ProvideX

Version 5.10

Automation in ProvideX

Automation Basics

Properties and Methods

ProvideX Extensions to the COM Interop

Error Handling

Events

Usage and Examples

Comparisons with Visual Basic

Advanced Topics

PVXOCX32 Misnomer

COM vs Flat API Function Calls

best

Publication Release: V5.10 6/19/03

Automation in ProvideX

Automation Basics

Automation (formerly known as *OLE Automation*) is a feature of the *Component Object Model* (COM), an industry-standard technology used by applications to ***expose*** objects, methods, properties, and events to development tools, macro languages, and other applications. For example, a spreadsheet application might expose a worksheet, chart, cell, or range of cells, each as a different type of object; and a word processor might expose objects such as applications, documents, paragraphs, bookmarks, or sentences.

When an application or library supports Automation, the objects exposed by the application can be accessed through ProvideX. These objects can be manipulated using ProvideX to invoke their methods and get or set their properties.

In order to better understand Automation, it is necessary to understand some basic concepts and terminology:

Object	Any item that can be programmed, manipulated or controlled. Interfacing with an object is done through property setting and getting, and calling of methods.
Property	A property is a characteristic of an object (an adjective). For example, properties of a `Textbox` object might include: `Name`, `Visible`, `Forecolor` etc.
Method	A function that performs an action on an object (a verb). For example, an `Application` object might expose a `Close` method.
Control	An object that exposes a user interface. Controls are now typically based on ActiveX technology, vs. the older OLE Control technology (OCX).
Late Bound	Obtaining a reference to an object without any prior information about the object. Property and method names are resolved at run time. This is the binding style used by ProvideX.

To program against an object, a reference to that object must first be obtained. This process is commonly referred to as ***binding***. Unlike other languages, ProvideX simplifies this process by providing one statement that can be used to create new objects, reference running objects, and connect to remote objects.

Note: The file associated with the ProvideX COM interface, `pvxocx32.dll`, is referred to as "the DLL" throughout this document. Refer to **PVXOCX32 Misnomer**, *p.31*, for further explanation.

Referencing an Object

The **DEF OBJECT** directive is used to create a new instance of a specified object.

DEF OBJECT *obj_ID*,[@(*col*,*ln*,*wth*,*ht*)]{, |=} *obj_name*$[;**LICENSE**=*key*][,**ERR**=*stmtref*]

Where:

obj_ID	Numeric variable that will be used to save the object reference.
@(*col*,*ln*, *wth*,*ht***)**	Optional left, top, width, and height values to be applied against the object. If the object is a control, then the control will be placed at coordinates left, top (ProvideX based) with the size specified by width and height. One underlying side effect of this argument is that it determines which ProvideX window will "own" the object.
obj_name$	String expression identifying the object to be referenced, as well as any object specific parameters. See *Object Name Contents*.
LICENSE=*key*	Optional license key that should be applied when attempting to bind to an object. See *Licensed Controls*/*Objects* for details.
ERR=*stmtref*	Optional program line number or statement label to which to transfer control in case of error. *Note:* If an error occurs during the **DEF OBJECT** statement, the error code will always equal 12. Use the **MSG(-1)** function to determine the exact reason for the failure.

Object Name Contents

The **DEF OBJECT** *obj_name$* string may contain one of the following:

*	Asterisk displays a pop-up window listing all 32 bit OLE and ActiveX controls installed on the system
CLSID	Class identifier GUID for the object in the format {*hhhhhhhh-hhhh-hhhh-hhhhhhhhhhhh*}, where the *h* indicates a hexadecimal character.
progID	Programmatic identifier name for the object. An example of this is `Word.Document`.

[FILE]*x:\filename*	Keyword indicates that the object should be created using the specified file name. An example of this would be a Microsoft Word document file.
[DCOM]*server;name*	Keyword indicates that the object being referenced is located on a remote system. *server* parameter is optional, and can be specified either by name, or by IP address. If not supplied then the object is considered local. *name* parameter is the *CLSID* or *progID* for the object.
[GLOBAL]*name*	Keyword indicates that a reference to an object exposed by the use of `PvxMakeGlobal` should be obtained. The *name* parameter is the name that was used to expose the object. The `PvxMakeGlobal` is described in detail later in this document.
[REGISTER]*x:\filename;name*	This syntax is used to ensure that the object information is properly registered before attempting to create an instance of the object. *x:\filename* parameter is the name of the executable file or library that exposes the automation object. *name* parameter is the *CLSID* or *progID* for the object.
[RUNNING]*name*	Keyword indicates that ProvideX should bind to a running instance of the named object, where the *name* parameter is given as *CLSID* or *progID*. An error will occur if the object is not currently running.
[RUNNING OR NEW]*name*	Keyword indicates the same functionality as **[RUNNING]** syntax, with one difference: if the object is not currently running, ProvideX attempts to create a new instance of the named object.

Licensed Controls/Objects

Redistribution of a third party COM control can sometimes involve the use of a ***license file*** (usually identified by a `.LIC` extension). The license file usually permits developer-level access to the control and is not for redistribution. In some cases, a ***license key*** must be extracted from the license file in order for the control to function in run-time mode.

The following steps outline how to extract the license key from the license file, and how to make it available to the control in a run-time environment:

1. On the system where the automation object and license file have been installed, obtain a reference to the object without specifying the license information.

2. Query the `PvxLicense$` property of the object for the license key. If the object is licensed, the key data is returned as a string of hex characters.

3. Append the **LICENSE**=*key* data to the parameter expression of the **DEF OBJECT** statement.

Once the new **DEF OBJECT** statement has been generated, the object reference can then be obtained on systems that do not have the license file installed.

Examples

Upon successful execution of the **DEF OBJECT** statement, ProvideX will place the object reference into the supplied numeric variable. Some examples of the **DEF OBJECT** statement include the following:

```
DEF OBJECT X, "*"
DEF OBJECT X, "Word.Application", ERR=*NEXT
DEF OBJECT X, @(1,1, 70, 20)="Word.Document"
DEF OBJECT X, "[dcom]MyServer;Shell.Explorer"
DEF OBJECT X, @(10, 2, 20, 10)="[file]c:\my documents\test.doc"
DEF OBJECT X, "VCF1.VCF1Ctrl.1;License=80412079727687420286696319 67"
DEF OBJECT X, "[running or new]Excel.Application"
```

The **DEF OBJECT** statement can also be used to bind child objects, which are returned as the result of either a property access or method call. The syntax for this binding is:

```
DEF OBJECT X
```

Releasing an Object Reference

In order to manage the lifetime of an object, the ProvideX developer has been provided with the **DELETE OBJECT** statement.

DELETE OBJECT *obj_ID*[,**ERR=***stmtref*]

Where:

obj_ID	Numeric variable name of object reference.
stmtref	Program line number or statement label to which to transfer control.

The **DELETE OBJECT** statement takes one parameter, which is the ProvideX variable that has been bound to an object reference. After execution of this statement, the reference to the object is terminated, and the object is released from memory. For child objects, it is an error to perform a **DELETE OBJECT** if the **DEF OBJECT** has not been performed first.

Properties and Methods

Once a reference to an object has been obtained, the next step would be to manipulate or control the object. To obtain a listing of all the exposed properties and methods, the ***tick-star*** (**'***) internal property can be evaluated:

PRINT *obj_ID*'*

This statement returns a comma separated list of all exposed properties and methods, with method names having trailing parentheses. This list is merely an overview of the object, and does not include data type or parameter information.

Note: Some objects return a listing that only contains the ProvideX extended properties and methods (explained later). This occurs when the object does not expose run-time type information.

Proper object documentation is very important. Without proper documentation, it will be impossible to tell:

- What parameters are to be passed to methods
- What return values can be expected from properties or methods
- What parameters are considered by reference, vs. those that are by value
- What method parameters are optional
- What properties are indexed, and what data types are used for the indexes

Best Software only provides support for the ProvideX object ***interface***. Specific interface information for an object must be obtained from the respective vendor.

Retrieving Property Values

To retrieve a property value, place the object variable and property name on the right hand side of the equation:

variable=*object*'*property*[**$**]

Automation is case insensitive; therefore, a property name can be written in upper, lower, mixed, or proper case. If the property is to return a string data type, then a **$** symbol should be placed at the end of the property name. Properties may also be collections or arrays, which would require a slightly different syntax when assigning values:

variable=*object*'*property*.**get[$](***index***[$], ...)**

The **.get** that is appended to the property name indicates to ProvideX that this is a property, and not a method call. For string type properties, the **$** symbol should be placed at the end of the **.get** and before the open parentheses. The *index* parameter indicates the property element to retrieve. Unlike ProvideX arrays, the index for the property might not be a numeric data type (check the object documentation).

Example:

```
STYLE=DOCUMENT'Styles.get("Normal")
```

Where `Styles` is a property and `"Normal"` indicates the indexed value to retrieve.

Note: Some objects allow indexed property access without specifying **`.get`** notation.

Assigning Property Values

To assign a property value, place the object variable and property name on the left hand side of the equation:

object'*property*[**$**]=*variable*

Automation is case insensitive; therefore, a property name can be written in upper, lower, mixed, or proper case. If the property is to return a string data type, then a **$** symbol should be placed at the end of the property name. Properties may also be collections or arrays, which would require a slightly different syntax when assigning values:

result=*object*'*property*.**put**(*index*[**$**], …, *data*[**$**])

This syntax is identical to a method call, but with a few exceptions. The *result* of the property assignment is always zero, and it can be disregarded. The **.put** indicates to ProvideX that this is a property, and not a method call. And finally, the *data* to be assigned to the property is passed in as the last parameter between the parentheses.

This syntax style is also required when assigning an object to a non-index/non-array property. The reason for this is due to a syntax conflict in ProvideX, where the following would be invalid:

object'*property*=*****other_object*

The correct syntax for the above example would be:

result=*object*'*property*.**put(***other_object***)***

Note: When assigning or passing an object, it is required that an asterisk (*) appear before the object reference. This allows ProvideX to differentiate between a variable holding a numeric value, and one that holds an object reference.

There may also be cases where the property assignment is expecting to be set by reference. A common example occurs when an object property is changed to refer to a new object. For most properties that are object types, the **.put** syntax will work correctly. If an error does occur, then the following syntax should be tried:

result=*object*'*property*.**putref(****otherobject***)**

Calling Methods

To call a method, place the object variable, method name, and parameter list on the right hand side of the equation:

result=*object*'*method***[$] (***param1***,** *param2* **[, ...])**

Automation is case insensitive; therefore, a method name can be written in upper, lower, mixed, or proper case. If the method is to return a string data type, then a **$** symbol should be placed at the end of the method name. Some methods are written to accept ***optional*** parameters. When choosing not to pass an optional parameter, a * should be passed in the place of the parameter:

result=*object*'*method***[$] (*******,** *param2*, *param3***)**

The preceding example passes two parameters in, skipping the first optional parameter. The one exception to this is that ProvideX will not allow the passing of * as the last parameter. When the last parameter is optional, and should be skipped, simply close the parenthesis after the last actual argument.

The following example shows incorrect syntax for skipping the last two optional parameters of a method call:

result=*object*'*method***[$] (***param1***, *, *)**

The correct coding would be:

result=*object*'*method***[$] (***param1***)**

Note: Documentation or a type library viewer should be used when trying to determine if a method uses optional parameters.

ProvideX COM Interface Extensions

Extended Properties and Methods

During the development of the ProvideX COM interface, it was realized that developers may require information and helper functions that are not exposed by all COM objects, such as the container window handle, or the internal COM object class name, etc. To accommodate this, a set of properties and methods were created for access by all COM objects instantiated in ProvideX. The DLL handles the execution of these internally, but to the developer, they function identically to the other COM properties and methods of the object.

The following is a list of additional COM members that are available for use with ProvideX.

PVXERROR[$] ***Read Only Property.*** Returns the last error code, or message (depending on $ suffix) that occurred when accessing a property or method of the object.

PVXEVENTS[$] ***Read Only Property.*** Returns the ProvideX class object that is handling events, or a list of events names (depending on $ suffix), for the object.

PVXHEIGHT ***Read Write Property.*** Used to set the height, in pixels, of an ActiveX or OLE control. If the object is not a control, then setting this property has no effect.

PVXLEFT ***Read Write Property.*** Used to set the left border, in pixels, of an ActiveX or OLE control. If the object is not a control, then setting this property has no effect.

PVXTOP ***Read Write Property.*** Used to set the top border, in pixels, of an ActiveX or OLE control. If the object is not a control, then setting this property has no effect.

PVXWIDTH ***Read Write Property.*** Used to set the width, in pixels, of an ActiveX or OLE control. If the object is not a control, then setting this property has no effect.

PVXNAME ***Read Only Property.*** Returns the string used to create the instance of the object. For sub objects, the string also includes the property/method names.

PVXHANDLE ***Read Only Property.*** Returns the window handle for the container window that is hosting the ActiveX or OLE control. If the object is not a control, a zero will be returned.

PVXMODE ***Read Write Property.*** For controls that differentiate between development and run time mode, this is used to set the "user mode" state. Setting this to zero will place the control in development mode, any other value will place it into run time mode. If the object is not an OLE control, then setting this property has no effect.

PVXALIAS(*member$***,** *user$***)**

Method Call. This method takes two string parameters. The first parameter, *member$*, indicates the actual COM member name to alias. The second parameter, *user$*, is the user defined alias name to use.

Upon success, the object can be accessed using the alias name in place of the actual name. For example, if an object had a property called DAY, an error 20 would occur when attempting to access it (ProvideX name collision). To correct this:

```
Z = OBJECT'PVXALIAS("DAY", "_DAY")
A$ = OBJECT'_DAY$
```

PVXID ***Read Only Property.*** This property returns the interop handle to the object. It can be assigned to another integer variable, which can then be used in a **DEF OBJECT** statement. This is useful for situations in WindX where the ProvideX local variable goes out of scope.

PVXFREE(["*children***"])**

Method Call. This method will release the instance of the object. This is useful for sub objects, when **DEF OBJECT** has not been called. This method will also accept one optional string parameter, which if passed, should be set to "*children*". This will cause all the children of the object to be released, but will not release the object itself. Any other setting will cause the object to be released.

PVXEXTDATA[$] ***Read Only Property.*** This property provides buffering for returned string data that exceeds 32K. If a result returns more than 32K of data, the first 32K bytes will be returned, and the remaining data can be accessed via this property. The numeric value is the amount of data remaining in the buffer, the string value is the next 32000 bytes, or whatever is remaining.

PVXPARENT ***Read Only Property.*** Returns the parent handle (`interop id`, also see **PVXID**) for the object, or zero if the object is top level.

PVXTYPELIB$ ***Read Only Property.*** Returns the file name that exposes type information for the object. (type information must be available)

PVXMAKEGLOBAL(*global$***)**

Method Call. This method accepts one string parameter, which is the name to make "global". If successful, another ProvideX session can access this object though a **DEF OBJECT** statement using the **"[global]"** parameter. This allows a ProvideX session to expose objects to other sessions in a server like fashion.

PVXLICENSE$ ***Read Only Property.*** Returns a hexadecimal string for objects that utilize license keys (providing the key is available). This string can then be used in a **DEF OBJECT** statement, which will allow the code to create an instance of the object without the key being available (runtime client sites).

PVXISA$ ***Read Only Property.*** Returns the internal COM class interface name from the associated type library, if available. If no type library is available, a blank string is returned.

PVXDESCRIBE$(***member$*)**

Method Call. This method accepts one string parameter, which is the actual COM member name to "describe" (requires type information to be available). If successful, returns a multi line string of information describing the member, its return type, as well as parameters and their types.

Extended Objects

In order to complete the COM functionality and provide a mechanism for future enhancements, extended objects were added. These pseudo objects are instantiated in the same way as all other COM objects within ProvideX, except for the fact that an asterisk must preface the object name.

DEF OBJECT "*{*extended name***}"**

It should be noted that while these objects appear identical to other COM objects, they are not COM based in nature. This means that extended objects: do not expose events, do not expose controls, cannot have member names aliased, cannot have member information "described", and cannot be made global to other processes.

The currently supported (internal) extended objects are listed below:

***VARIANT**	Wrapper around an OLE variant data type. Provides a means for passing data by reference, as well as converting data from a ProvideX type to Automation types.
***VARARRAY**	Wrapper around an OLE safearray of variants. Provides a mechanism for COM array data handling.
***MASTER**	Serves as a system object for the interop layer. Provides object iteration capabilities, version information, etc.
***ERROR**	Wrapper export for last error information.

The four extended objects are described in detail in the sections that follow.

***VARIANT** *Extended Object*

Being a wrapper around an OLE variant, this object can store any data type, including other objects. It also provides a means for converting data in place, and handling data types that are not directly supported by ProvideX. Its primary purpose is for method calls that expect data to be passed ***by reference***, but it also useful when a COM object will only handle data of a specific type.

The following members belong to the ***VARIANT** object:

LEN ***Read Only Property.*** Returns the length of the data held by the variant. Logical use is primarily for string data.

TYPE$ ***Read Write Property.*** Returns the data type for the data being held in the variant. When set, will attempt to convert the data to the requested data type. An error will occur if the conversion fails.Valid types:

A	Array data type
B	Boolean data type
C	Byte data type
M	Currency data type
D	Date data type
R	Double data type
E	Empty data type
N	16 Bit Integer data type
I	32 Bit Integer data type
O	Object data type
F	Single data type
S	String data type

PASSBYREF *Read Write Property.* Determines if the variant will be passed by reference. Setting the property to zero indicates false, any other value indicates true. The default setting for this property is true. Also see **EXPLICITBYREF.**

EXPLICITBYREF *Read Write Property.* Determines how the variant will be passed if **PASSBYREF** is set to true. When this property is set to false (zero), a pointer to the variant will be passed to the COM method. When set to true (non zero), a pointer to the actual data will be passed. The default setting for this property is false.

VAL[$] *Read Write Property.* Used to get/set the data held by the variant. It should be noted that setting the data may cause an implicit conversion. For example, if the current data type is B, setting the **VAL** property to 0 will cause a conversion to data type I. When assigning an object to the **VAL** property, the following notation must be used:

object'**VAL.PUT(***otherobject***)**

ADD(*data*[$])** *Method Call.* Add the value of the data parameter to the currently held variant data. Logical use is for building strings greater than 32K in length, but can be used to add numbers, etc. Calling this method will perform an implicit conversion to the data type of the passed parameter.

CLEAR( ) *Method Call.* Clears the contents of the variant. After the **CLEAR** finishes, the data type for the variant will be set to E(mpty). If the variant contained an object or array reference, the object will be released.

If an object's documentation indicates that a parameter for a method call is "by reference", then ***VARIANT** must be used. The ***VARIANT** object is always passed "by reference" to the object, thus allowing the developer to retrieve the new value after method execution.

Example:

```
0010 DEF OBJECT V, "*VARIANT"
0020 DEF OBJECT X, "MSScriptControl.ScriptControl"
0030 V'VAL=10
0040 Z=X'RUN("MySub", *V)
0050 ! Pretend that the object X has set the data for V to "Hello World"
0060 ?V'VAL$ ! Will print "Hello World"
```

The data for V was set to 10 and then passed as a parameter to the object's method call. Because it was passed "by reference", the method call can change the data to anything or any type it wishes. If unsure of the data type returned in the ***VARIANT** object, then check the **'TYPE$** property.

*VARARRAY *Extended Object*

Instances of this object type are used to facilitate methods that either accept or return COM arrays. What distinguishes this object from ProvideX arrays is the fact that each element can accept data of any type. For example, ***element 1*** may hold a string value, ***element 2*** an object, ***element 3*** an integer, etc. It should also be noted that this object is always passed by reference.

If a situation arises where this functionality is not desirable, a ***VARIANT** object can be used to pass the variant array:

```
10 DEF OBJECT VARIANT, "*VARIANT"
20 DEF OBJECT VARRAY, "*VARARRAY"
20 ! SET UP VARRAY
30 VARIANT'VAL.PUT(*VARRAY)
40 VARIANT'PASSBYREF=0
50 OBJECT'METHOD(*VARIANT) ! Pass the variant array by value
```

When an array is assigned to a ***VARIANT**, it is important to remember that the ***VARIANT** will end up with a copy of the array, not the actual array itself. Any changes, even releasing the array, will not affect the array held by the ***VARIANT**.

The following members belong to the ***VARARRAY** object:

CREATE(*elements***[, *elements*])**

Method Call. Initializes the variant array to the dimension count, and element count for each dimension. This method must be passed at least one element count (to create a single dimensioned array), and can accept a maximum of 20 element counts.

Examples:

```
! Create 1 dim array with 10 elements
OBJECT'CREATE(10)
! Create 2 dim array
OBJECT'CREATE(10, 100)
```

Either this method, or **CREATEVECTOR**, must first be called before attempting to access data. Calling **CREATE** multiple times is also allowed, but will clear any existing data.

DIMENSIONS

Read Only Property. Returns the number of dimensions in the variant array.

TYPE$(*index***[, *index*])** ***Method Call.*** Returns the data type for the data being held in the variant array element. It is legal to call this method without passing in the element index, in which case the index will default to zero. For example, the following is identical for a two dimension array:

```
T$=OBJECT'TYPE$()
T$=OBJECT'TYPE$(0, 0)
```

Unlike the variant object though, the **TYPE** property is read only. If conversion of a data type is required, then the use of a variant object is mandatory. See ***VARIANT**.

COPY ***Read Only Property.*** Returns a variant array object that contains an exact copy of the contents of the current variant array.

CLEAR(*index***[, *index*])** ***Method Call.*** Clears the data being held in the variant array element. It is legal to call this method without passing in the element index, in which case the index will default to zero. After the clear completes, the element data type is set to **E**(mpty).

LBOUND(*dimension***)** ***Method Call.*** Returns the lower boundary for the dimension in the array. Please note that this will always be zero for a valid dimension. When passing *dimension*, the value should be an integer between 1 and the number of valid dimensions; otherwise, an error occurs.

UBOUND(*dimension***)** ***Method Call.*** Returns the upper boundary for the dimension in the array. This will be one number less than the total count of elements in the dimension; i.e., **0 ...** *Element_Count*-**1**.

GETDATA[$](*index***[, *index*])**

Method Call. Returns the data being held in the variant array element. It is legal to call this method without passing in the element index, in which case the index will default to zero.

GETDATAEX(*index***[, *index*])**

Method Call. Returns the data being held in the variant array element as a variant object. It is legal to call this method without passing in the element index, in which case the index will default to zero.

SETDATA(*index***[, *index*], *data*[$])**

Method Call. Assigns the contents of *data* to the variant array element. It is legal to call this method without passing in the element index, in which case the index will default to zero. The *data* parameter can be any valid data type, including variant and variant array objects.

CREATEVECTOR(*data***[, *data*])**

Method Call. Initializes the object as a single dimension array with the element count equal to the number of parameters passed in. Each element in the array will be assigned the contents of the corresponding *data* parameter.

Example:

```
OBJECT'CREATEVECTOR("John", "Smith", 31, 150.3)
```

Will create a single dimension array with an **LBOUND** of zero, a **UBOUND** of 3, and element data types of:

```
[0] = "S", [1] = "S", [2] = "I", [3] = "R"
```

Calling **CREATEVECTOR** multiple times is also allowed, but will clear any existing data.

***VARARRAY** encapsulates the functionality of OLE SafeArrays. The data type for the array is always set to VT_VARIANT, which allows each array element to contain any valid data type (including other arrays!). Only the object's documentation can tell you if you need to pass an array or not. If you run into difficulties using the ***VARARRY** (i.e., changes are not getting passed back), try setting ***VARIANT**'s **VAL** property to the array, and then pass the ***VARIANT**.

Example:

```
0010 DEF OBJECT A, "*VARARRAY"
0020 Z=A'CREATE(10, 10)
0030 Z=A'SETDATA(0, 0, "Testing")
0040 DEF OBJECT V, "*VARIANT"
0050 Z=V'VAL.PUT(*A)
0060 ! Perform call passing *V, which is a variant holding a SafeArray
0070 ?OBJ'METHOD(*V)
0080 A=V'VAL
0090 ?A'GETDATA$(0, 0)
```

If the documentation indicates that the object's method call takes a "variable argument list", then ***VARARRAY** does ***not*** need to be used. For these methods, the individual parameters should be passed in as normal.

***MASTER** *Extended Object*

The master object provides direct access to the object list maintained by the interop layer, as well as any future global settings that may be introduced. With this access, a developer can quickly determine the number of objects in use, which is essential when tracking down code that may not be handling sub object references properly. The following members belong to the ***MASTER** object:

COUNT — ***Read Only Property.*** Returns the number of objects being managed by the interop layer (includes the master object in this count).

ITEM(*index***)** — ***Method Call.*** Returns the interop handle for the object at the specified *index* (see **PVXID**). Please note that **ITEM** should be treated as a zero-based array.

VERSION$ — ***Read Only Property.*** Returns the file version number for the interop library `PVXOCX32`.

CALLCOUNT — ***Read Only Property.*** Returns the number of command call executions since the interop library was initialized.

***ERROR** *Extended Object*

This static table is exposed as another internal object. It can be **DEF**'ed after an error has occurred and will still contain the last error information. One note in regards to this class: while multiple instances can be created, they all point to the same data block; i.e., clearing one ***ERROR** object will in effect clear all other instances. The following members belong to the ***ERROR** object:

OLEERROR — ***Read Only Property.*** Returns the last OLE `HRESULT` code that was raised.

NUMBER — ***Read Only Property.*** Returns last object code set for a dispatch call that failed with `DISP_E_EXCEPTION`.

DESCRIPTION — ***Read Only Property.*** Contents depends on if the last error was `DISP_E_EXCEPTION` or not. If so, it returns the object defined message. If not it returns the string representation of the OLE `HRESULT`.

HELPFILE — ***Read Only Property.*** If last error code was `DISP_E_EXCEPTION`, then this will be pulled from the `EXCEPINFO` structure.

HELPCONTEXT — ***Read Only Property.*** Same as **HELPFILE**, except for `HELPCONTEXT` field of structure.

CLEAR( ) — ***Method Call.*** Clears the last error information.

Error Handling

No matter how carefully crafted your code, errors can (and probably will) occur when programming against COM objects. Lack of proper documentation, misbehaved COM objects, and incorrect data types are just a few of the numerous reasons that errors will occur. When a COM error does occur, it normally appears in the form of an `Error #88: Invalid/unknown property name`. This error should only be taken as an indicator that a COM method (property) call ***failed***.

In order to further identify the problem, the following steps should be taken:

1. **Break multi tick (') statement lines into single tick statements.** Multi tick statement lines only specify one object variable. The other objects created to resolve the expression are temporary. If an error is reported on one of the temporary objects, it cannot be evaluated after the statement executes, thus the context of the error is lost.
2. **Check the result of MSG(-1).** When a COM error occurs, the interop layer will set the textual error message for **MSG(-1)**.
3. **Check the PVXERROR[$] for the object**. The error code returned by **PVXERROR** is also known as the `HRESULT` by developers in other languages. When working with third party developers, this information may be required. The string representation of this error is identical to what is displayed by **MSG(-1)**.
4. **Verify (with documentation or type library viewer) that the information you are passing is correct.** One of the most common errors is passing incorrect or invalid data to COM methods. The second most common error is passing too many, or too few, parameters.

In most cases, the steps above will be enough to resolve the COM errors that occur. Should this not be the case, then proper documentation will be critical in resolving the issue. If documentation is not available for the COM object, then the use of **PVXDESCRIBE** can be used as a last resort. This extended method takes one string parameter, which is the name of the object's property or method to generate information for.

If the object does not expose type information, then the interop layer will not be able to provide any detailed information.

Note: An object does not require type library information to be programmable. However, without proper documentation, it will be impossible to determine what member names are available, and how they should be called.

Events

Whenever a COM object wants to notify its clients that something has happened, it sends out a message. This message is called an ***event***, and the process of sending the message is referred to as ***event firing***. If an event is fired and nobody is listening, did the event ever occur? Obviously, the client application that is controlling the COM object has to be listening for the events. When a client application wishes to receive events from a COM object, it advises the COM object of this fact.

Receiving Events

In order to receive events from a COM object, a ProvideX class must first be designed. A class function must be written for each event to be handled. For example, if writing an event class for a COM object that exposed the event

ONCLICK(X ***as Integer*****, Y** ***as Integer*****, Button** ***as Integer*****, Shift** ***as Integer*****)**

.. the following would be required in your ProvideX class:

```
10 DEF CLASS "FOO"
20   FUNCTION ONCLICK(X, Y, B, S) ONCLICK FOR EVENT "ONCLICK"
30 END DEF
40 STOP
50 ONCLICK:
60 ENTER X, Y, B, S
70 PRINT "OnClick was fired"
80 RETURN
```

The function and label name are not required to match the name of the event. It is the string after the `FOR EVENT` portion of the statement that determines the event name that will be handled. The event name is not case sensitive, and argument names are not required to match the COM object's event declaration. When the OOP method name matches the COM event name, then **SAME** can be used to describe the name of the event.

Once the class is complete, an instance of the class must be instantiated in order to bind the COM object to the event handler. An example of the binding is listed on line 30:

```
10 DEF OJBECT FOO, "{Your Com Object Name}"
20 FOOEVENTS = NEW("FOO")
30 ON EVENT FROM FOO PROCESS FOOEVENTS
```

Once bound, the ProvideX event class function will be called when the corresponding event occurs.

Handling Events

It is highly recommended that event functions be kept relatively short and simple. One reason for this is that normal code execution will be suspended when an event is handled and will not resume until the event function is finished. It is also possible to introduce code (or use commands such as **MSGBOX**) to create a reentrancy issue in the event function.

Reentrancy is allowed; however, a limit has been imposed to prevent runaway objects. If the event call stack reaches a depth of 64, all further events will be discarded while waiting for the current events to finish. The following related TCB values have been added to ProvideX to expose this information:

TCB(120) Returns the interop handle (see **PVXID**) for the COM object that fired this event. Similar to the `_obj` when in a class function.

TCB(121) Returns the number of events that have been ***dispatched*** to ProvideX.

TCB(122) Returns the current depth of the event call stack.

TCB(123) Returns the number of events that have been ***discarded*** by ProvideX due to excessive outstanding event calls.

It is important that the function declaration in the ProvideX class matches that of the COM event. Otherwise, your function may not get called, or may cause an error when the event is fired. If a situation occurs where an event argument is defined as variant (and may receive either string or numeric data) then a second overloaded event function should be defined, e.g.,

OnData(*data*)

The following would be required in your ProvideX class to properly handle both cases:

```
010 DEF CLASS "FOO"
020   FUNCTION ON_DATA(N) ON_NDATA FOR EVENT "ONDATA"
030   FUNCTION ON_DATA(S$) ON_SDATA FOR EVENT "ONDATA"
040 END DEF
050 STOP
060 ON_NDATA:
070 ENTER N
080 PRINT "Numeric data ", N
090 RETURN
100 ON_SDATA:
110 ENTER S$
120 PRINT "String data ", S$
130 RETURN
```

Another aspect of event handling deals with the scope of passed arguments. All arguments passed into an event are local in scope, and should not be considered to exist when the event is finished. It is common for many event routines to pass COM objects in as parameters. These can be programmed against during the event, but should not be persisted (assigned for later use) when the event is finished.

Event Templates

The ***ProvideX Type Library Browser*** (TLB) was developed to provide extended type information for Windows COM objects, and to help simplify the process of creating event class objects. This utility (`pvxtlb.exe`) is freely downloadable from www.pvx.com. The following steps outline the event template generation feature of the TLB:

1. Use the *Type Library Browser* to open the desired type library. The COM object's **PVXTYPELIB$** and **PVXISA$** properties are useful for determining this information.
2. Locate the CoClass (COM Class) that supports the COM object, and locate the Default Event Interface in the Entity Documentation section.
3. Browse to the event interface by clicking the link in the Entity Documentation, or by double clicking the interface in the Members list.
4. In the Entity Documentation section of the utility, a link is available to generate an event template. Click this link to open the Save As dialog.
5. Save the template to the desired file name. The **DEF CLASS** statement in the template is automatically updated to reflect the filename assigned.

The event template (in text form) can then be edited in any text processor. All event functions, type casting, and def object statements will have been automatically created. The only thing that is required is to fill in the event function bodies with the necessary ProvideX code. Each function will have a commented section identified with the tag "< Insert code here >", which is where the custom code should be placed.

For further information on the Type Library Browser, refer to the documentation (`TLBDoc.pdf`) available from www.pvx.com.

Errors in Events

In a typical ProvideX program that uses class objects, an error raised by an object will be cascaded back to the calling program. Events, on the other hand, are called by COM objects and not ProvideX code. If an error occurs during the event handling, the COM interop layer will notify the COM object that the event failed. However, nothing is reported on the ProvideX side; i.e., there is no program to report to. For this reason, all error handling should be performed within the class object.

Also, it is not wise to release the calling COM object, or its event handler object, during the execution of the event. Releasing the COM object during the event can lead to unpredictable results. Objects that were passed in as event parameters may be safely released.

Usage and Examples

Sub-Objects

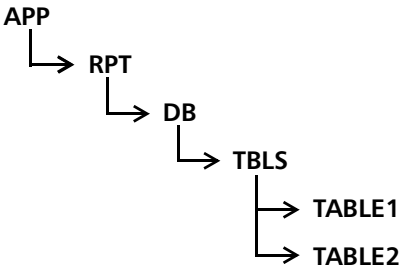
In ProvideX, a sub-object can be defined as any object that is the return value of another object's property or method call. How is this important? Since a sub-object is *owned* by the base object, it will be *destroyed* when the base object is destroyed.

Example:

```
0010 ! Crystal COM example
0020 DEF OBJECT APP,"CrystalRuntime.Application"
0040 RPT=APP'OPENREPORT("d:\crw8\chart.rpt")
0050 DB=RPT'DATABASE
0060 TBLS=DB'TABLES
0070 TABLE1=TBLS'ITEM(1)
0080 TABLE2=TBLS'ITEM(2)
0090 DELETE OBJECT APP
```

In the example above, RPT is a sub-object of APP, DB is a sub-object of RPT, TBLS is a sub-object of DB, and TABLE1 and TABLE2 are sub-objects of TBLS.

This ownership is best descriped as a tree list where:



After line 80 executes, all sub-objects owned by APP will be destroyed. This is done to ensure that the reference counting of APP is brought down to 1 before it is destroyed. This is not a feature of COM, but a feature implemented by the DLL. In a VB application, if APP was freed, the actual COM object would exist in memory until all the other sub-objects were freed. (Because each sub-object increments the reference count of the parent). While this feature is ideal for most situations, it can cause problems if you delete a parent object and then try to perform actions on one of its sub-objects.

Passing Optional Parameters

Many objects have methods where the parameters are defined as optional. This is an indication to the developer that the parameter can be excluded. But how is this accomplished in ProvideX? Below is an example of an ADO Recordset Open method in VB as translated to ProvideX:

VB: `RS.OPEN "GL1_ACCOUNTS", CONN,  ,  , 2`
ProvideX: `?RS'OPEN("GL1_Accounts", *CONN, *, *, 2)`

In ProvideX, the optional parameters are passed using "*". The one exception is that ProvideX will not allow "*" to be passed as the last parameter. If these parameters are not required for the method call, then they must be omitted; e.g.,

```
0010 X=RS'OPEN("GL1_Accounts", *CONN, *, *, *) ! Syntax Error
0020 ! The following line is the same as the line above
0030 X=RS'OPEN("GL1_Accounts", *CONN)
```

The three optional parameters are omitted in line 30. Although syntactically different, line 30 is functionally the same as line 10.

Reserved Word Conflicts - Aliasing

Occasionally there are cases where an object's property or method is identical to a ProvideX reserved word. When this happens, ProvideX generates a syntax error on any attempt to use the object's property or method name. For example, if a you queried a calendar control for a property called `DAY`:

```
> ?X'DAY
```

An `Error 20` is generated because `DAY` is a reserved word. The workaround for this is called ***aliasing***. Aliasing allows a developer to call a property or method by creating user-defined names. For example, the following could be done to correct the previous example:

```
0010 Z=X'PVXALIAS("DAY", "FOO")
0020 ! FOO is now a reference to DAY
0030 ?X'FOO
```

Aliasing may be used on any COM method or property. It is also useful for assigning more *meaningful* names to object methods and properties.

Method Invocation

Due to syntax restrictions, ProvideX does not allow object assignment to properties. The following example (given for illustation purposes) generates an `Error #23: Missing/Invalid variable`:

```
10 DEF OBJECT X, "*VARIANT"
20 DEF OBJECT Y, "*VARIANT"
30 X'VAL=*Y ! Error 23 occurs
```

To handle this situation, method calls are enhanced to accept "invoke hints". An invoke hint indicates to the DLL how a call should actually be invoked: as a *function*, *property get*, *property set*, or *property set reference*.

The following is a list of valid method invocation hints.

`.GET`	*Get property*
`.PUT`	*Set property*
`.PUTREF`	*Set property reference*

These hints are added to the end of a method/property name so that the call reads as *object*'*method*{*.hint*}(*parameters*). If you attempt to get or set array properties without using an "invoke hint", the DLL will attempt to resolve the calling type; however, it can only do this by trial and error (up to 4 separate attempts).

For speed reasons, as well as clarity, the developer should always specify a hint when calling a property using the syntax of a method call. In the previous example, line 30 should be changed to:

```
30 NULL=X'VAL.PUT(*Y) ! Assign object Y to X'VAL
```

The following is an example of a grid object that exposes a `CELL` property that is accessed using a row and column indicator:

```
10 DATA=GRID'CELL.GET(ROW, COL)        ! Get the cell value
20 DATA= DATA+10                       ! Add 10 to the value
30 NULL=GRID'CELL.PUT(ROW, COL, DATA)  ! Set the cell value
```

Examples

Example 1. Embedding IE5 or IE6. You must have Internet Explorer installed for this to work.

```
0010 DEF OBJECT handle, @(2,2,70,16)="Shell.Explorer"
0020 errcode=handle'Navigate2("www.pvx.com")
0030 input *
0040 DELETE OBJECT handle
```

Example 2. Dynamically creating an object from the OCX toolbox.

```
0010 PRINT 'CS'
0020 WAIT 0
0030 DEF OBJECT OCX,@(40,1,30,10),"*",ERR=0100
0040 PROG$=OCX'PVXNAME$
0050 ESCAPE
0060 END
0100 PRINT MSG(-1)
```

Example 3. Creating a Microsoft Form Frame and adding a button to it. This requires the MS Forms 2.0 Object Library to be installed on the system.

```
0010 DEF OBJECT FRM, @(40, 2, 30, 12), "Forms.Frame.1"
0020 FRM'PVXMODE=0
0030 CTRLS=FRM'Controls
0040 BTN1=CTRLS'ADD("Forms.CommandButton.1")
0050 BTN1'Top=20
0060 BTN1'Left=20
0070 BTN1'Caption$="Hello World!"
0080 input *
0090 DELETE OBJECT FRM
```

Example 4. Creating a Microsoft Word Document from a file reference. This requires MS Word to be installed on the system.

```
0010 GET_FILE_BOX READ X$,LWD,"Word Document","Word Document|*.doc,"
0020 IF X$="" THEN END
0030 DEF OBJECT WD,@(0,0,80,20),"[file]"+X$
0040 INPUT *
0050 DELETE OBJECT WD
0060 END
```

Example 5. Using ADO to display the tables in a Microsoft Access Database.

```
0010 GET_FILE_BOX READ X$, LWD,"Access Database"," Access Database |*.mdb,"
0020 IF X$="" THEN END
0030 DEF OBJECT CONN,"ADODB.Connection"
0040 CONNSTR$="Provider=Microsoft.Jet.OLEDB.4.0;Data Source="+X$
0050 X=CONN'OPEN(CONNSTR$)
0060 DEF OBJECT VARDATA,"*vararray"
0070 Z=VARDATA'CREATE(4) ! Create single dim array with 4 elements
0080 ! By default, all elements are set to empty
0090 Z=VARDATA'SETDATA(3,"Table") ! Set 4th element to string "Table"
0100 ! Get the schema recordset
0110 R=CONN'OPENSCHEMA(20,*VARDATA)
0120 N$=""; FOR I=0 TO R'FIELDS'COUNT-1; N$=N$+R'FIELDS(I)'NAME$+" | "; NEXT
      I; PRINT N$
0130 ! Get the first 20 table names, this will return an array
0140 RA=R'GETROWS(20)
0150 FOR I=0 TO RA'UBOUND(2)
0160 TXT$=""
0170 FOR II=0 TO RA'UBOUND(1)
0180 TXT$=TXT$+RA'GETDATA$(II,I)+" | "
0190 NEXT II
0200 PRINT TXT$
0210 NEXT I
```

Example 6. Test of Microsoft Script Control (***VARARRAY** support). This demonstrates passing a ***VARARRAY** in a ***VARIANT**, as well getting an object returned back "by reference".

```
0040 DEF OBJECT MSC,"MSScriptControl.ScriptControl"
0050 MSC'LANGUAGE$="" ! Need to clear it in order to get it set.
      Documented issue in MSDN
0060 MSC'LANGUAGE$="VBScript"
0070 MSC'ALLOWUI=-1
0080 NL$=CHR(13)+CHR(10)
0090 PGMTEXT$="Sub GiveMeObject(byref obj)"+NL$
0100 PGMTEXT$=PGMTEXT$+"set
      obj(1)=CreateObject("+QUO+"Excel.Application"+QUO+")"+NL$
0105 PGMTEXT$=PGMTEXT$+"msgbox obj(1)"+NL$
0110 PGMTEXT$=PGMTEXT$+"End Sub"+NL$
0120 Z=MSC'ADDCODE(PGMTEXT$)
0130 DEF OBJECT X,"*variant"
0131 DEF OBJECT A,"*vararray"; Z=A'CREATE(2)
0132 Z=X'VAL.PUT(*A)
0140 Z=MSC'RUN("GiveMeObject",*X)
0150 A=X'VAL ! Get the array back
0155 PRINT A'TYPE$(1) ! Will print "O"
0160 DELETE OBJECT X,ERR=*NEXT
0170 DELETE OBJECT MSC
```

Example 7. Using ***VARIANT** objects and assigning objects to properties.

```
0010 DEF OBJECT X,"*VARIANT"
0020 DEF OBJECT Y,"*VARIANT"
0030 X'VAL$="Hello World"
0040 Y'VAL=100
0050 Z=Y'VAL.PUT(*X)
0060 PRINT Y'VAL$ ! Will print "Hello World"
```

Comparisons with Visual Basic

Due to its popularity and wide spread use, many third party vendors tailor their code examples for Visual Basic. While translating these examples to ProvideX is relatively straight forward, there are some nuances in the Visual Basic language that can cause problems.

This section lists some of the more common translation errors, and describes how they should be handled.

Default Member Access

What is a default member? A default member is a property or method of an object that is invoked when a client does not specify a property or method name. Take the following Visual Basic example:

```
Dim rs As ADODB.Recordset
rs("CompanyName") = "SomeCompany"
rs!CompanyName = "SomeCompany"
```

The code above is actually a shortcut for:

```
Dim rs As ADODB.Recordset
rs.Fields("CompanyName").Value = "SomeCompany"
rs.Fields!CompanyName.Value = "SomeCompany"
```

The problem with this coding style is that the code is no longer self documenting. A programmer must have knowledge of the `ADODB.Recordset` object in order to determine what the code is actually doing.

Determining when a shortcut has been used is a little more difficult. If the code returns an object, and the assignment is performed using a string, number, or object (with out a **Set** statement), then a shortcut is most likely in use. To access the default member in ProvideX, an underscore _ must be used. See the corresponding ProvideX code:

```
10 DEF OBJECT RS, "ADODB.Recordset"
20 RS'_("CompanyName")'Value$ = "SomeCompany"
30 RS'_("CompanyName")'_$ = "SomeCompany"
```

Note: While this coding style is supported in ProvideX, its general use is discouraged.

For Each

The `For Each` statement is used in Visual Basic when an object exposes a collection interface (using `IEnumVariant`); e.g., `Excel.Application` exposes a collection called `WorkBooks`:

```
Dim ExcelApp As Object
Set ExcelApp = CreateObject("Excel.Application")
For Each WorkBook in ExcelApp.WorkBooks
  '...
Next
```

Unfortunately, ProvideX is unable to use the enumerator interface, and must use the Count and Item() property to gain access to the items in the collection:

```
10 DEF OBJECT ExcelApp, "Excel.Application"
20 FOR I=1 TO ExcelApp'WorkBooks'Count
30   WORKBOOK = ExcelApp'WorkBooks(I)
40   !…
50 NEXT
```

Note: All collections will expose an `Item( )` and `Count` property at the very minimum. The difficult part about converting the `For Each` code is in determining if the collection is 1-based, or (zero) 0-based. The MSDN indicates that older collections in Visual Basic tend to be 0-based, while newer collections are 1-based. If the documentation does not indicate which index base is used, then a trial and error technique must be applied.

Named Arguments

Some objects allow method arguments to be passed in using name positioning rather than index positioning. For example, the `Open` method of the Microsoft Excel Workbooks object (for opening a workbook) takes 13 arguments. All arguments are optional in the `Open` method, and could be written in Visual Basic as:

```
Workbooks.Open "book2.xls", , , , , , , , , , , , True
```

Given that Microsoft Excel will accept named arguments, the preceding code could also have been written as:

```
Workbooks.Open FileName:="book2.xls", AddToMru:=True
```

The use of a named argument is very easy to spot when converting Visual Basic code, given the `:=` syntax. The difficult part is converting the above sample to ProvideX, which must pass arguments by index. This is where documentation, or a good type library viewer, is necessary.

Once the index location of `FileName` and `AddTOMru` are found, coding the statement in ProvideX is simple:

```
Workbooks'Open("book2.xls", *, *, *, *, *, *, *, *, *, *, *, True)
```

This is very close (syntactically) to the original Visual Basic example, except for the use of asterisks as optional arguments.

Calling Conventions

This section discusses how to determine when you should, or should not, expect a result from a method call. If a Visual Basic statement passes parameters, but does not contain open and close parentheses, then the statement is performing a call to a procedure, and no result is returned. Using the previous Excel example:

```
Workbooks.Open "book2.xls", , , , , , , , , , , , True
-
10 Z = Workbooks'Open("book2.xls", *, *, *, *, *, *, *, *, *, *, *, True)
```

In this example, no data is returned from the Visual Basic call; but, in ProvideX, a zero would be returned to Z. The zero in ProvideX, for this situation, represents a null return value. Using another example:

```
Dim I as Integer
I = RecordSet.Fields(0)
-
10 I = RecordSet'Fields(0)'Value
```

The data returned for the field value might be zero, but it does have meaning in this context (it is the data value for the Fields). Finally, if the statement you are converting starts with a `Set`, then your code should be written to expect an object return value.

```
Dim Fld as Object
Set Fld = RecordSet.Fields(0)
-
10 Fld = RecordSet'Fields(0)
20 DEF OBJECT Fld
```

Advanced Topics

How COM Calls are Made by the DLL

Upon successful creation, `IDispatch` is retrieved from the newly created object. `IDispatch` is a COM interface that allows the DLL to perform three specific functions:

- Get the function and property listing for an object
- Resolve dispatch IDs for given names
- Invoke property and method calls on the COM object.

The first step taken by the DLL in order to get/set a property or method is to get the dispatch ID for a given object. The DLL checks the object's hash table for the name. (The hash table allows for storage and retrieval of previously resolved name/ID pairs). This is referred to as "IDbinding", which is a form of early binding. By using the hash table, the DLL makes only one COM call to resolve any given name, versus two COM calls when performing an action on a "late bound" object.

Next, the DLL packs the passed-in parameters. Strings are converted to OLE BSTRings, integers and doubles are packed *as is*, "*" optional parameters are converted to VT_ERROR types, and object parameters are converted to either IDispatch or VARIANT BYREFs, depending on the object type.

Once the packing is performed, the `invoke` function is called on the object's IDispatch interface. The COM object has control at this point and will unpack the data, execute the property or method, and set the result value. When control is returned to the DLL, it will check for errors. If no errors have occured, the result is packaged up and sent back to ProvideX.

OCX Controls

The DLL container control exposes all the "ambient" properties that are required by OCX controls. These include *foreground/ background* color, *message reflection*, *user mode*, etc.

User mode is the only ambient property that is directly available to the ProvideX programmer, the rest are handled by the DLL. User mode is special in that it tells the OCX control if it should act as a design-time control, or as a run-time control. For example, control ***A*** displays designer toolbars when *UserMode* = *false*, but hides these when *UserMode* = *true*. In ProvideX, this property is accessed via the **'PVXMODE** property. Setting **PVXMODE** = 0 sets *UserMode* to *false* (design-time). Any non-zero value will set *UserMode* to *true* (run-time). The DLL also supports OCX control property sheet activation by way of the F2 key. To display an OCX controls property sheet (if it has one), set focus to the control and press F2.

OLE Container Support

During the creation of the COM object, the DLL queries the dispatch interface to see if IOleControl is exposed. If it is, then the DLL will create a container window that will be embedded within a ProvideX window. The object is then shown or activated, depending on the flags it exposes. For objects that are "in place" activated (i.e., they expose the IOleInPlaceActiveObject interface) special note should be taken.

Some of these controls, (e.g., Excel) have routines that are input-synchronous. Double-clicking a cell in Excel is an example of this — the cell is put into edit mode and it captures focus. If you attempt to make a method or property call at this point, it would fail with an error indicating that the "call was rejected".

A workaround to this problem is to send a WM_ACTIVATEAPP to the ProvideX window before making the COM call.

Example:

```
10 HWND$=OBJ(0); HWND=DEC(HWND$(17,4))
20 X=DLL("USER32", "SendMessageA", HWND, 28, 1, 0)
30 X=OBJECT'METHOD(PARAM1, PARAM2, …)
```

Line 20 ensures that the control is activated and will reset ***most*** input synchronized states.

PVXOCX32 Misnomer

The "OCX" reference is a bit misleading, given the fact that ProvideX supports any object or control that exposes an IDispatch interface. The "OCX" reference was assigned during initial development, when the desired results were the use of OCX controls within ProvideX.

In its current implementation, ProvideX COM support includes OLE, OCX, and ActiveX technologies:

OLE

Out of the DDE protocol grew OLE version 1.0 (1991), which was made available to all developers as a standard. The acronym "OLE" is an abbreviation of "Object Linking and Embedding."

OLE 1.0 greatly enhanced the creation and management of compound documents. One placed "embedded objects" or "linked objects" in a document that retained the native data used to create them (or a "link" to that data) as well as information about the format. All the complexity of editing content was reduced to a double-click of the mouse (and ***presto!***) the object data was automatically brought back into the original editor.

OCX

The original OLE control specification required every control to implement at least nine specified interfaces, containing a combined total of 60 methods, in addition to whatever interfaces the control might implement to expose its own methods.

There were also seven other interfaces that might be required, depending on whether the control had a user interface, and whether it supported events, property change notifications, ambient properties, property sets, property pages, or external connections.

This standardization of interfaces made OLE controls easy to use, if not easy to program. A container application knew exactly what interfaces it could expect from any OLE control and could treat all OLE controls in a uniform manner. The one problem was that, in order to support all this functionality, OLE controls tend to be rather large, which makes them impractical for use on the Internet.

ActiveX

In 1996, Microsoft released a new specification for controls. This specification essentially only requires a control to support one interface, IUnknown, and two API functions, DllRegisterServer and DllUnregisterServer.

Because an ActiveX control does not have to support the standard set of interfaces required for an OLE control, a container has to have more specific knowledge about an ActiveX control than it would have to have in order to use an OLE control. On the other hand, an ActiveX control only needs to support those interfaces it actually requires to do its job. Therefore, ActiveX controls can be very small and very responsive.

COM vs Flat API Function Calls

DLL Problem	COM Solution
Path dependencies, multiple providers of a service.	Use the registry to map from abstract class identifiers to absolute server locations as well as mapping between categories and class identifiers.
Decentralized definition of identifiers.	Use GUIDs generated by an algorithm that guarantees uniqueness across time and space, eliminating the need for centralized identifier allocation.
Specific management APIs for each service category.	Supply a very simple generic and universal management API that accommodates all service categories, called Implementation Location.
Sharing instances across process/machine boundaries.	Marshaling of interface pointers, Location Transparency providing the ability to implement a server as an EXE or DLL.
Different in-process, local, and remote programming models.	A single model for all types of client-object connections supported through the interface structure, Location Transparency.
Lifetime management of servers and objects.	Universal reference counting through the base interface IUnknown that all objects support and from which all other interfaces are derived.
Multiple services per server module.	Use CLSID:IID:table_offset to absolutely identify functions instead of module:ordinal.
Versioning as well as interfaces that are strongly typed at both compile time and run time	Support the concept of multiple immutable interfaces.