EasyVENT

Release version 3.2 for Windows.

General.

EasyVENT is a Purebasic library (written for Windows only) which allows developers to easily attach event handlers to windows/controls/menus etc. for a predefined set of Windows events.

For example, attaching an event handler for a button's click event is accomplished by issuing the command:

Result = ***SetEventHandler****(hWnd, #OnButtonClick, @MyButtonHandler())*

where hWnd is the handle of the button gadget, and MyButtonHandler() is the name of a function within your code which acts as the event handler. That is, whenever a user clicks the button in question, the function MyButtonHandler() will run.

A non-zero return means there was no error in registering the handler.

The list of events covered by this library will increase with each release. The current list is detailed below.

NOTE that the library is currently only available for Purebasic 4 for Windows and is in the form of a source code include file.

Package, licence and terms of use.

This package contains all the EasyVENT source files, various demo programs (as appropriate) and this user guide.

The software contained within this package is free to use in any project (commercial or otherwise) or as a learning tool. I do, however, assert my moral right to be identified as the creator of this software (except where acknowledgements are given) and thus ask that due acknowledgement is given within any product/creation in which my source code forms a part. Use the software for any purpose whatsoever.

The EasyVENT software is provided on an *as is* basis, with no warranty either given or implied, meaning that I am not liable for any damage caused by its use (or misuse!) nor by damage caused by other programs based on its source code.

Acknowledgements.

Thanks to the following:

Timo *'Freak'* Harter for his excellent OLE drag and drop library which version 2.0 of EasyVENT makes use of.

netmaestro for pointing me in the right direction when I couldn't see two feet in front of myself!

ebs, kiffi and *Geotrail* for being the first to *risk* this library!

netmaestro (again!) for his work on the *#OnCursorEnter* and *#OnCursorExit* events. This area is now far far superior to the alpha versions of EasyVENT.

DoubleDutch for continued testing of EasyVENT 3.

To use EasyVENT within a Purebasic program.

Simply ensure that the following command resides within your Purebasic source code file before any attempt is made to attach an event handler to a window/control:

```
XincludeFile "EasyVENT.pbi"
```

You can now attach event handlers to either the window in question or any of its' gadgets or menus etc.

That's it.

Event handler functions.

Registering one of your own functions as an event handler for one or more events is simple. Use the command:

```
Result = SetEventHandler(hWnd, Event, @MyEventHandler(), [commanditem])
```

hWnd is the windows handle of the window/control to which the handler is to be attached.

Event is one of the intrinsic event constants (such as *#OnButtonClick*) which details the particular event and which are described in detail below.

commanditem is an optional parameter used only when attaching event handlers to menu items / toolbar buttons.

The function: ***RemoveEventHandler****(hWnd, Event, [commanditem])*

will remove a handler from the specified window/control's list of registered handlers. Again, *commanditem* is optional and only used when removing event handlers from menu items / toolbar buttons.

The function: ***PerformDefaultWinProcessing****(*sender.PB_Sender)*

allows an event handler to instruct the EasyVENT library to itself instruct Windows to perform the default processing associated with the underlying Windows message (if any).

See the section on 'Performing default Windows processing' for more details. [1]

A 'skeleton' event handler function.

Each event handler function that you use should assume the following form.

```
Procedure.l MyEventHandler(*sender.PB_Sender)

  ...... {Your code}

  ProcedureReturn result
EndProcedure
```

---

[1] This function is new for EasyVENT 3.0.0.

The PB_Sender structure is intended to provide information useful in the processing of the underlying event and is defined in the "EventModuleResident.pbi" source file as follows:

```
Structure PB_Sender
  hWnd.l
  message.l
  mouseX.l
  mouseY.l
  button.l
  item.l
  state.l
  text$
  originalmessage.l
  uMsg.l
  wParam.l
  lParam.l
EndStructure        ( + three additional fields which are reserved for internal use.)
```

Individual events shipped by EasyVENT will use a selection of these fields only (as detailed in the section describing each event), but each field generally serves a specific function as detailed:

hWnd	the windows handle of the window/control to which the event concerns.
message	one of the intrinsic *event constants* listed below.
mouseX	the x-coordinate of the mouse cursor at the time the event occurred. This is usually in client coordinates; except for the non-client mouse events, in which case this will be in screen coordinates.
mouseY	the y-coordinate of the mouse cursor at the time the event occurred. This is usually in client coordinates; except for the non-client mouse events, in which case this will be in screen coordinates.
button	for click events, this indicates which mouse button was clicked etc. It will assume one of the following values: *#EVENT_LEFTBUTTON* *#EVENT_MIDDLEBUTTON* *#EVENT_RIGHTBUTTON*
item	for certain events\gadgets this indicates which item is being affected etc.
state	for certain events\gadgets this provides extra information.
text$	for certain events\gadgets this provides extra information.
originalmessage	only used for the *#OnUnhandledWinMessage* event.

The remaining fields; uMsg, wParam, lParam are relevant only when the underlying event corresponds directly to a windows message (not all events do!) and contain the ubiquitous windows 'message values' (this is for more experienced programmers only).

Some events will allow you to change these values in order to affect changes to the corresponding windows message processing. You can even change the value of uMsg to change the actual windows message (useful for mouse messages).

Take care when changing these values!

You should exit your event handler with the statement:

> ***ProcedureReturn*** *result*

where, for those events in which a return value is important and **in all cases but one**, *result* assumes one of the following values:

#Event_ReturnDefault (default)

> instructs EasyVENT to return the result of performing the underlying Windows processing for this message (if any). This depends on the developer's event handler first calling the function ***PerformDefaultWinProcessing****()* (see below). If this function is not called then returning *#Event_ReturnDefault* defaults to *#Event_ReturnTrue*.

#Event_ReturnTrue

> instructs EasyVENT to return a value to Windows (depending on the particular event) which would be interpreted as 'proceed' or 'true' or 'yes' or 'valid' etc. depending upon the context and nature of the particular event. You will need to see the description of individual events to see exactly how this value is interpreted and utilised etc. For example, for the event *#OnItemSelecting*, a return value of *#Event_ReturnTrue* will permit the underlying selection to proceed.

#Event_ReturnFalse

> instructs EasyVENT to return a value to Windows (depending on the particular event) which would be interpreted as 'cancel' or 'false' or 'no' or 'invalid' etc. depending upon the context and nature of the particular event. You will need to see the description of individual events to see exactly how this value is interpreted and utilised etc. For example, for the event *#OnItemSelecting*, a return value of *#Event_ReturnFalse* will prevent the underlying selection from proceeding.

The exception to this is any handler attached to the *#OnUnhandledWinMessage* event in which case the value returned from the handler is passed directly back to Windows. [2]

Performing default Windows processing.

Many (but not all) EasyVENT events correspond to Windows messages. For example the event *#OnClose* directly corresponds to the Windows message #WM_CLOSE.

In such cases it is often desirable (and sometimes necessary) to allow Windows to perform it's '*default*' processing for this message.

A *classic* example will be the event *#OnMouseDown* (corresponding to the various mouse button messages; #WM_LBUTTONDOWN etc.) in which most handlers will require

---

[2] The *#OnANYevent* and *#OnUnhandledWinMessage* events require rather special attention and so the reader is advised to read the sections detailing these events very carefully.

Windows to first perform it's default processing before the handler adds it's own processing.

For this, there is now (introduced in EasyVENT 3.0.0) the command :

PerformDefaultWinProcessing*(*sender.PB_Sender)*

which can be called at any time during the execution of your event handler.
This way you have complete control over when (if ever) Windows gets to perform it's default processing etc.

The return from this function is that which Windows itself returns and is really only important for the *#OnUnhandledWinMessage* event. For all other events, returning the value *#Event_ReturnDefault* from your handler will ensure that the value returned from ***PerformDefaultWinProcessing****()* is passed along as appropriate.

NOTES.

i) In earlier versions of EasyVENT (prior to 3.0.0) Windows could only perform such default processing **after** an event handler had finished executing. Whilst convenient, this did restrict the EasyVENT library somewhat and led to a few inconsistencies (particularly with keyboard and mouse events).

ii) Having complete freedom over when (and if) such default processing is to occur does mean that things are a little more complex now.

The problem is that the developer now has to decide whether default processing is to occur and, more importantly perhaps, when?

Do you call ***PerformDefaultWinProcessing****()* at the beginning of your handler, or at the end?

Experienced Windows programmers will know just how important this can be for certain messages and how much difference changing this order can make to a piece of code!

If in doubt, position a call to ***PerformDefaultWinProcessing****()* at the end of your handler code, at least in the first instance. If this doesn't achieve the desired results, then reposition it at the beginning of your handler etc.

iii) So which events can, or should, you not include a call to ***PerformDefaultWinProcessing****()* ?

Basically any event where you do not wish any default processing to be undertaken!

Not including a call to ***PerformDefaultWinProcessing****()* can lead to problems with certain events, whilst with others it will make no difference at all.

For example, a *#OnKeyDown* handler could prevent presses of the return key from registering with a gadget by calling ***PerformDefaultWinProcessing****()* only if the user did not press the return key etc.

Indeed ***PerformDefaultWinProcessing()*** comes into it's own with the keyboard and mouse events in particular.

If an event corresponds to a *notification* sent by Windows (e.g. *#OnCollapseExpandSelection*) then there is usually little point in calling ***PerformDefaultWinProcessing()***. In such cases, the value you return from your event handler may or may not be important (depending on the event concerned).

See the demo programs for examples of how to structure your event handlers and to get an idea of when to call the ***PerformDefaultWinProcessing()*** function.

Current list of supported events.

The following list details the events currently supported and, where appropriate, accompanying notes.

The events detailed are used with the commands:

SetEventHandler()
and ***RemoveEventHandler***()

as detailed above.

#OnANYevent

This is a new addition to EasyVENT[3] and a discussion of this event is left to the section

Two specialised events.

#OnButtonClick

This event fires whenever the user clicks a button registered as receiving such events.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the button.
button	*#EVENT_LEFTBUTTON*
item	the button ID.

This event corresponds to the Windows **command** message #BN_CLICKED.

The return value is unimportant as this is just a notification. You need not call ***PerformDefaultWinProcessing()*** either.

#OnChange

This event fires whenever the user has taken some action which may have altered text in a string gadget or an editor gadget. E.g. entered text, pasted text etc.

Probably more useful with an editor (rich edit) gadget.

The values of the fields of the *sender parameter of interest are as follows:

---

[3] Introduced in EasyVENT 3.0.1.

hWnd	the windows handle of the control.

This event corresponds to the Windows **command** message #EN_CHANGE.

The return value is unimportant as this is just a notification. You need not call ***PerformDefaultWinProcessing()*** either.

#OnClose
This event fires whenever the user clicks the close button on the registered window.

This event corresponds to the Windows message #WM_CLOSE.

The return value is not important in that *#Event_ReturnTrue* and *#Event_ReturnFalse* have the same effect.
What is important is whether you allow Windows to perform the default processing by calling ***PerformDefaultWinProcessing()*** and then you will probably return a value of *#Event_ReturnDefault*.

#OnCollapseExpandSelection **Tree Gadgets and explorer tree gadgets only.**
This event fires whenever the user attempts to collapse or expand a parent node's list of child items.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the control.
item	the item number of the parent node.
state	*#EVENT_COLLAPSE* or *#EVENT_EXPAND* as appropriate.

This event corresponds to the Windows **notify** message #TVN_ITEMEXPANDING.

Return *#Event_ReturnFalse* from your event handler to prevent the parent node's list of child items being collapsed or expanded. Return *#Event_ReturnTrue* or *#Event_ReturnDefault* otherwise. You need not call ***PerformDefaultWinProcessing()***.

#OnContextMenuPopup
This event fires whenever the user attempts to right-click the registered control or otherwise invoke an associated popup menu.

This event can be used to prevent such a context menu from appearing (such as those belonging to string gadgets) or indeed can be used to replace the default menu with a custom one etc.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the control.
button	*#EVENT_RIGHTBUTTON*
lParam	the position of the cursor (in screen coordinates). The low-word holds the horizontal position, the high-word holds the vertical position.

This event can now be used for the edit control attached to an **editable ComboBox gadget**. Attach the appropriate event handler to the ComboBox gadget as usual.
This event corresponds to the Windows message #WM_CONTEXTMENU.

The return value is not important in that *#Event_ReturnTrue* and *#Event_ReturnFalse* have the same effect.
To prevent a default menu from appearing with certain controls simply do not call ***PerformDefaultWinProcessing()*** etc. In such cases you can display your own popup menu.

#OnCursorEnter

This event fires whenever the mouse cursor enters the **client area** of the control/window.[4]

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the control.

The return value is ignored.

#OnCursorExit

This event fires whenever the mouse cursor leaves the control/window.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the control.

The return value is ignored.

#OnDblClick

This event fires whenever the user double clicks a mouse button within either the client or non-client areas of the control/window registered as receiving such events.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the control.
button	*#EVENT_LEFTBUTTON* or *#EVENT_MIDDLEBUTTON* or *#EVENT_RIGHTBUTTON* as appropriate.
state	one of the values *#EVENT_CLIENT* or *#EVENT_NONCLIENT* as appropriate.

NOTE, the mouseX and mouseY fields of the *sender parameter will be in client coordinates or screen coordinates depending on whether this is a client or non-client event (just check the value of the state field.)

This event corresponds to the Windows messages #WM_LBUTTONDBLCLK, #WM_NCLBUTTONDBLCLK, #WM_MBUTTONDBLCLK, #WM_NCMBUTTONDBLCLK, #WM_RBUTTONDBLCLK and #WM_NCRBUTTONDBLCLK as appropriate.

The return value is not important in that *#Event_ReturnTrue* and *#Event_ReturnFalse* have the same effect.
What is important is whether you allows Windows to perform the default processing by calling ***PerformDefaultWinProcessing()*** and then you will probably return a value of *#Event_ReturnDefault*.

---

[4] I may add non-client support at a later date.

#OnDragDrop
This event fires whenever the user drops files onto a control/window registered with this event.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the control/window.
button	*#EVENT_LEFTBUTTON*
item	the number of filenames dropped.
wParam	the windows handle of the underlying drop structure which will be required when retrieving the dropped filenames (win api call will be required for this).

NOTE for experienced Windows programmers. When the event handler exits, the EasyVENT library automatically calls the DragFinish_() function to free up the internal memory used etc. Thus the application should not call this function itself.

This event corresponds to the Windows message #WM_DROPFILES.

The return value is unimportant. You need not call ***PerformDefaultWinProcessing**()* either.

#OnDragItemStart	See the separate section on drag/drop items for this event.
#OnDropItem	See the separate section on drag/drop items for this event.

#OnEditTreeLabels **Tree Gadgets and explorer tree gadgets only.**
This event fires whenever the user attempts to edit an item's label and also when the user has completed any subsequent editing.

The first call occurs when the user attempts to edit an item's label. In this case the values of the fields of the *sender parameter are as follows:

hWnd	the windows handle of the control.
item	the item number of the label which the user is attempting to edit.
state	#EVENT_*BEGINLABELEDIT*

Return *#Event_ReturnFalse* from your event handler to prohibit the label from being edited. Return *#Event_ReturnTrue* or *#Event_ReturnDefault* otherwise. You need not call ***PerformDefaultWinProcessing**()*.

The second call occurs when the user has finished editing an item's label. In this case the values of the fields of the *sender parameter are as follows:

hWnd	the windows handle of the control.
item	the item number of the label which the user is editing.
state	#EVENT_*ENDLABELEDIT*
text$	contains a copy of the new text which the user has entered.

Return *#Event_ReturnFalse* from your event handler to restore the label to its original content. Return *#Event_ReturnTrue* or *#Event_ReturnDefault* otherwise. You need not call ***PerformDefaultWinProcessing**()*.

This event corresponds to the Windows **notify** messages #TVN_BEGINLABELEDIT and #TVN_ENDLABELEDIT

#OnErase

This event fires whenever the control/window background requires erasing prior to painting etc. By using this event, you can take charge of many aspects of painting such as '*owner draw*', reducing flicker etc.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the control.
wParam	identifies the device context (hdc) which can be used for drawing.

This event corresponds to the Windows message #WM_ ERASEBKGND.

If you call ***PerformDefaultWinProcessing()*** then you will likely return *#Event_ReturnDefault*. Otherwise return *#Event_ReturnTrue* to inform Windows that you did not erase the background, *#Event_ReturnFalse* if you did. (This can have a noticeable effect if XP themes are disabled.)

#OnGotFocus

This event fires whenever the registered control/window receives the keyboard focus.

This event corresponds to the Windows message #WM_SETFOCUS.

The return value is unimportant. You will need to consider calling ***PerformDefaultWinProcessing()*** however, especially if the underlying control displays a caret.

#OnItemCheckboxChanging

This event fires when the user attempts to alter a checkbox in an item of a control which supports this event (currently ListIcon gadgets and Tree gadgets) but is sent **before** the checkbox has been altered.

The item field of the *sender parameter indicates which item's checkbox the user is attempting to change (either through clicking the mouse or through the keyboard use etc.)

The state field of the *sender parameter contains one of the values :

#EVENT_CHECKING
#EVENT_UNCHECKING

which denotes the action attempted by the user.

Return *#Event_ReturnFalse* from your event handler to prevent the alteration. Return *#Event_ReturnTrue* or *#Event_ReturnDefault* otherwise. You need not call ***PerformDefaultWinProcessing()***.

NOTE, as of Purebasic 4.4 (XP themes enabled) a double click of a tree gadget's checkbox (have not tested ListIcons, but I presume the same is true) will toggle the checkbox's state. EasyVENT does not check this by default and so if wishing to prevent such action, you may need to utilise a #OnDblClick handler as well in order to prevent the default processing etc.

#OnItemSelected

This event fires **after** the user selects a new item in a registered gadget. The following gadgets are supported:

ComboBoxGadget	ExplorerComboGadget
ListViewGadget	
ListIconGadget	ExplorerListGadget
TreeGadget	ExplorerTreeGadget
PanelGadget	

NOTES.

i) For ListIcon gadgets and ExplorerList Gadgets, this event fires whenever items are deselected as well as selected **or** (in the case that the ListIcon has checkboxes displayed in the first column), the user checks or clears one of the checkboxes. This of course means that this event can fire multiple times for a single user action. In these cases the item field of the *sender parameter indicates which item is being (de)selected and the state field assumes one of the following values:

#EVENT_DESELECT
#EVENT_SELECT

ii) For Tree gadgets and ExplorerTree gadgets, the item field of the *sender parameter indicates which item has been selected and the state field contains one of the following values to indicate how the item was selected :

#EVENT_UNKNOWN
#EVENT_MOUSE
#EVENT_KEYBOARD

iii) For Panel gadgets, this event fires whenever the user selects a new tab. The item field of the *sender parameter indicates which tab the user has selected and the state field contains one of the following values to indicate how the item was selected :

#EVENT_MOUSE
#EVENT_KEYBOARD

The return value is unimportant as this is just a notification. You need not call ***PerformDefaultWinProcessing****()* either.

#OnItemSelecting

This event fires when the user selects a new item in a registered gadget but **before** the system changes the selection. This gives the developer the opportunity to refuse the selection. The following gadgets are supported:

ListIconGadget	ExplorerListGadget

TreeGadget ExplorerTreeGadget

PanelGadget

The item field of the *sender parameter indicates which item is being selected.

NOTES.

i) For ListIcon gadgets and ExplorerList Gadgets, this event fires whenever items are deselected as well as selected **or** (in the case that the ListIcon has checkboxes displayed in the first column), the user checks or clears one of the checkboxes. This of course means that this event can fire multiple times for a single user action. In these cases the item field of the *sender parameter indicates which item is being (de)selected and the state field assumes one of the following values:
#EVENT_DESELECT
#EVENT_SELECT

ii) For Tree gadgets and ExplorerTree gadgets, the item field of the *sender parameter indicates which item is being selected and the state field contains one of the following values to indicate how the item was selected :
#EVENT_UNKNOWN
#EVENT_MOUSE
#EVENT_KEYBOARD

iii) For Panel gadgets, this event fires whenever the item field of the *sender parameter indicates which tab the user is attempting to select and the state field contains one of the following values to indicate how the item was selected :
#EVENT_MOUSE
#EVENT_KEYBOARD

Return *#Event_ReturnFalse* from your event handler to prevent the selection. Return *#Event_ReturnTrue* or *#Event_ReturnDefault* otherwise. You need not call ***PerformDefaultWinProcessing()***.

#OnKeyDown
This event fires whenever a non-system key has been pushed and the underlying control/window has been registered for this event etc.

Useful for intercepting #VK_DEL key presses.
The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the control/window.
wParam	the virtual key code of the key.

This event can now be used for the edit control attached to an **editable ComboBox gadget**. Attach the appropriate event handler to the ComboBox gadget as usual (although when this event fires, *sender\hWnd will contain the handle of it's edit field). It also works for DateGadgets.

This event corresponds to the Windows message #WM_KEYDOWN.

The return value is not important in that *#Event_ReturnTrue* and *#Event_ReturnFalse* have the same effect.
What is important is whether you allow Windows to perform the default processing by calling ***PerformDefaultWinProcessing****()* and then you will probably return a value of *#Event_ReturnDefault*.

#OnKeyPress
This event fires whenever the user presses a key on the keyboard and the underlying control/window has been registered for this event etc.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the control/window.
wParam	the character code of the key.

This event can now be used for the edit control attached to an **editable ComboBox gadget**. Attach the appropriate event handler to the ComboBox gadget as usual (although when this event fires, *sender\hWnd will contain the handle of it's edit field). It also works for DateGadgets.
This event corresponds to the Windows message #WM_CHAR and is typically used to prevent certain keys from registering with a control or even modifying the keys pressed; e.g. by capitalising all characters entered at the keyboard.

The return value is not important in that *#Event_ReturnTrue* and *#Event_ReturnFalse* have the same effect.
What is important is whether you allow Windows to perform the default processing by calling ***PerformDefaultWinProcessing****()* and then you will probably return a value of *#Event_ReturnDefault*. Without default processing the characters pressed will not register with the underlying control.

#OnKeyUp
This event fires whenever a non-system key has been released and the underlying control/window has been registered for this event etc.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the control/window.
wParam	the virtual key code of the key.

This event can now be used for the edit control attached to an **editable ComboBox gadget**. Attach the appropriate event handler to the ComboBox gadget as usual (although when this event fires, *sender\hWnd will contain the handle of it's edit field). It also works for DateGadgets.

This event corresponds to the Windows message #WM_KEYUP.

The return value is not important in that *#Event_ReturnTrue* and *#Event_ReturnFalse* have the same effect.
What is important is whether you allow Windows to perform the default processing by calling ***PerformDefaultWinProcessing****()* and then you will probably return a value of *#Event_ReturnDefault*.

#OnLinkClick

This event fires whenever the user clicks a link within an editor gadget registered as receiving such events.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the editor gadget.
button	*#EVENT_LEFTBUTTON* or *#EVENT_RIGHTBUTTON* as appropriate.
item	the index (zero based) of the first character of the link text within the text of the editor gadget.
state	the index (zero based) of the last character of the link text within the text of the editor gadget.
text$	the full text of the link clicked.

To mark selected text within an editor gadget as being a 'link', the developer has two choices. Either, manually set the #CFE_LINK style bit of the selected text (which may not conform to usual www…. URL's etc.) and highlight accordingly, or, to enable the editor gadget to automatically detect and highlight URL's within the text of the control, send the #EM_AUTOURLDETECT message to the editor gadget (EasyVENT does not send this message automatically).

Both of these will result in the *#OnLinkClick* message being sent by EasyVENT to your event handler whenever such a 'link' is clicked.

NOTE, automatic detection cannot be used together with the manual process of setting the #CFE_LINK style bit etc. Indeed, switching automatic detection on has the effect of continually clearing the #CFE_LINK style throughout the text.

This event corresponds to the Windows **notify** message #EN_LINK.

The return value is unimportant as this is just a notification. You need not call ***PerformDefaultWinProcessing****()* either.

#OnLostFocus

This event fires whenever the registered control/window loses the keyboard focus.

This event corresponds to the Windows message #WM_KILLFOCUS.

The return value is unimportant. You will need to consider calling ***PerformDefaultWinProcessing****()* however, especially if the underlying control displays a caret.

#OnMaximize

This event fires whenever the user is attempting to maximize the registered window.

This event corresponds to the Windows **syscommand** message #SC_MAXIMIZE.

Calling ***PerformDefaultWinProcessing****()* within your handler will allow the maximizing to proceed. Otherwise, to prevent the maxmizing operation, simply do not call this function.

#OnMenuItemSelect

This event fires whenever the user clicks a menu item / toolbar button registered as receiving such events.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the parent window.
button	*#EVENT_LEFTBUTTON*
item	the menu item /toolbar button ID.

This message is registered on a item by item basis; that is you can associate different handlers with different menu item / toolbar button. You can also, of course, associate more than one menu item / toolbar button to the same handler etc.

For example, the command

Result = SetEventHandler(hWnd, #OnSelectMenuItem, @MenuItemHandler(), 3)

will associate the particular event handler detailed with the menu item / toolbar button with ID 3. In this case *hWnd* must be the handle of the **main window** containing the menu or toolbar etc. (as opposed to the handle of the toolbar!)

The return value is unimportant as this is just a notification. You need not call ***PerformDefaultWinProcessing****()* either.

#OnMinimize

This event fires whenever the user is attempting to minimize the registered window.

This event corresponds to the Windows **syscommand** message #SC_MINIMIZE.

Calling ***PerformDefaultWinProcessing****()* within your handler will allow the minimizing to proceed. Otherwise, to prevent the minmizing operation, simply do not call this function.

#OnMouseDown

This event fires whenever the user clicks a mouse button within either the client or non-client areas of the control/window registered as receiving such events.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the control/window.
button	*#EVENT_LEFTBUTTON* or *#EVENT_MIDDLEBUTTON* or *#EVENT_RIGHTBUTTON* as appropriate.
state	one of the values *#EVENT_CLIENT* or *#EVENT_NONCLIENT* as appropriate.

NOTE, the mouseX and mouseY fields of the *sender parameter will be in client coordinates or screen coordinates depending on whether this is a client or non-client event (just check the value of the state field.)

This event corresponds to the Windows messages #WM_LBUTTONDOWN, #WM_NCLBUTTONDOWN, #WM_MBUTTONDOWN, #WM_NCMBUTTONDOWN, #WM_RBUTTONDOWN and #WM_NCRBUTTONDOWN as appropriate.

The return value is not important in that *#Event_ReturnTrue* and *#Event_ReturnFalse* have the same effect.

What is important is whether you allow Windows to perform the default processing by calling ***PerformDefaultWinProcessing****()* and then you will probably return a value of *#Event_ReturnDefault*.

#OnMouseOver

This event fires whenever the cursor moves within either the client or non-client areas of the control/window registered as receiving such events.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the control/window.
state	one of the values *#EVENT_CLIENT* or *#EVENT_NONCLIENT* as appropriate.

NOTE, the mouseX and mouseY fields of the *sender parameter will be in client coordinates or screen coordinates depending on whether this is a client or non-client event (just check the value of the state field.)

This event corresponds to the Windows message #WM_MOUSEMOVE, #WM_NCMOUSEDOWN as appropriate.

The return value is not important in that *#Event_ReturnTrue* and *#Event_ReturnFalse* have the same effect.
What is important is whether you allow Windows to perform the default processing by calling ***PerformDefaultWinProcessing****()* and then you will probably return a value of *#Event_ReturnDefault*.

#OnMouseUp

This event fires whenever the user releases a mouse button within either the client r non-client areas of the control/window registered as receiving such events.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the control/window.
button	*#EVENT_LEFTBUTTON* or *#EVENT_MIDDLEBUTTON* or *#EVENT_RIGHTBUTTON* as appropriate.
state	one of the values *#EVENT_CLIENT* or *#EVENT_NONCLIENT* as appropriate.

NOTE, the mouseX and mouseY fields of the *sender parameter will be in client coordinates or screen coordinates depending on whether this is a client or non-client event (just check the value of the state field.)

This event corresponds to the Windows messages #WM_LBUTTONUP, #WM_NCLBUTTONUP, #WM_MBUTTONUP, #WM_NCMBUTTONUP, #WM_RBUTTONUP and #WM_NCRBUTTONUP as appropriate.

The return value is not important in that *#Event_ReturnTrue* and *#Event_ReturnFalse* have the same effect.
What is important is whether you allow Windows to perform the default processing by calling ***PerformDefaultWinProcessing****()* and then you will probably return a value of *#Event_ReturnDefault*.

#OnMove

This event fires whenever the registered window is moved.

This event corresponds to the Windows message #WM_MOVE.

The return value is unimportant as this is just a notification. You need not call ***PerformDefaultWinProcessing()*** either.

#OnPaint

This event fires whenever windows or an application makes a request for part of the registered control/window to be repainted, including the non-client area (as of EasyVENT 3.0.2). You might use this event, for example, to perform any drawing operations directly onto the window etc.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the control/window.
uMsg	one of the Window's constants #WM_PAINT or #WM_NCPAINT as appropriate.

This event corresponds to the Windows messages #WM_PAINT and #WM_NCPAINT. Return either *#Event_ReturnDefault*, or *#Event_ReturnFalse* depending on whether you call ***PerformDefaultWinProcessing()*** or not.

#OnResize

This event fires whenever the registered window is resized.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the control/window.
lParam	the new width of the client area in its low order word and the height in the high order word.

This event corresponds to the Windows message #WM_SIZE.

The return value is unimportant as this is just a notification. You may need to call ***PerformDefaultWinProcessing()*** however.

#OnScroll

This event fires either when the user scrolls a control/window's standard scrollbar (horizontal or vertical) or scrolls a ScrollBar gadget or when a Spin gadget is clicked or when the user moves the slider of a TrackBar gadget.

In all these cases, some of the fields of the *sender parameter are particularly important as shown in the following table:

Field	Standard control/window	ScrollBar gadget	Spin gadget	Trackbar gadget
hWnd	handle of control/window	handle of ScrollBar gadget	handle of Spin gadget	handle of TrackBar gadget
item	scroll box position	scroll box position	value of Spin gadget	value of TrackBar gadget
state	scroll bar code*	scroll bar code*	scroll bar code*	scroll bar code*
uMsg	#WM_HSCROLL or #WM_VSCROLL	#WM_HSCROLL or #WM_VSCROLL	#WM_VSCROLL	#WM_HSCROLL or #WM_VSCROLL

*The '*scroll bar code*' assumes one of the values:

#SB_BOTTOM
#SB_LINEUP
#SB_THUMBPOSITION
#SB_ENDSCROLL
#SB_PAGEDOWN
#SB_THUMBTRACK
#SB_LINEDOWN
#SB_PAGEUP
#SB_TOP

of which #SB_ENDSCROLL is probably the most important as it informs us when the scrolling operation is at an end.
This event corresponds to the Windows messages #WM_HSCROLL and #WM_VSCROLL as appropriate.

The return value is not important in that *#Event_ReturnTrue* and *#Event_ReturnFalse* have the same effect.
What is important (except for spin and trackbar gadgets) is whether you allow Windows to perform the default processing by calling ***PerformDefaultWinProcessing***() and then you will probably return a value of *#Event_ReturnDefault*.

#OnSetCursor

This event fires whenever the mouse causes the cursor to move within a window and mouse input is not captured.

This is useful because the message is sent first to the parent window and thus gives the parent window control over the cursor's setting in a child window etc. For example, you can use this event to temporarily change the mouse cursor without first setting the class cursor to null etc.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the control/window.
wParam	the handle of the window containing the cursor.

This event corresponds to the Windows message #WM_SETCURSOR.

Return *#Event_ReturnFalse* to prevent a child of the window/control receiving the current event, from receiving an *#OnSetCursor* event itself. Otherwise return *#Event_ReturnTrue*.

Use ***PerformDefaultWinProcessing***() if appropriate.

#OnSizingWindow

The event fires whenever the size or position of a registered window is about to change. An application can use this event to override the window's default maximized size and position, or its default minimum or maximum tracking size, thus restricting a user's ability to size the window etc.

The values of the fields of the *sender parameter of interest are as follows:

hWnd	the windows handle of the registered window.
lParam	a pointer to a Windows MINMAXINFO structure.

The MINMAXINFO structure contains information about a window's maximized size and position and its minimum and maximum tracking size and consists of the following fields:

```
POINT ptReserved
POINT ptMaxSize
POINT ptMaxPosition
POINT ptMinTrackSize
POINT ptMaxTrackSize
```

(See the Windows help file for more information).

As an example, the following event handler effectively prohibits a user from sizing the underlying window in such a way that the width is less than 400 pixels or the height is less than 300 pixels:

```
Procedure.l OnSizingWindow(*sender.PB_Sender)
  Protected *pMinMax.MINMAXINFO
  *pMinMax.MINMAXINFO=*sender\lparam
  *pMinMax\ptMinTrackSize\x=400
  *pMinMax\ptMinTrackSize\y=300
  ProcedureReturn #Event_ReturnTrue
EndProcedure
```

This event corresponds to the Windows message #WM_GETMINMAXINFO.

The return value is not important and default processing is not advised as it would defeat the object of having such a handler!

#OnUnhandledWinMessage

This is an event requiring some careful handling and so a discussion of this event is left to the section

Two specialised events.

OLE Drag and drop items.

As of version 2.0 of EasyVENT, drag and drop uses the Purebasic OLE drag and drop library. This means that items (text, images, files etc.) can be dragged between applications etc. and is far more powerful than that offered by earlier versions of EasyVENT.

The drag / drop functionality of EasyVENT is thus really a convenient wrapper around the PB library commands, but it does have the advantage of easily allowing drags from any kind of gadget / window. See the demo programs for details.

The following events are used for dragging and dropping of items between gadgets / windows / applications :

#OnDragItemStart
#OnDropItem

A drag drop operation is instigated by the user on depressing the left mouse button and moving the mouse a predetermined amount. At this point the *#OnDragItemStart* event handler of the gadget / window with the focus is called. Here the developer would start the drag with one of the Purebasic drag commands, e.g. DragText() etc.

When the user completes the drop by releasing the mouse button, the *#OnDropItem* event handler of the gadget / window beneath the cursor is then called (if such a handler exists **and if the gadget / window has been enabled for drops with the Purebasic command EnableGadgetDrop() or EnableWindowDrop() etc.**)

The two events associated with drag/drop of items are detailed thus.

#OnDragItemStart

Registering a control with a handler for this event allows for easy drag and drop of items from the control. For example, you could drag multiple items from a ListIcon and deposit them in a second ListIcon gadget etc. (See the appropriate demo programs for an example of this.)

This handler is called when the user begins the drag operation. In this case the values of the fields of the *sender parameter are as follows:

hWnd	the windows handle of the control from which an item is being dragged.
message	*#OnDragItemStart*
mouseX	the x-coordinate of the mouse cursor within the client area of the underlying control at the time the drag was instigated.
mouseY	the y-coordinate of the mouse cursor within the client area of the underlying control at the time the drag was instigated.
button	*#EVENT_LEFTBUTTON*
wParam	will indicate whether various virtual keys are down including the control key. Use this if deciding whether to instigate an OLE copy or move etc.

Test the control key by using :

```
If *sender\wParam&#MK_CONTROL
```

etc.

Here the developer would start the drag with one of the Purebasic drag commands, e.g. DragText() etc.

The return from this handler is not important and is not used.

#OnDropItem

This event fires whenever the user finishes dragging an item from a gadget by releasing the mouse button over the registered control/window.

NOTE that the control/window must have previously been enabled for drops with the Purebasic command EnableGadgetDrop() or EnableWindowDrop() etc.

This event is preceeded by a *#OnDragItemStart* event.

You can register this event with any control/window.

In this case the values of the fields of the *sender parameter are as follows:

hWnd	the windows handle of the registered control/window, i.e. the **destination** of the drop.
message	*#OnDropItem*
mouseX	the x-coordinate of the mouse cursor within the client area of the **destination** control (i.e. the control receiving the drop).
mouseY	the y-coordinate of the mouse cursor within the client area of the **destination** control (i.e. the control receiving the drop).
button	*#EVENT_LEFTBUTTON*
item	the windows handle of the control we are dragging from, i.e. the **source** control.

The return from this handler is not important and is not used.

Two specialised events.

In this section we detail two inter-related events which do need a little care as, whilst they are included for convenience and (with the second event at least) for increasing the power of EasyVENT, they can lead to certain problems if they are used without due consideration.

***#OnANYevent*.**

This event provides a quick means of processing ALL events (not Windows messages) for a particular window\control in a single procedure (rather than individual procedures).

In earlier versions of EasyVENT (prior to version 3.0.1), if you wished a single event handler to process two or more events for the same window\control, then you would need to issue two ***SetEventHandler***() commands, one for each event.

For example, suppose you wished to handle *#OnKeyPress* and *#OnChange* for the same edit control and in the same handler function, then you would previously have had to issue the commands :

SetEventHandler(***GadgetID***(*#edit1*), *#OnKeyPress*, *@MyHandler*())
and ***SetEventHandler***(***GadgetID***(*#edit1*), *#OnChange*, *@MyHandler*())

separately.

With *#OnANYevent*, you can now use a single event handler function to process ALL events for the underlying window\control; kind of like a Window callback function (but of course all the events have been automatically translated from Window messages and notifications etc. by EasyVENT).

Note that when I say ALL events for the underlying window\control, I mean ALL events **with one exception** [5], there are no other exceptions! All events get directed to your handler whether you intend to process them or not!

So, for example, issuing the command :

SetEventHandler(***GadgetID***(*#edit1*), *#OnANYevent*, *@MyHandler*())

will cause absolutely all events (with just the one exception) appropriate for this particular edit gadget to be sent to the function named *MyHandler*().

Now, with this seemingly convenient arrangement comes a need to consider those events which you have no interest in. After all, EasyVENT has no way of knowinge exactly which events you process in your *#OnANYevent* handler. Consequently, you need to ensure that all those events which you do not process in your *#OnANYevent* handler receive the 'default processing' by using the ***PerformDefaultWinProcessing****()* function.

---

[5] *#OnUnhandledWinMessage.*

A suitable 'template' for a *#OnANYevent* event handler would thus be :

```
Procedure.l MyUnHandledWinHandler(*sender.PB_Sender)
  Select *sender\message

    ...... {Your code here – handle any messages you require.}

     Default  {for all unhandled messages).
       PerformDefaultWinProcessing(*sender)
       result = #Event_ReturnDefault

  EndSelect
  ProcedureReturn result
EndProcedure
```

The thing to remember with *#OnANYevent* is that, if one is defined for a particular window or control, then EasyVENT regards all events (except *#OnUnhandledWinMessage*) as having been dealt with and will thus not pass the underlying Window's message back to the system for default processing. Hence the 'Default' switch in the above template code.

Please see the demo program : "OnANYevent demo.pb" for an example of using such a handler function.

***#OnUnhandledWinMessage*.**

This event fires whenever a Windows message is not handled by an event procedure (including *#OnANYevent* handlers in which only the default processing occus) or indeed in cases where there is no corresponding EasyVENT event. This includes *most* (but not all!) unhandled EasyVENT events.

This event is sent in the form of the underlying Windows message and consequently you should handle this accordingly. For example, a *#OnButtonClick* event is, at the outset, sent to the *#OnButtonClick* handler attached to the underlying button. However, at the Windows level this message corresponds to a #BN_CLICKED command message sent to the **parent window** of the button. Hence, if you are looking to process this event in response to a *#OnUnhandledWinMessage* message, then you must look for it in the *#OnUnhandledWinMessage* handler attached to the **parent window** of the button. It's an important point and one which has caused a few developers a few headaches!

The values of the fields of the *sender parameter of interest are essentially those which correspond to the usual window procedure parameters; namely (hWnd, uMsg, wParam, lParam) and are as follows:

hWnd	the handle of the window receiving the message
uMsg	the Windows message constant, e.g. #WM_COMMAND etc.
wParam	
lParam	

In addition, *mousex* and *mousey* contain the client coordinates of the cursor at the time the message was sent, and the *originalmessage* value holds the EasyVENT message constant in the event that the Windows message would have generated an EasyVENT event.

The return value from this handler is passed directly back to Windows because if such a handler is present, it becomes responsible for all Windows messages which are not processed by any event handler (including the *#OnUnhandledWinMessage* event handler).

Like any Windows callback procedure you must ensure that any unhandled messages receive *default processing*. To do this simply call ***PerformDefaultWinProcessing()*** for all unhandled messages.

A suitable 'template' for a *#OnUnhandledWinMessage* event handler would be :

```
Procedure.l MyUnHandledWinHandler(*sender.PB_Sender)
  Select *sender\uMsg

      …… {Your code here – handle any messages you require.}

     Default  {for all unhandled messages).
       result = PerformDefaultWinProcessing(*sender)

  EndSelect
  ProcedureReturn result
EndProcedure
```

(Note the subtle differences to the template given for *#OnANYevent* handlers.)

For messages which you decide to handle within the *#OnUnhandledWinMessage* event handler, ensure that you return a valid result as befitting the underlying message. You will need to consult the API documentation for these details.

Remember that there is nothing which forces an application to utilise a *#OnUnhandledWinMessage* handler. In these cases, all unhandled messages are automatically sent for default processing by EasyVENT. [6]

Please see the demo program : "OnUnhandledWinMEssage demo.pb" for an example of using such a handler function.

---

[6] Not all unhandled EasyVENT events are passed to a *#OnUnhandledWinMessage* handler, as some do not correspond to Windows messages. The point is that *#OnUnhandledWinMessage* handlers should really only be used for direct processing of Windows messages and not EasyVENT events.

A note on the relationship between *#OnANYevent* and *#OnUnhandledWinMessage*.

First, note that handlers for these two events differ in their return values.

#OnANYevent returns one of : *#Event_ReturnDefault* or *#Event_ReturnTrue* or *#Event_ReturnFalse* (like any regular event handler), whilst *#OnUnhandledWinMessage* returns the actual result of processing a Windows message.

Secondly, *#OnANYevent* handlers take absolute priority over all other handlers.

For example, suppose we have issued the following commands for an edit control :

SetEventHandler(***GadgetID***(*#edit1*), *#OnKeyPress*, *@MyKeyPressHandler*())

SetEventHandler(***GadgetID***(*#edit1*), *#OnANYevent*, *@MyOnANYHandler*())

and ***SetEventHandler***(***GadgetID***(*#edit1*), *#OnUnHandledWinMessage*, *@WinHandler*())

Then, if a *#OnKeyPressed* event is raised, it will be sent to the *#OnANYevent* handler and not the *#OnKeyPressed* handler.

Similarly, if an event is raised which has no specific handler attached to it, then it too gets sent to the *#OnANYevent* handler.

Only Windows messages which have no corresponding EasyVENT event will be despatched to the *#OnUnhandledWinMessage* handler.

Contrast this to the following situation in which no *#OnANYevent* handler is defined :

SetEventHandler(***GadgetID***(*#edit1*), *#OnKeyPress*, *@MyKeyPressHandler*())

and ***SetEventHandler***(***GadgetID***(*#edit1*), *#OnUnHandledWinMessage*, *@WinHandler*())

In this case, if a *#OnKeyPressed* event is raised, it will be sent to the *#OnKeyPressed* handler as would be expected.

However, if an event is raised which has no specific handler attached to it, then it will be sent (if it corresponds to a Windows message) to the *#OnUnhandledWinMessage* handler.

A note on Panel gadgets and ScrollArea gadgets.

Events can be attached to the child controls of any container gadget (including Panel gadgets and ScrollArea gadgets) as with all other controls.

However, take a little care when trying to attach events directly to a Panel gadget's individual panels or a ScrollArea gadget's 'client area'. In the case of a Panel gadget, individual panels are 'Static' Windows controls and in the case of a ScrollArea gadget, the 'client' area is a customised window registered by Purebasic.

The best way of dealing with these are with the GetParent_() API command.

For example, to attach a *#OnMouseOver* event to an individual panel (as opposed to a gadget within the panel - which is no problem), use the command:

```
Result = SetEventHandler(GetParent_(GadgetID(#anygadget)), #OnMouseOver,
            @MyHandler())
```

where *#anygadget* is the identifier of any gadget placed within the particular panel.

See the demo programs for further details.

A note on Spin gadgets.

There is a risk of a program entering an infinite loop if using certain events without undue care. Specifically, if you set the value of a spin gadget within its *#OnChange* event handler then your program will undoubtedly enter such a loop.

For this reason I recommend using the *#OnScroll* handler when setting the value of a Spin gadget which, coincidentally, provides a lot of extra information anyhow.

See the SpinGadget demo program for more details.

A note on structuring your programs.

There is no requirement (or rule) which specifies that an event procedure can only handle one event or deal with one control/window etc.

Indeed, I now typically use a single event procedure for each control, sifting through the events with a Select \ EndSelect construct.

For example :

```
Procedure.l events_EditorGadget(*sender.PB_Sender)
  Select *sender\message
    Case #OnMouseDown, #OnDblClick
      result = #Event_ReturnFalse  ;No call to PerformDefaultWinProcessing() means that
no default processing will occur.
    Case #OnKeyPress
      If *sender\wParam <> #VK_TAB
        PerformDefaultWinProcessing(*sender)
        result = #Event_ReturnDefault
      EndIf
  EndSelect
  ProcedureReturn result
EndProcedure
```

Of course, the above example would require several uses of ***SetEventHandler**()*, one for each of the messages processed.

An alternative is of course the *#OnANYevent* event.

Occasionally I will use a single event procedure to deal with one event, but spread across several windows / controls. A typical use for this kind of scenario might be to use a *#OnErase* handler to prevent Windows from erasing the backgrounds of several gadgets in an attempt to reduce flicker etc. (This is a trick that can be used to good effect.)

```
Procedure.l event_EraseBackground(*sender.PB_Sender)
  ProcedureReturn #Event_ReturnFalse   ;This will prevent Windows from erasing the
background etc.
EndProcedure
```

Again, several uses of ***SetEventHandler**()* are required, this time one for each window / control to which the above handler is attached.

Upgrading from EasyVENT 1.xx.xx or 2.xx.xx.

Unfortunately EasyVENT 3 is not backwards compatible in that applications using earlier versions of EasyVENT may well refuse to compile. At the very least I would expect the resulting executables to function incorrectly or even crash completely.

This is because with the latest version of the library, an event handler can no longer rely upon EasyVENT to pass the Windows message corresponding to an event (if any) back to the system as soon as the event handler has finished its work. Instead the event handlers have to explicitly call the function ***PerformDefaultWinProcessing**()* in order for the underlying message to receive default processing.

This is a far more flexible way of doing things, much more powerful, but of course this does leave us with the extra burden of deciding whereabouts in our event handlers to call this function; at the beginning, at the end, or not at all?

It all depends of course on the particular event and the desired effect etc.

However, converting source code using earlier versions of EasyVENT to use EasyVENT 3.xx.xx may not be as difficult as it first appears. Remember that all default processing in earlier versions of EasyVENT occurred after the event handler had finished executing.

We can thus begin our conversions by placing all instances of ***PerformDefaultWinProcessing**()* at the end of our handlers.

In fact, if in your old source a handler returned the value *#PB_PRocessPureBasicEvents* in order for default processing of a Windows message to occur, simply replace this with :
***PerformDefaultWinProcessing**()*
***ProcedureReturn** #Event_ReturnDefault*

For those instances when zero was returned, then return the value *#Event_ReturnFalse* without calling the ***PerformDefaultWinProcessing**()* function.
I think this should cater for about 70% of the ammendments required to upgrade our source code to use EasyVENT 3.xx.xx.

Indeed, this was the case with all the demo programs which are included in the EasyVENT package.

Other changes will invariable involve having to place additional calls to ***PerformDefaultWinProcessing()*** etc.

Q&A : tips and tricks.

The following ‘tips’ have generally arisen out of questions I have answered from other developers making good use of EasyVENT.

Question 1) I have set a #OnDragItemStart handler for a ListView gadget. However the scrollbar of the listviewgadget does not react on a single click when the eventhandler is set.

The only explanation for this is that you must have a #OnMouseDown handler set up for the same gadget but are not calling PerformDefaultWinProcessing() within that handler.

Remember that #OnMouseDown now fires for non-client area messages as well as client area ones.