression starts. It will also occur before the expression is parsed and so it therefore happens before any UndefinedVariable() events. This is generally a good time to clear the storage in which you want to store the data generated by the FunctioNater Control.

`DivideByZero()`
`InvalidArgument(BSTR functionName,double arg)`

These events occur during execution when the indicated math exception occurs. You can, for example, call `Stop()` when you receive one of these events if you feel it is indicating something needs to be fixed.

`UndefinedVariable(BSTR variableName)`

This event passes the name of the variable that is not recognized in the expression. When you receive this event, you can call the `DefineIndependentVariable()` method, passing the provided variable name. If you do not do this, the FunctioNater Control will create a suitable variable.

`Parsed()`

This event occurs just after the FunctioNater Control has completely parsed the expression and has all the information it needs to evaluate the expression. When you receive this event, it is safe to call `Start()`.

`Finished()`

When the expression has been evaluated over the entire domain, this event is sent. There will be no more events stemming from the execution sequence for which this marks the end.

```
Started(long cntExpectedResults)
Paused()
Resumed()
Stopped()
```

These events occur as the user controls the evaluation of the expression or as some software calls the associated methods on the FunctioNater Control.

The `Started()` event passes the estimated number of iterations which will occur.

```
TakeResults(long iterationNumber,BSTR results)
```

As the expression is evaluated, this event will occur to provide your software with the results at each point on the domain.

The iteration number is a sequential number from 1 to the # of values produced by the expression. If you store this number locally, setting it to 0 when you receive the `Clear()` event, when you upon receipt of the `Finished()` event, that stored number will indicate the total number of values received.

The string passed in this event has the form:

"x = -1.000 y = -2.000 z = -3.000"

for an expression that provides the value z and has the independent variables x and y. Note that the value/variable pairs have an "=" sign (with a space at either end of the "=") and that the pairs are separated from each other by a single space.

```
TakeValues(long iterationNumber,
           long cntValues,
           BSTR variableNames(),
           double variableValues())
```

This method provides your software with the actual values of the expression at each point in the domain.

The iteration number is as described in the prior event. The second parameter indicates the number of members in the following arrays. Which are the names of the variables, and the values of the variables. The arrays contain entries in the same order in both arrays, i.e., a variable named x may

be in position 1 in the names array, it's value will be in position 1 in the values array.

Note that the value of the expression, i.e., the result of the expression given the current value of all independent variables, is the last element of the `variableValues` array. Also, the last element of the `variableNames` array is the name of the expression, which is "result" if there is no "=" sign in the expression, or the characters to the left of the "=" sign if there is one in the expression. Thus, the value of `cntValues` will be one more than the number of variables in the expression.

Epilogue

First, allow us to thank you for purchasing the GSystem FunctioNater Control and giving us a chance to help with your development efforts.

We never forget the privilege it is to be asked to join your team !

Though we try as best we can, it is inevitable that you might have discovered some bug in the tool or inconsistencies in its use with respect to this document.

In either case, we would very much like to hear from you if you've had any problems. Indeed, we appreciate any form of constructive criticism as well as any kudos or good ideas you can provide as well.

One of the very important reasons for this is that there are so many environments, types of applications, and other factors within which our software may get used that we could not possibly have thought of all the features we needed to put in the system. Your assistance can help us to get any needed features and fixes into the product as soon as possible.

We would also like to know how well this manual has presented its material and if you think some other approach would have been more effective. We believe that proper technical documentation is something of a young and fledgling science and we don't purport to be the best at it … yet!

With your help, however, we can get there.

Thanks again !

problems@innovisionate.com

ideas@innovisionate.com

support@innovisionate.com