Help Contents

Browse Custom Control version 3.0

developed by Gabriel Oancea

The Browse control is a shareware application, that means you are free to use and test it for 30 days. After this period of time, if you decide you like it you have to register it with the author (see address and Compuserve ID below).

You are free to distribute the demo version of the control in any way you like, if you distribute it with all the documentation enclosed. You are not allowed to sell it.

For inquiries please contact Gabriel Oancea, Compuserve ID: 70404,655, or contact Delta Soft Inc., 12 Danton Court, Ajax, Ontario, Canada, postal code L1S 3G1. Telephone: (416) 619-2018.

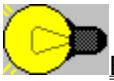
Description

The Browse control can be used for viewing the contents of a database (DBF) file in a tabular format. It allows you to select the number and content of the data columns to be displayed, select the records you want to view, the index to be used; also allows editing of the data in the browse window, with data pre- and post-validation - if editing for the respective column is enabled.

Please read the sub-chapters listed below in order, if you are new to Browse, vxBase or both.

NOTE: In order to use the control you must initialize the vxBase DLL, see your vxBase documentation. Include the VXBASE.BAS file provided in your projects, or insert its contents in one of your modules.

Introduction

 Example

The way the control appears to the users is very similar with a browse created with Clipper TBrowse class, or with the standard xBase BROWSE command: each column has a column header, each row is a record, the user can scroll horizontally and verically either using scroll bars or with the keyboard. Columns can be frozen at the left side of the control window, etc. There are many improvements compared to the DOS based browses, all the standard Windows and VB features are available. There are new features like index scoping and pseudo-filter, searching and many more.

There is no limit to the number of records in the database (max. is 2 billion), multi-user - multi-instance capabilities are built in. For database interface the browse control uses vxBase (a shareware DLL that can be found on most of the Bulletin Boards that have a VB section, on Compuserve MS-BASIC forum, etc.)

How it works:

First you have to create a browse on one of your forms: click the Browse icon in the ToolBox then draw the control. The default name is Brw1. At design time you can set properties like colors, placement, font and style.

At run-time, when you activate the browse you will have to initialize it: set the number of columns, the width (in characters) of each column, the column headers; Optionally you can set the text alignment (justification) for both the header and the data, enable or disable editing for each column, set the number of left frozen columns (columns that will be always visible at the left edge of the control window, also called nonscrollable), set special (enhanced) colouring for the columns you choose, and so on.

Then you open the DBF and assign the handle of the DBF to the Dbf property, in this moment the control will start displaying data, namely will fire the GetLine event, asking you to provide it with the data to be displayed, for the current line. You will write code for returning the data formatted as desired, in the cLine argument of the event. This system allows a maximum of flexibility with a minimum of coding required as summarized bellow:

1. Define the control at design time:	Set colors, font properties, placement and initial values for standard properties like Visible, Enabled, Tag etc, if required.
2. Initialize the control at run-time:	Set the no of columns, width of each column and optional: header, column text alignment, column width in pixels. Assign the Dbf property and optional: set index, scope(s), pseudo-filter, enable editing for the columns you want (if any), set the no of frozen columns, set the columns with special coloring, etc.
3. Write code for the GetLine event:	This event returns a line of data from VB, to be displayed by the browse; the line consists of the data for each column added in one string, undelimited.

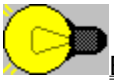
Introduction Example:

The example below assumes you have a form (**Form1**) on which there is a Browse control **Brw1**. You should also add the file VXBASE.BAS for the required declarations for the vxBase DLL.

```
Option Explicit
Dim nD as integer, nI as integer

Sub Form_Load()
   vxInit              ' initialize vxBase
   vxSetLocks FALSE    ' set Clipper compatible locking mechanism

   ' we want to browse 3 fields from the PURCHASE.DBF
   Brw1.Cols = 3           ' set the number of columns
   ' set the data width for each column from 0 to Cols - 1, that is 0 to 2
   Brw1.ColWidth(0) = 7    ' Customer no  (the field name is CUSTNO, numeric
   7.0)
   Brw1.ColWidth(1) = 10   ' PO number    (the field name is PONO, character
   10)
   Brw1.ColWidth(2) = 20   ' Cust name(the field name is CUSTNAME,
   character, 20)
   'Set the header text so that it matches the column widths set above
   '              1234567     1234567890     12345678901234567890
   Brw1.Header = "Cust.No" + "PO number " + "Customer name       "

   ' open the database PURCHASE.DBF and assign the handle to the Dbf
   property
   nD = vxUseDbf("PURCHASE")
   If nD <= 0 Then MsgBox "Error!": Exit Sub    ' if database cannot be
   opened
   Brw1.Dbf = nD
   ' open the index PURCHASE.NTX and assign the handle to the Ntx property
   nI = vxUseNtx("PURCHASE")
   If nI <= 0 Then MsgBox "Error!": Exit Sub    ' if database cannot be
   opened
   Brw1.Ntx = nI
   Brw1.SetFocus  ' We're done!
End Sub

Sub Brw1_GetLine( cLine as String )
   ' We must return a line of data, in this case our three fields
   ' The data will look like this:
   '  Col 0       Col 1          Col 2
   '  width=7     width = 10     width = 20
   '  1234567     1234567890     12345678901234567890
   ' "9999999" + "XXXXXXXXXX" + "XXXXXXXXXXXXXXXXXXXX"
   cLine = vxField("CUSTNO") + vxField("PONO") + vxField("CUSTNAME")
End Sub

Sub FormUnload( Cancel as integer )      ' close all open files and
deallocate
  Dim j as integer                       ' memory in the VXBASE.DLL
  j = vxCloseAll()                       ' !!only at the end of the program!!
```

```
  vxDeallocate
End Sub
```

Index Scope

 Example

One of the most important features of the control is the Index Scope, which will allow you to extract a subset of records from the database browsed without using a filter, which is much faster.

The Index Scope can be used in two ways:
you can extract all the records for which the index key begins with a character string, or
you can extract all records for which the index key falls between two character string expressions.

For example if you have a CUSTOMER database, indexed on the field called CUSTNAME - the customer name, you can extract all the customers with a name begining with 'SMITH' (that is 'SMITH', 'SMITHER', ...), or you can extract all the customers with a name that falls between 'Bill' and 'Joseph'.

In order to use the Index Scope you must have a database and index file open and the browse control initialized (see the Introduction on how to do this). To extract all the records for which the index key begins with a character string expression (for example 'MyString') you set the IndexExp property to the desired expression:
`Brw.IndexExp = 'MyString'`, which can be a literal string or an expression which evaluates to string.
The control will refresh all the visible records and position the record pointer on the first record that meets your criteria. If no record is found for which the index key begins with your expression, then the control window is cleared, the record pointer is moved to EOF and the EmptyFile property is set to **True**.
To retrieve the current setting for the scope:

```
If Len(Brw.IndexExp) = 0 then
     Label1.Caption = "Current Scope: <None>"
Else
     Label1.Caption = "Current Scope: '" + Brw.IndexExp + "'"
End If
```

You can also extract all the records which fall in a range. To do this set up your browse as above, then set the IndexExp property to the upper limit of the range. Then set the IndexExp2 property to the lower limit of the range. For example:

```
Brw.IndexExp  = Trim$(Text1.Text)
Brw.IndexExp2 = Trim$(Text2.Text)
```

The control will refresh itself, and position the record pointer to the first record in scope. If no record is in scope (that is no record falls between `Trim$(Text1.Text)` and `Trim$(Text2.Text)` then the record pointer is moved to the EOF, and the EmptyFile property is set to **True**.

Notes:

To clear one or both Index Expression properties set them to "" (an empty string), setting IndexExp to an empty string will also clear the IndexExp2 and FilterExp and will refresh all records in display.

If you use both IndexExp and IndexExp2 properties they must be set in this order. Subsequent change of the IndexExp can clear the IndexExp2 (if IndexExp > IndexExp2).

When adding records with a scope set, you should make sure the newly added records fall in scope, otherwise, after the REFRESHALL, the control can set the EmptyFile to **True** (the control resolves a REFRESHALL by maintaining the current record pointer, and if this is out of scope it will consider the file empty). If you have to add a record that is not in scope it will NOT be displayed, and it is recommend that you issue a GOTOP request instead of the REFRESHALL.

Index Scope Example

The following example illustrates how to create a browse in which the user will be able to set a scope.
It is assumed that you have a form **Form1** containing the following controls:
- a Browse control Brw1
- a Text Box: Text1
- a Command Button: Command1 (set the caption to "Set &Scope")
The code in the Form1 is listed below, you should add to the project the VXBASE.BAS module provided, including the vxBase declare statements required.

```
Option Explicit
Dim nD as integer, nI as integer

Sub Form_Load()
  vxInit
  vxSetLocks False

  nD = vxUseDbf("MyDbf")      ' open database and index
  nI = vxUseNtx("MyNtx")

  ' initialize the browse
  Brw1.Cols = 3
  Brw1.ColWidth(0) = 10
  ' set column widths, alignments, header etc....

  Brw1.Dbf = nD               ' set database handle
  Brw1.Ntx = nI               ' set index handle
  Brw1.SetFocus               ' start
End Sub

Sub Brw1_GetLine( cLine as String )      ' get line event
  cLine = vxField("MyField1") + ...' return data to display
End Sub

Sub Command1_Click()
  ' set the text entered in the Text1 as Index Scope
  Brw1.IndexExp = Trim$(Text1.Text)
End Sub
```

Pseudo-Filter

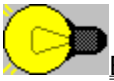
Another major feature is the Pseudo-Filter, allowed only when a scope is active. Basically the pseudo-filter is similar to a normal xBase filter, the only difference being that it is applied only to the records already in scope, that is sometimes a small sub-set of the whole database. To set a Pseudo-Filter you have to have an active Index Scope, then you assign a normal xBase filter expression to the FilterExp property of the control. It can be a literal character string:
`Brw1.FilterExp = "BALANCE>0 .AND. DATEEXP > ctod('01/01/93')"`, or an expression evaluating to a string:
`Brw1.FilterExp = Trim$( txtMyText.Text )` The expression must evaluate to a logical value. You can use any valid vxBase expression, including aliased fields (`"CUSTOMER->NAME"`).

You will discover that the Pseudo-Filter can be much faster than a normal filter, especially when the scope in effect is narrow. Both the scope and the pseudo-filter respect all the previous settings of the work area: standard filter, relations, etc. So you can have a filter, set a scope and then set a pseudo-filter and everything works fine.

Use the Example from the Index Scope and add another text box Text2, then modify the Command1_Click to look like:

```
Sub Command1_Click()
  ' set the text entered in the Text1 as Index Scope
  Brw1.IndexExp  = Trim$(Text1.Text)

  If len(Brw1.IndexExp) = 0 Then
    Text2.Text = ""
    Exit Sub      ' don't bother with the filter, we have no scope
  End If

  On Error Resume Next
  ' if the filter expression is not valid the control will issue an error
  message
  Brw1.FilterExp = Trim$(Text2.Text)
  If Brw1.FilterExp <> Trim$(Text2.Text) Then MsgBox "Wrong Pseudo-Filter!"
  On Error GoTo 0
End Sub
```

Editing

 Example

One of the most handy features of the control is on-the-spot editing, with data pre-validation and post-validation and an edit flag for each column.

How does the user see it: while browsing a file he/she can either press ENTER to start editing the current (highlighted) field, or just simply start entering the text. Press ENTER when done if you want to save the current field, press ESC to cancel.

All the nice features are there: Cut, Copy and Paste to/from the clipboard, undo, scrolling, etc. - like any normal text box. Almost all the normal xBase editing keys are available: press ENTER or PAGEUP / PAGEDOWN to exit edit mode and save, press ESC or click with the mouse outside the edit area to cancel changes. Press UP or DOWN arrows to save and move to the next record up or down.

How do you program it: very simple: just set the property ColEdit(x) = **True**, (where x is the column number for which you want to enable editing; you can enable editing of all columns, of one column or of none, *default is none)*. Then you write code for the EditWhen event if you want to do some data validation and finally in the EditValid event, if everything is fine use one of the vxReplXXXX vxBase functions to replace data (you might need to do some locking as well - if required). This procedure is suitable if you have expressions as column data, rather than fields (Click on Example above for details).

An alternative way: there is also an easier way to do it: let the control replace the fields for you, if the columns enabled for edit with this mechanism are database fields.
Set the property ColField(x) = "MyFldName" - for example, for each the column you want to be edit enabled (setting the ColField(x) to a valid field name will automatically set the ColEdit(x) to **True**).
The control will replace the field you supply with the text entered, taking care of all the locking required. A EditReplFail event will be fired if the record could not be locked. Using the **EditWhen** and **EditValid** events, you can check the validity of data. Returning False will either not allow the editing, or keep the user in edit until a valid value is entered or the edit is cancelled (Click on Example above for details).

You can have the 2 ways of editing mixed as you like. *Some hints and requirements:*
You must set the **ColEdit** and **ColField** properties after setting the Dbf property, otherwise the edit flags and field names will be cleared.
Write data validation routines if you replace the data youself, you can prevent user errors.
Do not enable for editing the key (indexed) fields, only if absolutely required; an alternative is to set the index order to 0 (natural order) - use vxDeselectNtx().
NEVER move the record pointer in the EditWhen / EditValid events (or if you do so restore it to where it was), you might overwrite another record's data. Also do not request any Action from the browse while in one of this events, there is no need for the REFRESHLINE, this is done for you.

Editing Example

The following example illustrates how to create a browse in which the user will be able to edit fieds on the spot. It is assumed that you have a form **Form1** containing the Browse control **Brw1**.
The code in the Form1 is listed below, you should add to the project the VXBASE.BAS module provided, including the vxBase declare statements required.

```
Option Explicit
Dim nD as integer, k as integer, nFld as integer

Sub Form_Load()
  Dim cH As String

  vxInit: vxSetLocks False
  nD = vxUseDbf("MyDbf")      ' open database

  ' initialize the browse
  nFld = vxFieldCount()' get no of fields, max BRW_MAX_NO_OF_COLUMNS
  If nFld > BRW_MAX_NO_OF_COLUMNS Then nFld = BRW_MAX_NO_OF_COLUMNS
  Brw1.Cols = nFld
  for k = 1 to nFld
    Brw1.ColWidth(k - 1) = vxFieldSize(vxFieldName(k))
    cH = cH + Mid$(vxFieldName(k) + Space$(10), 1,
    vxFieldSize(vxFieldName(k)))
  next k
  Brw1.Header = cH
  ' set column alignments if desired

  Brw1.Dbf = nD               ' set database handle
  Brw1.ColEdit(0) = True      ' enable editing for the first field
  for k = 2 to nFld
    Brw1.ColField(k) = vxFieldName(k + 1)       ' enable direct editing for
    all
  Next k                                        ' the other fields in browse
  Brw1.SetFocus               ' start
End Sub

Sub Brw1_GetLine( cLine as String )             ' get line event
  For k = 1 to nFld
    cLine = cLine + vxField(vxFieldName(k))     ' return data to display
  Next k
End Sub

Sub Brw1_EditWhen(nCol As Integer, cField As String, lCancel As Integer)
  ' check if editing can proceed, if not set lCancel to True
  ' default action is to start editing
End Sub

Sub Brw1_EditValid(nCol As Integer, cField As String, lOk As Integer)
  ' write validation routines for each field, set lOk to False if invalid
  data
  ' set the cField to some valid choice - if you want
```

```
  ' default action is to replace the data if the ColField(nCol) is not empty
  ' if ColField(nCol) is empty no action is taken, you must replace the field
  ' with the cField value
  If nCol = 0 Then           ' for field 1 we must replace data here
    If vxLockRecord() Then    ' let's suppose it's a Character field
      vxReplString (vxFieldName(1)), (cField)
      k = vxUnlock()
    Else
      MsgBox "Cannot lock the record!"
    End If
  End If
End Sub
```

Other features of the Browse control

Some other important properties and events are listed below:

Ability to lock one or more columns at the left edge of the control window: by setting the LeftFrozen property to a number between 1 and Cols, you tell the browse control to 'freeze' the respective number of columns - that means that they will be visible no matter how the user scrolls the other columns. You can change the LeftFrozen property dynamically, at run-time - or let the user do it. This can be very handy if you would like to keep one / more column(s) always in view. Frozen columns can be edited, sized, etc. like any other normal column.

Highlighting columns: you can have selected columns displayed in a special color you can set up, all you have to do is set the ColorSpecial and ColorTSpecial at design or run-time then set the SpcColor(nColumn) = True and the column will be displayed using the two special colors as background / foreground. Useful for underlining some special column - as totals of some sort, or the editable columns if you choose.

Run time adjustable column widths: the user can change the column width by dragging the column separator with the mouse. For this, move the mouse on the column headers, between two columns: the cursor shape will change; press the right mouse button and keep it pressed, the column will appear delimited with a focus rectangle, size it to the right or left, release the mouse button when done. You can also set the column width from code using the ColWidthPix property (the width of each column in pixels). This can be useful to save and restore settings of a query the user has defined: when the control is unloaded (for example in the Form_Unload() save the column width in pixels to some variable(s), then upon exiting the program to a INI file. In the Form_Load() procedure (after setting the ColWidth - data width in characters) set the ColWidthPix to the saved values, eventually the location of the window on screen.

Design-time / run-time adjustable colors: you can set the colors for the header, normal, highlighted cell, frozen columns and special columns; the defaults are the Windows system colors. You can offer the user the chance of setting them to his/her preferences (use for example the common dialog box control - Colors), and then you can save and restore them.

Refresh record data displayed on the screen: write code in the Change event, this is fired every time the row, column, record number or any combination of the above are changing. You can have a non modal form displayed with the whole record on one side of the screen and you can refresh it every time the user moves in another window containing a browse.

Find out when the user tries to move outside the current scope (past EOF or before BOF): the events HitTop and HitBottom are fired every time when the user tries to move outside the current limits. Also the HitTop and HitBottom properties are set to **True** when the user tries to move past EOF of above BOF. You can use these events / properties to ask if he wants to add a record, or just simply beep at him.

Reposition the record pointer / refresh the browse display from code: the Action property allows you to send requests to the browse control, like: RefreshLine, RefreshAll, GoTop, GoBottom, Up, Down, Left or Right. You can look at the Action property as a method, because no custom methods can be implemented in the actual version of the CDK.
Another important feature is to search for a certain expression, if the file is indexed, set the SearchKey property to any string, the Action property will be set to **True** if the expression was found.
You can also force the user to enter Edit mode from code, just set the **Action** property to BRW_ACT_EDIT, and the current column from the current record will be edited.

Browse Properties

The standard Visual Basic properties supported are:
NAME, INDEX, BORDERSTYLE, DRAGMODE, DRAGICON, ENABLED, FONTBOLD, FONTITALIC, FONTNAME, FONTSIZE, FONTSTRIKE, FONTUNDER, HEIGHT, HELPCONTEXTID, HWND, LEFT, MOUSEPOINTER, PARENT, TABINDEX, TABSTOP, TAG, TOP, VISIBLE, WIDTH. Please see your VB manual for details about any of these properties.

Custom Properties

Available at design time:

ColorNormal	Long	Background color of the normal data cells
ColorHeader	Long	Background color of the column headers
ColorHighlight	Long	Background color of the highlighted data cell
ColorFrozen	Long	Background color of the frozen columns
ColorSpecial	Long	Background color of the special colored columns
ColorTNormal	Long	Text color of the normal data cells
ColorTHeader	Long	Text color of the column headers
ColorTHighlight	Long	Text color of the highlighted data cell
ColorTFrozen	Long	Text color of the frozen columns
ColorTSpecial	Long	Text color of the special colored columns

Available at run-time only, read only at run-time

Rows	Integer	Total no of rows visible in the window
RowHg	Integer	Row height, in pixels
LineLenPix	Integer	Total browse line length, in pixels
LineLenChar	Integer	Total browse data line length, in characters
LeftCol	Integer	Left-most visible column no (from 0 to Cols-1)
RightCol	Integer	Right-most visible column
CurrCell	String	Highlighted cell text
LastLine	Integer	Last row of data in display
EmptyFile	Integer	TRUE if the database has no records, or no record matches the scope / filter
HitTop	Integer	TRUE if the user has tried to move above BOF
HitBottom	Integer	TRUE if the user has tried to move past EOF
xPos	Integer	Current left coordinate of the highlighted cell in pixels
yPos	Integer	Current top coordinate of the highlighted cell in pixels
HBarDim	Integer	Height of the horizontal scroll bar in pixels, 0 if no bar is required
VBarDim	Integer	Width of the vertical scroll bar in pixels, 0 if no bar is required

Available at run time only, read-write at run time

Header	String	Columns header
Cols	Integer	Total number of columns in browse
ColWidth()	Integer	Array of column data width, array index starts at 0 to Cols-1
ColWidthPix()	nteger	Array of column pixel widths, array index starts at 0 to Cols-1
ColAlign()	Integer	Columns alignment (left, right or center)
SpcColor()	Integer	True if you want the column highlighted in the special color
ColEdit()	Integer	True if you want edit enabled for the column
ColField()	Integer	Valid field name if you want edit enabled in auto-replace mode
Col	Integer	Current display column
Row	Integer	Current display row
LeftFrozen	Integer	No of columns to freeze to the left edge of the control window

HScroll	Integer	Nothing, this property is kept for compatibility with previous versions.
VScroll	Integer	Nothing, this property is kept for compatibility with previous versions.
Dbf	Integer	Handle of the Dbf work area, returned by vxUseDbf() / vxUseDbfRO()
Ntx	Integer	Handle of the Ntx, returned by vxUseNtx()
IndexExp	String	Scope based on the index key (alone or with **IndexExp2**)
IndexExp2	String	Scope based on the index key (with **IndexExp** only)
FilterExp	String	Pseudo-filter, must be a valid xBase expression
SearchKey	String	A character key to search in the selected set of records
Action	Integer	Tell the Browse control what to do

Browse Colors [Long]

```
[Var] = [Form!]Brw.ColorHeader     [ = LongExp ] ' header BackColor
[Var] = [Form!]Brw.ColorTHeader    [ = LongExp ] ' header ForeColor
[Var] = [Form!]Brw.ColorNormal     [ = LongExp ] ' normal cell BackColor
[Var] = [Form!]Brw.ColorTNormal    [ = LongExp ] ' normal cell ForeColor
[Var] = [Form!]Brw.ColorHighlight  [ = LongExp ] ' highlighted cell
BackColor
[Var] = [Form!]Brw.ColorTHighlight[ = LongExp ] ' highlighted cell
ForeColor
[Var] = [Form!]Brw.ColorFrozen     [ = LongExp ] ' frozen columns BackColor
[Var] = [Form!]Brw.ColorTFrozen    [ = LongExp ] ' frozen columns ForeColor
[Var] = [Form!]Brw.ColorSpecial    [ = LongExp ] ' special columns BackColor
[Var] = [Form!]Brw.ColorTSpecial   [ = LongExp ] ' special columns ForeColor
```

The default values are the Windows standard system colors: caption background and text, window background and text and selected item background and text. The ColorFrozen, ColorTFrozen, ColorSpecial and ColorTSpecial will be the same as ColorNormal and respectively ColorTNormal by default, you can change them at design stage if you choose to use them.

The colors can be changed at design time by either typing in a hex value, or by selecting the color from the standard VB color box (click on the '...' button or double-click on the property name to activate it).

You can also change colors at run time as a response to some user selection - for example selecting another color from the CommDlg Colors Selection dialog.

NOTE:

when the browse does not have focus or when it is not enabled, the header colors will change to the standard Windows inactive window caption colors.

if a color is set both to special and frozen status, the frozen color is used.

Example:

```
' set a flashing background for the current cell in browse
' we create a timer Timer1 and set the interval to 0.3 seconds
' It is nor recommendable to use the timer like this, but this
' is just an example for using the colors
Sub Timer1_Timer()
  Static lRed
  If lRed Then
    lRed = False: Brw.ColorHighlight = &HFF0000&     ' Blue
  Else
    lRed = True: Brw.ColorHighlight = &HFF&          ' Red
  End If
End Sub
```

Rows [Integer]

```
[Var] = [Form!]Brw.Rows
```

The Rows property is read-only at run time, and represents the no of rows visible currently, including the blank rows (if any). The header line is not included.

Example:

```
' adjust the height of the browse to fit 20 rows of data
Sub Form_Resize()
  ' if iconic don't bother
  If Me.WindowState = MINIMIZED Then Exit Sub
  ' some form resizing code here, make sure there is room enough
  If Brw.Rows <> 20 Then
    nOldScaleMode = Me.ScaleMode         ' save old scale mode
    Me.ScaleMode = PIXELS                ' defined in CONSTANT.TXT = 3
    ' recalc height: 20 rows+1 header line + Horz Bar height
    Brw.Height = (20 + 1) * Brw.RowHg + Brw.HBarDim
    Me.ScaleMode = nOldScaleMode         ' and restore scale mode
  End If
End Sub
```

RowHg [Integer]

```
[Var] = [Form!]Brw.RowHg
```

The RowHg property is read-only at run time, and represents the height (in pixels) of one data row, it will change when the font name or style changes

Example:

```
' adjust the height of the browse to fit 20 rows of data
Sub Form_Resize()
  ' if iconic don't bother
  If Me.WindowState = MINIMIZED Then Exit Sub
  ' some form resizing code here, make sure there is room enough
  If Brw.Rows <> 20 Then
    nOldScaleMode = Me.ScaleMode         ' save old scale mode
    Me.ScaleMode = PIXELS                ' defined in CONSTANT.TXT = 3
    ' recalc height: 20 rows+1 header line + Horz Bar height
    Brw.Height = (20 + 1) * Brw.RowHg + Brw.HBarDim
    Me.ScaleMode = nOldScaleMode         ' and restore scale mode
  End If
End Sub
```

LineLenPix [Integer]

```
[Var] = [Form!]Brw.LineLenPix
```

The LineLenPix property is read-only at run time, and represents the length of one full line of data (if all columns would be visible). It will change when the font name or style, no of columns, data width of a column or column width in pixels are changing.

Example:

```
' adjust the width of the browse to fit all columns in display
Sub Form_Resize()
  ' if iconic don't bother
  If Me.WindowState = MINIMIZED Then Exit Sub
  nOldScaleMode = Me.ScaleMode           ' save old scale mode
  Me.ScaleMode = PIXELS                  ' defined in CONSTANT.TXT = 3
  If Brw.LineLenPix + Brw.VBarDim <= Me.ScaleWidth Then
    Brw.Width = Brw.LineLenPix + Brw.VBarDim  ' reset the width
  End If
  Me.ScaleMode = nOldScaleMode           ' and restore scale mode
End Sub
```

LineLenChar [Integer]

```
[Var] = [Form!]Brw.LineLenChar
```

The LineLenChar property is read-only at run time, and represents the length, in characters, of one line of data. It will change when the no of columns or data width of a column are changing.

Example:

```
' pad with spaces the line of data we return to the control
' from the GetLine event
' NOTE: padding is not required, the control will add the required spaces
'       This is just an example
Sub Brw_GetLine( cLine As String )
  ' set data in cLine = MyField + a custom function: MyFunction$
  cLine = vxField( "MyField" ) + MyFunction$( vxRecNo() )
  If len( cLine ) < Brw.LineLenChar Then      ' check if data length is Ok
    cLine = cLine + Space$( Brw.LineLenChar - len(cLine) )
  End If
End Sub
```

LeftCol, RightCol [Integer]

```
[Var] = [Form!]Brw.LeftCol
[Var] = [Form!]Brw.RightCol
```

The LeftCol and RightCol properties are read-only at run time, and represent the left-most and respectively the right-most visible columns in the browse. They change as the user moves using the left / right arrow keys or the horizontal scroll bar. Useful when you need to know whether a column is visible or not, for edit purposes for example.

The LeftCol represents the left-most **NOT FROZEN** column, if any.

Example:

```
' we want to count all the empty EXPIRES date fields in a database,
' in a fancy way
' For this we set up a disabled browse Brw and write the function:
Function CountBlanks() As Integer
  Dim nCnt As Integer
  ' select the file and go top, we assume the browse is already setup
  ' and the EXPIRES field is in column 3, we have 2 columns frozen
  Brw.Action = BRW_ACT_GOTOP       ' go top
  Do While Brw.LeftCol < 3         ' put the EXPIRES column near the frozen
  cols
    Brw.Action = BRW_ACT_RIGHT     ' move one to the right
  Loop

  Do While Not vxEof() And Not Brw.HitBottom
    If vxEmpty("EXPIRES") Then nCnt = nCnt + 1
    Brw.Action = BRW_ACT_DOWN      ' the use can see the cursor moving down
    fast
  Loop
End Sub
```

CurrCell [String]

```
[Var] = [Form!]Brw.CurrCell
```

The CurrCell property is read-only at run time, and represents the text in the current cell (that is the active cell - the one in the highlighted color). It changes every time the user moves in the browse, you can use the Change event to find out when it changed, for example in order to update a Label control caption. The character string returned by the CurrCell property is what the user gets to edit, if edit is enabled for the specified column, it will be passed in the EditWhen event to you for pre-validation.

Example:

```
' we want to show the contents of the current cell in the Label1 control
Sub Brw_Change( nRowCol As Integer )
  Label1.Caption = "Current Cell: " & Brw.CurrCell
  If nRowCol <> BRW_CHANGE_COL Then      ' if RecNo has changed
    RedisplayRecord                      ' refresh the display
  End If
End Sub

Sub Brw_EditWhen( nCol As Integer, cField As String, lCancel As Integer )
  ' for example allow editing only for blank fields
  ' the cField is actually the same with CurrCell
  If Trim$(cField) <> "" Then lCancel = True   ' setting lCancel to True will
End Sub                                        ' prevent entering in edit
mode
```

LastLine [Integer]

```
[Var] = [Form!]Brw.LastLine
```

The LastLine property is read-only at run time, and represents the last line of data in the browse control window. If there is enough data (records) in the database to fill all the rows, LastLine = Rows and a vertical scroll bar is present (VBarDim > 0), otherwise LastLine is smaller and no vertical bar is displayed (also VBarDim will be set to 0). It can be smaller, although there are enough records to be displayed, if you use the vertical scroll bar by dragging the thumb near the end of the file. In this case the control will position the file on the new record, and will place it on the first line in the browse; if there are less records to EOF than lines visible, then LastLine will be smaller than Rows.

If the file is empty (no records, or no records in scope / filter), then LastLine is 0, and EmptyFile will be set to **True**. It is recommendable to use EmptyFile to test for the empty file condition.

Example:

```
' shrink the browse if less records than no of lines visible in browse,
' so that we do not display blank lines
Sub Form_Load()
  ' Initialize the browse, set the Dbf and Ntx, scopes etc.
  If Brw.LastLine < Brw.Rows Then
    ' Set the new browse Height
    Brw.Height = Brw.LastLine * Brw.RowHg + Brw.HBarDim + 1
  End If
End Sub
```

EmptyFile [Integer]

```
[Var] = [Form!]Brw.EmptyFile
```

The EmpyFile property is read-only at run time, and is set to **True** if the database file is empty - has no records or if there are no records meeting the criteria you set with index scope: IndexExp, IndexExp2 and eventually pseudo-filter: FilterExp. Also if the filter set with **vxFilter** Sub (vxBase) can determine the file to appear empty. If the EmptyFile is set to **True**, the LastLine is set to 0 and the data lines are cleared, no vertical bar is displayed, the current cell is set to "" (an empty string) and cursor (active cell) will not be allowed to move except on row 0 (the first row). The GetLine event is NOT fired when EmptyFile is **True**.

It is useful to check for empty file before writing a record to file, you should append a blank record if the file is empty before the replace statements.

Example:

```
' save a set of values entered by the user, if the file is empty add a
record
Function MyRecSave() As Integer
  Dim lOk As Integer

  If Brw.EmptyFile Then
    lOk = vxAppendBlank()     ' add a blank record
  Else
    lOk = vxLockRecord        ' lock current record
  End If

  If Not lOk Then
    MsgBox "Cannot lock/append record!"  ' lock/append failed
    MyRecSave = lOk                      ' return FALSE
    Exit Function
  End If

  ' replace data...
  lOk = vxWrite()             ' commit changes
  lOk = vxUnlock()            ' release lock
  If Brw.EmptyFile Then       ' EmptyFile is still True, if file was empty,
                              ' we did no refresh yet
    Brw.Action = BRW_ACT_REFRESHLINE
  Else
    Brw.Action = BRW_ACT_REFRESHALL      'refresh all after append
  End If
  MyRecSave = True
End Function
```

HitTop, HitBottom [Integer]

```
[Var] = [Form!]Brw.HitTop
[Var] = [Form!]Brw.HitBottom
```

The HitTop, HitBottom properties are read-only at run time, and they are set to **True** if the user has tried to move up before the beginning of file or before the first record in scope and, respectively past end of file or down past the last record in scope. In both cases first the HitTop / HitBottom events are fired, and then the two properties are set.
The two properties are useful if you use the `BRW_ACT_DOWN / BRW_ACT_UP` actions from code to move the record pointer, for example if you want to print all records in scope at any time - as in the example below.

Example:

```
' print all the records in scope, without knowing what the scope is, for
example
' if the scope can be set by the user for a generic query
Function PrintMyQuery() As Integer
  If Brw.EmptyFile Then
    MsgBox "Nothing to print"
    PrintMyQuery = False
    Exit Function
  End If

  Brw.Action = BRW_ACT_GOTOP
  ' print some headers
  Do While Not Brw.HitBottom
    PrintOneLine                  ' print one data line
    Brw.Action = BRW_ACT_DOWN     ' skip one record and refresh the display
  Loop
  Printer.EndDoc
  PrintMyQuery = True
End Function
```

xPos, yPos [Integer]

```
[Var] = [Form!]Brw.xPos
[Var] = [Form!]Brw.yPos
```

The xPos, yPos properties are read-only at run time, and they are the coordinates, in pixels, of the current highlighted cell's upper-left corner, relative to the browse control's upper-left corner.
NOTE: the two properties are always in pixels, no matter what the ScaleMode of the container of the browse control is. If you use a different ScaleMode, and you plan to use them, you should transform the coordinates from pixels to your mapping mode.

The width and height of the current cell are the width of the current column in pixels: Brw.ColWidthPix(Brw.Col) and the row height: Brw.RowHg respectively.

Example:
The following example shows how to change the cursor (MousePointer) shape when it is above the highlighted cell. Create a label control **Label1**, set the caption to "" (an empty string), set BorderStyle to None, BorderStyle to TRANSPARENT, and bring it to front (above) the Brw control, that will make the control invisible. Set the MousePointer property to a special shape, for example 2 (Cross). At run time we will change the position of this control every time the Brw current cell changes position, so that the control will be exactly above the current cell, that will make the mouse pointer to change to our new shape. The code in the Change event bellow is doing the repositioning.

```
DefType MYRECT              ' add this in a module
  left   As Integer
  top    As Integer
  width  As Integer
  height As Integer
End Type

Sub Brw_Change( nRowCol As Integer )
  Dim r as MYRECT
  ' we assume the form has the ScaleMode set to 3 (PIXELS)
  r.left   = Brw.Left + Brw.xPos
  r.top    = Brw.Top  + Brw.yPos
  r.width  = Brw.ColWidthPix(Brw.Col)
  r.height = Brw.RowHg
  Label1.Move r.left, r.top, r.width, r.height
End Sub
```

HBarDim, VBarDim [Integer]

```
[Var] = [Form!]Brw.HBarDim
[Var] = [Form!]Brw.VBarDim
```

The HBarDim, VBarDim properties are read-only at run time, and they represent the height of the horizontal scroll bar and the width of the vertical scroll bar of the browse control, respectively. They are 0 if the respective scroll bar is not required.
NOTE: the two properties are always in pixels, no matter what the ScaleMode of the container of the browse control is. If you use a different ScaleMode, and you plan to use them, you should transform the coordinates from pixels to your mapping mode.

The two properties can be used as an indicator of the presence of the scroll bars, and also they are useful for run-time resizing of the browse.

Example:

```
' adjust the height of the browse to fit 20 rows of data
Sub Form_Resize()
  ' if iconic don't bother
  If Me.WindowState = MINIMIZED Then Exit Sub
  ' some form resizing code here, make sure there is enough room
  If Brw.Rows <> 20 Then
    nOldScaleMode = Me.ScaleMode         ' save old scale mode
    Me.ScaleMode = PIXELS                ' defined in CONSTANT.TXT = 3
    ' recalc height: 20 rows+1 header line + Horz Bar height
    Brw.Height = (20 + 1) * Brw.RowHg + Brw.HBarDim
    Me.ScaleMode = nOldScaleMode         ' and restore scale mode
  End If
End Sub
```

Header [String]

```
[Var] = [Form!]Brw.Header [ = StringExp]
```

The Header property is a string made from the concatenation of all the column headers. You have to assign the Header only after you have defined the number of columns in the browse Cols and the data width of each column ColWidth. The header can be very easily made up by concatenating all the column headers, padded with spaces or trimmed to the data width of the respective column. Normally you would do this in the Form_Load sub, or in the function which starts the browse.

If the number of columns in browse, or the column data width for any column changes, the header is automatically adjusted (padded or trimmed). However, if the column contents changes, you should re-asssign the Header.

NOTES:

If you do not trimm / padd with spaces each column header to be exactly the size of the data width of the column, the header will be displayed erroneously. A simple pad function is presented below, it is included in the VXBASE.BAS module if you want to use it.
The last column does not need to be padded.
If you assign a string longer than LineLenChar characters, it will be trimmed, no error is generated.

Example:

```
simple function to padd a var with spaces or to trimm it to a given length
' it works both for strings and numbers, normally should return string, but
' who knows what we might use it for, so let's return a Variant
Function pad( ByVal cString As Variant, ByVal nLength As Integer ) As
Variant
  pad = Mid( cString & space$(nLength), 1, nLength )
End Function

Sub Form_Load()
  ' a simple database: CUSTOMER, 3 fields: CUSTNAME - Char 30,
  ' CUSTADDR - Char 30, CUSTNO - Numeric 10.0
  ' set the number of columns to 3, and column width as above
  Brw.Cols = 3
  Brw.ColWidth(0) = 30       ' CUSTNAME
  Brw.ColWidth(1) = 30       ' CUSTADDR
  Brw.ColWidth(2) = 10       ' CUSTNO

  ' and now set the header
  Brw.Header = pad("Customer name", 30) + pad("Address", 30) + "Number"

  ' continue with intitialization, set the Dbf, Ntx, ...
End Sub
```

Cols [Integer]

```
[Var] = [Form!]Brw.Cols [ = IntegerExp]
```

The Cols property is an integer, available only at run-time, which sets the number of data columns displayed by the browse. It must be a positive number and no bigger then BRW_MAX_NO_OF_COLS (see Constants).

Assigning the Cols of the control is the first logical step of initialization, although is not preemptive. But when you change the Cols everything is resized / changed so it is a good practice to set the Cols first, and when you change it, change the others related properties as well. The next step would be to assign the columns data width (ColWidth) and then the columns Header.

For a detailed explanation about the browse initialization see Introduction and the example there.

ColWidth [Array of Integers]

```
[Var] = [Form!]Brw.ColWidth(n%) [ = IntegerExp]
```

The ColWidth property is an array of integers, available only at run-time, which sets the width of the data displayed in each column, from 0 to Cols - 1, in characters. It must be positive and less than BRW_MAX_COL_WIDTH for each column (see Constants).

After assigning the Cols (number of columns) of the control, the next logical step of initialization is to assign the columns data width for each column and then the columns Header.

The value assigned for each column tells the control how much space to allocate for each column's data, and also for the column header. You should use these values also when assigning the Header and when writing code for the GetLine event. Data returned to the browse will be assigned to each column using the values in the ColWidth array.

The ColWidth property can be dynamically changed, this will resize the internal buffer maintained by the control, and will also resize the ColWidthPix property for the respective column. Please note that changing the ColWidthPix() or resizing the columns with the mouse will **NOT** affect the ColWidth!

For a detailed explanation about the browse initialization see Introduction and the example there.

ColWidthPix [Array of Integers]

```
[Var] = [Form!]Brw.ColWidthPix(n%) [ = IntegerExp]
```

The ColWidthPix property is an array of integers, available only at run-time, which sets the width of each column (the displayed width in pixels not the data width in characters as ColWidth does), valid indexes are from 0 to Cols - 1. This property changes when the ColWidth changes and also when the user resizes the column with the mouse, or when the font changes. Changing this property will not affect how many characters are stored in the internal buffer for each column.

The default value is the column width in characters multiplied by the average character width for the selected font.

You would have no reason to set this property from code, except one case: you want to restore a previous state of a user defined query window, see the example below.

Example:
The following example stores the last position and other settings of a user defined query into a user type, which is in turn saved to a disk file or INI file when the user exits the application. We will just show the code to save the last status of the window and restore it from the sStatus variable. Saving and restoring to/from file are not shown, you can use either a random file or some WinAPI calls like Get/SetProfileString.

```
DefType MYSTATUS
  left                            As Integer
  top                             As Integer
  width                           As Integer
  height                          As Integer
  ColWdt(0 to BRW_MAX_NO_OF_COLS) As Integer
  ' other data as DBF name, index name, scope, ...
End Type
Global sStatus As MYSTATUS

Sub Form_Unload( Cancel As Integer )
  ' save form position and browse column widths
  sStatus.left   = Me.Left
  sStatus.top    = Me.Top
  sStatus.width  = Me.Width
  sStatus.height = Me.Height
  For k = 0 to Brw.Cols - 1
    sStatus.ColWdt(k) = Brw.ColWidthPix(k)
  Next k
  ' save other data to sStatus
End Sub

Sub Form_Load()
  ' initialize the browse, ...
  ' then restore status, or start with default values if no status saved
  If sStatus.height = 0 Then Exit Sub    ' no previously saved status, exit
  Me.Left   = sStatus.left
  Me.Top    = sStatus.top
  Me.Width  = sStatus.width
  Me.Height = sStatus.height
  For k = 0 to Brw.Cols - 1
```

```
    Brw.ColWidthPix(k) = sStatus.ColWdt(k)
  Next k
End Sub
```

ColAlign [Array of Integers]

```
[Var] = [Form!]Brw.ColAlign(n%) [ = IntegerExp]
```

The ColAlign property is an array of integers, available only at run-time, which sets the way the column headers and data are aligned (justified) within the display rectangle. The headers and data can have different alignments (header left justified, data right justified for example). By default both the headers and the data are left aligned.

Available alignments are left (default), center and right.

You can set the column alignments to a combination of data + header alignment constants, for example `BRW_ALIGN_LEFT + BRW_ALIGNH_RIGHT` (see Constants for the values).

After assigning the number of columns (Cols) of the control and the column widths (ColWidth), you can set the column alignment for each column, as required.

Example:

```
Sub Form_Load()
  ' a simple database: CUSTOMER, 3 fields: CUSTNAME - Char 30,
  ' CUSTADDR - Char 30, CUSTNO - Numeric 10.0
  ' set the number of columns to 3, and column width as above
  Brw.Cols = 3
  Brw.ColWidth(0) = 30       ' CUSTNAME
  Brw.ColWidth(1) = 30       ' CUSTADDR
  Brw.ColWidth(2) = 10       ' CUSTNO

  Brw.ColAlign(0) = BRW_ALIGN_CENTER + BRW_ALIGNH_CENTER   ' both centered
  Brw.ColAlign(1) = BRW_ALIGN_LEFT + BRW_ALIGNH_CENTER     ' left, header
  right
  Brw.ColAlign(2) = BRW_ALIGN_RIGHT + BRW_ALIGNH_RIGHT     ' both right

  ' and now set the header
  Brw.Header = pad("Customer name", 30) + pad("Address", 30) + "Number"

  ' continue with intitialization, set the Dbf, Ntx, ...
End Sub
```

SpcColor [Array of Integers]

```
[Var] = [Form!]Brw.SpcColor(n%) [ = IntegerExp]
```

The SpcColor property is an array of integers (valid values are **True** and **False**). This property, in conjunction with ColorSpecial and ColorTSpecial, is used to display the selected column(s) in a different color. You can set the background and foreground color of the special colored columns at design time or run time, then after initializing the browse, set the SpcColor to True for all the columns you want displayed in the special color. By default no column is specially colored and the two special color are the same as the normal colors.

You can use this property to display the editable columns in a different color, or to highlight a calculated column, for example.

If editing is enabled for the special columns, the color of the edit window is the same as the columns color.

Example:

```
Sub Form_Load()
  ' a simple database: CUSTOMER, 3 fields: CUSTNAME - Char 30,
  ' CUSTADDR - Char 30, CUSTNO - Numeric 10.0
  ' set the number of columns to 3, and column width as above
  Brw.Cols = 3
  Brw.ColWidth(0) = 30       ' CUSTNAME
  Brw.ColWidth(1) = 30       ' CUSTADDR
  Brw.ColWidth(2) = 10       ' CUSTNO

  ' set color combination for the special columns
  Brw.ColorSpecial  = QBColor(12)  ' intense red background
  Brw.ColorTSpecial = QBColor(10)  ' intense blue text
  Brw.SpcColor(0)   = True   ' Display CUSTNAME in the special col.
  combination

  Brw.Header = pad("Customer name", 30) + pad("Address", 30) + "Number"

  ' continue with intitialization, set the Dbf, Ntx, ...
End Sub
```

ColEdit [Array of Integers]

```
[Var] = [Form!]Brw.ColEdit(n%) [ = IntegerExp]
```

The ColEdit property is an array of integers (valid values are **True** and **False**). This property, when set to **True**, will enable editing for the specified column. By default editing is disabled for all columns. If you set this property to **True** for a column, but the ColField() property for the same column is empty, you have to replace the user edited value yourself in the EditValid event.

At run-time the EditWhen event will be fired prior to enter Edit mode, then an EditValid event is fired when the user leaves the Edit mode with intention to save.

You can prevent the user to enter Edit mode by setting the **lCancel** argument of the EditWhen event to **True**, if for any reason you do not want edit for current record.

You can force data saved to be valid by setting the **lOk** argument of the EditValid event to **False**, you can even return a suggested value in the **cField** string argument.

See Editing for more details.

If the data entered by the user (the **cField** argument of EditValid) is Ok, you can replace it, the locking is your responsibility.

NOTES:

Do not issue a BRW_ACT_REFRESHLINE in the EditValid, refreshing is done for you. Do not change the record pointer while in the events, or if you do, then restore it.

The number of characters allowed in Edit mode is equal to the ColWidth() - the data width for the respective column.

Example:

```
Sub Form_Load()
  ' a simple database: CUSTOMER, 3 fields: CUSTNAME - Char 30,
  ' CUSTADDR - Char 30, CUSTNO - Numeric 10.0
  ' set the number of columns to 3, and column width as above
  Brw.Cols = 3
  Brw.ColWidth(0) = 30       ' CUSTNAME
  Brw.ColWidth(1) = 30       ' CUSTADDR
  Brw.ColWidth(2) = 10       ' CUSTNO

  Brw.ColEdit(0)  = True     ' enable edit for the CUSTNAME

  Brw.Header = pad("Customer name", 30) + pad("Address", 30) + "Number"

  ' continue with intitialization, set the Dbf, Ntx, ...
End Sub

Sub Brw_EditValid( nCol As Integer, cField As String, lOk As Integer )
  ' nCol indicates what column is in edit mode, we have only one
  If Trim$(cField) = "" Then       ' avoid empty customer names
    MsgBox "Customer name cannot be blank!"
    lOk = False: Exit Sub
  End If
  If vxLockRecord() Then                  ' lock the record
    vxReplString "CUSTNAME", (cField)     ' replace the field
    j = vxUnlock()                        ' unlock
  Else
```

```
    ' Lock failed, let the user know, eventually more elegant recovery
    MsgBox "Cannot lock the record, try later please"
  End If
End Sub
```

ColField [Array of Strings]

```
[Var] = [Form!]Brw.ColField(n%) [ = StringExp]
```

The ColField property is an array of strings (valid field names from the current work area). If you assign the ColField property for a column, editing will be enabled for this column, and the control will be in charge of replacing the data in the database as entered by the user. The ColEdit() property for the respective column is automatically set to **True**. The EditWhen and EditValid events will be fired as usual, but you do not have to write any code for data replacing, just for data validation, compare the example at ColEdit() with the one bellow.

The ColField property must be set after assigning the Dbf property, so that the control can check if the field names assigned are valid.

At run-time the EditWhen event will be fired prior to enter Edit mode, then an EditValid event is fired when the user leaves the Edit mode with intention to save.

You can prevent the user to enter Edit mode by setting the **lCancel** argument of the EditWhen event to **True**, if for any reason you do not want edit for current record.

You can force data saved to be valid by setting the **lOk** argument of the EditValid event to **False**, you can even return a suggested value in the **cField** string argument.

See Editing for more details.

NOTES:

Do not issue a `BRW_ACT_REFRESHLINE` in the EditValid, refreshing is done for you. Do not change the record pointer while in the events, or if you do, then restore it.

The number of characters allowed in Edit mode is equal to the ColWidth() - the data width for the respective column.

If the control cannot lock the record a FireReplFail event is fired, and the edit is cancelled

Example:

```
Sub Form_Load()
  ' a simple database: CUSTOMER, 3 fields: CUSTNAME - Char 30,
  ' CUSTADDR - Char 30, CUSTNO - Numeric 10.0
  ' set the number of columns to 3, and column width as above
  Brw.Cols = 3
  Brw.ColWidth(0) = 30       ' CUSTNAME
  Brw.ColWidth(1) = 30       ' CUSTADDR
  Brw.ColWidth(2) = 10       ' CUSTNO
  Brw.Header = pad("Customer name", 30) + pad("Address", 30) + "Number"
  ' continue with intitialization, set the Dbf, Ntx, ...
  ' after
  Brw.ColField(0)  = "CUSTNAME"    ' enable edit for the CUSTNAME column

End Sub

Sub Brw_EditValid( nCol As Integer, cField As String, lOk As Integer )
  ' nCol indicates what column is in edit mode, we have only one
  If Trim$(cField) = "" Then       ' avoid empty customer names
    MsgBox "Customer name cannot be blank!"
    lOk = False
  End If
End Sub
```

Col, Row [Integer]

```
[Var] = [Form!]Brw.Col [ = IntegerExp ]
[Var] = [Form!]Brw.Row [ = IntegerExp ]
```

The Col and Row properties represent the current column and row in the browse (where the highlighted cell is). The user can change any of them by moving with the arrows, clicking with the mouse, etc.

If you assign them, Col must be between 0 and Cols - 1, Row must be between 0 and LastLine - 1, otherwise a run-time error is generated.

By assigning Row and / or Col you force the control to move the highlighted cell there.

Example:

```
' this example shows a way to keep the user out of the frozen columns
Sub Brw_Change( nRowCol As Integer )
  If Brw.Col < Brw.LeftFrozen Then Brw.Col = Brw.LeftFrozen
End Sub
```

LeftFrozen [Integer]

```
[Var] = [Form!]Brw.LeftFrozen [ = IntegerExp ]
```

The LeftFrozen property sets the no of columns to be visible all the time, these columns start with column 0 to LeftFrozen - 1. The frozen columns will be excluded from the horizontal scroll, when the user moves to the right or left, either with the mouse or using the keyboard.

The LeftFrozen columns will be displayed in the ColorFrozen / ColorTFrozen combination, by default these colors are the same with the normal colors. Increasing / decreasing the LeftFrozen will dynamically add / remove fixed columns to the left edge. If a column is both Special color and Frozen the Frozen colors will take precedence.

If the combined width of all the frozen columns exceeds the control width, then the control cannot be scrolled horizontally. The LeftCol is the number of the left-most visible column excluding the frozen columns.

Example:

```
' freeze 2 columns to the left
Brw.LeftFrozen = 2
```

Dbf [Integer]

```
[Var] = [Form!]Brw.Dbf [ = IntegerExp ]
```

The Dbf property is a valid handle to an open database, the one you want to browse. You obtain this handle from one of the vxUseDbf() or vxUseDbfRO() vxBase functions. After initializing the browse: number of columns, column data width, alignment, header, the next step is to open the database file and assign the handle to the Dbf property.

If you want to set a standard filter, set it before assigning the handle to the Dbf property, in this way you will avoid a useless refresh.

When you set this property, all the database related properties like Ntx, IndexExp, etc., and most of the array properties (like ColField, etc.) are reset to the default values. After this the control will start to retrieve data, that is, it will fire a GetLine event for each row of data required; if the control is visible the data will be displayed as well.

Example: (for another example see also the Introduction and the example there)

```
Sub Form_Load()
  ' a simple database: CUSTOMER, 3 fields: CUSTNAME - Char 30,
  ' CUSTADDR - Char 30, CUSTNO - Numeric 10.0
  ' set the number of columns to 3, and column width as above
  Brw.Cols = 3
  Brw.ColWidth(0) = 30       ' CUSTNAME
  Brw.ColWidth(1) = 30       ' CUSTADDR
  Brw.ColWidth(2) = 10       ' CUSTNO

  Brw.Header = pad("Customer name", 30) + pad("Address", 30) + "Number"

  nD = vxUseDbf("CUSTOMER")  ' open the database
  If nD = 0 Then
    MsgBox "Cannot open CUSTOMER.DBF!"
  Else
    vxFilter ".NOT.DELETED()"
    j = vxTop()
    Brw.Dbf = nD' assign the handle and start the browse
  End If
End Sub
```

Ntx [Integer]

```
[Var] = [Form!]Brw.Ntx [ = IntegerExp ]
```

The Ntx property is a valid handle to an open index file, the one you want to use in the browse. You obtain this handle from the vxUseNtx() vxBase function. After initializing the browse: number of columns, column data width, alignment, header, and the Dbf assign the Ntx property.

When you set this property, all the database related properties like IndexExp, FilterExp, etc. are reset to the default values. After setting or changing the Ntx property the control will refresh itself, and reposition to the first record in the new index order. The Index Scope and Pseudo-Filter are cleared.

Example: (for another example see also the Introduction and the example there)

```
Sub Form_Load()
  ' a simple database: CUSTOMER, 3 fields: CUSTNAME - Char 30,
  ' CUSTADDR - Char 30, CUSTNO - Numeric 10.0
  ' indexed on CUSTNAME to CUSTOMER.NTX
  ' set the number of columns to 3, and column width as above
  Brw.Cols = 3
  Brw.ColWidth(0) = 30       ' CUSTNAME
  Brw.ColWidth(1) = 30       ' CUSTADDR
  Brw.ColWidth(2) = 10       ' CUSTNO

  Brw.Header = pad("Customer name", 30) + pad("Address", 30) + "Number"

  nD = vxUseDbf("CUSTOMER")        ' open the database
  If nD = 0 Then
    MsgBox "Cannot open CUSTOMER.DBF!"
  Else
    Brw.Dbf = nD             ' assign the DBF handle
    nI = vxUseNtx("CUSTOMER")      ' open the index file
    If nI = 0 Then
      MsgBox "Cannot open CUSTOMER.NTX"
    Else
      Brw.Ntx = nI               ' assign the NTX handle
    End If
  End If
End Sub
```

IndexExp, IndexExp2 [String]

```
[Var] = [Form!]Brw.IndexExp  [ = StringExp ]
[Var] = [Form!]Brw.IndexExp2 [ = StringExp ]
```

The IndexExp and IndexExp2 properties are used to define a Index Scope at run time. For a detailed description please see the Index Scope and the example there. An index must be open and active, and the handle of the index assigned to the Ntx property, in order to have an Index Scope, otherwise a run-time error is generated.

Setting the IndexExp will clear the IndexExp2, if the former is before the the later in index order. In this case the FilterExp will be also cleared. In any case the database will be positioned on the first record in Scope, you do not have to refresh all records in view.

In order to extract all the records for which the string resulting from the index key evaluation (shortly called the index key) begins with a certain expression (for example all invoices begining with "1993") you would set the IndexExp to that expression ("1993" for example), and IndexExp2 to "" (empty string).

To extract all the records for which the index key is between two string expressions, set IndexExp to the first (upper limit) expression, and IndexExp2 to the second (lower limit) expression. For example to see all names between "RANSOME" and "SMITH" set IndexExp to "RANSOME" and IndexExp2 to "SMITH".

To reset the index scope set IndexExp to "" - an empty string. If no records fall into the Scope then the EmptyFile property is set to True, and the control window cleared.

Example:
The following example presents a Sub in one of your modules to set a Index Scope, the Sub can be called from a menu item's or from a command button's click event, for example, and can be used for any active browse control. You would probably want to replace the InputBox$ with something less ugly.

```
Sub SetScope( Brw As Control )
  Dim cI As String

  ' display the current index key
  cI = "Current index expression: " & vxNtxExpr(vxNtxCurrent())
  ' get the IndexExp
  Brw.IndexExp  = InputBox$( cI, "Enter index scope", (Brw.IndexExp))

  ' if this one is empty, then exit
  If Brw.IndexExp = "" Then Exit Sub

  ' show also IndexExp, and get the IndexExp2
  cI = cI & chr$(13) & chr$(10) & "First scope: " & Brw.IndexExp
  Brw.IndexExp2 = InputBox$( cI, "Enter index scope", (Brw.IndexExp2))
End Sub
```

FilterExp [String]

```
[Var] = [Form!]Brw.FilterExp  [ = StringExp ]
```

The FilterExp property is used to extract a subset of the records in the Index Scope at run time. For a detailed description please see the Pseudo-Filter and the example there. An index must be open and active, and the handle of the index assigned to the Ntx property, and also a Index Scope must be active in order to have a Pseudo-Filter, otherwise a run-time error is generated.

The string expression is a xBase expression which can be evaluated to a logical value. If the expression evaluates to **.T.** (**True**) then the record is shown, otherwise is skipped. For example the expression can be something like `"BALANCE>=0 .AND. .NOT. PAID"`. The database will be positioned on the first record matching the criteria (no need to refresh), if none is found EmptyFile is set to **True** and the control window is cleared.

To reset the pseudo filter set FilterExp to "" - an empty string.

Example:
The following example presents a Sub in one of your modules to set a Pseudo-Filter, the Sub can be called from a menu item's or from a command button's click event, for example, and can be used for any active browse control. You would probably want to replace the InputBox$ with something better.

```
Sub SetPseudoFilter( Brw As Control )
  Dim cI As String

  ' display the current index key and scope
  cI = "Current index expression: " & vxNtxExpr(vxNtxCurrent()) + CRLF
  If Brw.IndexExp2 = "" Then
    cI = cI & "Index scope: " & Brw.IndexExp
  Else
    cI = cI & "Index scope from: " & Brw.IndexExp & " to " & Brw.IndexExp2
  End If
  ' set error handler and get the new FilterExp, the control will validate
  it
  On Error Resume Next
  Brw.FilterExp = InputBox$( cI, "Enter filter", (Brw.FilterExp))
  On Error GoTo 0
End Sub
```

SearchKey [String]

```
[Var] = [Form!]Brw.SearchKey  [ = StringExp ]
```

The SearchKey is a handy way to do a fast search on an indexed file, using the index key. An index must be opened and the Ntx property must be set to a valid index handle before you can use the SearchKey. You set the SearchKey property to the expression to search for and check the Action property to see if it is **True** - that means the expression is found, the database is positioned on the found record, and all records displayed are refreshed.

The way the SearchKey works depends upon the Index Scope setting:

1. no Index Scope: simple SEEK
2. single SCOPE (all records begining with "1993" for example): the control performs a relative SEEK within the scope, for example: SEEK "1993" + SearchKey.
3. double SCOPE (all record begining with: from "1991" to "1993" for example) the SEEK is absolute, and should NOT fall outside the scope and eventually the pseudo-filter, otherwise the refresh will generate an EmptyFile = **True**. It is your responsibility to make sure the search key falls between the two scopes.

The SearchKey remains unchanged until you reset it, so you can remember what was the user looking for last time.

Example:
The following example presents a Function in one of your modules to serch for something. You would probably want to replace the InputBox$ with something better.

```
Function MySearch( Brw As Control ) As Integer
  Dim cI As String, cS As String

  ' display the current index key and scope
  cI = "Current index expression: " & vxNtxExpr(vxNtxCurrent()) + CRLF
  If Brw.IndexExp2 = "" Then
    cI = cI & "Index scope: " & Brw.IndexExp
  Else
    cI = cI & "Index scope from: " & Brw.IndexExp & " to " & Brw.IndexExp2
  End If
  ' Get the new search key
  cS = InputBox$( cI, "Enter filter", (Brw.SearchKey))
  If Brw.IndexExp2 <> "" Then
     ' check if cS is in scope
     If Brw.IndexExp <= cS And cS <= Brw.IndexExp2 Then
       Brw.SearchKey = cS
       If Brw.Action = True Then MsgBox "Found!" Else MsgBox "Not found!"
       MySearch = (Brw.Action = True)
     Else
       MsgBox "Search criteria out of scope!"
     End If
  Else
     Brw.SearchKey = cS
     If Brw.Action = True Then MsgBox "Found!" Else MsgBox "Not found!"
     MySearch = (Brw.Action = True)
  End If
End Sub
```

Action [Integer]

```
[Var] = [Form!]Brw.Action [ = IntegerExp ]
```

The Action property is probably the one you will use most, and it is very important to understand what it does. This is the way you can send messages to the control, requiring it to perform some action. It does not hold any meaningful value - it is 0, except for one case: when you are searching for an expression it will hold **True** (-1) if it is found, and **False** otherwise (see SearchKey for details).

To request the control to perform a refresh line action - that is for example if you have edited the current record, set the Action property to `BRW_ACT_REFRESHLINE`. All the `BRW_ACT_*` constants are defined in the include file `BRW_INC.TXT` and shown in the Constants section. You can merge this file with one of your modules, it is recommendable to use the constants, not the numeric values.

Available actions:

`BRW_ACT_REFRESHLINE`: Refresh the current line of data. It is required after **EDIT** one record. Not required if editing was done in the browse. It will fire one GetLine event. IMPORTANT NOTE: if the database is indexed and there is an active index handle assigned to the Ntx property you should check if the index key was not modified by the current record edit, because if it was you might need to refresh ALL records. A simple algorithm for this is presented below:

store the current value of the index key to a variable before replacing the data:

```
cOldIndKey = Space$(255)
j = vxEvalString( vxNtxExpr(vxNtxCurrent()), cOldIndKey )
```

replace data (perform all the vxReplXXXX you need)

compare the new index key with the variable stored before and if they are different a refresh all is required, otherwise the index key was not affected by the edit / replace:

```
cNewIndKey = Space$(255)
j = vxEvalString( vxNtxExpr(vxNtxCurrent()), cNewIndKey )
If cNewIndKey <> cOldIndKey Then
   Brw.Action = BRW_ACT_REFRESHALL
Else
   Brw.Action = BRW_ACT_REFRESHLINE
End If
```

`BRW_ACT_REFRESHALL`: Refresh the lines visible in the control. It is required after APPEND, DELETE, or whenever you change the record pointer (position) outside the browse. Not required when you set any of the controls database properties line Dbf, Ntx, IndexExp, IndexExp2, FilterExp, SearchKey, in this case the refresh all is automatic. Refresh all also executes the refresh Vertical Bar, see next.

`BRW_ACT_REFRESHBAR`: Refresh the data of the vertical scroll bar, it is not normally required, you should use the refresh all (above) instead. Maintained for compatibility with previous versions of the control.

`BRW_ACT_GOTOP`: Move to the first record in the database or in scope, and refresh all visible records.

`BRW_ACT_GOBOTTOM`: Move to the last record in the database or in scope, and refresh all visible records.

`BRW_ACT_UP`: Move one record up (if posible), respects the scope and pseudo-filter.

`BRW_ACT_DOWN`: Move one record down (towards the end of file, if posible). Respects the scope

and pseudo-filter, you should use this instead of SKIP 1, when performing some action (like printing, copying to another file) on a user selected sub-set of records.

BRW_ACT_LEFT: Move one column to the left, if posible.

BRW_ACT_RIGHT: Move one column to the right, if posible.

BRW_ACT_EDIT: Enter edit mode for the current cell (Row and Col), if editing is enabled for the column. You should postion the cursor using the Row and Col properties before entering edit mode.

Browse Events

The standard Visual Basic events supported are:
DRAGDROP, DRAGOVER, GOTFOCUS, KEYDOWN, KEYPRESS, KEYUP and LOSTFOCUS. Please see your VB manual for details about any of these events.

Custom Events

Change(nRowCol as Integer)	- row, col or record no has changed
GetLine(cLine as String)	- data line required
HitBottom()	- the user hits the end of file
HitTop()	- the user hits the beginning of file
EditWhen(nCol As Integer, cField As String, lCancel As Integer)	- data edit pre-validation
EditValid(nCol As Integer, cField As String, lOk As Integer)	- data edit post-validation
EditReplFail(nCol As Integer, cField As String)	- data replace failed

Change(nRowCol As Integer)

```
Sub Brw_Change( [Index As Integer, ] nRowCol As Integer )
```

This event is fired whenever the user changes the position of the highlighted cell, using the mouse or the keyboard (for example moving up, down, left, right, page up, page down, top of file, bottom of file, etc.).

The **Index** argument uniquely identifies the browse control from an array of browse controls; the **nRowCol** argument holds information about what has changed and can be one of the constants below (see Constants for values):

The record no has changed, the Row and Col are the same	BRW_CHANGE_REC
The record no and the Row have changed	BRW_CHANGE_ROW
The column no has changed, the same record no	BRW_CHANGE_COL
Both the row and column no have changed	BRW_CHANGE_BOTH

Example:
The following example shows how to refresh data displayed from the current record in a set of controls (like labels and text boxes, etc.), so that it is actual at all times.

```
Sub Brw_Change( nRowCol As Integer )
  If nRowCol <> BRW_CHANGE_COL Then
    ' if the record number has changed then redisplay data in different
    ' controls like labels, text, list, etc.
    Label1.Caption = vxField("FIELD1")
    Text1.Text     = vxFieldTrim("FIELD2")
    Select Case (vxInteger("FIELD3"))
      Case 1 To 5
        List1.ListIndex = vxInteger("FIELD3") - 1
      Otherwise
        List1.ListIndex = -1: List.Enabled = False
    End Select
    ' ..., and so on
  End If
End Sub
```

GetLine(cLine As String)

```
Sub Brw_GetLine( [Index As Integer, ] cLine As String )
```

This event is fired when the control requires a line (row) of data to be displayed, from the current record. The data for one row (record) is made from the concatenation of the data for each column padded with spaces / trimmed to the data width defined in the ColWidth property, in other words in the way you have defined it when you initialized the control last time. The concatenated string must be assigned to the **cLine** argument.

Data for each column would be normally database fields, but can be anything, including variables, arrays, etc, transformed to a string.

No delimiters are required between two succesive column's data, for example, if we define the column width of the first two columns to the width of the fields CUSTNAME and CUSTADDR in the CUSTOMER database then we should use:

```
' in Form_Load()
Brw.ColWidth(0) = vxFieldSize("CUSTNAME")
Brw.ColWidth(1) = vxFieldSize("CUSTADDR")
'....

' in Brw_GetLine
cLine = vxField("CUSTNAME") + vxField("CUSTADDR")
' cLine = cLine + ... (the rest of columns)
```

The **Index** argument uniquely identifies the browse control from an array of browse controls; the **cLine** argument will return the data to the control. If left blank no information will be displayed, although the database is open and the record pointer correctly positioned.

Please see the Introduction and the example there.

HitTop(), HitBottom()

```
Sub Brw_HitTop    ( [Index As Integer] )
Sub Brw_HitBottom( [Index As Integer] )
```

This events are fired when the user has tried to move before BOF or past EOF, either using the mouse or the keyboard. Both events can be ignored, you can set a Beep statement in the event procedure, or ask the user if he/she wants to add a new record (in HitBottom) to emulate the standard xBase BROWSE command.

The **Index** argument uniquely identifies the browse control from an array of browse controls.

Example:

```
Sub Brw_HitBottom()
  If MsgBox("Add new record?", MB_ICONQUESTION + MB_YESNO) = IDYES Then
    ' add a blank record and reposition the browse
    If vxAppendBlank() Then Brw.Action = BRW_ACT_REFRESHALL
  End If
End Sub

Sub Brw_HitTop()
  Beep
End Sub
```

EditWhen(nCol As Integer, cField As String, lCancel As Integer)

```
Sub Brw_EditWhen( nCol As Integer, cField As String, lCancel As Integer )
```

This event is fired whenever the user tries to edit a field from column **nCol**, where **nCol** is a column enabled for editing. The **cField** argument is the current value of the field - the one that will be edited. The **lCancel** argument is a boolean flag (set to **False** by default), which tells the control if edit should be allowed; setting **lCancel** to **True** will not allow the user to enter in edit mode.

The **Index** argument uniquely identifies an element of a control array.

This event allows you to make a data pre-validation, before the user enters edit mode, and cancel the edit if you decide so, even if the column is enabled for edit.
See also Editing for details.

Example:
The following example will allow editing of the date field DUEDATE, in column 3, enabled for editing, only if the field DUEAMOUNT is greater than 0.

```
Sub Brw_EditWhen( nCol As Integer, cField As String, lCancel As Integer )
  If nCol <> 3 Then Exit Sub       ' check only column 3, DUEDATE
  If Val(vxField("DUEAMOUNT")) <= 0 Then
     ' no edit allowed
     Beep: MsgBox "The DUEAMOUNT must be positive!"
     lCancel = True
  End If
End Sub
```

EditValid(nCol As Integer, cField As String, lCancel As Integer)

```
Sub Brw_EditValid( nCol As Integer, cField As String, lOk As Integer )
```

This event is fired whenever the user tries to save an edited value for column **nCol**, where **nCol** is a column enabled for editing, and in this respect is much like a VALID clause in a standard xBase GET command. The **cField** argument is the value entered by the user - the one that will be eventually replaced. The **lOk** argument is a boolean flag (set to **True** by default), which tells the control if the value is Ok; setting **lOk** to **False** will prevent the control to exit edit mode, until a valid value is entered or the edit is cancelled.

You can set the **cField** to a string expression, for example as a default value, if the user has entered a wrong value in edit mode.

Depending on the way the edit for the nCol column is defined, you should:

1. If the editing mode is enabled with auto-replace, that is the control will replace the field for you - the ColField property must be assigned a valid field name for this, you should just check the data validity.
   You should use this edit mode whenever posible, that is when the column expressions are database fields and not expressions, it is faster and does not require so much coding.
2. If the editing mode is enabled without auto-replace, that is you are responsible for replacing the data in the database file - the ColEdit property is set to True, but the ColField is empty, you should check the data validity, and if Ok replace the value in the database.

In either case you do not need to send a REFRESHLINE message, this is done automatically if you return **True** in the **lOk** argument.

The **Index** argument uniquely identifies an element of a control array.
See also Editing for details.

Example:
In the following example we have two columns enabled for editing, one is the DUEAMOUNT, column 2, enabled directly, the browse control will replace the entered amount. The second one is the DUEDATE, column 3, for which we do the replace ourselfs.

```
Sub Brw_EditValid( nCol As Integer, cField As String, lOk As Integer )
    If nCol = 2 Then                ' edit DUEAMOUNT
      If Val(cField) < 0 Then       ' no negative numbers allowed
        MsgBox "Please enter a positive value!"
        lOk = False
      End If
    ElseIf nCol = 2 Then                  ' edit DUEDATE
      If IsDate((cField)) Then            ' Ok, replace data
        If vxLockRecord() Then            ' lock record
          vxReplDateString "DUEDATE", (cField)      ' and replace if lock Ok
          j = vxWrite()
          j = vxUnlock()
        Else
          MsgBox "Cannot lock the record, try later please!"
        End If
      Else
        MsgBox cField & " is not a valid date!"
        lOk = False
      End If
```

```
    End If
  End If
End Sub
```

EditReplFail(nCol As Integer, cField As String)

```
Sub Brw_EditReplFail( nCol As Integer, cField As String )
```

This event is fired when the edit for this column is in auto-replace mode (the control is responsible for data replacement) and for some reason the replace of the data failed. The most probable cause is that the record could not be locked, or the data format is invalid. In both cases the edit was cancelled.

nCol is the column beeing edited. The **cField** argument is the value entered by the user. You can try a recovery in this event procedure, eventually.

The **Index** argument uniquely identifies an element of a control array.
See also Editing for details.

Example:
The following example will try to recover a replace failed for the DUEAMOUNT column 2.

```
Sub Brw_EditReplFail( nCol As Integer, cField As String )
  If nCol <> 2 Then Exit Sub       ' check only column 2, DUEAMOUNT
  If vxLockRecord() Then           ' lock record
    vxReplDouble "DUEAMOUNT", Val(cField)     ' and replace if lock Ok
    j = vxWrite()
    j = vxUnlock()
  Else
    MsgBox "Cannot lock the record, try later please!"
  End If
End Sub
```

Browse Constants

All the constants listed bellow are available as text in the include file **BRW_INC.TXT** provided.

Limitative Constants

Maximum number of columns in the browse	BRW_MAX_NO_OF_COLS	64
Maximum column width (in characters)	BRW_MAX_COL_WIDTH	256

Error Codes

Invalid index referrence to one of the array properties	BRW_ERR_BADINDX	81
Invalid number of columns	BRW_ERR_NO_OF_COLS	2700
Invalid column no	BRW_ERR_COL_NO	32702
Invalid row number	BRW_ERR_ROW_NO	32703
Invalid data column width (characters)	BRW_ERR_COL_WIDTH	32704
Invalid column width (pixels)	BRW_ERR_COL_WIDTHX	32705
Invalid database handle	BRW_ERR_DBF_AREA	32706
Invalid index handle	BRW_ERR_NTX_AREA	32707
Invalid index SCOPE expression	BRW_ERR_INDEX_EXP	32708
Invalid pseudo-filter expression	BRW_ERR_FILTER_EXP	32709
Invalid search expression	BRW_ERR_DBF_SEARCH	32710
Invalid field name for the ColField property	BRW_ERR_FIELDNAME	32711

Available actions (for the Action property)

Refresh current line (will trigger one GetLine event)	BRW_ACT_REFRESHLINE	1
Refresh all visible lines / records (impl. RefreshBar)	BRW_ACT_REFRESHALL	2
Refresh the vertical scroll bar (not needed normally)	BRW_ACT_REFRESHBAR	3
Go to the last record in file / scope and refresh all	BRW_ACT_GOBOTTOM	4
Go to the first record in file / scope and refresh all	BRW_ACT_GOTOP	5
Skip one record up, if not posible a HitTop is fired	BRW_ACT_UP	6
Skip one record down, if not pos. a HitBottom is fired	BRW_ACT_DOWN	7
Move to the next column to the right (if any)	BRW_ACT_RIGHT	8
Move to the next column to the left (if any)	BRW_ACT_LEFT	9
Edit current cell, if editing is enabled	BRW_ACT_EDIT	10

Possible values of the nRowCol argument of the Change event:

The record no has changed	BRW_CHANGE_REC	0
The record no and the Row have changed	BRW_CHANGE_ROW	1
The column no has changed	BRW_CHANGE_COL	2
Both the row and column no have changed	BRW_CHANGE_BOTH	3

Column text display alignment for the ColAlign propriety

The column text is left justified (default):	BRW_ALIGN_LEFT	0
The column text is centered	BRW_ALIGN_CENTER	1
The column text is right justified	BRW_ALIGN_RIGHT	2
The column header text is left justified (default):	BRW_ALIGNH_LEFT	4
The column header text is centered	BRW_ALIGNH_CENTER	8
The column header text is right justified	BRW_ALIGNH_RIGHT	16

Keyboard Summary

Vertical movement:

UP	**UP**
DOWN	**DOWN**
CTRL-UP	**FIRST LINE IN WINDOW**
CTRL-DOWN	**LAST LINE ON WINDOW**
PAGEUP	**PAGE UP**
PAGEDOWN	**PAGE DOWN**
CTRL-PAGEUP	**TOP OF FILE**
CTRL-PAGEDOWN	**BOTTOM OF FILE**

Horizontal movement:

LEFT ARROW	**LEFT**
RIGHT ARROW	**RIGHT**
HOME	**LEFT-MOST VISIBLE COLUMN**
END	**RIGHT-MOST VISIBLE COLUMN**
CTRL-LEFT	**PAGE LEFT**
CTRL-RIGHT	**PAGE RIGHT**
CTRL-HOME	**LEFT-MOST COLUMN**
CTRL-END	**RIGHT-MOST COLUMN**

Edit keys:

ENTER	**Enter edit mode, if edit for current column enabled**
Any alphanumeric key	**Same as Enter + replace field contents with the entered character**
ESC	**Cancel edit**
PAGEUP / PAGEDOWN, ENTER	**Exit edit mode with saving, stay on the same row**
UP / DOWN ARROW	**Save field and move to previous / next row (record) - same column**