Creating Visual Basic VBXs:
The Property List

by Fred C. Hill

The second in a series of articles on the secrets of writing VBX controls in Borland Pascal.

This article will begin the process of creating a control to handle disk operations which are generally unavailable in Visual Basic. To start it is important that you understand the control property and how it is used. We begin by describing what a property list is and what it does.

Visual controls communicate via the property lists. Each of the items in the list is designed to allow the Visual Basic developer to change, enhance, hide or in some fashion alter the default behavior or to provide input for the control so it can do its job. This property list isn't provided by the Visual Basic programmer but by the VBX developer.

The list can be created just like any other list in any other program. That is it can be used as compiled or can be built at run time. Of course, it must be remembered that the first user of the VBX may not be the only user during its residency in memory so any run time creations must be ones which will effect all users. Considering this it is generally considered good strategy to set up all properties during compile time.

Developing the properties list is similar to laying out a COBOL programs data section although a lot more fun. Each of the required elements must be named and fixed in place so the Visual Basic runtime (rtl) will know where to find them and what type they are.

Each of the properties in the list is set up by initializing its unique property data structure. (see Listing 1).

```
const
pName: array[0..12] of char = ‘PropName'#0;
 Prop_Name: tPROPINFO = (
  npszName: tOffset(@pName);
  fl: DT_SHORT or PF_fGetData;
  offsetData: tOffset(@dataVariable);
  infoData:    0;
  dataDefault: 0;
  npszEnumList: 0;
  enumMax:     0);

type
  iPropIndex = (
    iProp_Name,
    iProp_Index,
    iProp_Tag,
    iProp_Left,
    iProp_Top,
    iProp_MyProp,
    iProp_Last);
```

```
const
  PropertyList : array[iPropIndex] of
  ofsPPropInfo = (
 pPropInfo_STD_NAME,
 pPropInfo_STD_INDEX,
 pPropInfo_STD_TAG,
 pPropInfo_STD_Left,
 pPropInfo_STD_TOP,
 tOffset(@pName),
 0);
```

Listing 1

The pPropInfo_STD_NAME property must be first in the list and must be followed by pPropInfo_STD_INDEX. After that the convention generally used, places all of the "standard" properties first in the list. After that you can place your properties anywhere you wish. If you find a VBX using PPROPINFO_STD_CTLNAME it is just a throw back to the VB 1.0 days.
Now that you have your property list defined let's discuss where the data goes and how does it make the journey from the Visual Basic program and back. The variables used to hold the controls properties can be anywhere in the VBX but is generally grouped into a single record structure. (See Listings 2 and 3)

```
type
PmyData = ^tMyData
 tMyData = record
  MyProp:  integer;
end;
```

Listing 2

A visual control supports the following data types:

DT_Bool Boolean	a short integer;
DT_COLOR	a long integer which holds an RGB value;
DT_ENUM	a short integer holding the enumerated value (0-255). Use of this data type activates the npszEnumList and the enumMax fields in the property data structure;
DT_HLSTR	String, this data type cannot be used in VB1.0 compatible VBX as support was added in Visual Basic 2.0. It is a VB string and can hold NULL values.
DT_NSZ	a string, terminated by a NULL character.
DT_LONG	A 32 bit signed integer
DT_OBJECT	Points to an Idispatch interface. Refer to the CDK distribution files for an explaination.
DT_PICTURE	

Picture structure stored as an HPIC handle.

DT_REAL

4 byte real.

DT_SHORT

16 bit signed integer

DT_XPOS

Long integer expressing an X coordinate

DT_XSIZE

Long integer expressing an X size in twips

DT_YPos

Long integer expressing a Y coordinate in twips.

DT_YSIZE

Long Integer expressing a Y size in twips.

Visual Basic requires that certain data types be passed between the control and VB in special ways. Each of the data types are defined in the VBAPI provided by Microsoft on the Custom Control Developers Kit (CDK) and have been exported in the VBAPI_.PAS unit provided on the accompanying diskette. The data types and the data handling flags are or'ed together to inform Visual Basic of the characteristics of the variable.

```
type
  enum_Values1 = (DiskA, DiskB, DiskC, DiskD);
  enum_Values2 = (DiskE, DiskF, DiskG, DiskH);
  pDataDefine = ^tDataRecord;
    tDataRecord = record
    bBooleanVal: boolean;
    usIntegerVal: integer;
    ulLongVal:    longInt;
    usEnumVal:    eNum;
    hszStringVal: Hsz;
    usAction:     Integer;
end;
```

Listing 3

Listing 4 is the structure defining the Boolean value. While the Pascal language defines and manages the boolean internally it is, in fact, a byte value and is considered such by Visual Basic. In the next issue we will discuss the methods used to handle the differences between Visual Basic and Pascal when it comes to data handling.

```
const
BooleanName:        array[0..12] of Char = 'Boolean'#0;
 Property_Boolean:   tPROPINF
const
 BooleanName:       array[0..12] of Char = 'Boolean'#0;
 Property_Boolean: tPROPINFO = (npszName: tOffset(@BooleanName);
 fl: DT_Bool or PF_fGetData or PF_fSetData or PF_fSaveData;
 offsetData: tOffset(@bBooleanVal);
 infoData:   0;
 dataDefault:   0;
 npszEnumList: 0;
 enumMax:   0);
```

Listing 4

Listing 5 shows the definition of an Integer value. Since Visual Basic, all VBX's and Borland Pascal are still 16 bit systems and have yet to be released in a 32 bit version, an integer is still only 16 bits long. A longInt is defined in a similar manner and is actually 2 computer words or 32 bits in length.

```
IntegerName: array[0..12] of Char = 'Integer'#0;
Property_Integer: tPROPINFO = (npszName: tOffset(@IntegerName);
 fl:  DT_Short or PF_fGetData or PF_fSetData or PF_fSaveData;
 offsetData:  tOffset(usIntegerVal);
 infoData:  0;
 dataDefault: 0;
 npszEnumList: 0;
 enumMax: 0);
```

Listing 5

Listing 6 contains the definition of an enumerated list. When Turbo Pascal 1.0 was first released I thought the enumerated list was almost magic. It simplified the language tremendously and made defining non standard sequences extremely easy. When an Enum data variable is defined in a VBX it is translated to a pop down list in the Property Window. The developer can select one of the choices or by double clicking the edit box can cycle through the choices until the correct one is found.

```
ENumName: array[0..12] of Char = 'ENum'#0;
EnumEntries: array[0..60] of Char =        'A:'#0'B:'#0'D:'#0'E:'#0'F:'#0'G:'#0'H:'#0'I:'#0#0;
Property_ENum: tPROPINFO = (npszName: tOffset(@ENumName);
 fl: DT_ENUM or PF_fGetData or PF_fSetData or PF_fSaveData;
offsetData: tOffset(@usEnumVal);
infoData:       0;
dataDefault:  0;
npszEnumList:        tOffset(@EnumEntries);
enumMax:     8);
```

Listing 6

After the Integer and LongInt, the string will undoubtedly be the most commonly used data type in the list. (See Listing 7) Except for the picture, which we won't be covering in this article, the string will also be the most difficult to use and control. The reason for this is that Visual Basic has provisions to handle most string types but prefers a Visual Basic string. Procedures are provided in the Control

Development Kit (CDK) to convert strings to and from NSZ (null terminated) and HLSTR, the preferred Visual Basic string which can contain imbedded nulls. Routines are also provided to create and destroy memory areas to hold and pass both temporary and permanent strings of both types.

In the Windows event paradigm it is important to understand that the address you had for a string during one pass through the VBX probably won't be the address during a subsequent pass.

```
StringName: array[0..12] of Char = 'String'#0;
Property_String:tPROPINFO = (npszName:tOffset(@StringName);
 fl:       DT_HSZ or PF_fGetData or PF_fSetData or PF_fSaveData;
 offsetData: tOffset(@hszStringVal);
 infoData:      0;
 dataDefault: 0;
 npszEnumList: 0;
 enumMax:    0);
```

Listing 7

As mentioned before, we will not be covering the DT_PICTURE variable in this article. We will attempt to cover it in a future article and in fact is complex enough to deserve an article of its own. We will also not be specifically discussing the DT_COLOR, the DT_OBJECT, or the DTXx and DT_Yx data types although in future articles I may use these types. When I do use them I'll make a point of descibing their use. I recommend that, if you get into writing controls in a big way that you invest in the CDK which will provide detailed documentation about the use of these data types.

The following is a list of property flags can be used in the **fl** variable of the property structure. Each of these initiates an automatic default action which is performed on the property at some point in the life of the control.

PF_fDefVal
: Causes Visual Basic to avoid saving and loading the property to disk when the value is equal to the default value in the PROPINFO structure.

PF_fEditable
: Enables the appliation developer to edit text in the settings box directly.

PF_fGetData
: Visual Basic gets data for this property by copying it directly from the programmer defined structure.

PF_fGetHszMsg
: Causes Visual Basic to send a VBM_GETPROPERTYHSZ message to the control whenever the property value is displayed in the Properties window. If this flag is not used Visual Basic will get the property value and displays it in the settings box.

PF_fGetMsg
: Causes Visual Basic to send VBM_GETPROPERTY message when the property value is requested. Either this flag or the PF_FGetData flag should be used.

PF_fLoadMsgOnly [2.0]
: Causes Visual Basic to send the VBM_LOADPROPERTY message when the form is loaded from a file. This property is similar to the PF_fSaveMsg, except that a PF_fSAVEPROPERTY

message is not generated, This allows controls to be backward compatible by loading properties in previous versions that are no longer supported in newer versions.

PF_fLoadDataOnly [2.0]
: Causes Visual Basic to load the property from a form file but prevents the property from being written back out. This is similar to the PF_fSAVEDATA except that property controls are not rewritten. This allows custom controls to be compatible with Visual Basic 1.0.

PF_fNoMultiSelect [2.0]
: Specifies that this property not be displayed when that control is part of a selected group.

PF_fNoInitDef
: Prevents Visual Basic from setting the property to the default during the control load. If PF_fDefVal is not set, this flag has no effect.

PF_fNoRunTimeR [2.0]
: Indicates that the property is write-only at run-time.

PF_fNoRunTimeW
: Indicates that the property is read-only at run-time.

PF_fNoShow
: Prevents the property from appearing in the Properties window. This is also the proper flag to add to the fl variable when retiring a property.

PF_fPreHwnd
: Causes the property to be loaded before the window is created. These properties generally affect the windows style and should be made in response to a WM_NCCREATE message.

PF_fPropArray
: Specifies that the property is an array. A data structure is required when getting and setting this property. The PF_fNoShow flag must be set when using this flag.

PF_fSaveData
: When the form is saved to a file Visual Basic gets its value from the control and reads its value from disk when loading the form.

PF_fSaveMsg
: Causes Visual Basic to issue a VBM_SAVEPROPERTY message when the form is saved and a VBM_LOADPROPERTY when the form is loaded.

PF_fSetCheck
: Cause Visual Basic to send a VBM_CHECKPROPERTY message before it sets the property value. This gives the control a chance to check the validity of the value before it is set.

PF_fSetData
: Visual Basic sets the value of the property by placing the data directly in the programmer defined structure.

PF_fSetMsg
: Causes Visual Basic to send a VBM_SETPROPERTY message when the user attempts to set the property. Either this flag or the PF_fSETData flag must be used. If both are used then the data is transferred before the message is sent.

PF_fUpdateOnEdit
: This flag causes the property value to be set each time a character is typed in the settings box of the Properties window. If this flag is not used, the property is not set until the change is committed.

A product generally goes through many versions from its early beginnings to the day it is retired.

During that lifetime properties can change. When you have new properties to add, by all means add them but **never, never ... never** remove the old ones. When a Visual Basic program is saved to disk, property values are saved with an ID equivalent to its index into your property table. If you add or delete from the middle of your table all subsequent properties will be thrown out of kilter. What this means to the developer is that upgrading from one version to another will be virtually impossible without throwing out the older forms and starting over. If you must retire a property you can always make it invisible to the user, **but leave it where it was first defined.** Visual Basic always sorted the property list into alphabetic order so where you place the property isn't critical.

In the next article we will cover how to handle each of the various data types. In particular we will discuss how to handle Visual Basic strings, verses pChar strings, verses Pascal strings, etc. Also the critical event processing paridgm which can cause a valid data pointer to become invalid from one use to the next.

Fred Hill is president of Micro System Solutions. His company produces DOS and Windows applications in Borland Pascal 7.0 and Visual Basic 3.0. He may be reached on Compuserve 76060,102