Creating Visual Basic VBXs:

The Event List

by Fred C. Hill

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

In this installment we'll explore then process by which the Visual Basic custom control allows the user to become an integral part of the operation of the control. The property list (See The Pascal Magazine, issue 2) provides a communication path but would be very unwieldy if it were the only way to manipulate a control. Imagine what your program would look like if you had to constantly monitor changes in the properties in order to know what your control was doing.

Declaring the Event

The Visual Basic Event List provides the additional level to allow the developer to interact on a real time basis. In addition to the Windows GUI, Windows 3.1 provides a true event processor. The current version of Windows still only provides cooperative multi-tasking and this is accomplished through the event processor.

When declaring an Event in the event list you're really only providing Visual Basic with the event name and its arguments. The actual processing behind the Event takes place in the control procedure.

In our project for this issue we will be converting the CIRC2 code provided in the Custom Control Development Kit (CDK) from Microsoft. This project is used to define how to add a custom event however we will be using it to explain both the standard and the custom events.

Event lists are declared in the same way as the Property lists. In our example we will be using an enumerated list and let the compiler determine the value assigned to the particular entry.

The Event Unit

```
unit event;
interface
uses
WinTypes,
vbapi_;

{ Event list
 Define the consecutive indices for the events
}
type
EVENT_Index = (
    EVENT_Circle_ClickIn,
    EVENT_Circle_ClickOut,
    EVENT_Circle_DRAGDROP,
    EVENT_Circle_DRAGOVER,
    EVENT_Circle_Last);
```

```
{Event procedure parameter prototypes}
const
ParamTypes_ClickIn:                    array[0..1]of word = (ET_R4, ET_R4);

EventClickInName:                      array[0..8] of Char = 'ClickIn'#0;
EventClickInParm:
    array[0..24] of char = 'X As Single, Y As Single'#0;

Event_ClickIn:
    tEVENTINFO   = (
        {----Name}
        npszName:                      tOffset(@EventClickInName);
        {----Number of parameters}
            cParms:          2;
        {----Number of words}
        cwParms:  4;
        {----Pointer to parm types}
        npParmTypes:                   tOffset(@ParmamTypes_ClickIn);
        {Pointer to Argument string}
        npszParmProf:                  tOffSet(@EventClickInParm);
        fl:             0
);

EventClickOutName:                     array[0..9] of Char = 'ClickOut'#0;

Event_ClickOut: tEVENTINFO   = (
    npszName:               tOffset(@EventClickOutName);
    cParms:         0;
    cwParms:        0;
    npParmTypes:            0;
    npszParmProf:           0;
    fl:             0
);

Circle_Events: array[EVENT_Index] of ofsPEVENTInfo = (
    ofsPEventInfo(@Event_ClickIn),
    ofsPEventInfo(@Event_ClickOut),
    PEVENTINFO_STD_DRAGDROP,
    PEVENTINFO_STD_DRAGOVER,
    0);

implementation

end.
```

Listing 1

In Listing 1 we first code the Event index. This index will be used later when defining the actual Event List (Circle_Events). Next we assign the types of parameters we will be passing between the VBX and Visual Basic. The Argument Type Flags are defined in the VB_API_ file provided on the disk that comes with this magazine as well as in list 2.

Argument Type Flags

Value	Description
ET_I2	16-bit signed integer
ET_I4	32-bit signed integer
ET_R4	4-byte real
ET_R8	8-byte real
ET_CY	8-byte currency
ET_HLSTR	String Strings are represented in the argument structure as a handle (HLSTR) rather than a pointer.

Listing 2

Next we assign the EVENTInfo structure. The first element in the structure is the Event Name (ClickIn) as it will appear in the Code window in Visual Basic. This is followed by the number of arguments being being passed and the number of computer words in the argument list. In the ClickIn event we will be passing 2 parameters consisting of 4 words. (Two ET_R4's or 2 4-byte reals.) Next come the pointer to the Argument Type array followed by the pointer to the Argument string. In C (or C++) this could be the actual string since C passed the address rather than the string. Finally we have the fl flag that can contain either a 0 or the EF_fNoUnload flag. When this flag is set Visual Basic will not allow unloading of the current form or any control on the form when the corresponding event procedure is being fired. This flag is primarily used for events that must assume that nothing is unloaded such as the Standard Paint event.

After defining the remainder of the Events we define the Event List itself. This list will be used in the model table to point Visual Basic to the array structures we just described.

Firing an Event

Firing the ClickIn Event causes the mouse current coordinates to be passed to the Visual Basic program.

Fire the ClickIn Event

```
{ TYPEDEF for parameters to the ClickIn event.
}
Type
tagCLICKINPARMS = record
    Y:    pointer;
    X:  pointer;
```

```
    Index: LPVoid;
end;


{ Fire the ClickIn event, passing the x,y coords of the click.}
procedure FireClickIn(Control: hctl; x, y: integer);
var
params:   tagClickInParms;
xTwips,
yTwips:   LongInt;
begin
{   float    xTwips, yTwips;}

xTwips := VBXPixelsToTwips(x);
yTwips := VBYPixelsToTwips(y);
params.X := @xTwips;
params.Y := @yTwips;
VBFireEvent(Control,                                ord(EVENT_CIRCLE_CLICKIN),
@params);
end;
```

Listing 3

The FireClickIn function converts the Windows X and Y coordinates from pixels to twips. (that the developer uses in the Visual Basic environment) To perform the pixel conversion the VBAPI_ routines **VBXPixelsToTwips** and **VBYPixelsToTwips** are called to ensure the coordinates are converted to logical twips that relate to the particular display device. Pointers to the converted variables are then passed to the argument structure and **VBFireEvent** is called.

The call to **VBFireEvent** does not return until the Visual Basic event procedure (assuming the developer coded one) returns. This ensures that all of the local variables in the control code are valid locations. If the event procedure changes any of the arguments then upon return the local variable will been replaced by Visual Basic.

```
function CircleCtlProc(Control: HCtl;
Wnd: Hwnd; Msg, wp: Word;
lp: LongInt):LongInt;
var
ps:     tPaintStruct;
LpCirc:    pCirc2;
hDcHold: hDc;
begin
case Msg of
    WM_NCCREATE:       begin
        LpCirc := VBDerefControl(Control);
        LpCirc^.CircleShape := 0;
        lpCirc^.FlashColor := 128;
```

```
        VBSetControlProperty(Control,                    ord(IPROP_Circle_BACKCOLOR),255);
    end;
    WM_LBUTTONDOWN,
    WM_LBUTTONDBLCLK:
    if (InCircle(Control, lp, HiWord(lp)))   then          begin
            hDcHold := GetDC(Wnd);
            FlashCircle(Control, hDcHold);
            ReleaseDC(Wnd, hDcHold);
                {---pass the mouse coord from the high and low words of lp)
FireClickIn(Control,lp,HiWord(lp));
        end else FireClickOut(Control);
    WM_LBUTTONUP:
        if (InCircle(Control, lp, HIWORD(lp)))                    then begin
            hDcHold := GetDC(Wnd);
            PaintCircle(Control, Wnd,                                  hDcHold);
            ReleaseDC(Wnd, hDcHold);
        end;
    WM_PAINT:
        if (wP <> 0) then
            PaintCircle(Control, Wnd, wP)
        else begin
            BeginPaint(Wnd, ps);
            paintCircle(Control, Wnd,                                  ps.hdc);
            EndPaint(Wnd, ps);
        end;
    WM_SIZE:   RecalcArea(Control, Wnd);
    VBM_SETPROPERTY:
        case wP of
            ord(IPROP_Circle_Shape): begin
                lpCirc :=                                          VBDerefControl(Control);
                lpCirc^.CircleShape := lp;
                RecalcArea(Control, Wnd);
                InvalidateRect(Wnd, nil,                                        true);
                CircleCtlProc := 0;
                exit;
    end;   end;   end;
CircleCtlProc := VBDefControlProc(Control,              Wnd, Msg, wP, lP);
end;
end.
```

Listing 4

The Circle control procedure fires the ClickIn event whenever the user clicks the mouse button while in the circle. The InCircle routine (see the code included on the disk) determines if the current mouse coordinates are within the circle and then the circle color is flashed and the FireClickIn procedure calls the VBFireEvent function.

I have just described how a custom event gets processed by the Visual Basic environment. A standard event is handled in much the same way except that the VB environment handles them automatically for you. In our event list we included the DRAGDROP and the DRAGOVER events that are two of the standard events provided for in Visual Basic. (See list 6)

Standard Events

Visual Basic provides 18 standard events. These events only have to be declared in the EVENTINFO table. The default control procedure **VBDefControlProc** will normally fire the standard events in response to the various Windows messages however if you wish you may fire a standard event in the same way you fire your own custom events.

Event Description

Click PEVENTINFO_STD_CLICK
WM_LBUTTONUP message. Fired when the control has captured the mouse. The standard MouseDown and MouseUp events occur before this event.

DblClick PEVENTINFO_STD_DBLCLICK
WM_LBUTTONDBLCLK message. Similar to Click.

DragDrop PEVENTINFO_STD_DRAGDROP
Fired after a VBM_DRAGDROP message is received.

DragOver PEVENTINFO_STD_DRAGOVER
Fired after a VBM_DRAGOVER message is received.

GotFocus PEVENTINFO_STD_GOTFOCUS
After receiving a WM_GOTFOCUS message a VBM_FIREEVENT message is posted. This, in effect, places the event in the normal queue and allows pending messages to be processed.

KeyDown PEVENTINFO_STD_KEYDOWN
Fired when a WM_KEYDOWN or a WM_SYSKEYDOWN message is received.

KeyPress PEVENTINFO_STD_KEYPRESS
Fired when a WM_CHAR message is received.

KeyUp PEVENTINFO_STD_KEYUP
Fired when a WM_KEYUP or a WM_SYSKEYUP message is received.

LostFocus PEVENTINFO_STD_LOSTFOCUS
Fired when a WM_LOSTFOCUS message is received. The same delay process as GotFocus is used here to ensure messages get processed in sequence.

MouseDown PEVENTINFO_STD_MOUSEDOWN
Fired if any BUTTONDOWN (left, right or middle) message is received. When this event is fired the mouse is captured by the control.

MouseMove PEVENTINFO_STD_MOUSEMOVE
Fired when a WM_MOUSEMOVE message is received.

MouseUp PEVENTINFO_STD_MOUSEUP
Fired if and BUTTONDOWN (left, right or middle) message is received. The mouse capture is released by this event.

Listing 5

Events unique to VB 2.0

These standard events are included for information only. See the CDK documentation for more information about them.

Last	PEVENTINFO_STD_LAST
LinkClose	PEVENTINFO_STD_LINKCLOSE
LinkError	PEVENTINFO_STD_LINKERROR
LinkNotify	PEVENTINFO_STD_LINKNOTIFY
LinkOpen	PEVENTINFO_STD_LINKOPEN
None	PEVENTINFO_STD_NONE

None is used as a placeholder for events you wish to remove. Using this placeholder allows applications to use different versions of a control with less likelyhood of a Visual Basic abort during load.

Listing 6

Reference: Professional Features Book 1, Microsoft Visual Basic 3.0 Custom Control Guide

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