

The GSystems Properties Component

Copyright/Disclaimer

This Document and the software described herein are protected under Copyright by InnoVisioNate Inc.

To the maximum extent permitted by applicable law, InnoVisioNate Inc. shall not be held liable for any damages; whether they be consequential, incidental, indirect, special, or of any other type, whatsoever. These may include but are not limited to; damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss arising out of the use of, or inability to use, this product.

Contents

Glossary	v
About the GSystems Properties Component	vi
<i>The component deployed at the application level</i>	<i>vii</i>
<i>The component deployed at the component level</i>	<i>viii</i>
About the Company	ix
About this Document	ix
<i>Part 0...</i>	<i>ix</i>
<i>Part I...</i>	<i>x</i>
<i>Part II...</i>	<i>x</i>
<i>Part III...</i>	<i>x</i>
<i>Part IV</i>	<i>xi</i>
<i>Part V</i>	<i>xii</i>
<i>Part VI</i>	<i>xii</i>
<i>Part VII</i>	<i>xii</i>
<i>Overall...</i>	<i>xii</i>
Part 0 - What is the GSystems Properties Component?	1
<i>What is persistence?</i>	<i>1</i>
<i>Provide simple “memory” persistence for your system.</i>	<i>2</i>
<i>Easily create a record-oriented “database” application - without the database</i>	<i>2</i>
<i>Manage contents of windows or controls</i>	<i>3</i>
<i>Provide properties settings dialogs</i>	<i>3</i>
<i>Provide run-time configuration and persistence of objects within your system.</i>	<i>3</i>

<i>Instantiating a GSystems Properties Component within your development environment.</i>	4
<i>An important note about timing</i>	5
If you are an MFC user	6
<i>Setting the properties of the GSystems Properties Component</i>	7
<i>GSystems Properties Component Summary</i>	8
Part I – What Are Properties ?	10
<i>Creating properties</i>	11
<i>The Types of Properties</i>	12
<i>Storing sets of objects within a single property</i>	13
<i>Providing Direct Access Storage</i>	16
<i>Using Normal Access to Storage</i>	18
<i>Combining an Array Property with an Array of Windows Controls</i>	19
<i>An important note about the “Set” of properties</i>	21
Part II – Provide Application Level Persistence	23
<i>Developing a persistable application – by example</i>	23
Instantiate a GSystems Properties Control object	23
Set the properties of the Properties Component	24
Create Properties	25
Provide the persistence mechanism to any other objects	27
Participate in the persistence process	27
Use the file manipulation capabilities	30
Load from an existing file	31
Saving to a file	31
Summary of the file manipulation methods	32
Part III – Provide Component Level Persistence	34
<i>Developing a persistable object – by example</i>	34
Quick Rules for Persistence	36
Step 1: Create an instance of the GSystems Properties component	37
Step 2: Create the set of properties maintained by the component	38
Step 3: Provide the path for your client to find the implementation of persistence interfaces through your object or application	39

Step 4 (Optional) provide access to your system for the GSystems Properties Component	44
Part IV – Intercepting the Save, Load, and InitNew functionality of the Properties Component	48
<i>A Clarifying Point Concerning Object Hierarchies</i>	48
<i>The mechanics of implementing the IGPropertiesClient COM interface</i>	48
Providing an interface in a VB project	49
Providing an interface in a MFC project	50
Providing an interface in a ATL project	51
<i>The specifics of the IGPropertiesClient interface</i>	52
Initializing a New Object	52
Saving the Object's State	53
Ensuring multiple objects are persisted	53
Ensuring actively changing numbers and types of objects are persisted	54
Restoring the Object's State	54
Part V - Using the Property Page implementation	55
<i>An Introduction to Property Pages</i>	55
The Mechanics of a Property Page	57
<i>Providing a property page window handle to the Properties Component</i>	58
Providing the easier properties windows – Summary	59
<i>Enable the use of IGPropertyPageClient and custom property page windows</i>	61
The Property Page Creation and Manipulation Processes	63
Set up properties comparison	64
Highly optional implementation of the IPropertyNotifySink interface	66
<i>Showing the property pages of other objects</i>	68
Adding property pages of any object	68
Editing properties of any object	69
Part VI – The relationship between properties and common windows controls	71
<i>IGProperty/IGProperties Window/Control Support methods</i>	71
Part VII – Interfaces reference information	76

<i>The IGProperty Interface</i>	77
Miscellaneous IGProperty methods	79
IGProperty value methods	81
The window relationship methods	84
<i>The IGProperties Interface</i>	84
Miscellaneous methods in IGProperties	85
File handling methods on IGProperties	88
Property management methods on IGProperties	89
Persistence methods in IGProperties	90
Property page methods in IGProperties	91
Initiate the edit properties process	91
Property Stack and Comparison Methods	92
Simple Property Page methods	93
More Extensive Property Page Support Methods	93
The window relationship methods	96
<i>The IGPropertiesClient Interface</i>	96
Persistence IGPropertiesClient Methods	96
<i>The IGPropertyPageClient interface</i>	97
Epilogue	100

Glossary

ActiveX	A Microsoft trademark. Sometimes used to indicate a COM component that can be used as a control on an application window. Within this document, the term ActiveX is interchangeable with “COM Object”.
COM	Component Object Model – Two complimentary sets of rules and regulations. Providers of functionality can follow one set in such a way that developers needing that functionality can follow the other set to utilize that functionality seamlessly and without trouble.
Client	The user, or consumer, of an object. When a COM object is placed on a Visual Basic form, the Visual Basic code accessing that COM object in the form is the client. The COM object itself and the binary code that implements the object is called the server.
Interface	A set of functionality. Strictly, a precise description of a group of functions, including the arguments and return values, is an interface definition. An interface <i>implementation</i> is a physical instance of this set of functions. This existence is as a table of functions which is the same as a C++ Virtual Function Table. <i>Any</i> software environment that can create and/or manipulate a table of functions like this can participate in COM.
Persistence	The act of saving a state at one point in time and then, at a later point in time, returning to exactly the same state.
Server	An external binary software resource accessed through COM that a client application, or any other software artifact, is able to create and use.
Storage	The place where properties are saved too and restored from. Technically a file but in an operating system format specifically geared towards the persistence of COM objects. Directly accessed through operating system provided functionality.

About the GSystems Properties Component

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

This particular component is one of the core tools in the GSystems family. It evolved as the need for persistence across all of the other components within this suite was recognized.

As a user of software, the developer of this tool feels that one of the obvious indicators of poor quality in software, is when the developers have not taken the opportunity to make it a personal experience for users. In other words, while using software, a user will “configure” their experience, for example, by sizing and placing a window of the system. That user is expressing a preference by doing so; for the system to recognize these preferences, and to respect the user enough to remember them the next time the software is run, is a clear indicator of quality in software.

There is a great deal of complexity in the COM persistence model. It is not this complexity, however, that drove the need to develop this product. It was that, after having gone through a lot of effort in implementing the stock COM persistence model, it turns out to be weak, inflexible, and a lot of work to repeat. There needed to be a way to provide a capable persistence and property pages mechanism that provides access to each of the steps in the processes involved yet which would limit the amount of software the developer would have to provide to accomplish the task.

Thus, the GSystems Properties Component embodies the right mix of ease of use while still providing access to the entire process. This offers the power and flexibility to allow developers to meet their initial needs quickly and easily while giving them the flexibility to evolve and improve the systems they develop as their own users’ needs become more complex.

It is true that certain “COM enabled” tools, such as Microsoft MFC (Microsoft Foundation Classes), implement persistence using wizards to generate code. However, as happens with all real projects, it doesn’t take long before you find these solutions are insufficient. Perhaps the first place you notice this is when your object utilizes other COM objects and you need to allow these objects to persist as well. No generic implementation of persistence can possibly “know” about your system’s object/ownership hierarchy, either as it exists at code generation time, or as it might (and probably will) evolve over time.

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.

The GSystems Properties component is designed to allow your application or object’s code to participate thoroughly in the entire process such that you can direct this process to work on behalf of your system how *you* see fit, not how the authors of MFC or ATL (ActiveX Template Library) presumed you would use it.

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 (kernel32.dll, user32.dll, common dialogs, etc.), and amongst themselves, for example, when one GSystems component creates and uses another GSystems component.

The component deployed at the application level

At the software application level, the GSystems Properties Component is the tool that can make the system present itself to the user, when started, exactly as that user had last utilized the system. Of course, this is more than just the size and placement of key windows in the application, it can extend to *any* part of an application or system which varies based on user action.

The major parts of this mechanism are;

1. Managing the state and value of properties as they relate to active windows controls on the user interface.
2. The saving and loading of properties to a Storage.
3. The interaction with the end-user to specify a save, open, or save-as file-name.

4. The presentation to the end user of property pages defined for the application as well as for any other COM object in the application that implements property pages.

An additional and powerful feature of the GSystems Properties component is that you can provide to your users the properties settings dialogs of *any* ActiveX or COM object while at the same time, the settings that the user chooses through these dialogs will be persisted right along with the other parts of your system.

Thus, for example, you can let users configure, at *runtime*, complex windows components, such as the list view control, as *they* prefer in the same way that you configure such objects in the development environment.

The component deployed at the component level

When you utilize the GSystems Properties Component in the development of your own components, you are generally providing persistence “services” that clients of your components will call upon to persist within their (the client’s) own context. In other words, you typically won’t deal with the “application” level persistence issues, such as prompting for file names, creating a storage and stream upon which to store the objects, and the saving and restoring of the all object’s properties.

Instead, you are providing the services which enable your client’s application to seamlessly provide persistence, at least for your objects, through these services.

These two “arenas” where you utilize the GSystems Properties Component do have overlapping boundaries. If it seems confusing why the distinction has been made, perhaps it will become clearer once you utilize the component in both of these kinds of development activities.

In any case, one reason for the distinction is that in this document there are places where the discussion or example is more relevant to one type of environment than to the other.

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 Properties Component 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...

Describes the GSystems Properties Component as the “Master of Properties” and as a component that can provide a wide range of services to your application or object.

Part I...

Describes “Properties” in the context of the GSystems Properties Component as a development tool. The term “properties” is used commonly in the component development arena to indicate characteristics, or attributes, of objects which are typically accessible through certain methods on the object’s default IDispatch interface. These methods generally set or retrieve the value of some variable within the implementing object’s variable space.

However, the properties discussed in this document are actually very powerful objects in their own right. The knowledge of the windows API embodied within these objects makes them much more completely integrated with the windows and controls of your application than mere variables are. For example, as objects, GSystems Properties can associate with the entries in a Combo-Box naturally without your having to write any code whatsoever, other than a single call to the appropriate method on the property object.

Part II...

Focuses on the use of the component in the development of applications. This covers the implementation of persistence for applications that you develop as opposed to within objects that are part of applications.

The basic framework for this discussion will be around the development of a system using Visual Basic. The general concepts will apply to any application development environment, though there would be differences in the syntax, etc., in other environments, such as PowerBuilder and Delphi. If you find a problem or have a question about implementation in any other environments, please don’t hesitate to contact us at support@InnoVisioNate.com.

Part III...

Deals with the development of COM/ActiveX objects which are intended for use by inclusion in some other software system built by a different developer. Here the persistence is somewhat different in that it is being “driven” by something outside of the code you are working in. For example, some development environment may be asking your object to persist, perhaps within a development environment. However, it *could* be that you are participating in run-time persistence, if, for example, your client’s client

(the end user) is running the application built by your client. One would hope that this developer would have used the GSystems Properties Component and followed the techniques described in Part I, though, of course, this is not necessary for the success of your object in persistence.

The discussion in Part II will be centered on the use of C/C++ in the construction of a COM object. Further, except for clarifying points, the discussion will make *no* reliance on any particular COM technique in C/C++, such as the MFC COleObject class or on ATL.

Even though our discussion does not center around MFC or ATL, there have been extensive examples provided with this system on how to use this tool effectively in each context. This document will reference these examples where clarification needs to be made.

To the extent that you are developing your COM objects in other environments, such as Visual Basic, it is hoped that the discussion in Part I will be able to answer any questions arising out of the difference in environment that are not answered in Part II.

In both arenas, we stress that it is not our intent to foster, or recommend, any one environment over another. One of the admirable goals of the COM model is that development ought to be language independent and people should be encouraged to work with the tools they know best while the fruits of their labors are available to all regardless of implementation language or environment.

The GSystems family of components is fully committed to this goal and through such commitment we ask you to please help out by letting us know when the component does not live up to the nature of the objective. For example, should you be developing in some environment and find that our components do not work, or work inconsistently within that environment, please let us know.

Comments are welcome at ideas@InnoVisioNate.com

Part IV

This section offers a thorough discussion of the methods you supply in the IGPropertiesClient interface. Since these methods are critical in the process of saving and restoring properties and managing the state of your application or object's internal data with respect to the properties, they have been given their own section in this document.

Part V

Part V will cover the rich set of capabilities the GSystems Properties Component presents which allow you to quickly present to the user, and importantly, manage the process of, property settings dialogs. These are the common user interface component that presents the typical settings dialogs that we all recognize as members of a tab-control. Where each tab in the control refers to the settings of a particular object, or class of properties.

This section will also cover the details of providing the properties dialogs of other objects used in your system to the user.

Part VI

This section will deal with the association between the properties and the window controls that you may be using in your application or object. There is a family of methods that you can use to quickly swap the value of a property between window objects, such as edit fields.

A discussion of the array properties and array of controls is also found here.

Part VII

Finally, for reference purposes, and for those who prefer to simply soak up all the relevant information without any supporting information; there is Part VII.

This section is pure reference and describes each interface and method without context, and hopefully, without reliance on any other part of the system.

Overall...

Any time a method (function) on one of the interfaces of the component is discussed, it will appear in a fixed font, such as `method()`. Note that the presence of the “()” is simply a further indication that it is a method. However, even though there may be nothing between the “(“ and “)”, this *does not* always mean that the method has no arguments. The () are simply there to reinforce that the indicated name is a method. There will generally be arguments required for most methods.

Part 0 - What is the GSystems Properties Component?

The GSystems Properties Component is a COM enabled object that you can use to provide a robust persistence mechanism within your own application or object. This tool also provides several tools that also support aspects of persistence, such as the manipulation of property values by users, the inclusion of arbitrary object's properties into the persistence mechanism, and managing the relationships between windows objects and property values.

What is persistence?

When an application runs, and the user makes changes to the application, to the interface, to the data embodied by the application, or to anything about that system that can change in any way; it is very nice, from the user's perspective, if the application retains anything of value from the perspective of the user.

Stated another way, lets say a user of your application has already answered some questions, or has arranged your application the way that is most pleasing to herself. Now, when she runs that application again and the system either re-asks the same questions (!) or does not remember the way that it was set up previously as clearly desired, then this user is not going to be very impressed with the system.

At a simple level, the GSystems Properties Component could be thought of as an automated implementation of preferences. Many programmers are familiar with this and know how to read and write entries into some form of persistence file, it is tedious but works. Use of the GSystems Properties component achieves this result easily and naturally.

However, that is the component at its simplest level. Perhaps this is all you need. Certainly, this goes a long way to providing the kind of "memory" an application should have as described above.

The GSystems Properties Component is able to provide much more far-reaching capabilities in the management of features within your application. Should you choose to use these features, you can achieve the following levels of persistence in your system.

Provide simple “memory” persistence for your system.

As stated previously, the GSystems Properties Component in its simplest form will allow you to “remember” how your user last used the system. You simply need to make “Properties” with the Properties Component and then use these properties just as if they were variables in your system. When the user exits the application, simply call `Save()` on the GSystems Properties Component control. When your system restarts, you call `OpenFile()` on the control and each of the properties will have the values they had when at the end of the last run, in any case, they will have the value that was last saved to the file.

The Properties Control creates properties that you can use *exactly* like any other variable in Visual Basic. In other words, your entire variable space can be saved and restored through runs of your system. To achieve this, all you have to do is to provide the startup code that creates the properties from the Properties Control, one call for each property (variable), and that makes the single calls to open the file on program startup and to save the file on program exit.

Easily create a record-oriented “database” application - without the database

Persistence on behalf of a specific window in an application can be thought of as representing a “record” with that window. Specifically, you can use the GSystems Properties Component to remember all of the contents of a window, and the properties and contents of objects upon that window. Then, you can provide a simple mechanism whereby the user selects one of many persistence files¹ and you use the Properties Component to load the appropriate screen with the data from that file. Saving and restoring the contents of a window just as if these were records in an actual database.

¹ The user doesn’t have to know about “files”. You could present some list to the user that “represents” files and which your application would know how to associate with the correct file. This list could itself be maintained by an application level property that is an array of strings.

Manage contents of windows or controls

The property object has methods that allow you to manipulate the contents of controls within your application and to associate the property with the control.

For example, you can call methods on properties that you have associated with drop-down list boxes, probably a string array property, to manage the relationship with the value of the property and the list box, for example, the current selection in the list box.

Provide properties settings dialogs

It is customary to provide a dialog that the user can raise to edit and/or browse the properties in the system. The GSystems Properties Component provides the framework that you can use to accomplish this task easily.

This framework includes presentation and manipulation of the dialog, using a window that you create, and it includes the “process” of communicating the user’s actions during this activity with your application. So you can, for example, correctly react to the “Ok”, “Cancel”, and “Apply” buttons as the user is working with the property settings.

Also, you can direct the GSystems Properties Component to display the properties settings of any of the objects or components in your system (that support property pages), either individually, or all together in one properties settings dialog². Thus, your application will appear to be providing the functionality of the COM objects you use, presenting an ever more seamless face to the end-user.

Provide run-time configuration and persistence of objects within your system.

Many windows objects contain properties that you set in your application development environment that control how these objects look and act. You can allow your users to set these same properties and these objects will look

² This is a powerful feature; using the GSystems Properties Component, you can now combine property pages from several objects and present these to the user as *one* property frame. That is, one dialog containing the property pages from distinct, and separately developed, ActiveX/COM objects.

and act as your users desire. All of your end user's preferences could not have been known when you initially placed these controls within the application.

This is very important when you are integrating complex objects within your application. Consider that these objects have properties (sometimes a large number of them) that govern much more than just the visual representation of the object. Also, consider that these objects may be a big factor in your entire application. You probably don't want your user's to have to specify these properties every single time they run the application. However, at the same time, you can imagine that your users may very well appreciate the rich functionality of these controls. Now, you can present all of the functionality of a given object directly to your users at run-time.

If ease of development is not reason enough, consider that, using the GSystems Properties Component, you don't even have to know *what* objects are going to be used by your system at run-time! For example, imagine a configurable "host" system that allows the user to select, at run-, or configuration-time, the actual object to use for a particular purpose. This is one of the truly powerful features of COM. To achieve goals like this, you must avoid having knowledge of specific COM objects built into your application code, to the extent possible, of course.

Instantiating a GSystems Properties Component within your development environment.

Because the GSystems Properties Component 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.

In a C/C++ environment, you can declare an instance to the GProperties interface, and then create this instance using a simple call to COM³, for example:

```
IGProperties *iProperties;  
CoCreateInstance (
```

³ The GSystems Properties Control does not care if it is instantiated as an ActiveX control or not, i.e., it does not require that it have some visual representation on a form or parent container and is built to not rely upon the existence of this kind of environment.

```

        CLSID_GSystemProperties,
        NULL,
        CLSCTX_INPROC_SERVER,
        IID_IGProperties,
        (void**) &iProperties);
    .
    .
    .
    iProperties -> 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 in the `OnFinalRelease()` method in an MFC generated ActiveX control. Note that the above C code needs to have the file `Properties_i.h` “included” for it to compile. Note also that the system that uses this code also needs to have the file `Properties_i.c` file “included” in one and only one source file in the system. Both of these files are provided with the GSystems Properties Component, you may simply need to manage the location of these files such that your development environment can properly find them at compile time.

An important note about timing

You may need to be aware of the sequence of events that will occur in your software while the process of persistence is underway. If you are creating an application, it is likely that this process is entirely under your control so you may not encounter this issue.

The first important thing to remember is that the GSystems Properties are objects that you must create within code when your object starts up, perhaps in your main `Form_Load()` method for an application, or in the constructor of your object if you are building a control or component.

In any case, remember that the property objects must exist as early on in the process as follows. Of course, since you generally use the GSystems Properties Component as the creator of properties, your instance of that object must also exist as early as possible.

To clarify the reasons for this, consider the timing issues involved if you are creating a component. Some client of your component will obtain an interface to your component through a call to COM, which will result in your object getting created through a mechanism provided either by you or the framework within which you built your object, ATL, for example. In any case, the only thing guaranteed to have happened before *every* client that will use your object is this “create instance” code. If you, within this code, create your instance of the GSystems Properties Control, *and*, through

that, create all of your properties⁴, you will be ready to persist whenever your client requests you to do so.

Without knowing all of the clients which may use your object, you can't know what those clients will do in terms of persisting your object, however, you can be prepared to provide persistence support irrespective of how any client asks for it.

If you are an MFC user

The `COleControl` class is the base class of objects generated with the ActiveX Control wizard. Since this base class (actually, one of it's parent classes) provides persistence support, it is necessary to take a few minor steps to ensure the GSystems Properties Component can be used to it's full potential in this case.

First, you must "re-direct" interface queries for the persistence interfaces on to the instance of the GSystems Properties Component. This is discussed in detail on page 40.

Next, you must be aware of the timing issues involved. A lot of clients will attempt to persist your object very early on (even just after creation). This has an impact on your instance of the GSystems Properties Component. Primarily, if you want to use the GSystems Properties Control as an ActiveX control integrated into your system through the resource editor, i.e., as a visual component. You must ensure that the window that contains this instance is created in the code in the constructor of your main object!

However, with `COleControl`, you can't simply create your main window in the constructor without other problems. Typically, your window is not valid until sometime after the `OnCreate()` method is called. Nevertheless, you can create a window, for example, the one containing the GSystems Properties Control, in this constructor if you create a parent for it first. Then, in `OnCreate()` you can switch that window's parent back to the originally intended parent. This is less difficult than it may seem. A complete example is provided in the `FunctionContainer` example.

⁴ If you implement `IGPropertiesClient`, you might as well put the code to create the properties in the `InitNew()` method on that interface. You could even put the code that creates the GSystems Properties Component there *as long as* you only create that object once during the instance of your object (or application).

Note that it is not necessary to have a visual component representation of the GSystems Properties Control and there is no real advantage to it; there are only a few properties on this component that you would set in a development environment anyway. This means that you can create your instance of the component direct through COM and the above issue will not exist.

Still, if you do have ActiveX controls on some window in your application (or object) that you want to persist, those controls must also exist when the system will try to “load” your object. In this case, you may again find it necessary to create the parent windows of these objects as early as possible.

Setting the properties of the GSystems Properties Component

The component does have a small set of properties that it will use primarily for file management when it is used in an application. Especially when its services regarding the opening and saving of files are used. Again, this is typically in an application rather than when the component is used in support of persistence in an object.

These properties are shown in the following table.

Table 1 GSystems Properties Control Properties

debuggingEnabled	True/False	Causes messages concerning invalid usage of the component to appear during run time. Intended for the developer while the application using the component is under construction
FileAllowedExtensions	string	A semi-colon delimited list of extensions that represent the “types” of files originally listed in the File-Open dialog box.
		Example: *.dat;*.exe;*.txt
FileName	String	The current active file
FileSaveOpenText	String	The text in the title-bar of the file open dialog

		open dialog
FileType	String	Descriptive text appearing in the “File Type” drop-down box of the File Open dialog box.

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

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

```
GProperties.FileName = "d:\temp\settings"
```

or, in a C environment,

```
iProperties -> put_FileName(L"d:\\temp\\settings");
```

where iProperties is a variable such as that retrieved on page 4. 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 Properties Component is placed on a form at design time, you can still override these properties with calls as in the above example.

GSystems Properties Component Summary

This section has described how to instantiate a GSystems Properties Component and use it to load a set of properties that the component will manage.

The true power of the Properties Component is more than just the simple storage of the values using the persistence mechanism. There are two other key areas where the GSystems Properties Component is intended to be useful. These are:

1. Integration into the *process* of persistence, by communicating with your object or application whenever the Properties Component is performing one of the significant steps in the persistence process. For example, just before saving the binary data to the storage, you have the opportunity to have your own software perform steps that are important for your object or application, such as ensuring the properties in sub-objects are also saved at the same time on the same storage.

2. Control and integration into the *process* of setting the properties via property pages. When the user asks to set the values of the properties, the Properties Component helps to manage the actual user interfaces that will be shown. More precisely, it shares the work in implementing the user interface components, which the system can display to directly manipulate these properties. At the same time, your object or application may, if you choose, be thoroughly involved in the events that are occurring while the user is manipulating these interfaces

Part I – What Are Properties ?

Each “property” you create with the GSystems Properties Component is a full featured object in it’s own right.

The fact that these properties can contain data, just like variables do, and that this data is persistent, that is saved and restored across instances of an application or object, is but one of the important aspects of properties.

Properties have other features that make them integrate well with important parts of your software system. This feature set easily maintains the “connection” between the property’s underlying data, and the current state of the system in such a way that frees your software from many of the details of managing the relationship between windows and the data within your system.

As an example, you can create an array property and provide it with the set of strings you’d like in a list box control. Then, you simply tell the property to set the contents of a list box control and the list box is populated. You no longer need to clear the control and loop over your array of strings inserting them one-by-one into the control. You don’t even have to maintain an array of strings, the Property object will be doing that for you, maintaining any additions or changes through user action along the way to boot.

Another way to use an array property is to associate it with an *array* of windows controls! This way, you can have a single property represent all of the data on an entire window or dialog. For example, if the array property contained an array of VARIANTS, and if you gather up the window handles of all of the controls on a dialog into another array, then, you can make a single call, passing the array of window handles, to the array property in order to update the entire window. Similarly, you make a single call to update the array property with the current window values. Add to this the powerful persistence features and process integration that the master GSystems Properties Component provides and you can persist an entire window with 1 property and 2 simple calls to its methods.

Thus, properties contain a rich set of features that you can use to make your connection between system data and windows easier to manage. Before discussing these, however, we will cover the details about how to make properties and what sort of “properties” properties have, such as type, contents, etc.

Creating properties

The GSystems Properties Component can be thought of as the Mother of all properties. Generally, you will “Create” a property object by making a call to the `Add()` method on the GSystems Properties Component:

```
IGProperty* pIGProperty_interface;  
GProperties.Add("property name",&pIGProperty_interface);
```

And, in Visual Basic:

```
Public formWidth As GsystemProperty  
Set formWidth = GSystemProperties1.Add
```

Where `GProperties` is an instance of the GSystems Properties Component (strictly, a reference to it's default interface). The second argument is the returned interface pointer to the created property object⁵. Both arguments to this call are optional; the name of the property can be left unspecified – in which case the system will generate one, while you don't have to obtain the reference to the created property (the second argument) unless you want to. For example, if you are going to call methods on this property in the future from within your application. The VB examples shows that the default interface to the created property is the return value of `Add()`.

If you do provide a name for the property, the Properties Control offers a method to retrieve that property if you need a reference to it on an occasional basis (which you already have if you retained the return value).

Note that you do not need to `AddRef()` and/or `Release()` the property interface returned because the GSystems Properties Components owns this object and will manage it's lifetime. However, the interface does represent a COM object, so if you `QueryInterface()` through the property to retrieve any other interface from it, you must, of course, `Release()` that interface.

Because a property is a pure COM object in it's own right, you can also create them through COM, for example, with `CoCreateInstance()`

⁵ This is a C/C++ example, VB and other environments typically have the last argument not appear in the call but rather be “assigned” to, such as

```
DIM property_interface as GSystemProperty  
Set property_interface = GProperties.Add("property name")
```

from C/C++, or, `CreateObject('GSystem.GProperty')` from VB, among other techniques. If you do this, you would then call the `Include()` method on the GSystems Properties Component. Further, you must manage the lifetime of the object, i.e., `Release()` the reference to it when you are finished with it.

The Types of Properties

There are several different types of properties that may be maintained by the system. These are represented by the `PropertyType` enumeration defined in the type library embedded in `Properties.ocx` and in the `Properties_i.h` header file.

Table 2 Property Types

Property Type	Value	Purpose
TYPE_LONG	0	Any long value. 32 bits
TYPE_DOUBLE	1	A double value.
TYPE_SZSTRING	2	A typical "C" null-terminated array of characters, the last being a '\0'
TYPE_STRING	3	A BSTR type, the OLE safe wide character string.
TYPE_BINARY	4	Any binary value, can be used to store any structure, such as the SIZEL or RECT structures.
TYPE_BOOL	5	True or False
TYPE_VARIANT	6	Any VARIANT value. Cannot be VT_DISPATCH or VT_UNKNOWN
TYPE_ARRAY	7	An array type, strictly a SAFEARRAY, can be an array of any type, i.e., an array of VT_VARIANT or an array of VT_I4, for example
TYPE_OBJECT_STORAGE_ARRAY	8	This type is explained on page 13

Though there is a “type” property (the method is `put_type()` for C clients or just “type = ___” for other environments) for each property object, you don’t typically have to set this value. When you set the value of a property, it will learn of its intended type. Depending on the development environment you are using, the argument passed to `put_value6()` for the property object may be different than what you are expecting. For example, since this argument is a VARIANT, one development environment might pass a literal integer as a VT_I2, while another might create a VT_I4 argument to pass.

Only in the case of problems in these interpretations would you find it absolutely necessary to set the type of the property.

There is one type of property, the `TYPE_OBJECT_STORAGE_ARRAY` that is for a very powerful special usage described next.

Storing sets of objects within a single property

It is important that you create properties in the same order all of the time so that the system can save and retrieve the properties without mixing up which part of the storage belongs to which property.

There is a special case where this is an inconvenient restriction, when you have an arbitrary number of COM objects that your system is using at any given time and when you are asked to persist your property space. This is an inconvenience because you don’t know up front how many of these objects you’d like to have stored on the storage, yet the system wants you to not “create” properties after the saving and loading processes have completed. Strictly speaking, you *could* add properties to the list of the properties after loading but before saving *if* you are sure to create the same number of objects, in the same order, the next time your application or object starts up. Though this latter case offers a solution to the problem, it is difficult to setup and manage.

A much better approach is to use the special `TYPE_OBJECT_STORAGE_ARRAY` property type provided for situations

⁶ Value is the default property of each property object. This means that in VB, you can assign the data the property will maintain with the statement:

```
property = theValue
```

which is the same as calling `property.put_value(theValue)` from C.

like these (though, you will see that this type of property has a great deal of potential in being able to store anything you want).

This special property type is used to indicate that you have 0 or more “sub” objects that you would like to store within your properties’ storage space, but at any given point in time, you don’t know exactly how many of these you will be storing. Since you create one instance of this property (or more if you need to, the point is you always create the same number of them) when the application or object starts, there will be no confusion when the system encounters this property in the stream of storage where the properties’ values are stored. Then, when you are notified by the system that it is a good time to prepare for saving, you “add” objects to this array of objects, all stored within this one property, so that the system can include those “sub” objects’ storage information all embedded within this one property.

Here is the call to “add” an object to the property:

```
pProperty.addStorageObject(IUnknown* pObj);
```

This call notifies the property of type = `TYPE_OBJECT_STORAGE_ARRAY` in the `pProperty` object to add some object to its list of objects that it will maintain the next time the property is saved.

Be aware that the property will attempt to retrieve an `IPersistStream` interface from the passed in `IUnknown` interface and, if successful, obtaining that interface will cause an `AddRef()` on the property.

For the above reason, you should call `removeStorageObject()` (or `clearStorageObjects()`) as soon as the following step is completed.

When you are ready to have the property update its internal value based on the state of the objects it is maintaining, you call

```
pProperty.writeStorageObjects()
```

and the property will then represent all of the persistence data of all of the objects previously added to the list of objects this property maintains.

Then, because the objects will have been `AddRef()`ed as stated previously, you should call `removeStorageObjects()` or `clearStorageObjects()`:

```
pProperty.clearStorageObjects()
```

which will cause the previously obtained pointer to the objects' IPersistStream interfaces to be released. Typically, you add all of the objects, write the storage, and clear the list all at the same time when you want the property to “contain” the current state of all of the sub-objects. Thus, you normally do this in the SavePrep() method on your IGPropertiesClient interface, see page 96 for more information.

One thing to keep in mind is that when you attempt to “restore” these properties from the loaded storage, you have to know how many there are. You can retrieve this information using the storageObjectCount() property:

from C/C++:

```
long cntObjects;  
pProperty.get_storedObjectCount(&cntObjects);
```

or, VB

```
Dim k as long  
k = property.storedObjectCount
```

Using this count of objects, you can create these sub objects at load time (using COM or whatever technique you used to create them before), add them to the property using addStorageObject(), and then, have the system restore them to their previous state:

```
pProperty.readStorageObjects();
```

Note again that these sub-objects are simply COM objects. There is no restriction on the size of their saved data, nor is there any requirement that these objects be built using the GSystems Properties Component. The only requirement is that they support the COM interface IPersistStream (which *any* object of merit will do). When the GSystems Properties Component saves these objects, it provides the objects with an implementation of IStream upon which they in turn write their persistence data. The GSystems Persistence Object then saves *this* data as the persistence data for those objects (the objects that have been “added” to the property).

As you can see, the use of this type of property has extensive possibilities, not the least of which is to use the technique to store any single object's information within your own object's data. However, for that purpose, i.e., storing one object when you know there will always be 1, the GSystems Properties Component provides another technique. This is the AddObject() method (page 27) which tells the master component to include a foreign object's persistence data anytime your own software's data is saved or loaded. This is the preferred method of saving “known” objects, that is, objects that exist in your development environment and for

which there will be no more or less of at any time during the running of your software.

Providing Direct Access Storage

Note: in this discussion, “storage” is used to denote your application or object’s memory space.

One of the tasks you undertake when using Properties is to ensure that the value of the property is appropriately “in-synch” with the data⁷ in your application so that when your properties are saved to the storage, they have the correct value. The necessity of performing this task is the reason why there is a `SavePrep()` call on the `IGPropertiesClient` interface, so that you can concentrate all of the code that would be required to do this in one place.

However, there is an automated way to ensure this association is always valid by simply giving the property physical access to the data’s memory. This technique is not available in all environments but where it is available it is easy and effective to use.

To use this technique, you pass the address of a variable, and the number of bytes it occupies in memory to the property which you want to associate that variable with.

```
propertyObject.directAccess(  
    propertyType, &memberVariable, sizeof(memberVariable));
```

Here, the property type is provided from the table on page 12, the `&memberVariable` is the address of some variable in your system, and of course, the `sizeof()` operator provides the system with the extent of this space.

Behind the scenes, this causes the property to use the data in that memory space every time it is asked to provide and/or set it’s underlying value. Though this technique is very efficient, you must be aware of the issues involved with using it.

⁷ If any, remember that in certain environments, you can use the properties as the data itself, i.e., you don’t have to maintain separate variables to swap in and out of the properties.

- 1) The property object must have the “rights” to access that memory space. It must be able to both read and write into that space as long as the property object is using it.
- 2) Manipulation of that space must have no impact to the development environment in use. That is, the memory will be manipulated external to the development environment and that must not impact the environment’s own manipulation of that memory. Keep in mind this has nothing to do with the ultimate “container” application that will be using an object developed using this technique.

In general, if the development environment will give you access to this memory, i.e., through a reference to it, then it is at least worth attempting whether you can pass that reference on to the property object to see whether it will work in that environment.

To clarify, if you are building an ActiveX control using MFC, then you can certainly pass the address of a variable to the property component and the property component can utilize that space. However, in an environment like PowerBuilder, it is unclear whether you can access underlying memory for variables and therefore whether you could then use this technique.

- 3) The “lifetime” of the variable whose memory you are providing must be at least as long as the property object.

This simply means that the memory must not “revert” to use by some other part of the system after giving it to the property object. For example, by passing the address of a variable created on the stack will cause immediate and severe problems when the property object tries to write to that address in the future.

However, the address of a member variable of some C++ object, for example, the object that is an ActiveX control you are developing, may be legally provided to the properties object until that object is deleted.

- 4) This technique will not work in a DCOM environment or across different processes or anywhere where marshalling is required.

The GSystems Properties Component was not developed as a distributed component. It is a development tool that is tightly integrated into your own code. Even if you do set the registration

information correctly to use the component in such an environment, this technique will not work properly under these conditions.

Using Normal Access to Storage

In lieu of the above technique of providing the connection between your data and the property's data, you need to ensure that the value of the property is valid at certain times.

Most of this discussion is held in the description of the `IGPropertiesClient` interface where this activity happens at the level of the `GSystems Properties Component` rather than at the level of each property.

However, what is important from that discussion is that at certain times, your code must set the value of the property so that the correct value will be saved on the storage. In particular, your software is notified just before this is going to occur.

One other technique, however, that you can use is to always keep the property updated whenever the underlying window object, or system variable, changes. For example, if you want to save the contents of an Edit control in a property, then, in the `OnChanged()` event of for that control, simply set the property there and the property will be valid when it is stored. Similarly, in your application's code that runs *after* the properties are loaded, simply set the contents of the edit control from the value of the property.

There is an entire class of functionality provided by the property object that can be used for this purpose, i.e., for exchanging the property's value between a windows object and the property object itself. These methods are discussed in detail in Part VI. However, the following will serve as a general introduction.

When you want to *set* the value of some window object, a control or other child window on a dialog or window, you can call the "`setWindow...`" methods on the property object.

The purpose of these methods is to allow the property to set it's value into the appropriate window control in a single call rather than retrieving the property's value and then using the windows API or other development environment technique to get that value into the control.

For example, the `setWindowListBoxList()` method will set the contents of a windows list box based on the current array of strings in the property⁸. You no longer need to manage the actual list box using this type of property. All you have to do is to be able to get the “handle” to the list box window, this is the `hWnd` property of any control on a Visual Basic form. Another class of these methods takes a child window identifier as well as the parent window handle. The above method name becomes `setWindowItemListBoxList()`, and the `hWnd` passed is the handle of the form while the numeric ID of the particular child window involved is also passed⁹.

Following the example, when you want to *retrieve* the value of a list box control, you call the `getWindowListBoxList()` method on the array property’s interface. Again, gone is the special code you have to write to loop through the list box to retrieve the values into your object or application. You don’t even have to manipulate the array entries in the array property, all of that is handled by the property itself.

In general, these methods are straight forward and self explanatory. For detailed reference level information for the rest of the methods in this class, please see page 71.

There is one type of property and window access method combination that is particularly powerful in a given type of use. That combination is described next.

Combining an Array Property with an Array of Windows Controls

As mentioned before, an array property may contain any number of elements holding any type of data in each element.

⁸ Which must be a `TYPE_ARRAY` property, though it can be an array of any “type” of variables, either `VARIANT` or any scalar type.

⁹ As with all of the property methods, there will be an “equivalent” method on the GSystems Properties Component itself. In this method, the only difference is that the name of the property is passed in the first argument, and that the first letter of the method name is capitalized. That technique causes the master component to find the appropriate property and call the associated method on that property, relieving your code from the necessity of keeping a reference to each property.

If you have defined a property of this type, and if you then create an array of longs representing each of the controls on a given window, you can save and restore the entire window using just this one property.

To get an array of controls, you simply need to create an array of long values with the same number of values as there are controls.¹⁰ Then, you fill up the values of this array with the window “handles” of each of the controls. Further, in some environments like Visual Basic, the system may provide convenient access to this “array” of windows by the use of an internal mechanism, like, in VB’s case, the “Control Array”.

In any case, once you have your control array populated¹¹ and you have an existing array property with the same number of values in it, you simply call:

```
pArrayProperty.setWindowArrayValues(controlArray);
```

which will populate each of the windows represented by `controlArray` with the corresponding value in the property `pArrayProperty`.

Note that the two arrays do not have to be the same in terms of dimensions, only that they have the same total number of elements. In fact, it is perfectly legal for the array dimensions to be different.

It is important to understand, however, the order that the entries in arrays are accessed by the GSystems Properties Component. For reference, this order is as follows:

- a. For 1 dimensional arrays, from the lower bound to the higher bound.
- b. For multi-dimensional arrays, the right-most index moves fastest, i.e., if the array is dimensioned like

[0-2][1-3][0-4]

¹⁰ Note that the VB `Dim controls() as long` statement will do this for you. In C/C++ environments you will have to create and populate a `SAFEARRAY` object of the necessary size.

¹¹ Note this is something you must do every time your window is created since the window handles themselves can 1) not be persisted and 2) will have different values each time the window is created.

the first element considered is (0,1,0), the second is (0,1,1), then, (0,1,2) and so on.

This order will always be followed. Therefore, you must populate the array values into the array property in such a way that the control you want to be associated with each value is “under” that value when the arrays are traversed side by side. The best way to understand this issue is to see it in action by trying it out in your own code.

When it becomes time to perform the opposite action and update the value of the properties from the current state of the window, perhaps just before saving the properties to storage, you make a similar one line call:

```
pArrayProperty.getWindowArrayValues(controlArray);
```

which takes the current contents of each window and puts it in the associated element in the array property. This property is now ready to be saved on the storage. When it is reloaded, the first action is taken and the entire window takes on the exact appearance it had when it was last saved. Using this technique, you can create “record oriented” windows in your system that swap entire records into and out of the window simply and efficiently. Indeed, saving the single property as one instance of the record.

An important note about the “Set” of properties

Again, your software system will create a group of properties and will have the values pertinent for each property maintained on a storage in such a way that the properties can have the same value they had when previously written to that storage.

Keep in mind that the order that these property objects will be restored from this storage is exactly the same order that they were saved to it, which is the order that the properties were `Add()`ed to the GSystems Properties Component.

Additionally, you may have instructed the GSystems Properties Component to save the “contents” of entire sub-components on the same storage that is used to maintain your own properties – persisting other objects such as common windows controls and other ActiveX controls.

Because of all of the data that is being maintained on this storage, it is important that you “create” your properties in the same order every time your application starts or your component is created. This is, of course, very natural because the order of execution of this code is highly predictable and

consistent from run to run. Except, perhaps, in rare cases would you be creating a different “set” of properties, or creating them in a different order.

By using the debugging property on the GSystems Properties Component, you will be notified if the system encounters data that has seemed to “shift” from run to run. If, for example, you add a property to your system, recompile it, and when running it, open up a file containing the persisted data, the system will see that the data no longer fits the “set” of properties defined and will pop up a message indicating this. Typically, when that file is re-saved (or a component is saved to it’s client supplied storage), then this problem will not occur the next time that file is opened.

Part II – Provide Application Level Persistence

Developing a persistable application – by example

A sample Visual Basic application has been included with your package which will demonstrate how to achieve results from the GSystems Properties Component in the development of an application.

This sample is the VB project PropertiesSample.vbp. If you cannot find the sample, please feel free to download it from InnoVisioNate.com.


If you do need to re-download the sample, it might also necessary to reinstall your copy of the GSystems Properties Component.

In any case, please ensure that, when you open the project in Visual Basic, the “G System Properties” component is referenced, press Ctrl-T to raise the components dialog and confirm this. The component should also be listed in the dialog accessed through the “Projects – references...” menu item. A sample ActiveX control, built, by the way, using the MFC Control example project, FunctionContainer.dsp, is used in the VB sample. This control’s component is registered as “FunctionContainer ActiveX Control module” which, as a component, should have been registered when the GSystems Properties Component was installed.

This sample application has many comments that are intended to take you step by step through the requirements of achieving persistence.

Here is the step-by-step procedure for using the GSystems Properties Component. If you prefer a less procedure oriented description, you may refer to Part VII starting on page 76, the non-procedural description of each aspect of the component.

Instantiate a GSystems Properties Control object

Put the control on your main form by dragging it off the palette. The icon on the palette looks like this .

You can reference the control throughout your system. In source files (or

modules) other than the main form. You may need to qualify the object with the name of the form where you have placed it. For example, if it is on Form1, and you have called it GProperties, reference it with

```
Form1.GProperties
```

This provides the convenience of being able to manipulate the control within the development environment.

However, keep in mind that you might have a situation where you want to have several of GSystems Properties Controls at run-time. Perhaps your application or system is divided into several “logical” areas where it makes sense to have a persistence mechanism for each. This allows you, for example, to save settings in one area of the system while saving different settings in another area. This is entirely conceivable, for example, when you have a “record” oriented screen and you might provide the selection of which record the screen represents in some other area of the application. In such a case, you would have persisted each record for the screen and would load the appropriate persistence file as appropriate as the user chooses to view a particular record.

Such a use makes the GSystems Properties Component viable as a simple and convenient “database” tool.

In cases where you will have more than a few, or even an unknown number of, instances of the Properties Control, you can instantiate the controls at run-time. Here we instantiate an nvGSystemProperties (non-visual) object:

```
Dim nvProperties As nvGSystemProperties  
Set nvProperties = New nvGSystemProperties
```

Then, we can use the variable `nvProperties` wherever we need as if it were a GSystems Properties Component placed on a form; the interfaces and capabilities are identical to each other, you just have to set the properties of the instantiated object at run-time

Set the properties of the Properties Component

Regardless of how you created your component. You may need to set the properties of it.

For example, you can provide a file name to open or save the persisted data to. There are only a few such properties in the Properties Component and setting them is optional; a default action will occur if the properties are not set. For example, if your application attempts to open a persistence file, but

you have not set the filename property, the system will raise a “select file” dialog box to the user.

Create Properties

The GSystems Properties Component is the master of all properties. You use the control to create the actual property objects. These objects are instances of the GSystems Property object, which has a rich set of capabilities in its own right (please see Part 0 on page 1).

At this point, you create the properties by calling the `Add()` method on your Properties Component¹². For example

```
GProperties.Add("property name")
```

creates a named property that will be used to hold any value, when this property gets a value assigned to it, it will take on one of the special property “types” previously described.

The property name is provided so that you can retrieve and/or access this property later on, for example, by calling the `Property()` method on the GProperties component

```
GProperties.Property("property name") = "string value"
```

or,

```
Dim prop as GSystemProperty  
Set prop = GProperties.Property("property name")  
prop.getWindowText Form1.someControl.hWnd
```

The first method allows you to use the property object directly without putting it into an intermediate variable. If you need to retain that property object, however, you can do so by holding onto the property object returned from the `Property()` method in an appropriate object variable. In such a case you might as well keep the property as soon as it is created using the `Add()` method on the GProperties object, note that in this case, an actual name is not necessary¹³ because you never have to go back to the

¹² Which, for this example is referred to by the name GProperties. When the object is placed on a VB form, it was probably called GSystemProperties1.

¹³ For various internal reasons, the system generates a name for the property.

GProperties object and search for the property of the given name, since you already kept a reference to it.

```
Dim formWidth as GSystemProperty
Set formWidth = GProperties.Add("")
```

will create a GSystemProperty object and provide a reference to it in the variable formWidth.

Now you can use this variable formWidth just like *any* other variable in your system¹⁴. When you reload the persistence file for the system, the value of the property formWidth will automatically contain the same value as when you last “saved” the persistence file.

Here’s how to persist on the width of your main window.

In the resize event of the form:

```
Private Sub Form_Resize()
    formWidth = Width
    formHeight = Height
End Sub
```

This code will keep the associated properties “up to date” with respect to the current window state. Now, when the persistence file is saved and subsequently reloaded, the values of the properties will be consistent with the last time (in the previous run!) that the window was saved.

```
Dim keepHeight as Integer
GProperties.OpenFile("persistence file name")
KeepHeight = formHeight
Form1.width = formWidth
FormHeight = keepHeight
Form1.height = formHeight
```

Just after opening the file, you can be assured that the properties have the correct value. Note the extra variable keepHeight, this is provided because the assignment to Form1.width is going to cause Form_Resize() to get called and that, in turn, will cause the formHeight property to get set, overwriting the value you really want to use in the Form1.height assignment statement.

¹⁴ In fact, the default “property” of each GSystemProperty object is its value. This means that the value of the property is what gets set when a variable (or expression) representing the property is on the left side of an equals sign. Because of this you must mentally make the distinction between the use of the “Set” statement and the assignment statement such that the reference to the property object does not get inadvertently changed.

Provide the persistence mechanism to any other objects

Since other objects are likely to be a part of your application, you can use the GSystems Properties Component to manage the run-time persistence of these objects too.

Note that these may be other components as well as some of the enhanced windows controls such as tree and list views. What is important is that the particular component must implement one of the COM persistence interfaces, which will usually be the case, otherwise, the control would be of little use as in real environments.

You allow the GSystems Properties Component to handle persistence for these objects with the `AddObject()` method.

```
GProperties.AddObject listView.Object
```

Here, the properties component is being passed the underlying “object” of the control on the form that has been named “listView”. Note that the “.Object” qualification is necessary as without it, you will be referring to an object that Visual Basic creates in support of hosting the actual object in the VB environment.

Once you have provided this information to the GSystems Properties Component, it will handle every aspect of persistence from that point forward. In particular, any time the storage is either saved or loaded, the Properties Component will instruct each associated object to save or load itself as appropriate. Additionally, if supported, objects will also have the opportunity to initialize their defaults whenever appropriate.

Participate in the persistence process

It is important to understand that when the GSystems Properties Component has saved and/or restored the properties from the storage, you as developer are responsible for updating your application with whatever state reflects the new values of these properties.

Whereas this may not be extremely complex, it is typically desirable to have all of the software that achieves these goals in one and only one place. It is also desirable to allow the GSystems Properties control access to this code at times of *its* choosing. This subtle point is because it is, after all, the GSystems Properties Component that knows precisely when the properties have changed, at least from having been loaded from a storage, or when the

properties need to be current, as, for example, just before saving the properties to a storage.

For these reasons, a client of the GSystems Properties Control, that is the software you are developing, is able to provide just such a set of functionality and can notify the GSystems Properties of where this functionality is and therefore let the Properties Component access these services as the needs arise.

This functionality is embodied in an interface called the `IGPropertiesClient` interface. Note the use of the suffix “Client” in the GSystems family of components means that the user of the GSystems component, the client, is going to *implement* the interface.

In the Visual Basic environment, you must create a “Class” by adding a new “Class Module” to your project. Note that the first statement in the new file is:

```
Implements IGPropertiesClient
```

This statement tells Visual Basic that you agree to provide all of the functions or subroutines that are defined as being in that interface. To see what these are, you can browse the object that *declares* the interface, press F2 for this function. If you look at the `GSystemPropertiesCtl` library, you will see the interface `IGPropertiesClient`. Selecting this interface will show you all the functions that you must implement in order for the class module to compile.

You can cut and past the entire `PropertiesClient.cls` file from the `PropertiesSample` application and simply delete any irrelevant code internal to each function/End Function or Sub/End Sub pair.

In fact, in general you may only need to implement three¹⁵ of these subroutines, the `SavePrep()`, `InitNew()`, and the `Load()` subroutines.

Here is the very simple intent of each of these functions.

SavePrep() is called when it is time to update the value of your properties because they are about to be written to a storage. Usually you have one line of code per property. This line of code will assign the value of

¹⁵ Though you must at least provide the stub of *all* the subroutines.

the property based on whatever that property is associated with in your application.

Of course, if you had been using the property as if it were a variable, the value will be up to date at all times and this step would not be necessary for that property. Often times, however, you may need to take the value in a control, for example, an edit field, and set the value of the property with this value. There are several “direct” ways to accomplish this, for example, to update the property’s value based on a particular control’s current contents,

```
theProperty.getWindowValue Form1.editControl.hWnd
```

where `theProperty` is a property object variable (or use the master Properties component’s `Property(“property name”) method`) and `editControl` is the name of an entry field on, for example, `Form1`. The `hWnd` property of any window is the window “handle”. You need to pass this to many of the Property methods when you want to associate a window control with the property value.

Note that this example retrieves the value showing in the edit control into the value of the property without having to format that value based on the “type” of the property. Specifically, if the property is of `TYPE_DOUBLE`, then the value of the property will be correctly formatted from the text in the edit control; you do not have to deal with that at this point. Part VI on page 71 deals exclusively with these tools.

InitNew() is generally only called when the GSystems Properties Component is used in support of a COM object that you may be developing. Specifically, when a COM object comes to life for the first time, i.e., it is not “loaded” from a storage, then the `InitNew()` method is called. In our case, since we are writing an application, it is likely that `InitNew()` will never be called.

However, it is convenient to use this method as a place for all the code that will be the “default” for your application. Then, at that point where the user asks for a “new” state of the application, your code can call¹⁶ `InitNew()`

¹⁶ Even though the `IGPropertiesClient` interface is defined for the GSystems Properties Component to call its methods, it is often very convenient, and perfectly legal, for code that you write to call these methods. In fact, you may find the structure and functionality of this interface suits many aspects of your application, such as calling `InitNew()` as a natural way to “reset” the state of the system quickly. The functionality within this interface is typically present in most applications, it is therefore convenient to place your code here for the benefit of both the entire application and for the GSystems Properties Component’s ability to enhance the application.

to set all of the properties to the default values, whatever they may be. Perhaps the defaults are stored in yet another storage managed by a different GSystems Properties Component elsewhere in the application.

In keeping with this strategy, then, first set all of your property's default values in this method, again, usually one line of code per property. Then, at the end of this, make a call to the `Loaded()` method, described next, because this latter method is where you update the visual state of the application based on the current value of all of the properties.

Loaded() is called by the GSystems Properties Component just after having read all of the properties from the storage. It is here that you make the opposite association than you did in `SavePrep()`. Generally, any updating of your windows or other content in your system that reflects the current properties is done here. Any aspect of your system that depends on the value of the saved (and now restored) property should be set in this subroutine.

Saved() is called by the GSystems Properties Component *after* the properties you are managing have been saved to the storage. It is important not to confuse this subroutine with the `SavePrep()` subroutine.

IsDirty() is called by the GSystems Properties Component to determine if any of your properties are “out of date”, or have changed, since the last time the properties were saved. This helps to determine if it is necessary for you to prompt the user for a file name (which you can use the GSystems Properties Component to do for you), or if you should provide similar logic to prevent the current state of the system from being overwritten. Your implementation must return a zero value for “Yes the properties have changed”, or a non-zero value for “No”.

Use the file manipulation capabilities

Since you are writing an application, your system must initiate the persistence process so that all of the properties maintained by the GSystems Properties Component ultimately get saved to the storage.

Remember that this is not the actual manipulation of the binary data representing the storage, rather, it is a set of higher level tools that allow you to provide functionality the user will use (or which you use entirely within code) to manage the set of properties as a file.

In order to use these features, you generally implement menu events that will initiate dialogs with the user as appropriate.

However, it is constructive to consider an elementary case first that will be used to describe the basic functionality.

Load from an existing file

Assuming there is an existing file on your user's system that had been previously saved using the GSystems Properties Component, you can load that file using code such as:

```
Dim fileLoaded As Boolean
GProperties.filename = filePathAndName
If Dir(GProperties.filename) <> "" Then
    FileLoaded = GProperties.LoadFile
Else
    GProperties.New
    GProperties.Save
End If
```

Which will initiate the process of restoring the persistence data within the file with the given name. Perhaps the `filePathAndName` value is one of the arguments passed to the application's executable, thus naming a "startup" file, for example. If the file does not exist, the `New()` method is called and the persistence data is immediately saved, thus creating the file. Note that you may ship such a file with your application if you'd like to specify the startup state of the application the first time it is run by your users.

Saving to a file

At any point in your application, you can start the saving process by instructing the GSystems Properties Component to save the properties to the file¹⁷ using the following code

```
GProperties.Save()
```

¹⁷ The `Save()` and `LoadFile()` methods both use the `FileName` property of the GSystems Properties Component as the destination and source files respectively. For `LoadFile()`, the file must exist or the method will not succeed and will return a non-zero value.

This will cause the system to store all of the properties to the file whose name is the value in the `FileName` property of the `GSystems Properties Component`.

For a fully automated system, you can have this method called as your application is shutting down, for example in the `Unload()` event of your main form. You must be sure, however, that any of the windows needed by the `SavePrep()` method you write are available to that method whenever `Save()` is called.

Summary of the file manipulation methods

You can have the `GSystems Properties Component` initiate dialogs that allow the user to specify where to save the current properties file and/or which file to open. Note that most of the “properties” on the `GSystems Properties Component` are provided in support of this activity.

In contexts where it makes sense for your users to save the properties you have defined for your application, you provide the menu items or command buttons that initiate these calls to the `GSystems Properties Component`:

New() will cause the properties to take on the “default” values specified through your implementation of `IGPropertiesClient` (if any). This method will result in the creation of a temporary file so the `Save()` (see below) method will work if called subsequent to this.

LoadFile¹⁸() is used to read the file associated with the current values of `FileName`. As described in the previous section.

OpenFile(string fileName) causes the named file to be opened and the “restore” process will begin. Note that this means the `Loaded()` method on your implementation of `IGPropertiesClient` will get called. With this method, the passed in parameter represents a file that already exists, the method will fail if it does not – successful operation is indicated by a 0 return value.

¹⁸ The `LoadFile()` method and the “Open” methods will pop-up a warning (if `debuggingEnabled = True`) if the current state of any properties is dirty, that is the properties will be lost if they are overwritten. No action is taken by the `GSystems Properties Component` in such a case, only a warning that processing logic allowed this to happen, which may be perfectly acceptable in the context of your system.

Open () will cause the Properties Component to raise a system dialog that will prompt the user for the name of a file to open. The method will return the name of the file chosen by the user, or an empty string if the user cancels the selection dialog.

If the user does not cancel, however, the persistence data from the file will be `Loaded ()` as in the previous method.

Save () will cause the system to save all of the current properties in the file whose name is the value of the `FileName` property.

SaveAs () will cause the system to prompt the user for a file within which to save all of the properties. As with the `OpenFile()` method, this will return the name of the file actually chosen by the user, or an empty string if the user canceled the operation.

SaveTo(string fileName) causes the named file to be the current file and all of the properties' values are saved to it.

The other major area of functionality the GSystems Properties Component can provide for you is with Property Pages and settings dialogs. The discussion of these powerful techniques is in Part V.

Part III – Provide Component Level Persistence

Developing a persistable object – by example

A sample object is included with your package. This object is a sample version of the GSystems Function Generator. It is equivalent to the full version of the control except that it does not fire some of the events necessary to achieve complete functionality with the control.

However, code relevant to this discussion is included where necessary to make the illustration. Further, this object is developed using raw COM, i.e., it does not use any MFC or ATL technique associated with COM. Therefore, the samples shown will not be reliant on any ActiveX development framework. However, two samples using each of these popular frameworks are provided with this system and certain sections of this part of the document will highlight differences in the approach where necessary because of the use of these other development tools.

If you are developing an object or component, one easy client, or container, you might find is the ActiveX Control test container accessible from Microsoft Visual Studio™. If you create an instance of an ActiveX/COM object in the test container, the container will persist the object when you save the it in the test container's file format; '.tcs files. For the purposes of testing persistence, this tool can be more than adequate.

Note that Part II of this document was built around a sample Visual Basic application. This application uses the sample object¹⁹ under discussion in this section. Therefore, you might find that the Visual Basic sample is also a suitable client within which to test your object as you learn about the topics in this section since using the VB sample will allow you to see the use of the GSystems Properties Component from its other main perspective – persistence at the application level.

The goal of this exercise will be to show precisely how a COM object should implement the GSystems Properties Component in such a way to

¹⁹ Strictly speaking, the sample application uses an ActiveX object which embeds the GSystems Function Generator Sample. That ActiveX Object is also provided, the FunctionContainer sample.

achieve persistence and property page support for the object. This will be accomplished by demonstrating the specific C++ code in the object that is related to the use of the GSystems Properties component.

If you haven't already done so, you need to install the GSystems Properties Component for this exercise. The GSystems components are shipped with an installation utility. However, the only important accomplishment of installation will be to "register" each component. All of the GSystems components are built such that this is also the only required step to make each component functional on any host system.

Specifically, all you have to do is to "open" each relevant '.ocx file with the program regsvr32.exe; make sure the '.ocx files are where you want to permanently store them and either use explorer to left-click and "open" the file with regsvr32.exe (regsvr.exe on Win 95), or type regsvr32 filename.ocx at a DOS prompt. Please make sure that the files Properties.ocx and FunctionSample.ocx are registered for this exercise.

Note: If you want to implement the Properties component using COM in a C or C++ environment, you may want to use the interface definition files provided for this purpose. The installation utility will place the files Properties_i.h, Properties_i.c, and Properties.tlb in the same location as the Properties.ocx file. You may need to manage the location of these files, i.e., put them where your development environment can find them.

In the ActiveX Test Container, create a new GSystems Function Sample using "Edit/Insert New Control..."

To see persistence in action for the FunctionSample object, change the expression and/or click on either of the "variable" tabs (x or y) and change any of the settings for the variables. Then, you can select 'File/Save' from the Test Container's menu, save the "object" to a file and later open that file, you should see the settings you made for the 'x' and/or 'y' variable have been retained.

Note that the properties that were changed, and subsequently saved, did not have to be set by the user using a "properties" dialog. While such functionality is a big part of the GSystems Properties Component, it is also easy and natural to connect the properties with running code in your object. For example, if you had written the function control, you might have the following code in your OnChanged() method for the expression entry field:

```
pIGProperties -> GetWindowItemValue(  
    "expression", hwnd, IDDI_EXPRESSION);
```

In the above, `pIGProperties` is your pointer to the GSystems Properties Component object's `IGProperties` interface, the "expression" argument is the name of the specific property, the `hwnd` argument is the window handle of the dialog containing the edit control and the `IDDI_EXPRESSION` value is the id of the edit control in the dialog (`hwnd`).

The next section discusses what the GSystems Function component did to achieve persistence.

Quick Rules for Persistence

The GSystems Function object obtains an instance of the `IGProperties` (a GSystems custom interface) interface by creating an instance of the GSystems Properties Component when the function is created. In particular, the function object's class has a member variable that is a pointer to type `IGProperties`, i.e.,

```
IGProperties* pIGProperties;
```

The definition of `IGProperties` (and other important information) is obtained by the inclusion of the header file: `Properties_i.h`.

Note: Anytime you use a specific definition such as provided by one of the interface definition header files, the compiler will be referencing a variable of type `CLSID`. This means that if you specifically include one of these definition files, then, in one and only one of the source files for your project, you must "include" the corresponding '.c' file, (`Properties_i.c`), which will instantiate the `CLSID` variable the compiler references in other files.

Note II: Some development environments have wizards and generated code, among other techniques, to make using COM objects easy. We choose not to center our discussion of the GSystems components within the context of these techniques for 2 reasons; 1) because not all users will have these environments and of those who do, not all are likely to utilize the techniques, and 2) for a minimal amount of increased complexity, the material covered is more in line with the reality of what is actually happening, a knowledge of which can make the technology much more accessible and understandable in the long run.

Note III: However, if you are using MFC, you can use the Class Wizard to create a `COleDispatch` wrapper class for the GSystems

Properties Control. Simply invoke Class Wizard, select the “Add Class ... From a type library” option, navigate to the Properties.ocx file, and select all of the available classes that appear in the subsequent dialog, or, as many as you want to use. Note that this will create a header and implementation file that get included in your project.

Step 1: Create an instance of the GSystems Properties component

Here is the code, in the constructor of the function object, which creates an instance of the GSystems Properties Component and obtains an interface pointer on it:

```
CoCreateInstance(CLSID_GSystemProperties,
                NULL,
                CLSCTX_INPROC_SERVER |
                CLSCTX_INPROC_HANDLER,
                IID_IGProperties,
                &pIGProperties);
```

The GSystems Properties Component is not written for, nor expected to be used in, a distributed COM environment. Thus, the CLSCTX_... constant should be one that ensures COM doesn't end up providing an instance of the server either out-of-process or from another machine on a network.

As with all COM objects, you must call

```
pIGProperties -> Release()
```

when you are finished with the GSystems Properties Component to allow the object to free itself.

Creating a COleDispatch instance in MFC objects

In an MFC project, you might create a reference to a COleDispatch derived wrapper for the object:

```
DGProperties& nvProperties = *(new DGProperties());
```

The DGProperties class is defined as per Note III concerning MFC above. This technique requires you to “connect” to the underlying object at an early point in the creation of your object, preferably, in the constructor.

```
nvProperties.CreateDispatch("GSystem.GProperties");
```

which allows you to begin using the nvProperties object.

An MFC project may also utilize the technique described next for ATL projects.

Importing the GSystems Properties Component for ATL and MFC projects

Another way to provide the control's functionality within your code is to "import" the type library from the '.ocx.

```
#import "Properties.ocx" raw_interfaces_only,  
        raw_native_types, no_namespace, named_guids
```

The above compiler directive will define all of the interfaces and methods on those interfaces from the control to your system. The effect of using this technique (with the above qualifiers) is the same as previously described by including the '_i.h and '_i.c header files.

A more convenient form of the #import directive may be to leave off the qualifiers. In this case the compiler will generate definitions for smart pointer access to the object. As this topic is fully covered by the Microsoft documentation, it is not repeated here except to remind the reader that it is an available technique.

Step 2: Create the set of properties maintained by the component

Properties are created and accessed through a name you give them. Here is code that creates a property for the GSystems Properties Component to maintain:

```
IGProperty* pIGPropertyExpression;  
pIGProperties -> Add("expression", TYPE_SZSTRING,  
                    &pIGPropertyExpression);
```

Note that this method takes the address of an IGProperty pointer variable so that you can keep a reference to the property for later use if you want. You can omit this last argument if you don't need this type of access to the property. You will see later ways in which you might use this interface.

(Optional) Provide convenient access to each property's value to your software

The GSystems Properties Component generally utilizes storage that it allocates for each property. Our example property definition will create a

property of type “string” (TYPE_SZSTRING) (these constants are defined in Properties_i.h and on page 12).

You may want to have the Properties component use storage provided by its client, i.e., your own software. This is a more convenient way to manage the relationship between the property and the storage (variable) that the property represents in your software. You do this by calling the `DirectAccess()` method on the `IGProperties` interface.

```
static char szExpression[256];
    .
    .
    .
IGProperties ->
    DirectAccess("expression", szExpression, 256);
```

This way, whenever the char array `szExpression` is changed, you can be assured that the appropriate value of the property itself is updated.

It is very important to review the notes concerning the use of this technique found on page 17.

Without the direct “connection” so established, you would need to “update” the value of the property just before the object is to be saved. However, by virtue of this connection, the relationship between your data and the property you want to represent that data is always up to date, which can lead to a much easier and natural integration of the GSystems Properties Component within your system.

Step 3: Provide the path for your client to find the implementation of persistence interfaces through your object or application

The GSystems Properties Component will provide the services that manage the storage and retrieval of all of the properties you have defined. However, when you are creating a COM object, you typically need to provide *access* to these services, where the client of your object will use these services *through* the access you provide. This section will describe how you provide this access.

The container application, i.e., Visual Basic, PowerBuilder, etc., will, at times, ask your object for an interface pointer to one of the standard persistence COM interfaces. It needs this interface pointer in order to tell your object when and where to store its persistence data. If you are building an application, you need to know the event or method in your application that will be called by the system just before or during persistence. Or, as

described in Part II of this document, you can take control of the process and initiate the saving and restoring of properties yourself.

If you are building a COM component, you must return the value of `QueryInterface()` on your saved `pIGProperties` interface pointer any time your object is asked for one of the persistence interfaces. In other words, you forward a request for an interface having to do with persistence on to the GSystems Properties Component that you are using.

Here is the relevant code from the Function Object's implementation of the `QueryInterface()` method of `IUnknown`:

```
.
.
if ( riid == IID_IGProperties )
    return pIGProperties -> QueryInterface(riid,ppv);
else
if ( riid == IID_ISpecifyPropertyPages )
    return pIGProperties -> QueryInterface(riid,ppv);
else
if ( riid == IID_IPersistStorage )
    return pIGProperties -> QueryInterface(riid,ppv);
else
if ( riid == IID_IPersistPropertyBag )
    return pIGProperties -> QueryInterface(riid,ppv);
else
if ( riid == IID_IPersistPropertyBag2 )
    return pIGProperties -> QueryInterface(riid,ppv);
else
if ( riid == IID_IPersistStreamInit )
    return pIGProperties -> QueryInterface(riid,ppv);
else
if ( riid == IID_IPersistStream )
    return pIGProperties -> QueryInterface(riid,ppv);
else
.
.
```

Note that if you are building your object using ATL or MFC, you need to take special steps to ensure that the external (from a client) call to `QueryInterface()` gets routed appropriately. These techniques are discussed next.

How to provide the interface connections when using MFC

MFC COM objects typically provide interfaces through the `INTERFACE_MAP` related macros. In the MFC internals, a table of programmer supplied “nested” interfaces is searched for the desired interface when `QueryInterface()` is called.

This technique does not work with the persistence interfaces implemented by the GSystems Properties Component because MFC requires the interface to be nested, i.e., physically part of the client object's class, when using the `INTERFACE_PART` macro. Further, the `INTERFACEAggregate` macro will not work because this is searched for interfaces *after* MFC resolves which interfaces the base class (usually `COleControl`) supports, thus not finding it necessary to search the aggregates²⁰.

The solution to all of this is to override the base class implementation of `GetInterfaceHook()`. MFC will call this hook before searching for any interface as a result of a call to `QueryInterface()`.

Here is the declaration and implementation of this method that will serve the purpose:

```
// Declaration in header file
IUnknown* GetInterfaceHook(const void*);

// Implementation
IUnknown* Function::GetInterfaceHook(const void* piid)
{
    REFIID riid = *((IID*)piid);
    if ( riid == IID_ISpecifyPropertyPages ||
        riid == IID_IPersistStreamInit ||
        riid == IID_IPersistStorage ||
        riid == IID_IPersistStream ||
        riid == IID_IPersistPropertyBag ||
        riid == IID_IPersistPropertyBag2)
    {
        IUnknown* p = CCmdTarget::GetInterfaceHook(piid);
        pIGProperties -> QueryInterface(riid, &p);
        if ( p )
            ExternalRelease();
        return p;
    }
    return CCmdTarget::GetInterfaceHook(piid);
}
```

This method will help MFC to find the interfaces (again, through the object's cached standard interface pointer on the GSystems Properties Component). The call to `ExternalRelease()` is necessary to ensure that the reference count will go to zero at the appropriate time; MFC will

²⁰ The `COleControl` base class is providing persistence services by implementing these interfaces. However, the GSystems Properties Component provides a much tighter integration of your code with the *processes* involved with persistence and properties. If you want to take advantage of this higher functionality, you must inhibit the `COleControl`'s implementation as described. Note that this does not break your COM object in any way, it simply provides you more control over its functionality.

call `AddRef()` on the object (your COM component) after getting the interface pointer, so you must call `Release()` to balance out that call to `AddRef()`.

Note: The redirection of the `IID_ISpecifyPropertyPages` interface is optional. You can simply use the MFC implementation of property pages automatically generated for you. If you use the GSystems Properties Component's implementation of property pages, you will have much tighter control over the processes involved in setting properties and you will need to include this interface among those that are "redirected".

How to provide the interface connections when using ATL

ATL has a similar mechanism for managing interfaces; it uses a "map" of interfaces and a looping mechanism over this map to find the appropriate interface at run-time.

However, ATL's implementation is much more powerful, for example, there are several different "types" of entries in the interface map. One of these instructs the looping mechanism to call a certain function when a given interface is requested.

Here is a sample from a header file for an object that will redirect persistence interfaces to the GSystems Properties Component.

```
BEGIN_COM_MAP(CATLPropertiesSample)
.
.
.
/* Overriding persistence and property pages interfaces
COM_INTERFACE_ENTRY(IPersistStreamInit)
COM_INTERFACE_ENTRY2(IPersist, IPersistStreamInit)
COM_INTERFACE_ENTRY(ISpecifyPropertyPages)
COM_INTERFACE_ENTRY(IPersistStorage)
*/
// Code Added for GSystems Properties Component Support:
COM_INTERFACE_ENTRY_FUNC(IID_IPersistStream, 0, QIHook)
COM_INTERFACE_ENTRY_FUNC(IID_IPersistStreamInit, 0, QIHook)
COM_INTERFACE_ENTRY_FUNC(IID_IPersistStorage, 0, QIHook)
COM_INTERFACE_ENTRY_FUNC(IID_IPersistPropertyBag, 0, QIHook)
COM_INTERFACE_ENTRY_FUNC(IID_IPersistPropertyBag2, 0, QIHook)
COM_INTERFACE_ENTRY_FUNC(IID_ISpecifyPropertyPages, 0, QIHook)
.
.
.
END_COM_MAP()
```



```
static HRESULT WINAPI
QIHook(void* pv, REFIID riid, LPVOID* ppv, DWORD dw);
```

Note that the wizard generated entries in this map have been commented out. This is because the GSystems Properties Component will provide these services. Note also that the classes dealing with persistence have also been removed from the object's inheritance structure, since they are no longer needed.

These interface map entries tell the ATL interface searching mechanism to call the function `QIHook()` any time an interface of the type indicated is requested. The declaration of `QIHook()` (it is a static member function of the class) is as shown above, its very simple implementation is:

```
HRESULT WINAPI
CATLPropertiesSample::QIHook(
    void* pv, REFIID riid, void** ppv, DWORD dw)
{
    CATLPropertiesSample*
        pThis = (CATLPropertiesSample*)pv;
    if ( ! pThis -> pIGPropertiesClient )
    {
        pThis -> QueryInterface(
            IID_IGPropertiesClient,
            (void**) &pThis -> pIGPropertiesClient);
        pThis -> pIGProperties -> Advise(
            pThis -> pIGPropertiesClient);
        pThis -> pIGPropertiesClient -> Release();
        pThis -> pIGPropertiesClient -> Release();
    }

    return
        pThis -> pIGProperties -> QueryInterface(riid, ppv);
}
```

The parameter “dw” is the “0” declared in the interface map entry macro.

In the above example, the interface request is simply forwarded on to the GSystems Properties Component; the `pIGProperties` variable is a pointer to the default interface on that component.

Note that the example also takes the opportunity to notify the GSystems Properties Component it will participate in the persistence process. It does this by getting a pointer to it's implementation of `IGPropertiesClient` and calling the `Advise()` method on the GSystems Properties Component.

This needs to be done before any persistence occurs and you know that no persistence can occur until the persisting agent (a client of your object) asks for a persistence interface. Therefore, the first time a client asks for one of

these interfaces is a good time to setup the advise mechanism. Note the two `Release()`s are required to ensure the calling object will get deleted at the appropriate time. The object knows it's the first time because its member variable `pIGPropertiesClient` is set to `NULL` in the constructor and will be `NULL` the first time `QIHook()` is called.

Step 4 (Optional) provide access to your system for the GSystems Properties Component

Persistence of your object or application's properties is a process that will occur as the user runs or interacts with his or her application (which would be your client when you are developing an object).

The GSystems Properties Component provides two ways that you can hook your own object into this process. The easier way involves the use of the windows messaging mechanism; you simply provide the Properties Component with a handle to a window that the component can then send messages to, using message ID values that you also provide.

The more involved method is to provide the Properties Component with an interface that includes methods that the Properties Component will call at certain points within the process.

The easier method is described first.

Provide a window handle for the persistence process

You call the `PutHWNDPersistence()` method on your copy of the `IGProperties` interface to use this simpler technique. You can call this method as many times, that is, for as many windows, as you'd like. For example, if you have implemented multiple windows, perhaps each depicting a separate object or part of an object, you can pass the handle to the Properties Component for each of them.

Calling this method will cause the GSystems Properties Component to send a particular message to your code (the associated window's message handler) at key points in the process. To help describe this, here is a sample call:

```

pIGProperties -> PutHWNDPersistence(
    hwndPersistence,
    hwndInit,
    hwndLoaded,
    hwndSavePrep);

```

This call passes the handle of the window in which several commands are available. Each “command” is associated with the windows command event on each of the 3 sub-windows (controls), typically command buttons, passed to the method. Provide a NULL window handle for functionality you do not implement.

Each of the control window handles is associated with the following actions:

- **hwndInit:** When the persistence mechanism is told by the container or application that the object should initialize, the GSystems Properties Component will send a command to this control.
- **hwndLoad:** When the container or application restores property values from the persistence storage created on the last save, the Properties Component will send this command to your window. When you receive this message, you know that all of your properties are up to date with respect to the last save and that you can safely redraw your object taking these values into account.
- **hwndSavePrep:** This command is sent just before actually saving the data to the binary storage. This gives your application an opportunity to update the values of the properties so they will have the intended value on the storage.

When you use this technique, you put code in your window’s message handler that intercepts the WM_COMMAND messages for the above command button controls²¹. Note you passed a handle to a window to `PutHWNDPersistence()` and that this window’s message handler will receive the messages. You may need to investigate the mechanics of windows messaging for your particular environment if you find that your code is not getting called in the expected places and times from the GSystems Properties Control.

²¹ Or, with an environment like Visual Basic, you code your logic into the “Click()” event on the associated command buttons and Visual Basic will route the request from the GSystems Properties Control to that code as appropriate.

Note: With this form of the persistence process, you need to make sure that your window (that handles the commands) has been created by the time the container or client application asks your object to load from the storage. In general, you can't know exactly when (after creation) this will occur, so you may need to create the associated window early in your object creation. Of course, you can't call the `PutHWNDPersistence()` method until *after* the window has been created. Note also that the window handles passed to the method must also exist throughout the duration when the persistence mechanism may be used, which may in turn be as long as your object exists.

Provide an interface to the GSystems Properties Component for the persistence process

The GSystems Properties Component is able to communicate with your application or object using an interface called `IGPropertiesClient`. This interface is defined in the file `Properties_i.h`.

The “Client” in the name of this interface refers to the GSystems Properties Component. In other words, the GSystems Properties Component is going to turn around and get services from your software by calling methods on this interface. While implementing this interface is more work on your part, the functionality thus enabled allows your software to participate completely within the process of saving your application or object to system storage.

In order to take advantage of the functionality described in this section, you must tell your instance of the GSystems Properties Component that you support the `IGPropertiesClient` interface. To do this, you must first obtain a pointer to your implementation of `IGPropertiesClient`, usually by calling `QueryInterface()` for the interface `IID_IGPropertiesClient` (defined in `Properties_i.h`).

MFC Users can call `InternalQueryInterface()`. Or, if you call `GetControlUnknown()`, this will supply an `IUnknown` pointer to your own object. You can use that `IUnknown` Pointer to obtain your implementation of `IGPropertiesClient`:

```
IGPropertiesClient* pIGPropertiesClient;  
pIUnknown -> QueryInterface(IID_IGPropertiesClient,  
                             (void**) &pIGPropertiesClient);
```

Carefully note, however, that you must “implement” this interface. This means you must write the code for the methods in the interface and “connect” those methods to your object or application. Typically, this

means using the `INTERFACE_PART` macro, which is one approach in MFC.

Once you have the interface pointer, provide it to the GSystems Properties Component:

```
pIGProperties -> Advise(pIGPropertiesClient);
```

The GSystems Properties Component will `AddRef()` the interface.

The `IGPropertiesClient` interface provides a rich set of methods that connect your software with the GSystems Properties Component. This section has just been an introduction to this topic. Following is an entire section of this manual covering the details of using the `IGPropertiesClient` interface. Also, you can refer to page 96 to see the full descriptions of these methods in a reference format.

Part IV – Intercepting the Save, Load, and InitNew functionality of the Properties Component

The COM persistence interfaces offer a standard mechanism for reading and writing binary data onto a storage space. Typically, to enable persistence, you need to implement one of the persistence interfaces and subsequently process the calls clients of your object, or, other parts of your system, will make on that interface to give you the opportunity to save and restore.

Because you are using the GSystems Properties Component, all of the details with regard to the reading and writing of your property data are handled for you. To make the system as flexible as possible, the GSystems Properties Control uses a certain process which happens when this activity (saving or loading data) occurs. The best way to explain the use of the methods involved is by following this process and encountering each method as it is used.

A Clarifying Point Concerning Object Hierarchies

It is very likely that you will utilize other objects in the development of your software. One of the powerful features of the GSystems Properties Controls is that it can completely cover the persistence needs of these “other” controls – seamlessly managing the properties of these controls just as if *they* had also used the GSystems Properties Control. In general, all you need to do is to provide the GSystems Properties Control with access to each of these objects and the control will store the objects’ data along with your own object’s data.

There will be much more detail on the techniques used to integrate other software artifacts into your system later.

The mechanics of implementing the IGPropertiesClient COM interface

Depending upon your development environment, there may be several ways that you “implement” an interface.

Implementing an interface generally involves two general concepts:

- 1) Writing the code for each of the methods in the interface. If you want to implement an interface, it is necessary that you write code, probably all within one source code file, for each method that is in that interface. These methods must have exactly the signature²² specified in the publicly accessible interface definition²³. The required methods will probably be known to your development environment so that the environment may be able to let you know if you have successfully provided all that is necessary.
- 2) Publishing the interface so that clients external to your software will be able obtain that interface from your object or application. It may sometimes be only necessary that your application or object *provide* a client with the appropriate interface through some “Advise” mechanism. Usually, in such a case, the “Client” defines the interface and the Server provides an appropriate instance of the interface to the client.

There are several ways to accomplish these steps. However, this document will cover only a few that can be used in popular development environments. If you are using an environment that is not able to use any of these techniques, you are welcome to check the InnoVisioNate.com web site for additional information that will become available in the future.

Providing an interface in a VB project

Visual Basic provides the “Implements” keyword that tells the compiler that this file (a class file) will form the interface (step 1 above). The 2nd step, publishing the interface is handled entirely behind the scenes within VB.

As you type in the methods (called Functions and Subs in VB) of the interface, Visual Basic will alert you if you have not gotten the “signature” of any of the methods correct, that is, if the arguments or return type are not as specified in the interface definition. Also, in the VB environment, note that the interface definition is accessible by pressing the F2 key and looking at the referenced objects’ type library information.

²² A “signature” denotes the types of each argument to the method as well as the type of the return value.

²³ This definition may come from a type library, from the “Components” information maintained by Visual Basic, or from a header file accompanying a COM object that may be expecting to use this interface.

When you create functions in your implementation file, note that the name of the function is pre-pended with the name of the interface. Thus, for the IGPropertiesClient interface's Saved() interface, the declaration is:

```
Private Sub IGPropertiesClient_Saved()  
End Sub
```

Until you correctly specify the stub of each of the required functions or subs, VB will not successfully compile the source file. Remember that the definition of these functions may also include the ByRef and ByVal keywords and that the return value of any functions must also be exactly as specified in the interface definition.

Providing an interface in a MFC project

ActiveX Objects using an MFC generated project are based on the COleControl base class. This class also has as one of its ancestors, the CCmdTarget base class, *which*, in turn, implements a table driven approach to finding and supplying interfaces to clients.

This table of interfaces is called the "Interface Map" and you are able to add entries into this map so that interfaces that you implement will be found there, by the underlying mechanism, and will be available to external clients.

The step by step procedure to implement the IGPropertiesClient interface in an MFC project is as follows:

- a. Include the "Properties_i.h" file in the header file. Usually the control you are building's main header file. This file is shipped with the GSystems Properties Component.
- b. Put the DECLARE_INTERFACE_MAP macro in the declaration of your control's class.
- c. Declare all the methods of the IGPropertiesClient interface between the BEGIN_INTERFACE_PART and END_INTERFACE_PART macros, also in the declaration of your control's class.

Here is the relevant code from the object's header file:

```
DECLARE_INTERFACE_MAP()  
  
BEGIN_INTERFACE_PART(GPClient, IGPropertiesClient)  
    STDMETHOD(SavePrep)();  
    STDMETHOD(InitNew)();  
END_INTERFACE_PART
```



```

        STDMETHODCALLTYPE (Loaded) ();
        STDMETHODCALLTYPE (Saved) ();
        STDMETHODCALLTYPE (IsDirty) ();
        STDMETHODCALLTYPE (GetClassID) (VARIANT *pCLSID);
    END_INTERFACE_PART (GPClient)

```

- d. Put the `BEGIN_INTERFACE_MAP` and `END_INTERFACE_MAP` macros in your control's implementation file.
- e. Between the above two macros, put the `INTERFACE_PART` macro that places the entry for the `IGPropertiesClient` interface into MFC's mechanism for finding interfaces.

Here is the relevant code from the object's implementation file:

```

BEGIN_INTERFACE_MAP (CFunctionContainerCtrl, \
                    COleControl)
INTERFACE_PART (CFunctionContainerCtrl, \
                IID_IGPropertiesClient, GPClient)
END_INTERFACE_MAP ()

```

Note that the example is for building the object whose main class is `CFunctionContainerCtrl`, directly inherited from `COleControl` and that the `GPClient` entry is the same entry in the first position of the `BEGIN_INTERFACE_PART` macro used in the header file.

- f. Finally, implement the code for the interface. The MFC sample included, the `FunctionContainer` sample, contains the necessary source in the `IGPropertiesClient.cpp` file, which can easily be copied to your own project as a stub if desired.

An additional note is that you must implement the `IUnknown` members of this interface (`QueryInterface()`, `AddRef()`, and `Release()`). In general, these methods simply “delegate” to the main object. The above referenced sample provides an example of this.

Providing an interface in a ATL project

Implementing an interface in an ATL project is somewhat easier than it is with MFC. In ATL, you right click on your object's main class in the “Class View” tab of the work prompster, and select “Implement interface”.

This will present a tabbed dialog that shows you certain Type Libraries which are known to the project and which describe interfaces. If the

GSystemProperties tab is not present here, click “Add TypeLib...” button, select “G System Properties”, and click OK.

From the GSystemProperties tab, select the IGPropertiesClient (and IGPropertyPageClient for later use if you like) interface and click OK.

This will add the methods defined in the interface to your object’s main header file, You may want to create a new source file and move these methods over to it, leaving the declarations in the header file, of course. Your object will now have the capability to provide the interface to any external client asking for it, in this case, a GSystems Properties Component. Note that ATL uses interface inheritance as its mechanism for implementing interfaces. Therefore, you will see that your object now has IGPropertiesClient in its inheritance structure.

The specifics of the IGPropertiesClient interface

Initializing a New Object

A new object will be created either by your object, or by the user initiating the creation of the object through some COM mechanism, for example, through Insert-Object on a menu. What is important is that you have “redirected” the `QueryInterface()` call so that the GSystems Properties Component provides the persistence services for you, see page 39.

At some point after object creation, most clients will call `InitNew()` to tell the object to load its “default” state. Note that the Properties Component will be receiving this call. In order to give your object the opportunity to perform its own processing at this time, the Properties Component will then call the `InitNew`²⁴`()` method on the IGPropertiesClient interface that you have implemented.

Essentially, the `InitNew()` method can be thought of as a “Reset State” function. It is perfectly legal and appropriate for your code to call this method from any place within your system when this kind of action is required, for example, a “reset defaults” implementation.

²⁴ Or, it will send a message to the `hwndInit` window passed in a previous call to `PutHwndPersistence()`.

Saving the Object's State

When the user attempts to save your object or application, the system will call `Save()` on some implementation of `IPersistStorage` (or `IPersistStream` as appropriate). Since you have “redirected” this method to the GSystems Properties Component, it will handle the actual saving of all of your binary (properties) data to the provided storage.

However, before writing the data, the GSystems Properties component will call the `SavePrep25()` method on your implementation of `IGPropertiesClient` to give you the opportunity to ensure that all of your properties have the appropriate values and/or to allow you to do some prior processing. If you do not keep your properties up to date with the data within your object, you will need to set the values of the properties here so that they will have the appropriate values before saving. A simpler method of keeping your data and the property in synch is to use the `DirectAccess()` method described on page 38.

That is all that is really necessary in `SavePrep()`.

After `SavePrep()` has been called on your object, by the Properties Component, it will actually save the data using the standard COM mechanism of writing the binary data to the storage provided by the system. When this is complete, *then* the Properties Component will call your implementation of `Saved()` on your `IGPropertiesClient` interface.

Ensuring multiple objects are persisted

When your application or object uses other COM enabled objects, such as ActiveX controls, you can instruct the GSystems Properties Component to persist their properties as well.

The `AddObject()` method on the Properties Component tells that component that you would like to persist that object's properties among your own objects properties.

```
GProperties.AddObject(listView.object)
```

²⁵ Or, it will send a message to the `hwndSavePrep` window passed in a previous call to `PutHwndPersistence()`.

The above is a typical VB call to instruct the Properties Component (named `GProperties` when it was placed on the form) to include the properties of another object (in this case, one named `listView` which might be an instance of the `ListView` common objects). The `.object` qualifier ensures VB passes the appropriate object; without it, VB will pass a special “container” object it uses to host ActiveX (or other COM) objects.

Note that you use this when there is a known number and type of “other” objects you are using in your system. If you are using an arbitrary number of objects, i.e., you can’t know the number of them, and at least the type of them, use the technique described next.

Ensuring actively changing numbers and types of objects are persisted

There are certain situations where you may not know how many, or even what type of, other objects may exist in your control or application at any given time. A clear example of this would be a development environment that must manage multiple different objects such as those populated on a form in Visual Basic.

There is a special type of property provided for just this situation; when you need to have one property that can manage the persistence information for any number and any type of objects²⁶.

The full description of this technique is provided on page 13.

Restoring the Object’s State

When the GSystems Properties Component gets the call to load the data, it will call your implementation of `Loaded`²⁷ () when this processing is finished. Typically, in this method, you have the opportunity to update the contents of your windows controls and/or fields because you can be sure

²⁶ Remembering that the sequence of properties or objects that the GSystems Properties Component expects to encounter when Loading the data *must* be the same sequence as when that data was saved. For this reason, you cannot arbitrarily add properties at run-time and a single type of property is needed that can act as a “place-holder” for many objects at once.

²⁷ Or, it will send a message to the `hwndLoaded` window passed in a previous call to `PutHwndPersistence()`.

that the properties have freshly received their values that existed just before the prior save.

Once you have performed the necessary processing in the `Loaded()` method, your object or application's state will be entirely up to date with respect to how it was last saved. If convenient at this time, you can draw your object(s) so that this state is visually presented to the user.

Part V - Using the Property Page Implementation

An Introduction to Property Pages

When your user wants to alter the properties in the system, he or she will select something in the application interface that initiates the process. If your software is an object within another application or container, then *that* application would be responsible for providing the capability to initiate the process.

In any case, there are two different ways that a client may launch the edit properties process.

1. By obtaining an `ISpecifyPropertyPages` interface from your object. Using that interface, a client builds a set of windows (building the owner dialog, typically a tabbed page and *retrieving* the *owned* dialogs from your object) and display these windows to the user.
2. By calling `DoVerb()` on your implementation of the `IOleObject` interface with the value `OLEIVERB_PROPERTIES` in the first argument in the call. In an MFC `COleControl` derived object, this causes the `OnProperties()` method to be called. If you manually implement the `IOleObject` interface, the `DoVerb()` method on that interface is called.

When the first method is used by a client, note that you need do nothing else, the edit properties process is already under way and your property pages (as described in the next section) will be shown.

However, in the second case, you need to start the process off yourself. This is simply accomplished by calling the `EditProperties()` method on your cached pointer to the `IGProperties` interface.

```
pIGProperties -> EditProperties(  
                                hwndParent,  
                                L"tab Text",pIUnknownThis);
```

This method is called with what will be the parent window of the resulting property settings dialog box, the text to put in the title bar of the dialog, and a pointer to the `IUnknown` of the object for which the properties are to be maintained. If you've used MFC, you can call the `GetControlUnknown()` method to retrieve the pointer for the `IUnknown` interface to your object.

Note that the parent window handle is likely to be the window of the container if your software is an object. In fact, method `DoVerb()` receives a window handle that is suitable for this purpose.

Note also that the `IUnknown` pointer can be for *any* object that you currently have instantiated within your object or system, not just an object you are creating. This is true even if the object does not support the `GSystems Properties Component` property pages as described next. Significantly, this allows you to initiate the edit properties process for *any* object for which you have an `IUnknown` pointer (which you can retrieve for all valid COM objects on the currently running system)²⁸

At the heart of this topic is the implementation of the `IPropertyPage` interface that is defined by the Microsoft COM standard documentation²⁹.

If you have previously used the above interface, you may have discovered some vagaries with the definition and use of the interface through the typical property page implementation, i.e., `OleCreatePropertyFrame()`. If so, you may appreciate that the

²⁸ You can edit properties in sub-objects of your object this way. However, you can also *combine* the property pages of all objects you are maintaining in your system using the techniques provided by the `GSystems Properties Component`'s implementation of Property Pages.

²⁹ You don't have to implement this interface, the `Properties Component` does it for you. However, it may help to understand how this interface works to most effectively use the `GSystems Properties component`.

GSystems Properties Component solves some of the problems with that implementation.

If, for example, you believe that cancel means cancel, you will find the GSystems Properties Component allows cancel in a way that truly makes sense. This is in reference to the typical implementations of apply that (correctly) apply property changes to the system context, for example, updating the window with the *proposed* changes in effect, but which do not then allow the rollback of those changes. In particular, when the user presses cancel, the state *should* revert to the state of the system when the user first selected “change properties”, however, this is typically not the case.

The GSystems Properties Component fully supports both types of implementations.

The Mechanics of a Property Page

The GSystems Properties Component is able to display a window containing child windows and controls that contain the values and provide edit opportunities for the properties. In all objects or applications that support the display of property values, some window has to be created to host these controls and entry fields. Users familiar with MFC know that the property page generated for an MFC component is essentially just a dialog.

The GSystems Properties Component shares the work of creating the user interface necessary for properties manipulation with you, the developer of the application or object that will be using the Properties Component.

Specifically, you are responsible for creating the dialog window and placing a set of desired controls on that window, using whatever tool you have to do so. In Microsoft MFC, you can create a simple dialog based on the CDialog MFC class. In Visual Basic, you can simply create another form, etc. What is important is that all of the events that happen to that window, for example, when the user is manipulating a control or entering text, are under your software’s full control. The GSystems Properties Component does nothing to interfere with what your window wants to do in its own implementation. The Properties Component simply provides the mechanism to display the window at the appropriate time.

The important thing for you to do, then, when using the GSystems Properties Component is to tell the component where this window is. There is an easy way (and perhaps entirely sufficient, depending on your

circumstances), and there is a more complex, yet feature rich, method you can use to accomplish this.

The easier method is described first.

Providing a property page window handle to the Properties Component

If your application creates a simple properties settings dialog, you can provide the GSystems Properties Component with a handle to that window to display when the user manipulates the properties. You accomplish this by calling the `PutHWNDPropertyPage()` method provided by the Properties Component.

```
pIGProperties -> PutHWNDPropertyPage(  
    BSTR displayName,  
    long hwndProperties,  
    long hwndStart,  
    long hwndOK,  
    long hwndApply,  
    long hwndCancel);
```

Note that in the above syntax, the `BSTR` variable provides the “displayed” name of the window in a group of windows, for example, the text appearing on a tab when there is more than one window available. This is a string variable, your particular environment probably understands this. In Microsoft C++, the value can be passed as `L”the Text”`, as an example.

Note also that you may call this method with as many windows as you want, which allows you to “categorize” your properties in a way that makes sense with your application. You can call the associated `RemoveHWNDPropertyPage()` method to take any property page window back out of the group.

In the method call, the `HWND` variable is the “window handle”, in PowerBuilder, there is a function called `Handle()` that provides this value. In an object derived from an MFC window class, you would use the `GetSafeHWND()` method to return this value, and in Visual Basic, this is the `hWnd` property of a form.

The values for `hwndStart`, `hwndOK`, `hwndApply`, and `hwndCancel`, are the handles of the command buttons your window would implement if the window were to be shown by your own application or object (which it can do if you’d like). For example, the `hwndOK` control (probably a push button control) is one of these that an MFC generated dialog would use for

the “OK” button. What is important is that your code *does* have to have the event code implemented. In MFC you must have an event “registered” and implemented for the dialog whether or not you have actually placed the actual control on the dialog (which is not needed or desired for the GSystems Properties Component’s use).

You can simply put a command button on the dialog and pass that button’s handle to the `PutHWNDPropertyPage()` method. It is your implementation of the command event for that particular button that is called when the user presses a command button on the properties dialog³⁰. Remember, the apply button on the properties is supplied by the system, *not* the one you created on your properties dialog. In fact, you should make that button invisible when the properties dialog becomes visible. There will be more about this later.

Providing the easier properties windows – Summary

The steps in the process that occurs when you have taken this simpler approach are outlined to provide another frame of reference for using this information.

1. First, as mentioned before, you develop the window that will contain the “settings” or properties for your application or object. You would have added an event handler for 4 commands in this window; the “Start” command, the “OK” command, the “Apply” command, and the “Cancel” command. The actual command buttons should be hidden at run time. Note that you have full control over this window and the events and messages that the window processes.
2. At some point after you have created an instance of the GSystems Properties Component within your object or application, you will call the `PutHWNDPropertyPage()` method on the interface to that component, telling it that you want to add the window handle as a property page and also providing window handles of the buttons that contain major parts of the processing on that property page.

³⁰ Keep in mind that the actual properties dialog is built and displayed by the system. The dialog that you create, whose window you pass to `PutHWNDPropertyPage()`, is displayed as a *child* of the system generated properties dialog.

3. Either the user or the system initiates the “setting” of properties such that the windows containing the interface used to set properties are to be shown. At this point, your property window (or windows) will receive the command notification associated with the clicked event of the “start” button you created, i.e., the code you wrote as the “action” of the start button will execute. It is during the processing of this message that you may, if necessary, initialize your dialog or window so that all of the fields are “up to date” as far as their association with the properties.
4. Now the user is manipulating the controls and entry fields in your properties dialog. Your dialog’s window procedures and/or methods will get called just as if you had presented the dialog to the user in your own code. You can, in particular, process the changing values in controls on this dialog, updating the values in the corresponding properties, for example.
5. As the user presses the Apply button, the code you created for the command associated with the “apply” button you provide will execute. In particular, you specified the window handle associated with Apply when you notified the GSystems Properties Component of your intent to use this window for properties (step 2). You can set up your processing of this command as appropriate, taking the actions necessary to support “Apply” in your application, such as updating the view of your object with the current property values.
6. Also, when the user presses the OK or Cancel button, the property settings dialog is going to close. Before this happens, *all*³¹ of the windows that your object or application is using for property pages will get a processing opportunity through either the window handle for the Ok or for the Cancel command button.

Note that the windows that you use for these purposes should exist at any time that the user (or system) is likely to start the activity of setting properties. In MFC, for example, you might create your dialog and keep it around as long as your object is in use. You couldn’t just “create” the dialog when it is needed because the GSystems Properties Component needs to be given a handle to a window (`PutHWNDDPropertyPage()`) that will be

³¹ Remember that you can supply any number of windows to the GSystems Properties Component if you would like to have different “sub-sets” of properties.

used at some arbitrary point in the future. In other words, there is no way (using this simpler method) for the system to notify your object that it is going to need a properties setting window soon and for you to make one, you essentially need to always have one around for this purpose.

Enable the use of IGPropertyPageClient and custom property page windows

Using the IGProperties interface to your instance of the GSystems Properties Component, call the `AddPropertyPage()` method. This method requires that you pass a pointer to the IUnknown interface on an object that implements the IGPropertyPageClient³² interface.

Note: The IUnknown interface is for *your* component, the object you are developing. The Properties Component will query this interface for an instance of the IGPropertyPageClient interface which *your object* implements and which the Properties Component will use to communicate with your object during the properties setting process the user is performing. The Properties Component will *not* `AddRef()` the IUnknown interface, it will, however `QueryInterface()` through the IUnknown for the IGPropertiesClient interface, which will cause an `AddRef()` for that interface.

You don't have to provide the handle of the window to use in the interface as you did in the prior section because the GSystems Properties Component will retrieve this value later on through the IGPropertyPageClient interface.

The `GetPropertyPagesInfo()` method (that you implement) is used by the Properties Component to get the *collection* of property pages you want to show in the set properties dialog box. When the Properties Component calls this method, you need to provide a count of the number of windows involved, an array of strings to display as the page names, an array of strings specifying the help files associated with each page, and array of window sizes. This last array will contain 2 entries (longs) for each window

³² It is actually not required that the passed in object support IGPropertyPageClient. Significantly, if this object supports ISpecifyPropertyPages, which any object must do in order to present property pages, then this object's property pages will be included along with your own, if any, in the set of property pages; seamlessly added to yours as if you implemented them. See page 68 for more information.

you are providing; the width and the height of the 1st window are in the first 2 values, for example.

Note: The `GetPropertyPagesInfo()` method fills in several `SafeArrays`³³ that are passed to the method. These `SafeArrays` contain space for *one* set of information upon entry. If there will be more items returned, it is required that the implementation “Re-Dim” the provided arrays. Here is the code for an object that implements 8 property pages on behalf of the object G:

```
HRESULT G::GetPropertyPagesInfo(long* pCntPages,
                                SAFEARRAY** thePageNames,
                                SAFEARRAY** theHelpFiles,
                                SAFEARRAY** thePageSizes)
{
    if ( ! pCntPages ) return E_POINTER;

    *pCntPages = 8;
    SAFEARRAYBOUND rgsabound[1];

    rgsabound[0].cElements = *pCntPages;
    rgsabound[0].lLbound = 0;

    rgsabound[0].cElements = *pCntPages;
    rgsabound[0].lLbound = 0;

    // Resize the array to pass back the titles and help files
    SafeArrayRedim(*thePageNames,rgsabound);
    SafeArrayRedim(*theHelpFiles,rgsabound);

    // Resize the array to pass back the window sizes
    rgsabound[0].cElements = 2 * ( *pCntPages );
    SafeArrayRedim(*thePageSizes,rgsabound);

    long index = 0;
    BSTR bstrPageName = SysAllocString(L"Pos-Size");
    SafeArrayPutElement(*thePageNames,&index++,bstrPageName);
    SysFreeString(bstrPageName);
    bstrPageName = SysAllocString(L"Style");
    SafeArrayPutElement(*thePageNames,&index++,bstrPageName);
    SysFreeString(bstrPageName);
    .
    .
    .
}
```

³³ Your development environment should know what this is. In Visual Basic, the argument to the subroutine for the “names” array will appear as:

```
...,pageNames() As String,...
```

You must “ReDim” this array as appropriate before putting the strings into it.

```

    .
    long w = 100;
    long h = 100;
    index = 0;
    SafeArrayPutElement (*thePageSizes, &index++, &w);
    SafeArrayPutElement (*thePageSizes, &index++, &h);
    .
    .
    return S_OK;
}

```

The important point here is that the callee may need to re-dimension the `SafeArrays` to pass back to the `GSystems Properties Component`.

Because the `GSystems Properties Component` is built to support the process of manipulating property pages, The rest of the property page support is described within that process.

The Property Page Creation and Manipulation Processes

The process starts when the user elects to edit properties in your object or application. At this point, your application will receive a `QueryInterface()` call for the `ISpecifyPropertyPages` interface. Your implementation of `IUnknown` will delegate this call to your cached instance of the `IGProperties` interface (see page 39). This starts the chain of events.

The `GSystems Properties Component's` implementation of `ISpecifyPropertyPages` will call the `GetPropertyPages()` method on your application's implementation of `IGPropertyPageClient`³⁴. This method, which you write, returns the window handles and the textual description of each property page you want to display. Remember, you create `SafeArrays` to pass the strings and window handles back to the `Properties Component`.

After all of the property pages have been defined, the `BeforeAllPropertyPages()` method is called on each object that is supporting `IGPropertyPageClient`. This call is an opportunity to do whatever your application needs to in preparation for the property pages. It could, for example, load values into the controls on the property settings window.

³⁴ All of this assumes that your application or object implements the `IGPropertyPageClient` interface which it must do in order to use this property page implementation. If it does not, you may still use the simpler windows messaging based technique.

Note that it is possible to “mix” the simpler windows messaging technique with the technique implementing the `IGPropertyPageClient`. In such a case, *both* these processes are occurring together. Specifically, your windows involved in the simpler method will also receive their command messages at the appropriate time. See page 58 for more information.

Set up properties comparison

One very good thing to handle at this point is to setup the ability to detect changes in the properties. In particular, this is a good time to “Push” the properties so that you can later “Compare” the pushed set of values with the current set. Table 3 shows the methods on the `IGProperties` interface pertinent to this activity.

IGProperties Method	Performs the following
<code>Push()</code>	Puts the current set of properties on a “stack”, essentially saving a copy of the entire property set that can later be restored (Popped).
<code>Pop()</code>	Pops the topmost (or last pushed) set of properties off the stack, overwriting the “current” set of properties.
<code>Discard()</code>	Pops the topmost (or last pushed) set of properties off the stack but does not overwrite the current set of properties.
<code>Compare()</code>	Returns <code>S_OK</code> if all properties in the current set of properties equals all properties in the topmost (or last pushed) set of properties. Returns <code>S_FALSE</code> otherwise. Returns <code>E_FAIL</code> if the stack is empty.

Table 3 Property Set Manipulation Methods on `IGProperties`

In order to have the ability to both “Apply” and “Cancel” a set of properties, you `Push()` the current properties twice at the start of the process. When the user presses “Apply”, you `Discard()` the top of the stack, then re-`Push()` the current values. The stack remains 2 deep no matter how many times the user presses “Apply”. Now, if the user presses “Ok”, you can simply `Discard()` both of these 2, the current set of properties and values are what the user wants. If, however, the user presses “Cancel”, you simply `Discard()` once, then you `Pop()` the stack, returning the values of all properties to the values they had when the entire process started.

Now we will continue with the events that occur as the user is manipulating the properties.

Remember that your code created the windows that are showing in the properties dialogs that are being presented to the user. In particular, these properties dialogs are implemented using the `OleCreatePropertyFrame()` system call (which the GSystems Properties Component sets up and calls on your behalf). This is a “Tabbed” dialog with each tab constituting one of the windows that you provided to the Properties Component.

During the user’s manipulation of controls in the dialog, the system will be making calls to the `IsPageDirty()` method on your implementation of `IGPropertyPageClient`. You should provide the value `false` if the current set of properties is exactly the same as the “saved” set of properties. In other words, if the user has changed any property (through his/her interaction with your window and subsequently due to processing that is happening in your window procedures), return `true`, the properties are “dirty”.

Note that this method is called with page number³⁵ of the particular window that is currently active. In other words, if you were providing 3 windows to the system you could use the page number provided to ascertain, at run-time, what particular “sub set” of the properties are being asked about. In general, it is easier just to consider all the properties. In any case, if the `Compare()` method returns `S_OK`, then you can be sure that no property has changed.

As the user navigates the tab control, the system will activate each window as it is first shown to the user. The Properties Component will call `CreatePropertyPage()` as necessary to indicate to your code that you need to create the specific window to show. You will know which window to create based on the property page number passed in.

If the user clicks the Apply button on the properties dialog, the Properties Component will call the `Apply()` method. This implies a global action that indicates you should update the object representation such that *all* properties are taken into account³⁶.

³⁵ The pages are numbered from 1 through the number of property pages you specified in your implementation of the `GetPropertyPagesInfo()` method, and are in the order that you placed them in the arrays passed to that method.

³⁶ This is the easiest approach. If you want to break your set of properties into sub-sets, each represented by its own property window, you may need to avoid using all of the properties when you draw your object. Keep in mind that, if you are updating properties in your property window procedures, these values *will* have been updated when each respective window was

During the `Apply()` processing, it is a good time to `Discard()` and `Push()` a new set of (current) properties on the stack, as previously discussed:

```
pIGProperties -> Discard();  
pIGProperties -> Push();
```

This supports the proper implementation of the Cancel action.

When the user clicks the Ok or Cancel button, the properties dialog will be dispatched and the property settings session is (almost) over. Your code will receive the call to `AfterAllPropertyPages()`, with a parameter indicating whether the user pressed Ok, or Cancel. If the user clicked Ok, you simply need to `Discard()` twice, if they cancelled, call `Pop()` twice, then, update your windows controls in your object to reflect the now current values of the properties³⁷.

Finally, the GSystems Properties Component will call the `DestroyPropertyPage()` method, again, passing the page #, once for each page defined, giving you the opportunity to free up the resources used by the property page window(s). Note you do not have to create and destroy these windows in this process. If you like, you may have had them already created before the process starts, then, in `CreatePropertyPage()` you would switch the parent of that window, and in `DestroyPropertyPage()`, switch the parent back to one of your own object's or application's windows.

Highly optional implementation of the IPropertyNotifySink interface

There is one final detail that may optionally occur after all property pages have been dispatched.

There is a system defined interface called the `IPropertyNotifySink` interface that is used to notify containers that one or more properties in an object have changed. This is a “sink” interface which means that a container

visible, then if you want to see the effects of “Applying” only the properties that are on the *current* window, you must somehow separate out the “other” properties in the representation of your object. Of course, you can always have more than one instance of the GSystems Properties Component.

³⁷ If you implemented `IGPropertiesClient`, simply call your implementation of `Loaded()` on that interface.

implements it and the object calls the methods. Much like the GSystems Properties Component calls the methods on the `IGPropertyPageClient` interface that your object (the sink) provides to that component.

When a container instantiates an object, for example, the one you are writing, it, the container, may request of your object for an instance of the `IConnectionPointContainer` interface. This is a way for the container to find out if the object has the ability to call methods on “outgoing” interfaces, or, if the object is capable of calling methods on interfaces that will be provided by the container³⁸. If you are actively³⁹ implementing event interfaces in your object, you will be able to detect when the container is asking for a “connection point” for the `IPROPERTYNOTIFYSINK` interface.

When the container ultimately provides your object with an actual instance of this interface, you can pass the interface to the GSystems Properties Component. This will give the Properties Component the information it needs to tell the container, in turn, that properties have changed when the user has dispatched the property pages.

Even though the implementation of events and outgoing interfaces is very complex, you don’t need to forgo this ability if you don’t have control over the parts of your object where this is implemented.

There is a simple interface called `IQuickActivate` which, if you or your development environment implement (and which you can hook into), which will receive a pointer to an `IPROPERTYNOTIFYSINK` interface as soon as your object is created⁴⁰. During the execution of the `QuickActivate()` method on this interface, pass the `IPROPERTYNOTIFYSINK` interface pointer on to the GSystems Properties Component:

```
HRESULT Function::QuickActivate(  
    QACONTAINER* qaContainer,  
    QACONTROL* qaControl)  
{  
    .  
    .  
    .  
}
```

³⁸ When an object defines a certain “event” interface, it is essentially asking a container to implement the interface, and expects the container to in-turn, ask the object if it is able to call methods on that interface if it, the container, will agree to implement it.

³⁹ Meaning that you have control of the pertinent parts of the process in your object, or, that you can somehow “hook” the process if it is under the control of MFC or ATL, for example.

⁴⁰ If the container knows about and seeks to use this interface.

```

.
.
if ( qaContainer -> pPropertyNotifySink )
    iProperties -> ConnectPropertyNotifySink(
        qaContainer -> pPropertyNotifySink,
        &qaControl -> dwPropNotifyCookie);
.
.

```

Which will give the Properties Component the information it needs to, for example, notify Visual Basic that properties have changed after the user has exercised the properties dialog(s) that you have implemented.

Showing the property pages of other objects

The `EditProperties()` and `AddPropertyPage()` methods of the GSystems Properties Component fully support the properties editing process for *any* object that supports the `ISpecifyPropertyPages` interface. In general, this is the vast majority of COM/ActiveX objects. Any object which provides property pages *must* implement this interface⁴¹.

Adding property pages of any object

This means that you can easily present the property pages of other objects to the user of your object or application as if you implemented those property pages yourself.

Further, you can mix these property pages amongst your own just as if you developed these “external” objects, integrating them completely within your system.

As an example, consider the property pages for the windows common controls, such as a `ListView` control. Such a control might have as many as 10 separate pages that control every detail about the appearance and operation of the control. Typically, you as developer only see these property pages within the development environment you are using to place these controls within your application.

Now, using the GSystems Properties Control, you can show these same “design-time” property pages to users of your application or object! At the

⁴¹ In the case of your object, you are allowing the GSystems Properties Component to do this for you.

same time, you can place these property pages directly beside your own, and of other objects, such that the entire integrated package is presented to your user at once.

All that is required to achieve this result is that you call the `AddPropertyPage()` method on the GSystems Properties Component. Note that you can call this as many times as you'd like and that each time you call it, you don't just add a single "property page", you add an entire object⁴² and all of the property pages that it will implement, to the ultimate "set" of property pages that will be shown to the user. If some of those objects support the `IGPropertyPageClient` interface, then the system will use that interface during the processing of *those* objects, however, this makes no difference to the GSystems Properties Component; it handles any object that supports `ISpecifyPropertyPages` – a Microsoft defined interface required of objects implementing property pages.

Whenever your user is ready to see property pages, you can start the process by calling the `EditProperties()` method on the GSystems Properties Component.

```
pIGProperties -> EditProperties(  
                                hwndParent, strTitle, NULL);
```

Where the `hwndParent` argument is the window handle of what will be the parent object of the property settings dialog that will be shown. When you pass `NULL` as the third argument, you are telling the system to edit the properties of the object(s) previously added through the `AddPropertyPage()` method.

Editing properties of any object

If the call to `EditProperties()` passes a specific object (its `IUnknown` interface) in the third parameter, then the system will raise the property settings dialogs of *that* object, as long as it supports `ISpecifyPropertyPages` as discussed previously.

This means that you can present to the user any object's, including the system common controls, such as Tree-Views and List-Views, properties dialogs.

⁴² Again, *any* object that supports `ISpecifyPropertyPages`

This capability is typically only available to developers of the application or object using the embedded object. Presenting these tools to your users and, on top of that, remembering the settings the user provided across runs of your application is a truly powerful example of seamless COM integration

.

Part VI – The relationship between properties and common windows controls

A time consuming task in windows development is the manipulation of the contents of windows “controls” on user interface components. Specifically, setting the text of controls and retrieving the user’s changes in these controls and making the appropriate changes within the application.

The GSystems Properties Control assists in these tasks by supplying a rich set of methods (on both the IGProperties and the IGProperty interfaces) that “shortcut” the processing path between the data in the system and the display and manipulation of that data in the windows controls.

To use these features, you call the methods on the Property objects that you have created using the GSystems Properties Component. There are two important “directions” of action inherent in these methods; “Setting” the contents of the windows controls with the property value and “Getting” the property values from the windows controls.

These methods are available in two different places, directly from the GSystems Properties Component, as well as from each of the property objects that the component created. The difference being that you must use the name of the property when calling the methods from the main component so that it may locate the particular property you want to use. Further, the first letter of the method name implemented through the main component is capitalized, whereas with the Property object, it is not.

The remaining discussion in this section is reference oriented in nature because there is no specific “process” which would encompass the use of these techniques. Therefore, the following describes each of these methods in detail. In this material, the `pIGProperties` variable is a pointer to an IGProperties interface while the `pIGProperty` variable points to a IGProperty interface. You may have an environment in which you create objects and can reference their properties with dot syntax, in which case, you would, of course, replace the “->” operator with “.”.

IGProperty/IGProperties Window/Control Support methods

These methods on both the IGProperty and IGProperties interface are intended to assist you in moving the values of the property to and from the windows and/or controls in your system.

In particular, you are not dealing with the actual “value” of the property through these methods. You are passing a window handle to the methods and letting the updating of that window or property happen within the method. You don’t need to, for example, retrieve the value of a property, write that onto an intermediate string, and use that string to update the display in a window. All of this is handled through a single call to one of the methods described here.

These methods start with either “set” or “get”. This should not be confused with the “get_” or “put_” in the “values” methods. It may help to keep the following in mind:

- “set” means you are telling the GSystems Properties Component to “set” the contents of the window represented by the handle passed in.
- “get” means you are telling the Component to “get” the value of the property *from* the window passed in.
- “item” means for a particular control, or child, on the window passed in. The child identifier is passed as well as the parent window handle. Methods without “item” in the name mean for the window itself, which is probably a control.

For example, if you have an edit control on a dialog window and that control’s id number is 5100, when the user changes the text in that control you would call:

```
pIGProperty -> getWindowItemValue(hwndDialog,5100);
```

This means “get” the value of the property from the control whose id is 5100 in the dialog represented `hwndDialog`.

You must be aware of the difference between `hwndDialog` and the window handle of the actual control itself. Note that some event handlers that you use in generated MFC C++ code may receive the actual window handle of the specific control⁴³ that is being manipulated by the window. This is in contrast to the above call which would be passing the *parent* window of that window handle.

If you know this to be the case, simply call:

⁴³ Every control on every window has a window handle and is itself a window

```
pIGProperty -> getWindowValue(hwndControl)
```

passing the window handle of the control that contains the text you want to use to update the property's value⁴⁴. If you have debugging for the property enabled, the system will notify you when you have passed an invalid window handle to any of these methods. However, the system has no way to determine if you are passing the handle of the correct window. If you pass a window handle that belongs to an entirely different control or window, the GSystems Properties Component will faithfully set or retrieve the value from *that* window.

Note that the descriptions of the methods on the IGProperties interface (the default interface for the GSystems Properties Component) are not specifically given here. This is because these methods are exactly the same with the exception that they each have an additional argument, the name of the property, as the first argument of the method. Also, the first letter of the method name is capitalized. Thus, any description of the Property level interface method will have an associated Properties Component level method as in this example:

```
IGProperty* pIGProperty_FileName;  
.  
.  
pIGProperty_FileName-> getWindowText(hwndWindow);
```

is equivalent to:

```
pIGProperties -> GetWindowText("file name",hwndWindow);
```

where, at some point prior to the second form of the call, the GSystems Properties Component had been used to create a property with the name "file name". While, at some point prior to the first form, the variable pIGProperty_FileName was given a value, probably during the creation of the property through the Properties Component.

Here are the methods used to assist in managing the relationship between controls and properties.

```
getWindowValue(long hwndControl)  
getWindowItemValue(long hwndDialog,  
long idControl)  
setWindowValue(long hwndControl)  
setWindowItemValue(long hwndDialog,
```

⁴⁴ Note that the "type" of the property does not have to be a text (TYPE_STRING for this to work. The GSystems Properties Component will properly format the value from the text in the control if, for example, the type of the property is TYPE_DATE.

long idControl)

Note: To prevent clutter, the remaining descriptions will omit the “..Item..” entries. Simply remember that these are identical, except that the window handle argument represents a parent window, and an additional long parameter represents the ID of the specific child window of interest

As described above, these methods work with properties that have some logical representation in text⁴⁵, i.e., so they can be shown in a window control, typically an edit box. Note, however, that any control that displays text can be used with these methods. This includes edit boxes as well as static text labels. Note that the “text” of a dialog window itself is usually that which displays in the title bar, if anything. Even this text can be used as a property, of course, the user isn’t able to directly manipulate that text.

setWindowEnabled(long hwndControl)

These methods can be used to enable or disable the associated controls. In particular, if the value of the property can be represented as TRUE (has a binary representation of non-zero), then the window or control is enabled, otherwise, it is disabled.

setWindowComboBoxSelection(long hwndControl)

getWindowComboBoxSelection(long hwndControl)

setWindowComboBoxList(long hwndControl)

getWindowComboBoxList(long hwndControl)

These methods support the use of Combo Boxes (aka DropDown Lists) as controls in your windows. You must populate the combo boxes with values pertinent to their context, you can do this by calling the `setWindowValue()` or `setWindowComboBoxList()`, method on an array property with the handle of a combo box control. Once this has been done, you can use the properties to set which selection will be made and after the user has made a selection, to set the value of the appropriate property based on that selection.

Again, you “set” the state of the control, and “get” the value of the property. Therefore, when you call the `setWindowCombo...` method, as an example, the GSystems Properties Component will use the *current* value of the

⁴⁵ Array properties, however, do have a property called `arrayIndex` (put..., `get_arrayIndex`) which indicates the current “active” member of the array, which can be shown in a scalar context.

property and set the text in the combo box to that value. If you have debugging enabled (see above) *and* if the appropriate value is not in the combo box, a message to this effect will be shown.

The following methods are exactly like the “ComboBox” methods except that they work on a “ListBox” control.

```
setWindowListBoxSelection(long hwndControl)  
getWindowListBoxSelection(long hwndControl)
```

```
setWindowListBoxList(long hwndControl)  
getWindowListBoxList(long hwndControl)
```

These methods are for check boxes and radio button controls. You pass the appropriate window to the “set” methods and the current value of the property (TRUE or non-zero, or FALSE or zero) determines the state of the check box.

```
setWindowChecked(long hwndControl)  
getWindowChecked(long hwndControl)
```

The following methods are useful with “Array” properties. Here the “..Item..” version is also shown, it takes a SafeArray argument representing the IDs of the windows associated with the array of window handles.

```
getWindowArrayValues (  
    SAFEARRAY(long) * hwndControls)  
getWindowItemArrayValues (  
    SAFEARRAY(long) * hwndDialogs,  
    SAFEARRAY(long) * idControls)  
setWindowArrayValues (  
    SAFEARRAY(long) * hwndControls)  
setWindowItemArrayValues (  
    SAFEARRAY(long) * hwndDialogs,  
    SAFEARRAY(long) * idControls)
```

To clarify, these arguments are pointers to SafeArrays containing longs. The “hwnd...” arrays contain the window handles of the controls, in the case of the “..Item..” methods, they contain the window handles of the parent windows and the `idControls` argument contains the child window IDs.

These methods are intended for use with properties that have an array type. When you call these methods, you must pass an array of window handles.

Even if all of the controls are on the same window, you must pass an array of the window handles.

The point of these methods is to manipulate the contents of many controls at once given the value of a single property which represents an array.

Part VII – Interfaces reference information

When you need to communicate with the GSystems Properties Component, you use the interfaces described in this section. These interfaces are dual automation interfaces which means you can use them in one of two ways, as dictated by the needs of your software and/or environment.

You can access the GSystems Properties Component using early or late binding. In particular, you can have your development environment learn of, and use, the capabilities of the component 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 Properties Component 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 n as integer
n = property.Value
property.Value = n
```

In reality, however, there are two distinct "methods" on the IGProperty 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}_longValue({long | long *} theValue)
```

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

```
put_longValue(long theValue)
```

The other is:

```
get_longValue(long* theValue)
```

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

```
property.longValue = theValue
```

or,

```
theValue = property.longValue
```

In this case, “property” will be an instance of the Property object, which is an instance of the object class that the GSystems Properties Component maintains to form the set of properties.

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 method⁴⁶. 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 IGProperty Interface

Every property that is added to the GSystems Properties Component for management has a unique copy of this interface you can use to manipulate the property.

There are four categories of things you can do with each property, these are:

⁴⁶ With some exceptions, usually where passing an “object” would be more efficient by passing a pointer to an object instead.

1. **Miscellaneous:** There are various manipulation methods you can use for each property.
2. **Values:** You set or retrieve the value of the property using these methods. Generally, the name of the method indicates the “type” of the information set or retrieved. It is generally safe to mix these methods. For example, if you have a “double” property, you can call `get_szValue()` for that property to retrieve its text representation. It is recommended you call `put_debuggingEnabled(true)` during development so that you will be notified when a situation occurs where this is not generally advisable.
3. **Window/Control Support:** These methods are used to exchange the value in the property with the text and/or value in a common window control. For example, you quickly update the value of a property by passing the handle of a window to one of these methods that will then use the contents of that window (really just a control) as the value of the property. In a likewise manner, you can have the value of the property update the contents of the window. Using these methods, you need make only one call to the `IGProperty` interface in your window or dialog procedure rather than using intermediate code to get the value and then update the controls in your window.
4. **The persistence of external objects:** This includes the ability to manipulate the entire property settings of an array of external objects within a single `GSystems Properties` property. Thus, an arbitrarily complex collection of external objects can be saved and restored as a single property within the system.

The methods on this interface begin with a lower case letter. This is purposefully done to distinguish them from methods in the `IGProperties` interface⁴⁷. Since the latter generally deals things at a higher level, while this interface is dealing with single items, i.e., each property.

⁴⁷ Note that certain MFC wizards and perhaps other tools get confused between the lower case letter on the `IGProperty` interface and an exactly named, except with an upper case initial letter, method on the `IGProperties` interface. Sometimes, these tools will “think” the actual name of the method on the latter interface has the lower case initial letter, probably because of the close proximity in definition of the two interfaces. There is no effect of this bug except that you may find it necessary to “go along” with the incorrect assessment by the development tool.

Miscellaneous IGPProperty methods

get_binaryData (BYTE pBinaryData) ;**

This method is primarily intended for internal use by the GSystems Properties Component. However, if you want to retrieve and store the internal binary representation of the property for some reason, you may use this method. You might do this, for example, to quickly compare the value of a complex property, such as an array property at different points in time. Note that the pointer returned is a pointer to memory space within the GSystems Properties Component, if you want to hold on to this binary representation, you should allocate memory in your own memory space (note the `get_size()` method will tell you how much you need⁴⁸) and copy the data returned from this method into it.

copyTo (IGProperty* pIGProperty_destination)

You can copy the value of one property to another by providing the IGPProperty interface to the “destination” property. The source property is the property that has provided the IGPProperty interface that you use to call this method.

**{put | get}_debuggingEnabled(
 {BOOL | BOOL*} setEnabled)**

During your early stages in development with the GSystems Properties Component, it pays to turn debugging on so that the Component can notify you, by displaying a message box, if you make an error in your use of the Component. For example, it is easy to pass an invalid window handle to the window support methods and you want to be notified about this during your development work so that you can correct the problem. The system will return `S_OK` in most cases if debugging is turned on (after displaying it’s information) so that you’re software under test may continue. However, with debugging off these methods generally return `E_FAIL` which will cause a Visual Basic application to terminate.

**directAccess (void* nativeStorage,
 long sizeofStorage)**

⁴⁸ The size information is dynamic, i.e., do not hold onto a size value returned and use it later, call `get_size()`, allocate the storage, call `get_binaryData` and then copy the data into your allocated memory, all in sequence.

This method allows you to provide the property to access memory within your object or application.⁴⁹ In C or C++ environments, you can pass the address of allocated storage (or a member variable) so that the property can use that storage directly as the location of the property's value. This is specifically used so that you won't have to make the value setting and retrieving calls (described later) to keep the property's value up to date. In particular, each time you set the value of a variable in your system, if that variable's address had also been passed to the associated property through this method, then that property is always up to date. Likewise, you can set the property's value and the variable is automatically updated.

On the other hand, in environments where this kind of memory access is not supported, you can use the properties generated by the GSystems Properties Component directly as if they were variables. Providing the same level of convenience as this method provides.

```
{put | get}_encodedText(  
    { BSTR | BSTR*} encodedText)
```

You can retrieve a non-binary representation of the property using the above method. An "encoded" text string is a hexadecimal representation of the entire property that can be saved by your application such that you can recreate the value of the property at any point in the future. This would be done by passing that encoded text string back in `put_encodedText()`. Of course, persistence of the properties is the job of the GSystems Properties Component so you don't *need* to do this, however, you might have a side reason for it, such as, for example, storing the property in a "text" database field.

```
{put | get}_isDirty({ BOOL | BOOL*} isDirty)
```

You can ask the property if it's value has changed since it was created, since you last called this method, or since it was last written to a storage medium.

```
{put | get}_name({ BSTR | BSTR *} name)
```

Sets the "name" of the property. Generally, when the property is created through the `Add()` method on the `IGProperties` interface, this name would have been specified there. There may be times, however, that you want to retrieve the properties' name. A `BSTR` represents the OLE safe string type. This is the same as the `String` type in Visual Basic.

⁴⁹ This is optional. Please see the discussion of the issues involved on page 17.

get_size(long* pSize)

To determine how much storage the property uses. Note that this method can be used on the right side of an '=' in environments that support this. In VB:

```
dim s as integer
s = property.size
```

However, there is no way to use this method as if you were assigning a value to it, which is true of any method that does not have an associated "put_...". This method would typically have no use outside of the internals of the GSystems Properties Component, however, access to the property's binary data is available if some client wants it, therefore, this method's usage would typically couple with the `get_binaryData()` method.

{put | get}_type (**{ PropertyType | PropertyType *}type)**

The property type determines the underlying data that is stored with the property, i.e., it is roughly equivalent to variable types in many environments. Typically, a property will "learn" of its type whenever a value is assigned to that property for the first time, i.e., it is not usually necessary to specify the type. However, there are certain times when you must explicitly set the type, for example, when the property is a `TYPE_OBJECT_STORAGE_ARRAY`, it is necessary to call `put_type()` to indicate this to the system. At other times, you will probably call the `get_type()` method to determine the type of the property if you needed to know this for some reason.

get_variantType(VARTYPE *vtype)

When you set the value of the property using the `put_value()` method, your development environment will "assemble" some `VARIANT` to pass as the argument to that method. If you'd like to see the actual type of the passed `VARIANT`, use this method. As an example, if you pass a literal integer from Visual Basic, VB will, in turn, create either a `VT_I2`, or a `VT_I4` variable for the property, this method allows you to see what VB did in this case.

IGProperty value methods

These methods are used to set and retrieve the "values" of the property. They are covered in alphabetical order.

{put | get}_arrayValue (**{ SAFEARRAY(VARIANT) | SAFEARRAY(VARIANT)* }** **theArray)**

These methods use SAFEARRAY's to maintain the value of the property. Used in conjunction with list and combo-boxes, this type of property can be very useful. Note that the SAFEARRAY passed in is copied into the property and that it may be an array of any valid VARIANT type, including other SAFEARRAY's. However, VT_UNKNOWN and VT_DISPATCH are not useful in this context.⁵⁰

Four methods pertinent to array properties are described here as they are relevant only to array properties.

{put | get}_arrayIndex (
 { long | long* } theIndex)

Sets which element of the array is “current”. This “property of the property” is saved and restored along with the array property. This way, not only are the contents of a list box saved, so is its selection.

{put | get}_arrayElement (
 long elementIndex,
 { VARIANT* | VARIANT ** } theValue)

Adds, replaces, or removes one of the members in the array. Passing a VARIANT value (or retrieving a VARIANT) and one of the values from the following table:

ARRAY_INDEX_REMOVE	-4	Take the element out of the array (the element passed is the index of the element)
ARRAY_INDEX_ADD	-3	Add the element to the end of the array
ARRAY_INDEX_FIRST	-2	Replace the first element or add if no elements
ARRAY_INDEX_LAST	-1	Replace the last element, or add if no elements
Any other value		Replace the element at that position, if the value supplied is within the range.

get_arrayElementCount (
 long *elementCount)

Lets you know how many elements are in the array.

ClearArray ()

⁵⁰ Again, the TYPE_OBJECT_STORAGE_ARRAY may be useful for maintaining other objects.

Removes all elements from the array.

```
{put | get}_binaryValue(long countBytes,  
                        { BYTE* | BYTE ** } theValue)
```

You can store any type of binary data in a property. The GSystems Properties Component makes no attempt to validate the data, it only requires that the property be a TYPE_BINARY property if you use this method. Using this type of property, you can save any structure or object as long as you can get its size and address⁵¹ to supply to this method. The property's data is ultimately written to a system supplied storage mechanism and there is no restriction on what type of data that may be. However, note that if your binary data contains pointers to other data, you must have some mechanism for recovering that "other" data after your property has been "restored" from a storage. In particular, the pointers will not be valid after your object (containing the property) is restored. Perhaps you have "other" properties utilized to account for this situation, or, you can use the TYPE_OBJECT_STORAGE_ARRAY property if applicable.

Note that the TYPE_BINARY property is very convenient for structures defined either by windows or by your application or object. For example, the SIZE structure defined in windows can be passed to a property of this type so that the size of the object's main window can be stored with the properties. In a certain sense, the binary property is a "catch all" for other types of data not represented in the other types. The term, however, should not be considered to denigrate the value of the property type as it is quite useful for any type of data. Understanding and using the `directAccess()` method in conjunction with this type of property is very helpful.

```
{put | get}_boolValue(  
                        {BOOL | BOOL *} boolValue)
```

To set or retrieve the value of a property that contains TRUE or FALSE.

```
{put | get}_doubleValue(  
                        {double | double* } doubleValue)
```

This method sets or retrieves the double value of the property.

⁵¹ In this case, the address is not used by the GSystems Properties Component beyond simply copying this binary data from that location at the time this method is called. In other words, the Properties Component allocates its own internal storage to hold this binary data and you need not be concerned with the issues involved with the `directAccess()` method.

```
{put | get}_longValue(  
    {long | long *} longValue)
```

Again, these are the same as the associated “double” properties. Here, the stored values are long integers (32 bits).

```
{put | get}_stringValue(  
    {BSTR | BSTR* } bstrValue)
```

These methods manipulate the property when it is an OLE compatible BSTR. These are a special representation of character data that is safe for use across disparate environments, for example Visual Basic and a C/C++ object.

```
{put | get}_szValue(char *szString)
```

To set or retrieve a C style null terminated “string” property, use the above methods. In the GSystems Properties Component, these strings are an array of ascii characters, the last one of which is a value of ‘0’. In other environments that use OLE strings (BSTR strings) use the `put(get)_stringValue()` methods instead.

Note the `get_...` and `put_...` methods both pass a pointer to char. The caller must pass a character array that has enough space to receive the string when using the “`get_...`” method. Use the `get_size()` method if you need to know the size of this string.

The window relationship methods

These methods are the subject of Part VI of this manual. Please refer to page 71 for more information.

The IGProperties Interface

This interface is the primary interface used for managing the *group* of properties as a whole. However, many of the methods can be used to “get at” particular properties. For example, by using the name of the property that was used when the property was created. This is intended as an easy way to access one of the properties in the set without having to maintain a pointer to each property’s interface.

The interface contains groupings of methods:

1. **Miscellaneous:** These methods are used for general control of the GSystems Properties Component.

2. **File handling:** Support, typically for application rather than object development, for file handling, such as saving to and restoring from an application defined file.
3. **Property management:** Creation and manipulation of the “set” of properties that are used by the client.
4. **Persistence support:** These methods are used to achieve the persistence, saving and loading, of the properties. The general overview of the use of these properties starts on page 48.
5. **Property Page Support:** These methods support the process of providing a user interface which can be used to edit properties. There are 2 levels of functionality that you can use with these methods. The easier way involves a single function call while the more complex way provides much more control over the entire process. The general discussion of this functionality starts on page 55.
6. **Window/Control support:** This group of methods is the same as that discussed in the description of the individual property interface; `IGProperty`. The difference here is that an additional first parameter is provided in these methods that simply names the particular property of interest, so that the master `Properties` Component can locate and direct the call to the specific property requested.

Miscellaneous methods in `IGProperties`

These methods provide general control over the `GSystems Properties` Component. Where the `IGProperty` method names start with a lower case letter, these methods (with the exception of the `get_` and `put_` methods) start with an upper case name. This convention is intended to help differentiate the methods where the functionality is very similar. In particular, methods that are like named in these two interfaces will deal with a *particular* property in the `IGProperty` interface, where, in the `IGProperties` interface, the method's first argument is the name of the property and that method will internally find and reference the appropriate property.

Here are the miscellaneous methods on this interface.

`CopyTo (IGProperties* pIGProperties_destination)`

This method is used to copy the entire set of properties (and their values) into another instance of the `GSystems Properties` Component. You can load

the properties of one object into those of another object, typically of the same general “type” of object. Only like named properties are copied using the target object’s IGProperties interface. In other words, each property in the source (defined as the object which owns the IGProperties interface that this method is called on) is copied into the destination only if a property with the same name exists in the destination.

get_Count(long* theCount)

This method returns the number of properties currently being maintained by the Component.

**{put | get}_DebuggingEnabled(
 {BOOL | BOOL *} setEnabled)**

You can enable the display of informative messages during the early stages of your work with the GSystems Properties Component. This assists in working out minor errors such as the misspelling of property names or the passing of an invalid window handle. When you set this value, all of the existing properties are affected, as well as all properties created by the component subsequent to this call. If you want to turn off the debugging capability of just one property, you can call put_debuggingEnabled(FALSE) on that property’s IGProperty interface.

**DirectAccess(BSTR propertyName,
 void* accessibleAddress,
 long sizeofMemory)**

This method allows you to specify that a property use the address of a particular variable or allocated space in your object or application. Please see the equivalent description of this method on page 80.

**{get | put}_FileAllowedExtensions(
 {BSTR | BSTR* }fileExtensions)**

This property dictates what will appear in the standard system File Open (or Save) dialog box in the drop-down list that indicates the allowed extensions of the file to open.

{get | put}_FileName({BSTR | BSTR* }fileName)

If the system were asked to store the properties at any given time, this property will specify what the name of that storage file will be. If this value is not specified when an application calls Save(), the system will raise a file save common dialog.

get | put}_FileType({BSTR | BSTR* }fileType)

In conjunction with `FileAllowedExtensions`, indicates text that will appear in the System File Save-As or File Open dialog boxes.

get | put}_FileType({BSTR | BSTR* }fileType)

Also in the system File Open, Save, dialogs, the text that appears in the title-bar of those dialogs.

**GetPropertyInterfaces(
 long* countInterfaces,
 IGProperty*** theInterfaces)**

This method allows you to retrieve the interfaces to all of the properties in the set of properties currently maintained by the GSystems Properties Component. Note that you pass the address of a long integer that will, upon return, contain the number of actual properties being maintained. Also note that the “triple” pointer to the returned interfaces may be thought of as a returned “array” of pointers to these interfaces. The GSystems Properties Component allocates all of the necessary space for this array. It is the caller’s responsibility to call `CoTaskMemFree()` on this returned array in order to free the memory.

Here is example code calling this interface:

```
long cntOthers;  
IGProperty** ppIGProperty_Array;  
pIGProperties -> GetPropertyInterfaces(  
                                    &cntOthers,  
                                    &ppIGProperty_Array);  
for ( int k = 0; k < cntOthers; k++ )  
{  
    IGProperty* pDestination = ppIGProperty_Array[k];  
    pDestination -> . . .  
    .  
}  
CoTaskMemFree(ppIGProperty_Array);
```

SetClassID(BYTE* pCLSID)

Often times a container will need a unique identifier for your object. The GSystems Properties Component will handle the call to supply this value. However, the Component needs to have an appropriate value to provide the container. This method is provided to allow you, the user of the GSystems Properties Component, the opportunity to provide this information to the GSystems Properties Component.

In particular, you should call this method soon after you have created the Properties Component or after your application starts if you have statically included the Component in your application.

If you are building an object, you should pass a pointer to the CLSID of your object to this method. For controls based on the MFC class COleControl, you can call the `GetClassID()` method to retrieve this value. If you don't have access to a CLSID⁵² identifier for your application or object, you can make one with the "guidgen" utility.

Note: This method does not need to be called *if* you supply the IGPropertiesClient interface as described on page 48. This is because that interface has a `GetClassID()` method that the GSystems Properties Component can call when and if it needs this information.

File handling methods on IGProperties

New ()

This method will initialize the set of properties managed by the GSystems Properties Component. This will overwrite the current state of all of the properties. If debugging is turned on and the current set of properties are "dirty", i.e., need saved, a warning message to this effect is displayed. You should call `Save ()` before `New ()` if your intent is not to overwrite the properties.

LoadFile (BOOL* pWasSuccessful)

This method will cause the storage saved in the file whose name is the `FileName` properties of the GSystems Properties Component. This method returns FALSE if the file could not be opened, TRUE otherwise.

Open (BSTR* pStrFileName)

This method prompts the user for a file to open. Note that the properties on the GSystems Properties Component specify the details for the Open File dialog shown to the user, for example, the allowed extensions for the files. The method returns the name of the file selected by the user. If the user cancels or the file cannot be opened, the system returns an empty string.

OpenFile (BSTR strFileName)

⁵² This is really just a 128bit magic number that provides a unique identifier for any given object or item that needs one.

This method opens the existing file with the name given, at the same time, it sets the `fileName` property to reflect the “current” active file.

Note that this method will also cause the current set of properties to be overwritten. Again, with debugging enabled, a message to the effect that unsaved properties are overwritten will be issued. And, the method will overwrite the properties.

Save ()

Causes the current set of properties to be saved in the “current” file. Note that the GSystems Properties Component the `FileName` property to indicate the name of the current file. Remember that calling `OpenFile(strFileName)` will also reset the value of this property to reflect the file named in that call.

SaveAs (BSTR* pStrFileName)

This method will prompt the user for a file name and then save the current set of properties to that file. As with the `Open ()` method, the system returns the file name selected.

SaveTo (BSTR strFileName)

The final method in this set saves the properties in the file whose name is provided. This method also sets the “current” active file by setting the `fileName` property of the GSystems Properties Component.

Property management methods on IGProperites

Add (BSTR name, [optional] IGProperty ppIGProperty)**

This method adds a new property into the collection of properties that the GSystems Properties Component manages. The `BSTR` argument is the standard OLE/COM string argument. The second argument is a pointer to a pointer to an `IGProperty` interface. If this parameter is supplied (i.e., non-NULL), the method will return a pointer to the interface on the property that is created.

Include (IGProperty* pIGProperty)

Used to “Add” a property created either directly through COM or from some other source, for example, an additional GSystems Properties Component. The receiving Properties Component will `AddRef()` the interface passed and will `Release()` it when the Properties Component is completely released.

Property (BSTR propertyname, IGProperty pProp)**

This method is used to return a property which has the given property name. Before returning the IGProperty interface, if found, the system `AddRef()` s it, therefore, you need to `Release()` it when you are done with it.

Remove (BSTR propertyname)

Simply removes the property from the set of properties maintained by the system.

Persistence methods in IGProperties

These methods are used to allow your object or application to participate in the “process” of saving and retrieving data to or from a storage.

More specifically, these methods are generally used during execution of methods in the IGPropertiesClient interface (discussed below) that you write for the purpose of integrating into the persistence process. This process is discussed in general terms starting on page 48.

Advise (IGPropertiesClient *pIGPropertiesClient)

This method provides the Component with an implementation of the IGPropertiesClient interface that it can use to communicate with your object or application⁵³. Note that this communication is for the purposes of the persistence processes as described on page 48, not for the property pages implementation. When you use the property pages features of the GSystems Properties Component, the Component will ask your software for an implementation of IGPropertiesClient that it can use, you don’t need to call `Advise()` for that interface.

PutHWNDPersistence (hwndPersistence, hwndInit, hwndLoad,

⁵³ Remember that your software implements the IGPropertiesClient interface.

hwndSavePrep) ;

As described on page 44, this method is used to provide simple access to your code in the persistence process. You supply the window handles of command button controls that are implemented in the passed parent window handle. At certain points in the persistence process, the GSystems Properties Component will send the window the appropriate command message, which will cause the command handlers to get called by the system.

You may call this method for each window that you want to be notified of the actions in the persistence process. If you do not implement one of the commands shown, pass zero for the handle of that command window.

IsDirty()

This method is called to ask your object if the properties are dirty (need to be saved). Return S_OK for TRUE (the object needs saving) or S_FALSE for FALSE. This is in response to the system asking this question of the GSystems Properties Component. The process of saving data is not guaranteed just because TRUE is returned; that decision is up to the client or container that initiated the request.

Property page methods in IGProperties

These methods are used to assist your object or application in managing the windows displayed to the user for the purpose of “editing” the properties.

As previously mentioned, there are two ways to accomplish property page support with the GSystems Properties Component. Methods that do not depend on which of the two ways you select to implement property pages are described first.

Initiate the edit properties process

**EditProperties(HWND hwndOwner,
BSTR tabText,
IUnknown* forObject)**

This method is used to initiate the edit properties process. Typically it is called when your application, or the client of your object, requests the OLEIVERB_PROPERTIES verb be performed through the IOleObject interface. You can call this method at any time to cause the GSystems Properties Component to create and display the property pages on behalf of your object (or application). Note that it doesn’t matter whether you have

implemented the easier method, or the more complex method of property pages integration. This method will initiate the process for whatever method you have chosen.

In fact, if you had chosen to keep the MFC, or any other, implementation of property pages, this method will still bring up those property pages.

Note also that you pass the IUnknown interface pointer for the object whose properties you want to edit. This can be any object on the users' system whatsoever. In other words, it does not matter how or who implemented the property pages for a given object; as long as that object has been created and you have a pointer to an IUnknown interface on the object, you can get the edit properties dialogs to display for the property⁵⁴.

Property Stack and Comparison Methods

These methods are used to manipulate the “stack” of properties. The GSystems Properties Component allows you to “push” and “pop” the entire property set and to compare the “current” set of properties with the set of properties on the top of the stack (last pushed). This is most convenient for determining the extent to which properties have changed during, for example, the properties editing process.

Push ()

This method is used to place the entire set of properties on the “stack”. This is a method by which you can save the entire property set for later restoration. Each time you `Push ()` the properties on to the stack, a copy of every property is saved such that you can restore the set of properties with a simple `Pop ()` of the stack.

The number of times you can `Push ()` the stack is limited only by available memory.

Pop ()

When you `Pop ()` the stack, you restore all of the properties to the state they had when you last `Push ()`-ed the stack. Note that you can't `Pop ()` more times than you call `Push ()`. If you have debugging enabled,

⁵⁴ This does require that the object implement the `ISpecifyPropertyPages` interface, which is a standard COM interface expected to be implemented by all objects, regardless of author or platform, that want to have the capability of showing property pages.

attempting extra `Pop()`s will generate a message. The action is otherwise benign.

Discard()

Similar to `Pop()` except the properties do not revert to the values from when they were last `Push()`-ed. For example, if the user accepts the current state of the properties, you can go ahead and discard the top of the stack as it won't be needed for restoration.

Compare()

Does a binary “compare” between the current set of properties and those on the top of the stack (last `Push()`-ed). Returns `S_OK` if all of the properties are identical, or `S_FALSE` otherwise. You can use this method to determine if any of the current set of properties have changed since they were last `Push()`-ed onto the stack.

Simple Property Page methods

```
PutHWNDPropertyPage(BSTR displayName,  
                    HWND hwndProperties,  
                    HWND hwndStart,  
                    HWND hwndOK,  
                    HWND hwndApply,  
                    HWND hwndCancel);  
RemoveHWNDPropertyPage(HWND hwndProperties);
```

The mechanics of the simpler property page implementation method are discussed in detail on page 58. In summary, the display name is the text displayed on the tab for this particular window. The `hwndProperties` argument is the handle to the dialog or window that will be shown. And the four remaining `HWND` values are the control window handles of command buttons that are handled by the window procedure for the given (parent) window handle.

The `RemoveHWNDPropertyPage()` method is used to take the given window back out of the list of windows that will be shown when property pages are next shown. Note that the list of property pages shown can change dynamically, each page showing a different subset of the entire set of properties, for example.

More Extensive Property Page Support Methods

**AdvisePropertyPageClient(
IGPropertyPageClient* pIGPropertyPageClient)**

This method provides the Properties Component with access to your application or object through the interface you have implemented. For complete details of the IGPropertiesClient interface, please see the next section. Note that this call will AddRef () the passed in interface.

To clear the list of property pages implemented, call this method with a NULL value. That indicates to the system that it should Release() all of the pointers to either ISpecifyPropertyPages or IGPropertyPageClient that it is holding. Once this is done, you can add a different set of objects into the list of objects providing property pages.

Note also that it is necessary for the passed in interface to implement IGPropertyPageClient, however, that implementation may return 0 property pages when asked through the interface's GetPropertyPagesInfo () method. This is useful if, for example, your object doesn't really implement properties or property pages, however, it manages one or more embedded objects that do have property pages.

AddPropertyPage(IUnknown* pIUnknownObject)

This method notifies the GSystems Properties Component that you are using the more complex and feature rich process in order to edit properties.

The IUnknown interface pointer should be a valid IUnknown interface to any object that implements *either* the IGPropertiesClient interface, or the ISpecifyPropertyPages⁵⁵ interface. It is noteworthy that this can be virtually *any* object that implements one of these interfaces (or both). It does not necessarily have to be your particular object which owns the IGProperties interface through which you have called this method.

Note that this method will attempt to get one of the two interfaces from the object, starting with the IGPropertyPageClient interface, and that doing so successfully will cause 1 AddRef () on the object.

**ConnectPropertyNotifySink(
IPropertyNotifySink *pINotify,
DWORD dwCookie)**

⁵⁵ A Microsoft defined interface that any object that intends to offer property pages must implement.

This method notifies the GSystems Properties Component that your container would like to be notified when a property changes. If you implement or override an instance of the IOleObject, you may get passed a valid pointer to an IPropertyNotifySink⁵⁶ interface. If you pass this interface on to the IGProperties interface, the latter component will notify your container (the implementor of the notify sink interface) whenever the user has edited properties through the property pages dialogs. This technique doesn't necessarily require your object to implement connection points (event interfaces), but for you to at least have enough control over the implementation to be able to hook or implement the following method on the IConnectionPointContainer at the appropriate time.

```
FindConnectionPoint(  
    REFIID riid IConnectionPoint **);
```

If you do pass on a container's IPropertyNotifySink to the GSystems Properties Component, as in the previous method description, you also need to pass a request of your object to make the connection on to the GSystems Properties Component as well.

In particular, if you support connection points, you implement IConnectionPointContainer. On this interface, you may receive a call to connect (through the FindConnectionPoint() method) to an IPropertyNotifySink interface. When you receive this call, simply delegate the call to the GSystems Properties Component using the above method.

Here is an implementation of this technique by an object that supports a connection to an event interface.

```
STDMETHODIMP Function::FindConnectionPoint(  
    REFIID riid,IConnectionPoint **ppCP)  
{  
    *ppCP = NULL;  
  
    if ( riid == DIID_IGSFunctionEvents )  
        return connectionPoint.QueryInterface(  
            IID_IConnectionPoint, (void **)ppCP);  
    /*  
    The connection is for some other interface, delegate  
    To the properties control.  
    */  
    return iProperties -> FindConnectionPoint(riid,ppCP);  
}
```

⁵⁶ A Microsoft defined interface that lets a container know (because it is implemented by a container) when a contained object's properties have changed.

In the example, if the requested identifier of the sink interface is not one the method knows about, then it just passes the request on to the GSystems Properties Component, through a cached IGProperties interface represented by the variable iProperties;

The window relationship methods

These methods are the subject of Part VI of this manual. Please refer to page 71 for more information

It bears repeating, however, that the methods on this interface do have the additional (first) argument which specifies the name of the property. The system will simply forward the appropriate call to the IGProperty interface of the property with the given name, if found.

The IGPropertiesClient Interface

This interface represents the points in your object or application that will be participating in the persistence process that the GSystems Properties Component manages on your behalf.

Don't forget, this is an interface that you write. The GSystems Properties Component simply specifies what the methods should be, not how, why, or if they are ultimately implemented.

The "Client" in this interface is the GSystems Properties Component itself. If you have supplied the capability, the Properties Component will be calling methods on this interface.

Persistence IGPropertiesClient Methods

Again, these methods are used by the Properties Component in support of the process of saving and restoring data from the system storage.

These methods are discussed in detail in Part IV starting on page 48. The methods are repeated here in brief form for clarity.

SavePrep ()

This method will be called just before writing the properties to the storage. This gives your object the opportunity to update the values of its properties or to perform any preparatory processing.

InitNew ()

When the Properties Component is notified by the container or client (of your object) that a new instance has been created, it will in turn notify your code by calling this method. Generally, this is a good time to set all of your properties to their default values. Once this has been done, you can call the `Loaded()` method to update the system windows/controls with the new values of the properties.

Loaded()

This method is called after the GSystems Properties has read all of the properties (that belong to the object which instantiated the Properties Component) from the binary storage. At this point, your properties have the same state as the last time the user saved your object. You can take this opportunity to re-paint your object to reflect the new state of these properties.

GetClassID (BYTE *)

When the container or client needs to have the GUID of the object being saved, the GSystems Properties Component will call this method so that your code can supply this value.

For a thorough discussion of these issues, see the description of the `SetClassID()` method in the section starting on page 87.

The IGPropertyPageClient interface

This interface consists of methods (again, that you write) which are involved in the processes that occur when the user edits properties using property pages. These methods were discussed in detail starting on page 61. The material here will be more summary information.

AfterAllPropertyPages (BOOL userCanceled)

This method is called when the user has finished the properties settings session. Specifically, it is called before the dialog containing all of the property pages closes. It is called only once regardless of the number of property pages used. The `userCanceled` argument will be `TRUE` if the user pressed the “Cancel” key to close the dialog, `FALSE` otherwise.

Apply()

This method is called when the user presses the “Apply” button on the property settings dialog.

BeforeAllPropertyPages()

This method is called once before all of the property pages are shown. You can take advantage of knowing this method is called only once per

properties setting session. For example, by `Push()` ing all of the properties as discussed previously.

```
CreatePropertyPage (  
    long pageNumber,  
    HWND hwndParent,  
    RECT rectSize,  
    BOOL isModal,  
    HWND * pHwndPropertyPage)
```

When the system is ready for you to create your property page (the one given the appropriate #, see the `GetPropertyPagesInfo()` method), it will call this method. Note that it will pass the window handle of what will become the parent of your property page window, the recommended size, and whether it is a modal window, as in a modal dialog, for example. You use whatever mechanism you have to create this window, using the passed window handle as it's parent. Alternatively, you can simply "set" the parent window handle of an already created window.

```
DestroyPropertyPage (long pageNumber)
```

The opposite of the above method. Simply notifies you that the system no longer needs the window. Note that if you simply set the parent of a window in the previous method, it is important to set it back in this method. This is because that parent will most likely be destroyed soon after this call has been made.

```
GetPropertyPagesInfo (  
    long* cntPages,  
    SAFEARRAY (BSTR) * pageNames,  
    SAFEARRAY (BSTR) * helpDirs,  
    SAFEARRAY (long) * pageSizes)
```

When the system needs to know how many pages you are going to supply, it will call this method. You return the number of pages in the `cntPages` parameter and you Re-Dimension (if you have more than 1) the `SAFEARRAY`s and place the required information in each of these.

A detailed example of this method is shown on page 62.

```
Help (BSTR bstrHelpFile)
```

If, in your implementation of `GetPropertyPagesInfo()`, you supplied a help file name for any of the property pages, this method is called when the user click's Help while that page is active in the dialog. If you return `S_FALSE` from this method, the GSystems Properties Component will attempt to launch the help file using the old-style WinHelp. If you want handle the display of help yourself, return `S_OK`.

IsPageDirty(long pageNumber,BOOL *isDirty)

From time to time, the Properties Component will ask your object if the properties represented by a particular window have changed. The Properties Component uses this information to know whether the Apply button should be enabled. You should set the passed parameter accordingly.

Use the `Compare()` method to determine if properties have changed.

Epilogue

First, allow us to thank you for purchasing the GSystem Properties Component 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 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