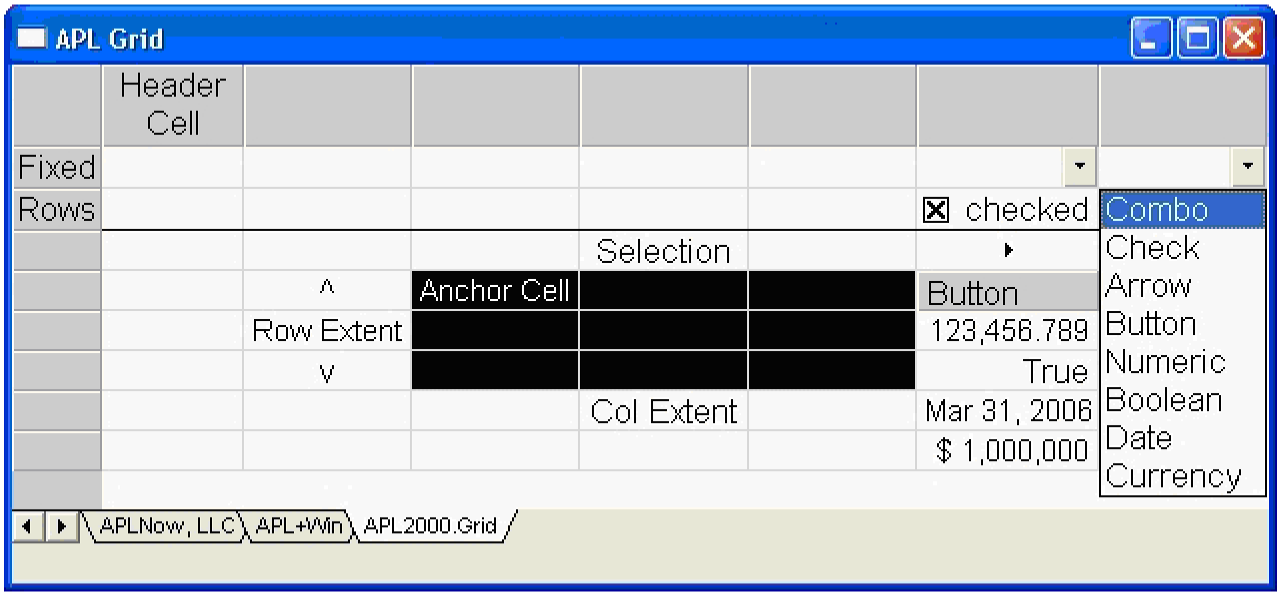
Using the APL Grid



APLNow LLC.
2640 Route 70 Bldg. 4
Manasquan, NJ 08736
(301) 208-7150

Copyright

APL.Grid for APL64

This edition published 2022.

Using the APL Grid

Overview

The APL Grid is an ActiveX control that has been tailored to fit smoothly into an APL64 user interface. It allows you to have a spreadsheet module in your application that you can define for your specific purposes. Each instance of an APL Grid is the equivalent of a workbook that allows you to have multiple sheets, each containing an arbitrary number of cells arranged in rows and columns. As with any spreadsheet, you can have numeric or character data in cells, as well as several special format data types, for example, currency.

There are two main conceptual differences between the APL Grid and a standalone spreadsheet application. One is that there is no calculation engine built into the grid; required calculations are performed in APL. This gives you the capability to manipulate data from ranges of cells as regular APL vectors or matrices. The second difference is that you, as the programmer, control what the user can do. You can make the grid as restrictive and prescriptive as any tightly controlled interface, or you can allow the user great latitude, with your program accepting the attendant responsibility for handling whatever the user tries to do.

As with any ActiveX control in APL64, the grid has a suite of properties and methods, and it recognizes specific events, for which you can write event handlers. This description assumes you have a working knowledge of the APL64 Windows interface and that you are familiar with using ActiveX therein. If you need more information on these topics, consult the APL+Win *Windows Reference Manual*; for ActiveX specifically, see the chapter, "Using ActiveX in APL+Win."

Installing the APL Grid

The APL Grid is contained in the file `APLGRID.DLL`. It is installed and registered in the same folder in which `APL64.EXE` is installed. When the APL Grid is moved to a different folder like C:\APL64, it must be registered again. Register the APL Grid file by running `REGSVR32` from the DOS Command Prompt console run as Administrator, specifying the full path where the APL Grid file is located, for example:

```
REGSVR32 C:\APL64\APLGRID.DLL
```

Uninstalling the Grid

To uninstall the grid, run `REGSVR32` with the `-u` switch, for example:

```
REGSVR32 -u C:\APL64\APLGRID.DLL
```

Distributing APLGrid.DLL

If you use the APL Grid in an application that you distribute to others, your installation program must register the APL Grid, or you must instruct your users to do so.

Identifying the Grid

Note that the grid has two version numbers. As for any ActiveX control, there is a version number that applies to the control itself, identifying the interface it presents to the outside world. You can query this in the version property of an instance of the object or see it in the fourth column of the xclasses property on the system object. Also for ActiveX controls, there is an internal version number for the specific DLL that implements the control; this number reflects the status of the DLL's internal behavior. You can see this four-part number (*major minor rev patch*) by invoking the XGridVersion method on an instance of the grid, or by right clicking on the grid file in Windows Explorer and viewing Properties/Version. If you request help from APLNow or send a message about the behavior of the grid, this is the version number to cite.

If you need to access the grid using low-level calls to the Windows API, the handle of the grid window can be retrieved from the hwnd or xHwnd property, once the grid is open in Windows.

Getting Started

Programming Alert: While you are programming the grid, you can invoke quick reference help on almost every ActiveX property, method, and event handler by executing a ⎕wi statement with a question mark in front of the name of the action you are interested in. The help shows the syntax in APL format, the names of the arguments, and usually (excluding the most complex) a short description of the relevant codes, values, or uses.

```
      ⎕wi '?xRow'                          ⍝APL expression to request help
xRow property:                             ⍝Help output: header
  Value@Long ← ⎕WI 'xRow'                  ⍝Help output: reference syntax
  ⎕WI 'xRow' Value@Long                    ⍝Help output: setting syntax
Page value; row index of active cell       ⍝Help output: description
Value: integer scalar or one-element vector ⍝Help output: description (continued…)
```

The syntax lines (such as the indented second and third lines of output shown above for xRow) are included in the **Syntax** section for each property, method, and event.

When you first create a grid control, it is simply a blank rectangle with a tab at the bottom of the grid that says "Page1"; it is up to you to define the number of rows, columns, and pages by setting the xRows, xCols, and xPages properties, respectively. Note that row and column settings apply only to the active page, which is defined by the property xPage, whose value is an origin 1 index. That value does not change if you rename the page by setting the xPageName property, for which you supply both the index and the name you wish to apply. You can set the name of a page other than the active page, and you can set multiple page names in the same statement. The tabs displaying the page names adjust individually to the lengths of the names you set.

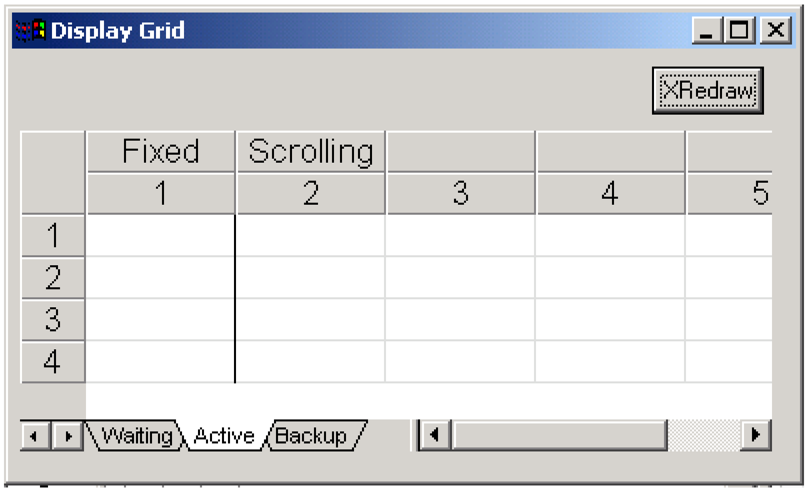
At the bottom of the grid, on the left, there is a pair of scroll buttons that allow the user to scroll the page tabs, followed by the tabs in page number sequence; on the right, if there are too many columns to fit the space, is a scroll bar that allows the user to scroll the grid itself horizontally. The space allotted for the tabs and their scroll buttons is specified in virtual-pixels in the xTabWidth property. Although it is not obvious, the user may change the width of this space by dragging the edge of the scroll bar on the right with the mouse cursor. If you set xTabWidth to 0, the tab strip disappears and the user cannot navigate to a different page of the grid.

By default, a grid page has a row of column headers and a column of row headers, which are outside of the body of the grid. You can use these header cells for labels to identify or characterize the content of a row or column. If the user clicks on a header cell, it selects the entire row or column to which it applies. You can increase the number of these headers, but note that the number of headers over each column is defined by the xHeadRows property (because the headers over columns are a row) and the number of row headers is defined by the xHeadCols property. Since the headers are outside the body of the grid, they remain visible even when you scroll; that is to say, the rows of column headers remain visible when you scroll vertically and the columns of row headers remain visible when you scroll horizontally.

You can also establish a number of rows of regular cells at the top of your grid and a number of columns of regular cells at the left of your grid that do not scroll. These values, which are zero initially, are in the xFixedRows and xFixedCols properties.

Example:

```
'fmDoc' ⎕wi 'Create' 'Form' ('where' 0 0 15 50) ('caption' 'Display Grid')
'fmDoc.bnPaint' ⎕wi 'New' 'Button' ('where' 0.5 40 1.5 7) ('caption' 'XRedraw')
'fmDoc.bnPaint' ⎕wi ('onClick' "'fmDoc.grid' ⎕wi 'XRedraw'")
⎕wself"'fmDoc.grid' ⎕wi 'New' 'APLNow32.Grid' ('where' 2.5 0.5 10 47)
⎕wi 'xPages' 3
⎕wi 'xPage' 2
⎕wi 'xRows' 4
⎕wi 'xCols' 5
⎕wi 'xPageName' (⍳3) ('Waiting' 'Active' 'Backup')
⎕wi 'xHeadRows' 2
⎕wi 'FixedCols' 1
```



Note: The statements that set the text in the column and row headers are explained later. The statements are reproduced here for your convenience, but if you paste them from Word to APL, you may have to fix the Format symbol (⍕).

```
⎕wi 'xText' ¯1 (⍳(⎕wi 'xCols')) (⍕¨⍳(⎕wi 'xCols'))
⎕wi 'xText'  (⍳(⎕wi 'xRows')) ¯1 ((⎕wi 'xRows') 1⍴ ⍕¨⍳(⎕wi 'xRows'))
⎕wi 'xText' ¯2 (⍳2) ('Fixed' 'Scrolling')
```

As with the numbers of rows and columns, the settings for the numbers of headers and fixed cells are page-specific, referring to the active page when referenced or set. Also page specific is the xGridLines property. By default this flag is 1, which means there are pale lines delineating regular cells. You can set this property to zero for a page to provide a blank slate on which only the positions of the contents of cells indicate their existence.

This property also affects lines between header cells, but the appearance of headers is different; this is an effect of the xBorderStyle property which has a different default appearance for headers and regular cells (see below for setting cell values on both regular cells and headers).

Programming Alert: For most programmatic actions you take that affect the content or appearance of the grid, the result is not immediately apparent on the screen. Either you or the user must take an action that causes the grid to be repainted. This behavior has the advantage that making multiple changes before painting reduces flicker. It carries the responsibility that when you are finished with a change or a series of changes, you should invoke the XRedraw or Paint method to bring the screen display up to date. Setting xGridLines is the first example of this phenomenon; the alert applies throughout the rest of this description. You can assign row and column arguments to this method to paint just the affected area, or you can paint the entire visible grid.

Grid Coordinate System

In order to understand how to set values, you must know how to address cells and ranges of cells.

Specifying Regular Cells

You identify a single cell by a pair of integers representing the row number and column number. (Row and column numbers for regular cells always begin with `1`; they are not `⎕io`-sensitive.) You identify a pattern of cells by supplying a vector of row numbers and a vector of column numbers. This is similar to indexing into a matrix except you don't use brackets, you don't separate the arguments with a semicolon, and you do not generate an error if you specify indices greater than the number of rows or columns on the active page. You can specify the rows and columns in any order; if you reference values, you specify only the rows and columns; if you are assigning values, the values you want to assign follow the coordinates you specify.

Thus, a statement like the first one below results in a grid pattern as shown on the left, while a statement like the second one results in a grid pattern as shown on the right:

```
      'fmA.grid' ⎕wi 'xText' (2 4) (1 3 5) (2 3ρ 'aa' 'bb' 'cc' 'dd' 'ee' 'ff')
      'fmA.grid' ⎕wi 'xText' (4 2) (3 5 1) (2 3ρ 'aa' 'bb' 'cc' 'dd' 'ee' 'ff')
 -- -- -- -- --       -- -- -- -- --
 aa -- bb -- cc       ff -- dd -- ee
 -- -- -- -- --       -- -- -- -- --
 dd -- ee -- ff       cc -- aa - bb
```

Programming Alert: When you assign values to cells, the grid control does not force the amount of data in the third argument to exactly match the number of cells in the coordinate arguments. If you supply fewer data than the row and column arguments imply, the grid reuses data from the beginning by tiling the values. If there are more data than implied by the arguments, the grid control attempts to use all the data by placing values in cells adjacent to the last one specified. In the first example above, if you specify the same row and column arguments but assign a `3`-by-`3` matrix of data, the control would place values in the fifth row of the grid. If you assign a `2`-by-`4` matrix, it would place values in the sixth column. Even if you have only five columns in your grid, the data are not lost. If you add a sixth column at a later time, the values appear. Note, however, that this behavior is not necessarily true when assigning attributes to cells.

Note also that in the second example, if you assign a `2`-by-`4` matrix, the additional data items would appear in the second column because that is adjacent to the last one specified. If you assign a `2`-by-`5` matrix, the last data items would overwrite the third column.

Using Scatter-Point Indexing

Scatter-Point indexing allows you to access noncontiguous, nonrectangular collections of cells scattered throughout the grid in a single `⎕wi` call rather than iterating with multiple `⎕wi` calls. This is especially useful in cases such as collecting the values of cells that have changed or in handling the scattered array of cell coordinates in the onXVirtualLoad event. Both cases involve a matrix of scattered cell coordinates that are awkward to access with the normal rectangular block indexing.

For example, the xChanges property returns a two-column matrix of cell coordinates scattered around the grid. You can use a two-column matrix such as this in place of the Rows argument in any property such as the xText property that takes a pair of Rows Cols arguments. For example:

```
      changes " ⎕wi 'xChanges'
      changes
 1 3
 2 5
 4 1
      mv " '#' ⎕wi 'missing'
      values " ⎕wi 'xText' changes mv
      values
 aaa bbb 999.00
```

The changes matrix in the example above tells us that the cells at coordinates 1 3 (row 1, column 3), 2 5, and 4 1 have changed. We can use it in place of the normal Rows argument to specify scatter-point indexing. When you use a matrix for the Rows argument, the system missing value (from result of the system object's missing property) must be specified as a placeholder for the Cols argument.

You can also set values using scatter-point indexing using a similar syntax. As when referencing the property, the scatter-point matrix of cell coordinates is used in place of the Rows argument and the system missing value is used as a placeholder for the Cols argument when setting the property.

For example, you can set the above changed cells to values aa, bb, and cc as follows:

```
      ⎕wi 'xText' changes mv ('aa' 'bb' 'cc')
```

This would result in values being stored into the grid at the positions shown below:

```
 -- -- aa -- --
 -- -- -- -- bb
 -- -- -- -- --
 cc -- -- -- --
```

You can use a singleton value to store the same value at all locations such as this:

```
      ⎕wi 'xText' changes mv 'xx'
```

This would result in values being stored into the grid at the positions shown below:

```
 -- -- xx -- --
 -- -- -- -- xx
 -- -- -- -- --
 xx -- -- -- --
```

Programming Alert: It is important to note that the shape of the result value is always a vector when using scatter-point indexing to reference a property. And the value argument must always be a vector or scalar when setting a property using scatter-point indexing. This is in shape contrast to the matrix shaped value used with normal indexing. Each element in the value correspond to the coordinates specified by each row of the scatter-point matrix argument.

You can also use scatter-point indexing in the three-dimensional xImage property that normally takes a Places, Rows, and Cols arguments. In this case you specify a three-column matrix of place, row, and column coordinates instead of the normal Places argument and use missing values for the Rows and Cols arguments. For example:

```
      coordinates
 1 10 9
 2 10 9
 2 15 1
 4 11 2
      images ← ⎕wi 'xImage' coordinates mv mv
      images
55 66 77 88
      ⎕wi 'xImage' coordinates mv mv 0
```

The coordinates matrix above specified scatter coordinates 1 10 9 (place 1, row 10, column 9), 2 10 9, 2 15 1, and 4 11 2. When the xImage property is referenced it shows us that image indexes 55, 66, 77, and 88 are set for the cells and places previously. Finally we reset all of these image indexes back to zero (0).

Specifying Cells for Default Values

You can specify default values for cells, either their content or attributes, for example, the background color of a non-selected cell. If you want to specify such a default for an entire page, you specify row and column coordinates of `0 0`. If you want to set a default property for a column you specify zero for the row and a positive scalar or vector for selected columns. If you want to set a default property for a row you specify a positive scalar or vector for selected rows and zero for the column.

Programming Alert: Default values exist in a hierarchy. If you set values for a page with `0 0`, and also set default values for a column, the column values override the page values. Similarly, if you set default values for a row, the row default values override both the column default values and page default values. If you set values for specific cells using positive number coordinates, those values override all three sets of defaults that may apply to that cell.

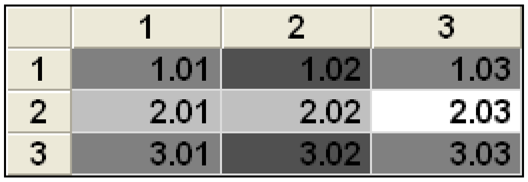
The order in which you set defaults makes no difference; the overrides apply. Once you set a default column value, changing the page default does not affect that column for that property and once you set a default row value, changing the page default or a column default does not affect that row for that property.

If you make a mistake in specifying cells, you can undo the effect by using the XDeleteCells method on specific cells or a default range. If you use this method, for example, on a column for which you set a column default value, the page defaults then apply. Note that using this method with one or both coordinates set to zero affects only the respective default values. Settings for specific values within the range are unchanged. If you use the method on specific cells, all the explicitly set values are erased, but the two levels of defaults apply.

As an example, consider the following settings of the page default, column default, row default, and a specific cell for the background color:

```
⎕wi 'xColorBack' 0 0 (256⊥128 128 128)    ⍝ Page default:   Middle Gray
⎕wi 'xColorBack' 0 2 (256⊥80 80 80)       ⍝ Column default: Dark Gray
⎕wi 'xColorBack' 2 0 (256⊥192 192 192)    ⍝ Row default:    Light Gray
⎕wi 'xColorBack' 2 3 (256⊥255 255 255)    ⍝ Specific cell:  White
```

This will give you a grid something like this:



	1	2	3
1	1.01	1.02	1.03
2	2.01	2.02	2.03
3	3.01	3.02	3.03

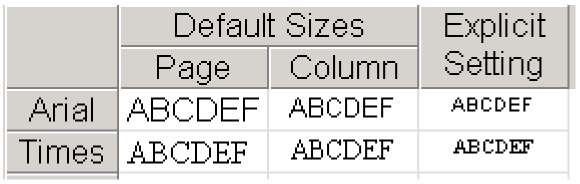
In the example above, we have the dark gray column default in column 2 overriding the middle gray page default, the light gray row default in row 2 overriding the column (and page) defaults, and the specific cell white value in row 2 column 3 overriding the row (and page) defaults.

Note that when you reference a column of cells using zero for the row argument, the grid returns a default value; that is, any default value you have set for the column . If you reference the page using arguments of `0 0`, the grid returns a default value for the page (or the grid) itself. This does not tell you if there is a specific setting for any cell in the column. If you want to query all the values for a given property in a column use a statement like:
`'fmA.grid' ⎕wi 'xProperty' (⍳ (⎕wi 'xRows'))` *colNumber*

Programming Alert: The inverse of this is equally true and may be surprising or misleading. If you query the explicit values for a range of cells for which only defaults apply, the grid may return an empty result or a Microsoft default value (this is not consistent across properties). If you want to see the default values, you must query with default (zero) arguments, or set xImplicitMode to `1`.

Implicit Mode

The picture below shows a grid with font settings as an example. The page defaults, which are set automatically when you create a grid, are Arial font (xFontName) and `19` pixels (xFontSize). Column `2` is given a column default setting of `16` pixels, and the cells in column `3` are given an explicit setting of `13` pixels. The cells in row `2` are given an explict setting of Times for the font. Note the values returned differ depending on whether you specify default or explicit coordinates; these are different settings regardless of what actually displays. Note also that the column default overrides the page default, which still exists, and the explicit settings override both defaults, although they also still exist for those cells. Finally, observe the effect of changing xImplicitMode on the values returned when you query with explicit coordinates those cells that have only default settings.



	Default Sizes		Explicit Setting
	Page	Column	
Arial	ABCDEF	ABCDEF	ABCDEF
Times	ABCDEF	ABCDEF	ABCDEF

```
      ⎕wi 'xFontSize' 0 0            ⍝  Reference page default font size
19
      ⎕wi 'xFontSize' 0 2 16         ⍝  Setting column default font size
      ⎕wi 'xFontSize' (⍳3) 3 13      ⍝  Setting explicit font size value
      ⎕wi 'xFontName' 2 (⍳3) 'Times'  ⍝  Setting explicit font name value
      ⎕wi 'xFontName' 0 0            ⍝  Page default
Arial
      ⎕wi 'xFontName' 0 2            ⍝  No column default setting

      ⎕wi 'xFontName' 1 2            ⍝  No explicit setting in this cell

      ⎕wi 'xFontName' 2 2            ⍝  Explicit setting in this cell
Times
      ⎕wi 'xFontSize' 0 0            ⍝  Page default
19
      ⎕wi 'xFontSize' 1 (⍳3)         ⍝  No explicit settings in columns 1 or 2
¯1 ¯1 13
      ⎕wi 'xFontSize' 0 (⍳3)         ⍝  Column default value only in column 2
¯1 16 ¯1
      ⎕wi 'xImplicitMode' 1
      ⎕wi 'xFontSize' 1 (⍳3)         ⍝  With implicit mode on, returns actuals
19 16 13                             ⍝  in all three columns
      ⎕wi 'xFontName' (⍳2) 1         ⍝  and both rows
Arial
Times
```

Specifying Header Cells

You identify header cells with negative numbers. The default row and column are `¯1`; if you assign additional header rows or columns (by setting xHeadRows or xHeadCols), they become `¯2`, `¯3`, etc., each successive value representing one step farther above or to the left of the regular grid. You can assign defaults to all headers using `¯2 ¯2`. The cell in the upper left corner between the header row and header column is `¯1 ¯1`. There is only one corner cell no matter how many additional headers are assigned, but you must have at least one of each to have a corner cell. You can set values, either default or specific, including colors and a label for the corner cell. **Behavior:** If the user clicks on the corner cell, it selects all the regular cells on the page.

Note that header cells have certain default settings (`¯2 ¯2`) that differ from regular cell default settings; these settings are not obvious if you reference the explicit value of a header cell property, because the default explicit value (`¯1`) allows the cell to inherit the page default settings. These special property settings include border style, background color, alignment and column size (for row headers); the default header settings for font name, font size, and row height (for column headers) are the same as the regular page default settings.

If you want to simply number your rows and columns for identification purposes, use these statements (if you paste them from Word to APL, you may have to fix the Format symbol):

```
      ⎕wi 'xText' ¯1 (⍳(⎕wi 'xCols')) (⍕¨⍳(⎕wi 'xCols'))
      ⎕wi 'xText'  (⍳(⎕wi 'xRows')) ¯1 ((⎕wi 'xRows') 1⍴ ⍕¨⍳(⎕wi'xRows'))
```

Programming Alert: If you want to assign a series of values to a column, you must specify a one-column matrix; specifying a vector argument places the first value in each row of your specified column and the remaining values in adjacent cells to the right. Further, if your specified column is the first column of row headers, the second value is not visible, since it applies to "column zero"; that is, the header column is `¯1`, and the first regular column is number `1`, which gets the third value from the vector.

The Active Cell and Specifying a Selection

At any given time, there is one cell that is the active cell on the active page of the grid; if you switch pages, the active cell may have different coordinates on the new page. If the grid has the focus, and the active cell is the only one selected, it has a dark, heavy border on all four sides. If the user presses a character key, the active cell enters edit mode and, by default, the character appears in the cell. In some cases, such as with the XEditStart method, the coordinates of the active cell are default arguments. You set the xActiveCell property with a two-element integer vector argument designating the row and column. You can also reference or set the arguments individually by using the xRow and xCol properties. (Note the similarity in names to the xRows and xCols properties that set the number of cells on the page.)

Although there is only one active cell, there can be more than one cell selected, in which case all the selected cells except the active cell are highlighted, and the active cell contrasts with the highlighted cells by having the default background color and a thin dark border on all four sides.

A user can create a selection by clicking in a cell and dragging the mouse or by holding the Shift key down and arrowing in one or two directions to create a single rectangular selection range. (Holding the Shift key down and pressing PageUp, PageDown, Home, or End also has the effect of creating a selection.) You can reference or set a selection with the xSelection property, which, by default and for a single selection range, is a one-row, four-column matrix. The first two elements are the coordinates of the anchor cell, the upper left corner of the range, and the last two elements are the extents of the range; that is, the third element is the number of rows and the fourth element is the number of columns in the selection. This property is similar to a where property using cells as the scale.

You can set the xSelection property by specifying any corner of the range; positive numbers for the extent elements are down and to the right. If the third element is negative, the number of rows is up from the corner you specify; if the fourth element is negative, the number of columns is to the left of the corner cell. Note, however, that the property will return the range as though you had specified the upper left corner and positive extents. That is, the xSelection property always identifies the upper left corner as the anchor cell.

Behavior: Once a selection range is established, you can change the active cell within the range under program control while the coordinates of the anchor cell and the extents of the selection remain the same. A user can Tab and use the Enter key to change the active cell within the range, so long as the xStayInSelection property is set to its default value of `1`. If you change the xStayInSelection flag to zero, then tabbing from the last column of the range or pressing the Enter key while the active cell is in the last row moves the active cell out of the range. The selection is canceled and the new active cell becomes the selection with extents of one in each direction. Note that if the user holds down the Shift key while tabbing or pressing Enter, the effect is to move the active cell in the negative direction, to the left or up.

If the xSelectionMode property is set to `1`, its default, you can specify multiple independent ranges of selection, which adds rows to the xSelection matrix. The user can generate multiple independent ranges of selection by holding down the Ctrl key while dragging or arrowing from a new anchor cell. The user can generate a succession of one-cell selections by clicking on individual cells while holding the Ctrl key. As cells or ranges of cells are added, the newest selection is the topmost row of the matrix. Each current selection is numbered sequentially as it is added to the matrix, although this "block number" is not visible in the matrix. The number of different ranges or single cells that are selected is identified in the xSelectionCount property. The value of this property is equal to the number of rows in the xSelection matrix.

Programming Alert: The rows are numbered in the order in which the selections are established, but each new row is added at the top of the matrix. Thus, the last row of the matrix is block number `1`, the next to last block number `2`, and the top row of an n-by-`4` matrix is the *n*th selection block.

Note that the xStayInSelection behavior applies only to the most recent selection, that is, the first row of the xSelection matrix. If you move the active cell outside that range, the entire selection is reduced to the new active cell. If the top row of the matrix is a single cell, tabbing or pressing the Enter key moves the active cell and changes the selection. If the selection is reduced to a single block, the block numbering starts over with `1`.

You can restrict the selection possibilities in various ways. If you set the xSelectionMode property to zero, only one cell at a time may be selected. Setting it to `2` or `3` means only whole rows are selected.

You can set the xAllowSelection property for any cell or range of cells to zero to exclude cells from being selected. Setting this flag to zero for a cell means it cannot be the active cell nor part of a selection range; clicking on the cell has no effect, you cannot tab, arrow or press the Enter key to get to the cell. If a user attempts to drag the mouse so a selection includes the cell, the dragging is prohibited anywhere past the row and column of the prohibited cell from the direction of the anchor cell. You cannot create a single selection range that surrounds such a cell, but you can create multiple selections to surround a prohibited cell. Note that there is no visual indication that a cell cannot be selected; you may want to change its appearance to indicate to a user that the cell is different.

In spite of not allowing the cell to be selected, you can still allow a user to enter data in the cell by invoking the xEditStart method with that cell's coordinates as the argument. If you want to prevent a cell's contents from being changed by the user, set the xProtect property to `1`. This setting allows the user to move to the cell or to include the cell as part of a selection, but keyboard input is restricted. You can set the text, make the cell active, or initiate editing under program control.

Establishing the Appearance of the Grid

There are a number of ways to define the appearance of grid cells, including color, font, size, etc. Some of these are merely initial settings; for example, a user can, by default, change the width of a column by dragging the border between its header and the header for an adjacent column. (You can suppress this capability, if that is necessary.) Other settings, such as color, take their defaults from Windows Control Panel settings, but changes are outside the user's control, unless you make specific programmatic tools to provide that control.

Colors

When you set a color property in the grid, you cannot use the typical three-element vector; you must specify a single integer that is equal to using the base value primitive function with three values to the base `256`; that is, `256 ⊥` *blue green red*.

There are five color properties you can set. Four of the properties apply to cells on the active page; for any of these four, you can set page defaults, column defaults and explicit cell values. The background color for non-selected cells is the xColorBack property. The background color property for selected cells other than the active cell is the xColorSelBack property. The text color for non-selected cells is the xColorText property; for selected cells other than the active cell it is the xColorSelText property.

Behavior: The active cell when in a selection has the unselected background and text colors. If the active cell is in edit mode, its colors come from a Windows default even if you have set xColorSelBack or xColorSelText.

To set any of these properties, you must supply row and column arguments (`0 0` for page defaults, `0` *columnIndices* for column defaults, or *rowIndices columnIndices* for explicit settings, where *Indices* are either integer scalars or vectors for specifying the cells to set), followed by an appropriately shaped value argument. These properties apply to row and column header cells as well as regular cells; you use negative integer arguments to identify headers. See the "Grid Coordinate System" above.

The fifth color property is xColorGrid; you reference this property with no argument or set it with a scalar color value. This establishes a color for the area of the grid that is below and to the right of regular cells, if cells do not occupy the entire available area.

Fonts

When you set a font in the grid, you set the name and the size in different properties: xFontName and xFontSize. You can assign any name, but only ANSI fonts work; symbol fonts do not take effect. If you assign a meaningless name, the system reverts to the default font without signaling an error. The font size is expressed in virtual-pixels. The default settings for the grid are Arial and `19`.

You can assign any of the style values: bold (`1`), italic (`2`), underline (`4`), and strikeout (`8`) in the xFontStyle property, but you cannot successfully invoke any of the Pitch, Family, or Quality values. Each of these properties applies to the active page only, and you can assign page and column defaults as well as explicit cells. They take row and column arguments, followed by an appropriate value if you are setting the property.

Cell Sizes

Cell sizes are expressed in virtual-pixels; they apply to an entire row or column. You can reference or set the height of a row with the xRowSize property and the width of a column with the xColSize property. Each of these takes a scalar or vector argument designating the rows or columns you wish to query or set. By default, a user can change the size of a row or column by dragging the border of the header below a row or to the right of a column. The default settings for the regular grid cells are column width `75` and row height `21`. The default width of the column of row headers is `33`.

There are six events, three for rows and three for columns, that fire in relation to resizing. When the user positions the mouse over a border that can be dragged, the onXCanColSize or onXCanRowSize event fires, perhaps repeatedly. This can occur even without a mouse button being pressed. You can prevent any resizing of the row or column affected by setting `⎕wres[2]` to `0`. If not suppressed, the cursor transforms into a double-headed arrow, which allows the user to grab the border. As the user drags the border, you also receive repeated onXColSizing or onXRowSizing events that report the column or row being dragged and the size. When the user releases the mouse, it triggers the onXColSized or onXRowSized event once. You can use any of these four events to control a resizing or to set a new size.

You can set a default row or column size by setting the first argument for xRowSize or xColSize to zero, followed by the size in virtual-pixels. These settings do not affect rows and columns already on the grid. They apply only to new rows and columns that you add. If you set these default values before you specify xRows and xCols, then they apply to the entire page. You can also establish a column size at any time by invoking the XFitCol method with a scalar or vector column argument. The grid adjusts each column to which this is applied to be just wide enough to display the contents of the cell (including a header cell) with the longest content string. Using the XFitCol method does not trigger either the XColSizing or the XColSized event.

Cell Borders

Each cell can have a border on any or all four sides; however, there are few circumstances where you would want to set the border on adjacent cells on all four sides. A cell border is not between cells; it is within the area of the cell itself. Thus, setting a border on the right side of column `1` and another on the left side of column `2` gives the appearance of a double border or one border of double thickness.

You can use a border to highlight a single cell or place one around the edges of a block of cells. If you want borders on each cell in a range of cells, it is useful to adopt a style of setting one vertical and one horizontal border on each cell, such as the right edge and the bottom. Note that cell borders are distinct from the gridlines that delineate all cells.

There are various types of borders: thin, medium, thick, and double, which you can place on any combination of the four edges. There are also values for borders that give the appearance of the cell being recessed or raised. The first group of values can be used in combinations; the latter two are exclusive. See the description of the xBorderStyle property for the specific values.

Images

A cell can also have images in any or all of its four corners, and one centered along any one of the four sides or in the very center of the cell. It should be obvious that unless your cell sizes are very large, images you use must be very small. There are two ways of specifying a container for the images you want to use. You can build an APL64 Imagelist object, and specify its handle, not its name, in the xImageList property. The handle of an imagelist is found in its himage property. Alternatively, you can specify the name of a bitmap file in the xImageFile property. These two methods are exclusive; if you set the imagelist handle, the system resets the value in xImageFile; similarly, if you set xImageFile, the system resets xImageList. These properties are not page-specific; one file serves all pages of the grid.

If you use a bitmap file, presumably it has a number of images tiled into a single bitmap. You can specify the width of the bitmaps in the xImageWidth property. For purposes of identifying individual images, the control divides the width of the entire bitmap file by the image width you specify and treats each segment as a separate image numbered consecutively starting with `1`. The height of the bitmaps is determined by the contents of the file; if you leave xImageWidth set to zero, the system assumes the images are square. For either container, you specify the indices as the fourth argument to the xImage property.

When you reference the xImage property, you must supply three scalar or nested-vector integer arguments. The second and third arguments specify rows and columns as with other properties. The first argument specifies which places you are querying. The corners are numbered one through four, upper left, upper right, lower left, and lower right; the number `5` refers to another location which is specified by the xAlign property. (Although you can specify a corner in xAlign, it makes little sense to do so for this purpose.) You can center an image vertically along either side, horizontally on the top or bottom, or both, which puts it in the very center of the cell.

The control returns a three-dimensional array whose shape is the same as the catenation of the lengths of the three arguments. The first plane of the result holds all the values for the first location that you specify; the rows and columns of each plane match the rows and columns of the grid that you specify. Thus, if you query the bottom corners of a three-row, two-column range of cells, the result will look like the following:

```
      'fmA.grid' ⎕wi 'xImage' (3 4) (1 3 5) (2 4)
 53 15     ⍝ Row 1   |------
 32 27     ⍝ Row 3   | lower left corner
 44 60     ⍝ Row 5   |------

 18 17     ⍝ Row 1   |------
  4 58     ⍝ Row 3   | lower right corner
 25  9     ⍝ Row 5   |------
   ⍝ ↑___Column 4
 ⍝↑________Column 2
```

The first cell of your array, row `1`, column `2`, has the images whose indices are `53` and `18`.

Setting Image Indices

When you set images, you supply the indices as the fourth argument. Ideally, you provide those indices in a three-dimensional array of the same shape as the array created by the first three arguments. Thus, you should supply a `2`-by-`3`-by-`2` array of indices for the above statement if you want to reset the images in those cells. You can use zero as an image index to have no image in a specified corner of a specified cell.

When you specify indices in a shape that does not exactly match the first three arguments, there are various ways in which the data are reshaped, extended, or accommodated. If you specify a two-dimensional array of indices, the control replicates the data for each specified corner, so you have multiple copies of one image in each cell. If you specify a vector, the control extends the values along each column as well as each corner. If you do not specify enough rows or columns of indices, the control replicates the data along the lacking axis. If you specify more rows or columns of indices than you specify in the arguments, the control extends the specification along either or both axes by incrementing the last row or column number by one for each extra row or column of data. If you specify more planes of data than you specify corners, the first settings are overwritten by the later ones.

Examples:
The file `APLGrid.bmp`, which is included in the `Examples` subdirectory with this release, contains a bitmap that is `6` pixels high and `180` pixels wide. If you name the file in the xImageFile property, you will have available `30` images, which include: a set of four red triangles oriented to fit in each of the corners of a cell in order; followed by three similar sets of four triangles differing from the first only in color; up and down red arrows; up and down green arrows; a green plus sign and a red minus sign; and eight white blocks (which show up only against a different background color). You can use these images singly or in combination to highlight cells, and you can use this file to experiment with the grid.

If you want to assign the first `24` images in order to a `2`-by-`3` range of cells, you could use the following expression. Note that the image indices must be arranged so that one index for each cell is in one plane of the fourth argument.

```
      Indices " 2 3 1⍉ 2 3 4 ρι24
      'fmA.grid' ⎕wi 'xImage' (ι4) (4 5) (1+ι3) Indices
```

It may not be obvious how the images are arranged; here is an equivalent statement:

```
      (2 3 1⍉ 2 3 4 ρι24)≡(ι4) ∘.+ 4× 2 3ρ(¯1+ι6)
1
      Indices
  1  5  9
 13 17 21

  2  6 10
 14 18 22

  3  7 11
 15 19 23

  4  8 12
 16 20 24
```

When you set the images for multiple cells, you must be careful that you specify the image indices in a shape compatible with the three-dimensional *place*, *row*, *column* arguments you supply. If you attempt to specify four images, one each in the upper right corner of a column of cells, but carelessly specify the images as a four element vector, you will get images in `16` cells (by virtue of the extension rules); all four of the cells you specify will have the first image. The grid interprets a four-element vector as a shape `1 1 4` array and a four-by-one matrix as a shape `1 4 1` array for purposes of this property.

Thus, a statement like the first one below results in a pattern of images indices as shown on the left, while a statement like the second one results in a pattern as shown on the right:

```
      'fmA.grid' ⎕wi 'xImage' (2) (⍳4) (2) (19 20 21 22)
      'fmA.grid' ⎕wi 'xImage' (2) (⍳4) (2) (4 1 ⍴ 19 20 21 22)

 -- 19 20 21 22       -- 19 -- -- --
 -- 19 20 21 22       -- 20 -- -- --
 -- 19 20 21 22       -- 21 -- -- --
 -- 19 20 21 22       -- 22 -- -- --
```

Establishing the Content of the Grid

Each cell in the grid has an xCellType value that determines its function. A normal cell holds text or numeric data. Cell types other than normal allow buttons, arrows, check boxes, or combo boxes in a cell. These special cell types provide means for users to manipulate the display of the grid, and they enrich the interaction between the user and your application. A normal cell is type zero. The default value of ¯1 means a value has not yet been set. You can also use ¯1 to clear the setting for a cell.

Normal cells have value types (a different use of the term "type") that determines how the data are interpreted; in particular, cells that contain numbers can be straight numeric, dates, Boolean, or currency. There are properties that establish the value type; there are also a set of properties that affect the formatting of data of a particular type within the cell.

Value Types for Normal Cells

Each cell has an xValueType property. By default, this is ¯1, which means no value type has been established. You can also set this property with ¯1 to clear a cell's value type. If you query the value of a cell for which no value has been assigned or entered and which has the default value type, you do not get a readily usable value. Querying with the xText or xValue property returns a one-row, one-column matrix that is empty and whose ActiveX datatype is 8200, a matrix of strings. If you query with one of the properties that implies a numeric value, you get the conversion error value (see below).

You can set the XValueType property explicitly for a cell to an integer value in the range of zero through 4, or for some types, you can set it implicitly by setting a content property with a value. By default, a user can type in a normal cell, and the text displays; however, this does not cause a change in the xValueType property. Setting either the xText or the xValue property to a character string does cause a change; you can reference either of these properties to see the text contents of a cell.

Text Cells

The value type zero designates a text cell. You can set the xValueType property to zero, or you can set the xText property with a character vector, and that action sets the xValueType property to zero. There are five different sub-types of a text cell. You specify these sub-types in the xCellTypeEx property. The default value is ¯1; the default behavior is sub-type zero.

In a normal **normal** (sub-type 0) cell, with the default left alignment, if the length of the text vector exceeds the width of the column, the text displays over the cell to the right if that cell is empty. If you are close to the edge of the control, the text may wrap over the cell below, if it is empty. Otherwise the visible text is truncated.

In a normal **constrained** (sub-type 1) cell, the visible text is unconditionally truncated. The entire text still exists in the xText property, and if you widen the column, more of it shows.

In a normal **multi-line** (sub-type 2) cell, the text wraps at word breaks or at ⎕tcnl characters. If the length of the text exceeds the column width and row height of the cell, the visible text is truncated.

In a normal **ellipsis** (sub-type 4) cell, if the length of the text exceeds the width of the cell, the control displays an ellipsis at the end of the truncated visible text to indicate there is more.

In a normal **label** (sub-type `8`) cell, the text that displays comes from the xValueEx property. A user can type into the cell, but the results are not displayed. This sub-type is more like a special purpose cell type than it is like a normal cell. You may want to protect such a cell with either the xProtect property or an event handler to keep a user from thinking he can type into the cell.

Programming Alert: What the user types, or what you set in a label cell, exists in the xText and xValue properties, even though it is not displayed. If you query either, you get the hidden value, not the visible text. It is your responsibility to know which cells are labels and to use xValueEx when appropriate.

Number Cells

If a user types numerals into a default normal cell, the grid treats the entry as text. For the grid to treat the entry as numeric, you must set the xValueType property to `1` or set the xNumber property with a numeric value. Once the cell is established as numeric, a user can type in numeric data as an integer, a floating point number using the period, negative values using the minus sign (which APL displays as a high minus when you reference the value) or in exponential format, for example `1E4`. (You can use uppercase or lowercase E, as in APL notation or uppercase or lowercase D, as in Fortran notation. In some languages, these distinguish single and double precision float values, but APL uses only double precision floats.)

Each cell can hold only one value; if you assign a vector, the grid puts the additional values in columns to the right of the specified cell. If you query the value of empty cells for which you have set xValueType to `1`, you get a conversion error indication, not a zero.

Programming Alert: There is a difference in setting the xValueType property to `1` as an explicit setting and as a default. If you type numerals in a cell and set xValueType explicitly, the control converts the contents of the cell to a number. However, if you set xValueType as a column default, the system treats the existing characters as text and changes the cell type to Text.

A conversion error occurs when the entry into a cell cannot be treated as a valid value according to the cell's value type. The most obvious case is an alphabetic text entry in a numeric field. The grid returns a value that indicates this condition when you reference the value of a cell that has such an entry. The default conversion error value is a negative number of large magnitude. You can set a value of your choice in the xConversionErrorValue property. Note that the conversion error value does not appear in the grid itself, even after you invoke a paint. If you want a visible indication of an improper value, you must do the value checking and formatting yourself.

Boolean Cells

If you set the xValueType property to `2`, the grid accepts any number or the values True or False as valid entries. When you query the value, any number, positive or negative, other than zero is treated as `1`; True and False are `1` and zero, respectively.

Note that there is not a distinct property to set a Boolean entry as there is with Text and Number cells (and Date and Currency, as explained below). You must query the xValue property to see the Boolean result. If you query the xNumber property, the grid returns any full numeric value and does not recognize True or False.

Date Cells

If you set the xValueType property to `3`, or set the xDate property with a valid value, it denotes a date cell. What constitutes a valid entry and how it displays depends, by default, on operating system settings on the host machine for locale and date formats. You can specify a particular locale by setting the xLocale property for a cell; these settings are four- or five-digit integers that Windows recognizes to represent a country/language combination. You can format date cells using the xFormat property. See the "Details on the xFormat Property" section for more information on exact formats.

For a typical US setting, you can type in or specify dates as Jan 31, 99; 2/29/00, or March 31, 2001, for example. If you specify a non-zero two-digit year that is less than 30 (this number is a host machine system setting), the grid assumes this century. If you specify a number greater than 29, it assumes the 1900s. If you type in an entry, the grid displays it as entered; if you set the values, the grid displays in one format, for example 2/29/2000. You can query the xText property to see what is displayed.

If you query the xDate or xValue property, you get the underlying value in the cell, which is numeric. If it is a floating point number, the fractional part represents a time measured as a percentage of 24 hours. Thus, if you specify .75, it represents 1800 hours or 6:00:00 pm. An integer or the integral part of a floating point number represents a date, where day 1 is December 31, 1899 (don't ask). January 1, 2002 is day 37257.

You can specify non-zero negative numbers to get dates earlier in the 19th century, but there are some anomalies around zero. If you specify zero, the grid assumes a time of midnight; if you type in zero, you get a conversion error. If you specify Dec 30, 1899, you may have a problem. Otherwise the grid is accurate, including February 1900 (which was not a leap year). If zero is a potential problem, you can obviate it by setting the xFormat property for the date cell; see the "Details on the xFormat Property" section, later in this document, for details.

Note also that if you specify a negative floating point number, the grid calculates a time on the day represented by the negative integer; that is, ¯1.8 is later than ¯1.7. Effectively, the minus sign applies only to the integer portion of the number. Thus, the values 0.52 and ¯0.52 represent the same time on the same day.

Currency Cells

If you set the xValueType property to 4, or set the xCurrency property with a valid value, it denotes a currency cell. Currency cells are simply numeric cells; you can set, reference or type in numeric integers or floating point numbers. You must set the xFormat property to invoke the currency representations and formatting.

Sub-types for non-text cells

You can set the xCellTypeEx property for non-text cells as well as a text cell. The ellipsis value (4) works for any value type, and you can put any type of display in a label cell (8). You can constrain (xCellTypeEx = 1) the display of a date, but cells with numeric values do not extend their display to the right, so the setting has no meaning for those value types. None of the value types other than text can be multi-line.

Special Purpose Cell Types

You can set the xCellType property to a number greater than zero to get a special purpose cell. These cells allow specific navigation or manipulation effects using combo box drop-down lists, check boxes, arrows and buttons. These cells are automatically protected; that is, the user cannot enter data into them by typing.

Combo Box

Setting the xCellType property value to 1 creates a Combo box in a cell; this is essentially a style 2 Combo, a drop-down list with a display field. You specify the list in the xValueEx property as a vector with the list items delimited by ⎕tclf or ⎕tcnl. There is an onXCellDropDown event that fires when a user clicks to see the list; note that this event-handler property applies to all combo boxes on any page of the grid. You can use the onXValueChange event, which also applies to the entire grid, to respond to the user selecting a choice from the list. Both of these events return the row and column values in ⎕warg; you can check the xValue property or xText to determine the text of the list item the user selected. There is also an onXCellDropWindow event that fires immediately before displaying the dropdown window. This is similar to the onXCellDropDown event except it fires immediately before the dropdown window and includes the window handle that is about to be displayed.

Programming Alert: You can use `⎕tclf` or `⎕tcnl` to separate list items. There is no numeric identifier of the list item; you must read the text, unless you specify the cell as having xValueType `1` (numeric), in which case you put numerals (text) in your list but read the xNumber property.

By default, a Combo box cell displays the most recently selected value and the drop down arrow. You can specify that the cell look like a normal cell, with only the value displayed, until it becomes the active cell, at which time the drop down arrow appears. You specify this behavior by setting the xCellTypeEx property to `1`.

Check Box

Setting the xCellType property value to `2` creates a Check box in a cell. You specify the caption in the xValueEx property. You can use the onXValueChange event, which applies to the grid, to respond to the user clicking the Check box. You can check the xValue property to determine if the box is checked (`1`) or unchecked (`0`).

You can choose the appearance of the check box by setting the xCellTypeEx property to have these effects:

`0`	flat box with **x** marker
`1`	recessed box with **x** marker
`2`	raised button with **x** marker
`4`	flat box with check mark.

Arrow

Setting the xCellType property value to `3` creates a cell with an arrow (triangle). By default there is a single arrow pointing to the right. You can set the xCellTypeEx property to have these effects: `1`, single right-pointing arrow; `2`, single left-pointing arrow; `4`, double right-pointing arrows; `8`, double left-pointing arrows. You can use the onXCellClick event, which applies to the entire grid, to respond to the user clicking an arrow.

Button

Setting the xCellType property value to `4` creates a Button cell. This cell does not look very different from a normal cell, so you may want to set a special background color (xColorBack) to distinguish it. You set the text of the button in the xText property. You can use the onXCellClick event to respond to the user clicking a button.

Miscellaneous Properties that Affect Visible Content

Alignment: You can specify the alignment of the contents of a normal cell by setting the xAlign property. Horizontal alignment values are: `1`, left justified (the default for text, date, and Boolean cells); `2`, right justified (the default for numeric cells); and `4`, centered. Vertical alignment values are: `8`, top; `16`, bottom; `32`, centered. You may choose one value from each group and specify their sum.

Merged Cells: You can make a large cell that takes up the space of a rectangular array of cells by using the XJoin method. You specify the upper left corner of the array you want to merge as the anchor cell and the number of rows and columns to the right and down, just as you specify a selection. You address this merged cell by the coordinates of the anchor cell. It has the properties of the anchor cell; values set for other cells in the block are suppressed until you undo the merge. You can return the cells to their individual states by invoking XJoin on any individual cell in the block. You can query the read-only xMergedCells property without an argument to see a four-column matrix of all the merged cell blocks on a page.

You can use this property effectively with header cells, for example to create a single header that spans multiple columns. You could convert a block of cells where input would be irrelevant into a single large label. The corner header block that you specify as `¯1 ¯1` when you have multiple row and/or column headers is effectively a permanently merged cell.

Visible range: If you have more rows and/or columns than are visible at one time, the cell in the upper left corner of the scrollable area and the number of rows and columns that are wholly or partially visible is contained in the xView property. If you have any fixed rows or columns, they are excluded from this calculation. You can query this property with no argument or set it with a row and column (or row, column, and extents; but the extents are ignored) to move the visible portion of the grid.

Visible Page: You can effectively hide a page by setting the xPageVisible flag to zero. The tab disappears and the user cannot navigate to the page, but the page still exists.

Visible Selection: You can query the location of a visible regular cell on the active page by invoking the XWhereIs method with a row and column argument. The grid returns the location of the upper left-hand corner relative to the grid's client area and the height and width in virtual-pixels. You can also query the location and size of a selection by using a four-element argument of row, column, and extents. If the anchor cell is not visible, the method returns four zeroes.

Scrolling Options: By default, if all your rows and columns are visible, there are no scroll bars in the grid. If you specify more of either than will fit, the appropriate scroll bar or bars appear. You can set the xScrollMode property for the active page with the values listed below to control the behavior of scrolling on that page.

1	Enable vertical scrolling hints
2	Enable horizontal scrolling hints
4	Enable vertical scrolling tracking
8	Enable horizontal scrolling tracking
16	Leave vertical scroll bar in place if not needed
32	Leave horizontal scroll bar in place if not needed
64	Hide vertical scroll bar
128	Hide horizontal scroll bar.

Scrolling hints are tooltip-like boxes that appear when the user presses the left (or primary) mouse button on the scroll box (thumb). By default, these show "Row *n*" for vertical scrolling and "Column *n*" for horizontal scrolling where n is the topmost or leftmost visible row or column. The numbers change as the user scrolls. You can change the text of the visible tip just before it appears by using the onXScrollHint event handler.

Tracking means the grid moves column by column or row by row as the user drags the thumb; without these settings, the grid does not reposition itself until the user releases the mouse from the thumb. By default a scroll bar appears if needed and disappears if not needed; that is, if all the cells fit in the visible area. You can have a disabled scroll bar, meaning the channel stays in place without a box to scroll if not needed. If you hide a scroll bar, it does not appear even if needed. (A user can still move through all the cells using the arrow keys.)

Tracking window

A tracking window can be any Windows object that has a window handle; the grid displays it when you specify the handle in the xTrackingWindow property and invoke the XMoveTrackingWindow method with a row and column argument. The window is positioned beside the specified cell. Typically, you use this facility as a super tooltip, triggering the display by defining an onXCellMouseEnter event handler and providing information to the user about the cell the mouse is over. If you invoke the method with a zero row or column argument, the window is again hidden. You can specify only one tracking window at a time for the grid. Although the window could be most anything, this facility is best used to provide information to the user rather than allowing interaction, since the tracking window is displayed beside, not over, the specified cell. If you want a special window for the user to enter input, you can define an edit window for a particular cell (see "Managing User Editing," below).

Example:
Note that the CellMouseEnter event works only on regular cells, not row and column header cells.

```
      ⎕wi '.trk.Create' 'Label' ('size' 2.5 8) ('visible' ¯1)
      ⎕wi 'xTrackingWindow' (⎕wi '.trk.hwnd')
      ⎕wi 'onXCellMouseEnter' 'TrackerFn'

    ∇ TrackerFn
[1]   (newrow newcol)←2↑⎕warg
[2]   :if 1=newcol
[3]     ⎕wi '.trk.caption' ('Click button to your left to select Row ',⍕newrow)
[4]     ⎕wi 'XMoveTrackingWindow' newrow newcol
[5]   :else
[6]     ⎕wi 'XMoveTrackingWindow' 0 0
[7]   :end
    ∇
```

Manipulating the Grid

Once the grid is set up and the initial content is defined, you may want to manipulate the grid under program control. There are a set of methods that allow you to perform basic grid actions.

Adding to and Deleting from the Grid

You can add rows, columns, or pages to the grid. You can insert rows by using the XInsertRows method with a scalar or vector argument indicating where to put the new row. If the argument is a single integer, one new row is inserted before the specified row. Subsequent rows are renumbered. If the argument is a vector of integers, new rows are inserted before each of the existing rows that are specified. Then, all the rows are re-indexed. That is, if you specify an argument of `5 6`, the new rows will be the fifth and seventh rows, while the rows that were previously fifth and sixth will be sixth and eighth, respectively. To insert two new adjacent rows, duplicate an index in the argument. You can also insert rows by specifying floating point numbers; using an argument of `4.5` inserts a row between the current fourth and fifth rows.

You can add columns in a similar fashion using the XInsertCols method. You can insert pages by using the XInsertPages method; however note the following. **Behavior:** The names on the page tabs are created in sequence regardless of their order in the grid. The page numbers, on the other hand, are adjusted to match their order. Thus, if you insert two pages into an existing three-page grid, the tabs may be in this order, even if you have renamed the first three pages: Page1, Page5, Page2, Page4, Page3. The page numbers will be in sequence left to right; if you click on the page whose name is Page5 and reference xPage, the grid returns `2`.

You can delete rows, columns, or pages by using an integer scalar or vector argument to the XDeleteRows, XDeleteCols, or XDeletePages methods, respectively.

Changing the Selection

You can change the selection under program control in two fashions. You can clear all the existing selection blocks and assign a new selection by setting the xSelection property with a four-element integer argument specifying the anchor cell and the row and column extents for the block you want to have selected (or a four-row matrix for a multiple selection). You can add a selection block by invoking the XAddSelection method with the same type of four-element integer argument. The new block is added at the top of the xSelection property matrix with the highest block number.

Reacting to User Navigation with Grid Event Handlers

There are two major ways in which a user interacts with your grid; by navigating through it or changing the contents of a cell. The event handlers introduced in this section allow you to track closely where the user has moved his attention.

Reacting to Mouse Actions

The movement of the mouse over the grid generates events regardless of whether a mouse button is pressed or not. Each time the mouse crosses a regular cell boundary, it triggers an onXCellMouseEnter event handler. If the mouse was already on a regular cell, the XCellMouseEnter event is preceded by an onXCellMouseLeave event handler. If the move is onto a cell the XCellMouseEnter event is followed by an onXCellMouseMove event handler.

The system variable `⎕warg` shows the cells involved in each event. During onXCellMouseLeave, `⎕warg` is a four-element integer vector with the row and column coordinates of the cell the mouse left, followed by the coordinates of the cell to which the mouse moved. These two pairs are reversed during onXCellMouseEnter; the coordinates of the cell to which the mouse moved are the first and second values, while the coordinates of the cell that was exited are the third and fourth values.

If the mouse is moved off the body of the grid, the second pair of coordinates in `⎕warg` for onXCellMouseLeave and the first pair for onXCellMouseEnter are `0 0`. If you move onto a cell from a row or column header or from off an edge of the grid, the second pair of coordinates for onXCellMouseEnter is `0 0`. There is not an XCellMouseLeave event for movement from off the grid, so if the user moves the mouse onto and off the grid, there will be one more XCellMouseEnter event than XCellMouseLeave.

Note: In the discussion below, it is assumed that the left mouse button is the primary button. If you have reversed the effect of the mouse buttons, read "primary" for left, and "opposite" for right.

If the user presses the left or right mouse button while the mouse is over a grid cell (including a header cell), that action triggers an onXCellMouseDown event handler. If the user releases either button while the mouse is anywhere over a cell or the grid area (that is, space that could be occupied by cells, but not the tab row or space that might be occupied by headers), it triggers an onXCellMouseUp event handler. If the user presses and releases the left mouse button while staying within a cell, the XCellMouseUp event is followed by an onXCellClick event handler.

An XCellMouseDown event and an XCellClick event in a regular cell, but not a header cell, are each followed by an XCellMouseMove event. An XCellMouseUp event is followed by an XCellMouseMove event if the mouse is over a cell but was moved off the cell where the button was pressed. In this case, the onXCellMouseUp event handler fires without an XCellClick event. If the mouse has been moved off a cell onto another area of the grid, the onXCellMouseUp event handler fires without either an XCellClick or an XCellMouseMove event. If a mouse button is pressed and released while the mouse is over the grid but not on a cell, only the XCellMouseUp event occurs.

During each of the onXCellMouseDown, onXCellMouseUp, and onXCellMouseMove event handlers, `⎕warg` is a five-element vector; during onXCellClick, `⎕warg` has six elements. The first two elements are the row and column where the event occurred (`0 0` for XCellMouseUp when it is off the body of the grid or was moved onto a header cell after the mouse down). The third element tells which mouse button caused the XCellMouseDown, XCellMouseUp or XCellClick event: `1` for the left mouse button; `2` for the right button. The third element is always zero for an XCellMouseMove event.

The fourth element is the sum of the values of the buttons that are pressed when the event occurs: `1` for the left mouse button; `2` for the right button; and `4` for the center button. The fifth element is the sum of the values of keys that are pressed when the event occurs: `1` for the Shift key; `2` for the Ctrl key; and `4` for the Alt key.

For an onXCellClick event handler, the sixth element is `0` for the first click, or `2` for a double-click. Note that double-clicking in a cell, by default puts the user into edit mode; this triggers an onXEditStart event handler as well as onXCellMouseLeave and onXCellMouseEnter with the cell coordinates and `0 0` in `⎕warg`. (See below for the event handlers that allow you to manage user editing.)

There are also two mouse events that apply only to the page tabs. Clicking a page tab, either for the current page or to change the page (see below), triggers the XTabMouseDown event; this is followed by the XTabMouseDouble event if a second click follows quickly enough. For both the onXTabMouseDown and onXTabMouseDouble event handlers, `⎕warg` is a four-element vector with the page number, the mouse button that caused the click, the sum of the values of the buttons that were pressed when the event occurred, and the sum of the values of the Shift, Ctrl, and Alt keys that were pressed.

Reacting to Changes in the Active Cell or the View

When the user causes the active cell to change by clicking on a cell or causing the focus to move by pressing a key (Tab, Enter, arrows, PageUp, PageDown, Home, or End), such action triggers the onXCellChange event handler. During the handler, `⎕warg` is a four-element vector as with onXCellMouseEnter: the first two elements are the coordinates of the new active cell; the third and fourth elements are the coordinates of the previously active cell. This event handler is also triggered if you set the xActiveCell property or the selection under program control.

If the user specifically uses the Tab or Enter key to change the active cell, this also triggers the onXKeyMove event handler before the XCellChange event. During onXKeyMove, `⎕warg` has six elements. The first four are the same as for the XCellChange event: row and column for the new active cell and previous active cell. The fifth element is the virtual key code: `9` for the Tab key; `13` for the Enter key. The sixth element is zero if the Shift key is not pressed; `1` if the Shift key was down when the event was triggered. You can override the effect of the user action by setting `⎕wres[1 2]` to the row and column coordinates of the cell you want to become active.

Establishing a selection by setting the xSelection property or invoking either the XSetSelection or XAddSelection method triggers the onXSelectionChange event handler. During this event handler, `⎕warg` is a five-element vector; the first four elements are the new selection (row and column of the anchor cell; row and column extents) while the fifth is the block number (when multiple blocks are selected, the sequential number of this block). Changing the active cell within a selection block also triggers this event handler even though the selection itself doesn't change as a result. Note that any change of the active cell triggers this event handler, because the active cell can be a selection with extents of one in each direction. The XSelectionChange event follows the XCellChange event when both are triggered.

When the view (the range of visible cells to the right and below any fixed columns and rows) changes, either by the user scrolling the grid or by setting the xView property under program control, an onXViewChange event handler is triggered. During this event handler, `⎕warg` has the same values as the new xView property values: row and column of the upper left visible, scrollable cell and the row and column extents of visible cells. Note that setting xView does not change the the active cell; you can scroll the active cell off the screen.

Reacting to Page Changes

When the active page changes, either by the user clicking a new tab or setting xPage under program control, two or three event handlers fire in succession, unless you take an action to prevent the change. The first event handler, onXPageChanging, fires before the change is effective. Both `⎕warg` and `⎕wres` are three-element vectors, containing the number of the page about to become active, the current page number, and a flag that is `1` if the action is a result of a programmatic action, or zero if the action was initiated by the user. If the change request is a user action, you can prevent the change by setting `⎕wres[1]` to zero.

The last event handler in the sequence, if you did not suppress the page change, is onXPageChanged. This event handler fires after the change has become effective; `⎕warg` contains the same values as above: newly active page number, previously active page number, and program control flag.

If the page that is about to become active has not been opened previously, the middle event handler in the sequence is onXPageOpen. You can use this event handler to initialize the content of a page, if you do not want to initialize the entire grid when it is created. The event fires only once per page unless you close and reopen the grid; ⎕warg contains the page number.

If the user changes the page by clicking a page tab with the mouse, the order of the events is XTabMouseDown, XPageChanging, XPageChanged, and XTabMouseDouble (if applicable).

Note that cell events, such as XCellChange or XSelectionChange are not triggered by a page change.

Managing User Editing

The reason for the existence of a grid control is usually related to the contents of its cells. If you allow user editing, you have a combination of properties, methods, and event handlers that you can use to manage the editing process and control the input. Although it appears that a user can simply type values into a cell, the process is more complex. In order for editing to occur, the grid creates a temporary control, similar to an edit box, as a surrogate for the cell. This control exists in its own window, captures the focus, and receives key presses and mouse clicks. When editing for the cell is complete, the grid transfers the content of the temporary control to the cell in the appropriate format, at which time the grid retrieves the focus and can be the recipient of events.

Initiating the Editing Process

You can initiate the editing process on a cell independent of a user action by invoking the XEditStart method with cell coordinates as the arguments. Optionally, you can include a third argument, which is the ANSI index to a character you want to appear as the first character of the edit string. (These are the same values that exist for the onKeyPress event handler in a built-in APL64 control.) By default, editing also begins on the active cell when the user presses a key that generates a character, presses the F2 key, or double-clicks in a cell (which makes it the active cell if it wasn't already). The xAutoEditStart property, whose default value is 7, contains the sum of three values that represent which of these actions initiates editing: 1, character input; 2, F2 key; and 4, double-click. You can reset this property to a smaller number to restrict the valid user actions. This setting applies to the whole grid.

Any action that initiates editing triggers the onXEditStart event handler. During this handler, ⎕warg is a six-element vector containing a code that initially is 1, a vector with the character or string that initiates the session, the row and column of the cell being edited, a code that is zero if no character is being assigned or the ANSI character code of a character that is starting the session, and the handle of the edit window. You can set the code in ⎕wres[1] to zero to cancel the editing.

If the user initiates editing by pressing a character key in the cell or by pressing F2, it triggers the onXCellKeyDown event handler, which precedes the XEditStart event. During this event handler, ⎕warg is a four-element vector: the virtual key code (113 for F2); Shift state (the sum of 1 if the Shift key is pressed, 2 for the Ctrl key, and 4 for the Alt key, or zero if none of the three is pressed); row and column of the cell. Note that pressing the Shift key generates a separate XCellKeyDown event from a character key. Note also that after the character key initiates editing, events occur in the temporary edit control, so there is no XCellKeyUp event.

By default, the edit window expands to hold the entire input during the editing process. If you want to limit the size of the edit window to the size of the cell, set the xEditAutoSize property to zero. Initially, this property value is ¯1, which means it inherits its value; you can set this either as a default property on a column or a page or explicitly on a cell. If you set it to 1, the window expands to fit the values entered as it does by default. You can reset the property to ¯1 to clear an explicit value.

Tracking Input

Once editing is initiated and the temporary edit control is active, you can track the input with event handlers. Each keystroke generates an onXEditKeyDown and an onXEditKeyUp event handler. As with onXCellKeyDown, `⎕warg` contains the virtual key code, shift state, and the cell coordinates. If the key generates a character, it also triggers an XEditKeyPress event in between XEditKeyDown and XEditKeyUp. During the onXEditKeyPress event handler, the first element of `⎕warg` is the ANSI index to the character; the remaining three elements are the same as for onXCellKeyDown. You can assign a different index value to `⎕wres[1]` to change the character that the key press generates or set `⎕wres[1]` to zero to ignore the character.

Ending the Edit Session

When the user attempts to exit the cell, it serves as a request to end the edit session and accept the new value. Typically, when editing, a user presses the Enter key or Tab key to move to an adjacent cell. He can also arrow up or down (the left and right arrows work within the temporary edit control), click on another cell, or click somewhere else in the application. Any of these actions triggers the onXEditEnd event handler, as does invoking the XEditEnd method under program control during the edit session.

During the onXEditEnd event handler, `⎕warg` is a four-element vector akin to the XEditStart event. The first element is a code, initially one; the second element contains the value that the user entered during editing; the third and fourth elements are row and column of the cell. If you do not change `⎕wres`, when the onXEditEnd event handler exits, the new value is assigned to the cell, the active cell (or the focus) changes, and the onXEditComplete event handler is triggered.

If you wish to intercept the completion of editing in the onXEditEnd event handler, for example, because the entry is invalid, you can set `⎕wres` to invoke several options. Initially, `⎕wres` has the same values as `⎕warg`.

You can leave `⎕wres[1]` set to `1` and assign any valid value (enclose a text string) to `⎕wres[2]`. This places the value you assign into the cell and triggers the onXEditComplete event handler.

If you set `⎕wres[1]` to zero, this places the user back in the cell in edit mode with the entry highlighted.

If you set `⎕wres[1]` to `2`, this leaves the existing value in the cell unchanged, but triggers the onXEditComplete event handler. You can use this value either to restore the value that existed before the edit session started or to keep a value that you assigned during the event handler.

If you set `⎕wres[1]` to `¯1`, this cancels the edit session, restores the pre-existing value to the cell, and triggers the onXEditCancel event handler.

By default, if the user clicks outside the application it does not request an end to the edit session. This behavior is specified in the xUnfocusEndEdit property, which applies to the whole grid and is zero by default. If you set this flag to `1`, any action that causes the grid to lose the focus also triggers the onXEditEnd event handler. However, setting `⎕wres[1]` to zero under this circumstance cancels the edit session rather than restoring it.

When you invoke the XEditEnd method without an argument or with the default argument of zero, it triggers the onXEditEnd event handler just as though the user had moved off the cell and attempted to end the edit session. The only difference is that the focus is on the just-edited cell when the event handler returns. If you invoke XEditEnd with a numeric non-zero argument, it forces the edit session to completion (if the code in `⎕warg[1]` during onXEditEnd is `1` or `2`) or to cancellation (if the code in `⎕warg[1]` during onXEditEnd is `¯1` or zero). The event handler cannot return the user to the edit session.

At any time, before, during, or after the editing process, you can query xEditText. Before editing, this read-only property contains the text that would be in the edit control if you initiated editing without specifying a new character; this value can differ from what appears in the cell, for example if you have special formatting for numbers. During the editing process, this contains the value that was in the cell at the beginning of the process, or the value most recently assigned under program control during editing; this is the value that cell would contain if you or the user canceled the editing. At the end of the editing process, after completion of the XEditEnd and XValueChange events, this contains the new value of the cell. Again, this value can differ from the value of xText, if there is some special formatting.

During the onXEditComplete event handler, ⎕warg contains only the coordinates of the cell; the value of the cell has already been established, so you cannot meaningfully set ⎕wres. You can use this event handler to adjust other cells that may be dependent on the most recently edited cell, for example, recalculating a total that includes the just-edited cell.

Canceling the Edit Session

An edit session can be canceled, as noted above, from the onXEditEnd event handler, by invoking the XEditCancel method before onXEditEnd is triggered, or by the user pressing the Escape key. When an edit session is canceled, the pre-existing value is restored to the cell, the values entered during the edit are ignored, and the onXEditCancel event handler is triggered. Neither the XEditEnd nor XEditComplete event occurs.

During the onXEditCancel event handler, as during onXEditComplete, ⎕warg contains only the cell coordinates.

Outside the Edit Session

Whenever an edit session completes successfully, that is by completing the edit not by cancellation, the system also triggers an onXValueChange event handler. This event occurs even if the value is no different, for example by double-clicking a cell to initiate an edit session but immediately tabbing out of the cell. As with onXEditComplete, during this event handler, ⎕warg contains only the cell coordinates.

For a normal cell (xCellType zero), that is, one that contains text, numeric, or specially formatted data, this event occurs immediately after XEditComplete. You could use either event handler equally effectively. However, the XValueChange event also occurs for special cell types that do not accommodate an edit session. For a Combo box (xCellType 1), the event occurs when a user clicks an item from the drop down list. For a Check box, (xCellType 2), the event occurs when the user toggles the check either on or off. For a Button (xCellType 4), the event occurs when the user presses the Enter key, which moves off the cell but does not trigger the onXCellClick event handler, or the spacebar, which leaves the focus but also does not trigger onXCellClick. For an Arrow cell, (xCellType 3), the event does not occur.

In addition, the XValueChange event occurs outside an edit session on a normal cell following the XValueClear event. Pressing the Delete key on any cell other than during an edit session triggers the onXValueClear event handler. (Pressing the Delete key during an edit session has the normal editing behavior; it removes either highlighted text or the single character following the selection cursor.)

During the onXValueClear event handler, ⎕warg is a three-element vector: the first element is a code that you use to cancel or allow the deletion; the second and third elements are the coordinates of the cell. Initially, for a normal cell, in both ⎕warg and ⎕wres, the first element is one. If you set it to zero, it cancels the deletion and restores the existing value to the cell (similar to canceling an edit session). In the latter case, the XValueChange event is suppressed.

If a block of cells is selected and the user presses the Delete key, the onXValueClear event fires on each cell in succession, followed by onXValueChange, if appropriate. If multiple blocks of cells are selected, the Delete key applies to all the cells in all the selection blocks.

For special cell types (that is, xCellType other than zero, normal), pressing the Delete key triggers onXValueClear, but the first element of ⎕warg (and ⎕wres) is 1. It is not clear what action the system should take, so setting the first element of ⎕wres to zero has no effect. However, you can make any appropriate changes to the special cell in the event handler. The delete action does not trigger onXValueChange on special cells regardless of the action of the onXValueClear event handler.

Note that invoking the XDeleteCells method deletes both attributes and values of a cell, but does not trigger either the onXValueClear or the onXValueChange event handler.

Invoking a Special Edit Window

All the above discussion about editing assumed the default edit window that the grid creates. Just as you can invoke a special tracking window, you can invoke your own edit window for the user to enter input. Create the control that you want the user to see, and assign its handle (found in the hwnd property of the control) to the xEditWindow property of any cell to which you want it to apply. This window replaces the default edit window of the grid. Unlike the tracking window, you can have multiple special edit windows assigned at a time.

This is not a simple feature to implement properly. Note that the events that occur for the editing session are the events of the defined control and not the events of the grid; for example, there will not be an onXEditKeyPress event. Nor are the XEditEnd, XEditCancel, or XValueChange events triggered. You are responsible for recognizing the end of the edit session (for example by capturing the Enter key), retrieving the value entered by the user, and assigning it to the appropriate cell. You are also responsible for sizing the control, if necessary, and for assigning a new active cell. If you move the focus, you may need to invoke XRedraw.

Managing Replication Mode

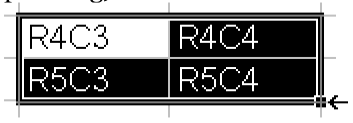
The grid supports a form of selection called “replication mode” that allows you to drag on the replication handle and extend the selection to perform application-defined operations such as duplication of values into adjacent cells.

The xReplMode property must be set to enable replication before it is available to users. The following enables replication along two axes at the same time:

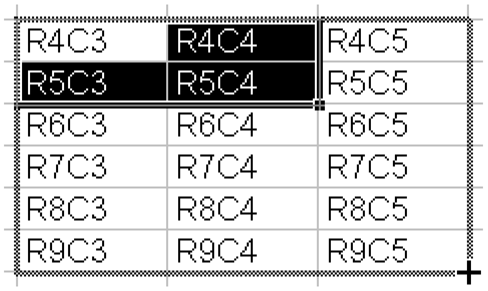
```
⎕wi 'xReplMode' 2
```

Or you can restrict the operation to only one axis at a time by setting xReplMode to 1. Consult the documentation for the xReplMode property for other options.

You start using replication mode by selecting a range of source cells on the grid. Then move the mouse over the replication handle (black box in the lower-right hand corner of the selection block; see below where arrow is pointing):



In this location the mouse cursor changes to a thin lined black cross indicating you can press the left mouse button down and begin dragging. As you drag the replication rectangle follows the mouse showing which cells are selected as the target range. You can extend this range above, below, right or left of the original selection as shown below:



R4C3	R4C4	R4C5
R5C3	R5C4	R5C5
R6C3	R6C4	R6C5
R7C3	R7C4	R7C5
R8C3	R8C4	R8C5
R9C3	R9C4	R9C5

When you release the mouse the onXReplEnd event fires to allow the application to perform whatever operation it desires on the target range of cells. Usually this operation will take data from the source range and move it into the target range after doing some kind of transformation. But anything that makes sense in the context of your application is possible.

You can cancel replication mode while you are still dragging the mouse by pressing the ESCAPE key. This will cause the onXReplCancel event to be signaled rather than the onXReplEnd event.

All of this may sound familiar because it is similar to what Excel™ does when you drag its "fill handle" to create a fill series or repeating fill value. You may want to consult Microsoft's Excel™ documentation if you are unfamiliar with this feature:

http://office.microsoft.com/en-us/assistance/HP051994981033.aspx

The grid's replication mode is similar to Excel™'s fill mode while the user is dragging the mouse. However, once the user releases the mouse button the behavior is completely under the control of the application. There is no automatic series filling or other features such as those built into Excel™. It is up to your application to determine what to do. The grid simply fires an event notifying you that replication mode has ended and giving you the coordinates of the original selection, the ending selection, and some other status flags. Everything else is in your hands.

In order to use replication mode your application must do two things, at a minimum:

1. Set the xReplMode property to enable replication mode and specify whether the selection can be extended along only one axis (like Excel™) or along two axes at the same time. The property also lets you control whether protected cells and cells that don't allow selection should block the target range from being extended across them. This is a property of each page. Settings to the xReplMode property that you make while one page is selected do not affect other pages of the grid.
2. Define a handler for the onXReplEnd event to perform some action when replication mode ends.

Three other events are fired during replication mode that may be useful in some cases:

1. The onXReplStart event is fired when the user presses the left mouse button down while hovering over the replication handle. This event occurs before the replication tracking rectangle has been drawn and you can set `⎕wres[1]←0` to disable replication mode from starting. You may want to use this event to update the status bar with instructions about moving the mouse to expand the target selection. If so, you must be sure to handle both the onXReplEnd and onXReplCancel events to restore the status bar to its normal state when replication mode ends.
2. The onXReplTrack event is fired during replication mode each time the selection changes. It is also fired whenever the keyboard shift state (Ctrl, Alt, and Shift keys) changes. You may want to use this event to update the status bar to show the size of target selection during tracking.
3. The onXReplCancel event is fired when replication mode is cancelled by the user pressing the ESCAPE key. In this case the onXReplEnd event is not fired.

Managing Virtual Mode

When the grid is used to handle large amounts of data, memory requirements may exceed available memory and data loading can be very slow. These problems can be addressed using “virtual mode”.

Virtual mode allows very large data sets (more than 10 million cells) to be handled efficiently. The cells of the grid are loaded on demand, as needed to display on the screen (as well as for printing, clipboard and XML operations) rather than requiring all data to be preloaded before the grid is displayed. If your application needs to handle large amounts of data and is able to respond quickly to random access requests for small blocks of data then you may be able to benefit from using virtual mode.

This section may be useful to you even if you do not plan on using virtual mode. The progress tracking mechanism supported by virtual mode is also available to non-virtual grids during long running printing, clipboard, and xml operations. See the “Progress Reporting” topic later in this section for details.

It is relatively easy to use virtual mode. The minimal implementation involves setting one or two properties (xVirtualMode and optionally xVirtualParts) and handling one event (onXVirtualLoad). Once you have set up virtual mode for displaying data on the screen, you don’t have to worry about printing, clipboard, or xml operations. They are all supported automatically via the same dynamic loading mechanism.

The VDEMO workspace contains an extended set of examples illustrating why virtualization makes sense and how to use it. It guides you step by step from the basics to a fully functional applet that handles data loading and includes a progress window with cancel button for operations that run longer than 1 second. You can use the included sample functions vLoader, vHandler, vProgress, and vProgressInit as a basis for building virtualization into your own applications. The examples are packaged as a `]DEMO` script that you can execute by loading the workspace and running the user command as shown below:

```
)LOAD C:\Users\johnw\AppData\Roaming\APLNowLLC\APL64\Examples\VDEMO.ws64
]DEMO vDemo
```

Three steps are required to use virtual mode:

1. Set the xVirtualMode property to a value indicating the kinds of virtual data loading your application requires. The default value, 0, disables virtual mode. Setting its value to 1 enables cells to be virtually loaded on demand when needed for display on the screen or for printing, clipboard, or xml operations. Other codes can be used in combination with 1 to selectively disable virtualization of printing, clipboard, and xml operations. There is no code to disable virtual loading for display on the screen. If you enable virtual loading at all, you will get virtualization of the screen display. There is also a code to disable the automatic unloading of data that has been virtually loaded. The xVirtualMode property is set independently for each page.

2. Optionally set the xVirtualParts property to control which parts of the grid are virtualized. Six parts of the grid can be virtualized: body, column header, row header, corner button, column defaults, and row defaults. All six parts are virtualized by default when virtual mode is enabled. This property lets you disable virtualization of any parts that are not used by your application. For example, if your application does not use row defaults (or uses them sparingly) you could disable them and manually preload the few rows where row defaults are needed, if any. This will avoid unnecessary onXVirtualLoad events from being fired for parts that are not used by your application and improve performance. The xVirtualParts property is set independently for each page.
3. Define an onXVirtualLoad event handler to load values on demand. The argument includes a reason code indicating why the event was fired (to load values for display on the screen, printing, clipboard, or xml operations, or for progress reporting without loading any cells), a matrix of coordinates for the cells that require values to be loaded, a range specifying the minimum and maximum positions of any cells being loaded, a code indicating which part is being loaded (body, row headers, column headers, corner button, row defaults, or column defaults), and three progress tracking metrics: estimated percent complete, actual elapsed time, and estimated time remaining. Your handler must compute the appropriate values for the requested cells and loads them into the grid by invoking `⎕wi` to set properties such as xText, xNumber, or xValue.

The xVirtualMode property can be set with the following codes (or use the default `0` to disable virtual mode):

1	Enable virtual mode plus any combination of the following:
2	Disable loading of cells for printing
4	Disable loading of cells for clipboard cut/copy operation
8	Disable loading of cells for clipboard paste operation
16	Disable loading of cells for xml output
32	Do not automatically unload cells

The xVirtualParts property can be set with any combination of the following codes (or use the default `¯1` to select all parts):

1	Body cells	`(Row > 0    and    Col > 0)`
2	Column header cells	`(Row < 0    and    Col > 0)`
4	Row header cells	`(Col < 0    and    Row > 0)`
8	Corner button cell	`(Row = ¯1   and    Col = ¯1)`
16	Column default cells	`(Row = 0    and    Col > 0)`
32	Row default cells	`(Col = 0    and    Row > 0)`

The onXVirtualLoad event has the following syntax:

```
⎕wevent ↔ 'XVirtualLoad'
⎕warg   ↔ Reason Cells Range Part PercentDone ElapsedTime RemainingTime Extra
⎕wres[1] ← Reason
```

Where argument and result are defined as follows:

`⎕warg[1]`	Reason	Reason event was fired (see table below)
`⎕warg[2]`	Cells	Matrix of cell coordinates to be loaded
`⎕warg[3]`	Range	Minimum and maximum range of coordinates in Cells (see below)
`⎕warg[4]`	Part	Part of grid being loaded (see below)
`⎕warg[5]`	PercentDone	Percentage of work completed (always zero for screen)
`⎕warg[6]`	ElapsedTime	Actual elapsed time (always zero for screen)
`⎕warg[7]`	RemainingTime	Estimated time remaining or `¯1` for screen or if not yet estimated
`⎕warg[8]`	Extra	Reserved for future use
`⎕wres[1]`		Can be set to 0 to cancel long running operations

The Reason code can be one of the following:

1	Loading cells for display on the screen
2	Loading cells for printing
4	Loading cells for clipboard cut/copy operation
8	Loading cells for clipboard paste operation
16	Loading cells for xml generation

The Cells argument is a two-column array of cell coordinates. `Cells[;1]` are row coordinates and `Cells[;2]` are column coordinates. If no cells are being requested for loading (Part=0: progress reporting events) then Cells will be an empty matrix of shape 0 2.

The Range argument is a four element vector specifying the minimum and maximum position of all items in the Cells argument. It is reported as a standard grid range: Row Col RowExtent ColExtent.

The Part argument indicates which part of the grid is being loaded. It uses the same codes as the xVirtualParts property plus zero (0) for progress reporting events where no data is being requested for loading.

After your handler returns, the grid checks the Cells list that it requested you to load. Any cells it finds on this list that have been loaded during the event get their virtually-loaded attribute set. You can query the virtually-loaded attribute (or change it) via the xVirtualLoaded property.

Programming Alert: Any cells you load during your onXVirtualLoad handler that are **not** part of the Cells list will **not** get their virtually-loaded attribute set. Thus, they will become permanently loaded, perhaps by accident! See the “Loading and Unloading of Cell” topic below for details.

Any cells that were requested for loading but were not loaded will be requested again if needed (since they remain clear and continue to meet the qualifications necessary for loading).

Consult the documentation for these properties and the event for more detailed descriptions.

Example

The following example is adapted from the VDEMO workspace. A grid is created, virtual mode is enabled, unused parts are excluded, and the vLoader function is used to handle the onXVirtualLoad event:

```
      ⎕wself←'f' ⎕wi 'Create' 'Form' ('where' whereForm)
      ⎕wself←'f.g' ⎕wi 'Create' 'APL.Grid' ('where' 0 0,'f'⎕wi'size')
      ⎕wi 'xRows' 100000
      ⎕wi 'xCols' 99
      ⎕wi 'xColSize' ¯1 (⎕wi 'xColSize' 1)
      ⎕wi 'savecontent' 0
      ⎕wi 'onXVirtualLoad' 'vLoader'
      ⎕wi 'xVirtualParts' (+/1 2 4 8)
      ⎕wi 'xVirtualMode' 1
```

Just to be safe the xVirtualMode property is set last, after the event handler and parts have been selected. Also, note that savecontent is set to 0. This disables saving of content when the grid is closed which can be *very* time consuming and is generally not a good idea unless you *really* need that functionality. So it is usually best to disable it, whether running a virtual grid or not.

Here's the vLoader function:

```
    ∇ vLoader;cells
      ⍝ Simple onXVirtualLoad handler to load data into grid
      cells←2⊃⎕warg
      :Select ⎕warg[4] ⍝ what part are we loading?
        :Case 1 ⋄ ⎕wi 'xNumber' cells mv (cells[;1]+0.01×100|cells[;2]) ⍝body
        :Case 2 ⋄ ⎕wi 'xNumber' cells mv (cells[;2])              ⍝col header
        :Case 4 ⋄ ⎕wi 'xNumber' cells mv (cells[;1])              ⍝row header
        :Case 8 ⋄ ⎕wi 'xText' ¯1 ¯1 'X'                           ⍝corner btn
      :EndSelect
    ∇
```

The function uses scatter-point indexing to simplify storing values into cells that may be scattered across non-rectangular, non-contiguous parts of the grid. This avoids the less efficient looping and multiple ⎕wi calls that would otherwise be necessary. See the "Using Scatter-Point Indexing" topic in the "Getting Started" section for details. Scatter-point indexing uses the system missing value as a placeholder for the Cols argument. This special value can be obtained by referencing the missing property of the system object as follows:

```
      mv ← '#' ⎕wi 'missing'
```

Loading and Unloading of Cells

It is important to understand when cells are candidates for loading and unloading by the grid. Otherwise the grid's behavior might surprise you in an unpleasant way. A cell can be virtually loaded only when it is completely "clear". Setting any properties for a cell makes it ineligible for virtual loading and the onXVirtualLoad event will not ask your handler to load it again. After a cell is virtually loaded, the grid automatically marks it as "virtually loaded" (this attribute can be queried or reset via the xVirtualLoaded property).

Once a cell has been loaded, it will only be unloaded if its virtually-loaded attribute remains set and its changed attribute remains clear (the changed attribute can become set due to user input or if you mark the cell via the xChanged or xChanges properties).

Therefore, if you want to prevent a cell from being loaded, set any property. And if you want to prevent a cell from being unloaded, set its changed attribute or clear its virtually-loaded attribute.

This means that if you preload some cells the virtual loading mechanism will leave them completely alone. It will not load them again because they already have some properties set. And it will not unload them because they will not have their virtual-loaded attribute set (unless you explicit set it via the xVirtualLoaded property).

Preloading can be valuable for handling exception cases (such as highlighting of cells requiring special attention) because it can simplify your handler from having to worry about these special cases. And this means your handler can be streamlined to focus only on loading of values without concern for special cases.

Programming Alert: But this can be a double-edged sword. If you mistakenly set some attribute for a cell but neglect to set its value, that cell will appear blank because virtual loading will not ask you to load its value. One place to be especially careful is when restoring the changed attributes for a set of cells after resuming a suspended edit session. You cannot simply invoke the xChanges property because this might set the changed attribute on unloaded cells and create problems. You must also make sure to restore the value that would have been loaded!

Unloading of cells that have been virtually loaded happens in two ways: manual and automatic.

Manual Unloading: Cells that are loaded for display on the screen are **not** unloaded automatically. They are "sticky" and stay loaded until you take steps to unload them explicitly via the XVirtualUnload or XDeleteCells methods. Users often revisit the same cells repeatedly while browsing a grid. This sticky behavior promotes faster performance when they scroll back over cells that were previously visited since these cells don't need to be loaded again. And usually, the user will only view a very small fraction of the total cells in a virtual gird. So keeping those cells loaded is not a problem in most cases.

When it is a problem, you can periodically call the XVirtualUnload method to unload them. Unlike the XDeleteCells method (which deletes all cells you specify), the XVirtualUnload method scans a range of cells and only deletes those that meet the following conditions:

1. The cell's virtually-loaded attribute must be **set**. You can query or reset this attribute of a cell via the xVirtualLoaded property.
2. The cell's changed attribute must **not** be set. You can query or reset the changed attribute of a cell via the xChanged property. You can get a list of all cells with their changed attribute set on the current page via the xChanges property. You can also use this property to clear the changed attribute from all cells on the current page.

The XVirtualUnload method can run incrementally in the background, sweeping through the entire grid a few milliseconds at a time (it can unload thousands of cells in a few milliseconds) in order to have minimal impact on the performance of your application or others.

Automatic Unloading: When cells are loaded for purposes other than display on the screen (printing, clipboard, or xml operations) they do not exhibit this "sticky" behavior. They are unloaded automatically. For example, cells are virtually loaded during printing in blocks of about 5000 cells. After each block of cells has been printed, the grid automatically unloads them as if you had called the XVirtualUnload method. Cells are loaded and unloaded in a wave passing through the rows and columns of the grid.

Saving Changes

Unless you are displaying a read-only grid, you will eventually need to detect which cells have changed due user input and collect their final values.

The xChanges property makes it easy to detect which cells have changed. It returns a matrix of coordinates for every cell that has been changed due to user input (or user input emulated by setting the xChanged property for a cell).

Collecting the values of changed cells is easily accomplished via scatter-point indexing (see the "Using Scatter-Point Indexing" topic in the "Getting Started" section for details). The following example references the xChanges property and uses the matrix of cell coordinates it returns to reference the xValue property for all the changed cells. This yields the coordinates and values of all cells that have changed.

```
changes ← ⎕wi 'xChanges'
values  ← ⎕wi 'xValue' changes mv
```

The mv variable used above must be set to the system missing value as shown below:

```
mv ← '#' ⎕wi 'missing'
```

If you are saving and continuing rather than exiting from the grid, you will want to clear all changed flags by setting the xChanges property with an empty numeric vector or matrix. For example:

```
⎕wi 'xChanges' ⍬
```

Do not try using an empty character vector argument. Because of the rules of ActiveX argument handling, it will be misinterpreted as a scalar string of length zero and cause an error.

Optimization

There are several things you can do to get optimal performance in a virtual grid. These include preloading of exceptional cells, preloading of column defaults and headers, suppression of events being fired for the corresponding parts of the grid that you have preloaded (or don't care about loading), and suppression of unloading. These things reduce the number of virtual load events you have to handle and streamline the number of properties you have to set during each virtual event.

The most important optimization is usually to define as many attributes as possible as column defaults and let them be inherited by the cells in each column (assuming your data is column oriented). You should set properties such as cell type, alignment, formatting, etc. as column defaults that are shared by all cells in the column with only a few exceptions. This allows your virtual loader to focus on just setting data values rather than worrying about appearance and therefore it can run faster. Often you will only need to set a single property such as xText, xNumber, or xValue.

When possible, use the grid's built in formatting functionality provided by the xFormat property (see the "Details on the xFormat Property" section for details). This will allow you to use the numeric valued properties xNumber or xDate rather than formatting the value in your handler and setting the xText property. Numeric properties are faster to load into the grid than text properties, especially when passing several thousand values at a time. And formatting via the xFormat property is often substantially faster than coding the same formatting in APL.

Most virtual grid applications will have a relatively small number of columns compared to the number of rows. Rather than dynamically loading column headers and column defaults on demand, you can usually preload them all at once at start up and suppress those parts of the grid from virtual loading. The same thing applies to the corner button header cell. Doing this results in fewer events being fired and streamlines your handler code to have fewer and simpler cases.

Since very large grids have many more rows than columns (since there are limits on the number of columns but not on the number of rows), this strategy won't work as well for row headers and row defaults. If you can avoid using row defaults then simply disable that part from loading. But even if you use row defaults sparingly (such as for highlighting exceptional rows) then you can preload their highlighting attributes as row defaults for any exceptional rows and disable virtual loading of that part of the grid.

Unless your application does not use row headers, you will not be able to disable them from virtual loading since preloading usually won't be a viable option. But since row labels can often be generated directly from the row number without accessing a database they will usually be fast to load. For example, the row header might simply be the row number. In that case your onXVirtualLoad event handler can set the xNumber property with the row number.

In some cases, you may be able to optimize performance by not unloading cells used in printing, clipboard, and xml operations. This can be done by setting the "no unload" option of the xVirtualMode property. Once cells are loaded for any purpose, you can keep them around and avoid having to reload them again for subsequent operations. This only makes sense when your grid's total amount of data is small enough to all be loaded into memory at once. In that case, virtualization could be used to avoid the performance hit of preloading all data during startup, but over time, the grid's data would become progressively loaded into memory and then stay there.

A variation on this theme would be to run a timer in the background to progressively load all cells into the grid a few thousand at a time until it is fully loaded. You can use the xVirtualLoaded property to check which cells are already loaded and avoid reloading them. Once such a grid is fully loaded you can turn off virtual mode.

Pre-Sizing of Column Width

You must preset column widths. Column width is not a cell attribute and is not virtualized as a separate part of the grid. Therefore, you will not receive any events asking you to set column width on demand, when the column is first being displayed or printed. But even if you hooked column sizing to events about column default cells, it won't work properly. Dynamically sizing column when they are about to be displayed sometimes causes painting problems that can be very confusing and annoying to users. This is a known bug that has not yet been resolved.

In addition, dynamically changing column widths during horizontal scrolling leads to confused positioning since the grid calculates how far to scroll based on the old widths and ends up displaying data with a different set of widths.

All of these problems can be avoided by setting the xColSize property for all columns when the grid is first created. You may find the XTextSize method useful for helping to determine an appropriate size for your columns based on actual content and font.

Progress Reporting

Some operations on large grids can run for such a long time that they may appear to be frozen because they don't give users feedback on their progress or offering them a way to cancel. The most problematic of these are printing, clipboard, and xml operations. Virtual Mode fully supports progress reporting during these operations via three arguments of the onXVirtualLoad event: estimated percent done, actual elapsed time, and estimated time remaining to completion.

These capabilities should not be overlooked for non-virtual grids. Even when virtual mode is disabled the onXVirtualLoad event is periodically fired during these long running operations *without requesting data to be loaded*. These events use a code to indicate they are for "progress reporting" and that no data is expected to be loaded. This means that both virtual and non-virtual grids can use the exact same progress reporting mechanism for long running printing, clipboard, and xml operations.

All you need to do is monitor the progress arguments when processing the event and display, update, or hide the form as appropriate.

You can use the elapsed time to decide when to show the form. It is usually a good idea to wait 1 or 2 seconds before displaying a progress window. If the operation doesn't end up taking very long the rapid fire opening and closing of the form will create more of an annoying flicker than useful feedback to the user. Once you reach the desired delay time, examine the estimated time remaining. If that value is too small don't display the form, since it will be closed again very soon and create the same flicker problem as described above. But if the remaining time is negative (indicating that the grid has not yet been able to estimate completion time) then you should go ahead and show the form.

You can use the percent done argument to set the position of a progress gauge and as a signal for closing the progress window (when it reaches 100%). All of these operations fire a final event with a percent done value of 100 when their work is fully complete. Prior to completion, they return a value of 99 or less. So 100% is a definitive signal to kill the progress form that won't be received twice for the same operations.

You can use the estimated time remaining to display a message indicating how much longer is expected for the operation to complete. This can be useful if the user wants to take a coffee break, etc. Keep in mind that ¯1 means *not yet computed*. The grid waits a few seconds until it has enough data to begin estimating remaining time. When you see a negative remaining time value you should display a blank status line for the time estimate (or perhaps a message saying estimate not yet available).

The VDEMO workspace gives a complete example for displaying a progress window with a cancel button. Parts of that demo are repeated below. The vHandler function below can be used to handle display of a progress window. Set it as the handler in place of the vLoader function that was used in the previous example.

```
    ∇ vHandler;s
[1]   ⍝ Handle onXVirtualLoad including progress window
[2]
[3]   ⍝ display progress window and detect cancel
[4]   :if vProgress ⎕warg[1 5 6 7]
[5]     ⍝ user cancelled... abort operation
[6]     →⎕wres[1]←0
[7]   :endIf
[8]
[9]   ⍝ load the requested data
[10]  vLoader
    ∇
```

The vHandler function is not much more complicated than the vLoader function (which it calls to load the cells). It adds a call to vProgress where most of the work is done in handling the progress window:

```
  ∇ kill←vProgress arg;⎕wself;x;pd;et;rt;re
    ⍝ Show progress window with cancel button
    ⍝ return 1 if user cancelled operation; otherwise return 0
    ⍝ arguments are: Reason, PercentDone, ElapsedTime, RemainingTime

    ⍝ parse arguments
    (re pd et rt)←arg

    ⍝ ignore event if loaded for display (it is just a paint event)
    →(re=1)/kill←0

    ⍝ create progress window centered on the form containing grid
    ⎕wself←vProgressInit ⎕wself ⎕wi ':self'

    ⍝ give UI a chance to handle cancel button
    ⎕wgive 0

    ⍝ check for cancel button
    :if ~⎕wi 'opened'
      :if ⎕wi ''open'
        kill←1
      :endif
      :return
    :endif

    ⍝ check for 100% done
    :If pd=100
      ⎕wi ''open' 0 ⍝ close cleanly without cancel
      x←⎕wi 'Close'
    :endif

    ⍝ update caption (if changed)
    :if re≠⎕wi ''re'
      :select re
        :case 2
          re←'Printing...'
        :case 4
          re←'Copying data to clipboard...'
        :case 8
          re←'Pasting data from clipboard...'
        :case 16
          re←'Generating XML...'
        :else
          →0 ⍝ ignore anything else
      :endselect
      '.msg' ⎕wi 'caption' re
    :endif

    ⍝ update progress gauge (if changed)
    :if pd≠⎕wi ''pd'
      ⎕wi ''pd' pd
      '.pro' ⎕wi 'value' pd
```

```
[52]  :endif
[53]
[54]  ⍝ update estimated time remaining (if changed)
[55]  :if rt≠⎕wi ''rt'
[56]    ⎕wi ''rt' rt
[57]    :if rt≥0
[58]      '.rem' ⎕wi 'caption' ((⍕rt),' seconds remaining')
[59]    :else
[60]      '.rem' ⎕wi 'caption' 'Estimating remaining time...'
[61]    :endif
[62]  :endif
[63]
[64]  :if 0>⎕wi 'visible'
[65]  :andif et≥1
[66]  :andif (rt≥1)∨rt<0
[67]    ⎕wi 'visible' 1 ⍝ make sure form is visible
[68]  :endif
    ∇
```

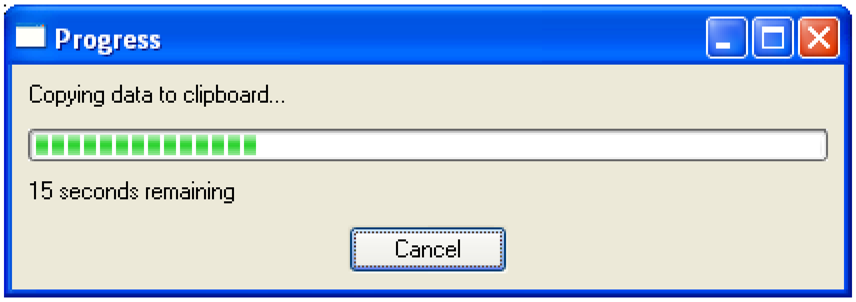
The final function in this suite is vProgressInit. It is called by vProgress to create and initialize the progress window and does a special trick to make it into an “owned window” that is owned by the form containing the grid but without having to do the normal Wait operation that can’t be used in this context.

```
    ∇ name←vProgressInit owner;h;s;w;p;a;⎕wself
[1]   ⍝ [re]create progress window as a child of <owner> form
[2]   ⍝  unless it already exists.  Return name of created form.
[3]   ⍝ Set its caption equal to owner form's caption.
[4]   ⍝ Use onPreCreate event to make progress form an ←owned window←
[5]   ⍝  that is always on top of the owner and gets hidden when the owner
[6]   ⍝  is hidden or minimized, etc.  This simply requires setting
[7]   ⍝  ⎕wres[9] to the hwnd of the owner form (little known trick).
[8]
[9]   ⍝ test existance
[10]  name←'vProgress_',owner
[11]  :if ''≡⎕wself←name ⎕wi 'self'
[12]    ⍝ make sure owner form is open so we can create progress
[13]    ⍝ form as an owned window (need a non-zero hwnd)
[14]    h←owner ⎕wi 'hwnd'
[15]    ⎕error(h=0)/'Owner form must be open before calling vProgress'
[16]    ⎕error(~'Form'≡owner ⎕wi 'class')/'Owner must be a top level form'
[17]    ⎕error('.'∊owner ⎕wi 'self')/'Owner must be a top level form'
[18]
[19]    ⍝ place progress form centered above owner form
[20]    ⍝ unless that would push it off edge of owner or desktop.
[21]    ⍝ In that case adjust position so that as much as possible
[22]    ⍝ of top and left edges of progress form are visible.  But
[23]    ⍝ consider bottom/right edges of owner and desktop first so
[24]    ⍝ that as much as possible of who progress form will be visible
[25]    s←7.0 52 ⍝ size of progress form
[26]    p←owner ⎕wi 'where' ⍝ position of owner form to be centered above
[27]    w←(p[1 2]+p[3 4]×0.5)-s×0.5 ⍝ ideal location before corrections
[28]    w←((p[1 2]+p[3 4])⌊w+s)-s ⍝ don't go past bottom/right of owner
[29]    a←'#' ⎕wi 'workarea' ⍝ desktop workarea
[30]    w←((a[1 2]+a[3 4])⌊w+s)-s ⍝ don't go past bottom/right of desktop
[31]    w←a[1 2]-p[1 2]-w ⍝ don't go past top/left of owner or desktop
[32]
[33]    ⍝ create progress form and its children
[34]    ⎕wself←name ⎕wi 'Create' 'Form' 'Close'
[35]    ⎕wi 'where' w
[36]    ⎕wi 'size' s
[37]    ⎕wi 'border' (1 128) ⍝ +16 gives close button and system menu
[38]    ⎕wi 'caption' (owner ⎕wi 'caption')
[39]    x←'.msg' ⎕wi 'New' 'Label'    ('where' 0.5 1  1.0 50)
[40]      '.msg' ⎕wi 'caption' '...'
[41]    x←'.pro' ⎕wi 'New' 'Progress' ('where' 2.0 1  1.0 50)
[42]      '.pro' ⎕wi 'style' 1
[43]    x←'.rem' ⎕wi 'New' 'Label'    ('where' 3.5 1  1.0 50)
[44]      '.rem' ⎕wi 'caption' ''
[45]    x←'.can' ⎕wi 'New' 'Button'   ('where' 5.0 21 1.5 10)
[46]      '.can' ⎕wi 'caption' 'Cancel'
[47]      '.can' ⎕wi 'style' 2
[48]  :endif
[49]
[50]  ⍝ open form in hidden state if closed
[51]  :if 0=⎕wi 'opened'
```

```
[52]    ⎕wi 'onPreCreate' ('⎕wres[9]←',⍕owner ⎕wi 'hwnd') ⍝owned window trick
[53]    ⎕wi ''open' 1 ⍝ open for business (used to detect cancel)
[54]    x←⎕wi 'Hide'
[55]  :endif
    ∇
```

There is plenty of room for improvement of these functions including disabling and displaying an hour glass pointer in the form containing the grid (so the user cannot interact with it during the long running operations).

Here's a sample of the progress window displayed by vProgress:



Copying data to clipboard...

15 seconds remaining

A couple of important things should be noted about handling progress events and allowing users to cancel. Some operations return an error code indicting if they have succeeded or been cancelled. Other operations signal an error when they are cancelled. In both cases your application must be prepared to handle cancellation.

The clipboard operations invoked via the XCopy, XCut, and XPaste methods and their corresponding built-in shortcuts Ctrl+C, Ctrl+X, and Ctrl+V do **not** signal any error nor do they return a value indicating whether they were cancelled. These operations are normally hooked to user interface actions (using the shortcuts or menus) and the user already knows if he cancelled them and no special action is required.

The XPrintPages method returns a value of 0 if cancelled and 1 if successful. But the related XPrintPage method does not have a return value (and one could not be added for historical reasons). Therefore it signals an error if cancelled. For similar reasons, all of the XML properties also signal errors when cancelled.

Therefore you will want to trap errors for these cases by placing them inside a `:try` block and taking appropriate action in the `:catchall` block that executes in response to an error (of use your own favorite error handling scheme).

The following error is signaled when cancellation occurs for the XPrintPage method and the XML properties:

```
⎕WI ERROR: APL.Grid exception 80004005 CANCELLED: The operation was cancelled by
the user
      ⎕wi 'errormessage'
CANCELLED: The operation was cancelled by the user
      ⎕wi 'errorcode'
¯2147467259
```

Managing Clipboard Operations

The grid supports standard clipboard functionality for simple tab/newline-delimited data. Shortcut keys Ctrl+X and Ctrl+C do cut and copy of the selected cells from the grid to the clipboard (the current selection must be a single cell or a single block; multiple selection disables the cut and copy operations). Shortcut key Ctrl+V does a paste operation from the clipboard into the grid starting at the active cell (if only one cell is selected). If the current selection contains multiple cells in a single block the clipboard contents are pasted into that block and any excess content on the clipboard is ignored. If the selected target block is larger than the clipboard content, the content is *not* replicated to fill the selected size. When multiple blocks are selected paste is disabled.

There are six methods for performing clipboard operations and testing if the grid can support these operations (such as testing for protected cells or multiple selections). These are XCanCut, XCanCopy, XCanPaste, XCut, XCopy, and XPaste. These methods are intended to be hooked to your application's menus and toolbars to control whether menu items are enabled and to perform clipboard operations.

Each of the above methods causes a similarly named event to be fired that you can override to change the default behavior. These are onXCanCut, onXCanCopy, onXCanPaste, onXCut, onXCopy, and onXPaste.

Built-in shortcut keys such as Ctrl+X first invoke the "Can" version of a method (causing its corresponding event to be fired) and if it is successful they then invoke the "Do" version of the method (causing its corresponding event to be fired). For example, pressing Ctrl+X first invokes the XCanCut method (causing the onXCanCut event to be fired) and if that is successful (i.e., if it returns 1) it then invokes the XCut method (causing the onXCut event to be fired).

However, when you invoke the methods directly they only cause one event to be fired. For example, the XCut method does not fire the onXCanCut event. It only fires the onXCut event!

The following list describes the algorithm used for each method:

XCanCopy method:

1. If multiple blocks are selected this method returns zero (0) immediately without firing the onXCanCopy event.
2. The onXCanCopy event is fired with the Enable argument initialized as one (1). Your handler can do whatever checking it needs to determine how to set the Enable result `(⎕wres[1])` value. However, this should be done as rapidly as possible (see the remarks below).
3. If your handler leaves the Enable result `(⎕wres[1])` set to one (1) or if you do not define a handler, the method returns one (1). Otherwise it returns zero (0).

Remarks:
The onXCanCopy event is fired *without* doing virtual loading of any cells in the selected block (see the "Managing Virtual Mode" section for details). If your handler needs to have cells virtually loaded remember that doing so can make the operation too slow to be useful for enabling of menu items and toolbar buttons.

XCanCut method:

1. If multiple blocks are selected this method returns zero (0) immediately without firing the onXCanCut event.
2. If any cell in the selected block is protected (xProtect property) or has a non-default cell type (i.e., if its xCellType property is explicitly or implicitly set to a non-zero value such as Combo=1 or Check=2) then the Enable argument of the onXCanCut event will be set to zero (0). Otherwise the Enable argument will be set to one (1).
3. The onXCanCut event is fired with the Enable argument initialized as described above. Your handler can do whatever checking it needs to determine how to set the Enable result `(⎕wres[1])` value. However, this should be done as rapidly as possible (see the remarks below).
4. If you do not define an onXCanCut event handler the value of the Enable argument computed as described above will become the return value for this method. If your handler sets the Enable result `(⎕wres[1])` to one (1) the method returns one (1). Otherwise it returns zero (0).

Remarks:
The test described in item (2) above is done *without* virtual loading of any cells in the selected block (see the “Managing Virtual Mode” section for details). Virtual loading is done during the XCut method and does enforce protection settings. But these checks are potentially too expensive for this method because it must have a very low overhead to be useful for enabling of menu items and toolbar buttons.

Because of this the wrong default value may be computed for the onXCanCut event’s Enable argument. However, if you are using a virtual grid with preloaded cell types defined as column defaults and if protected state is set on the whole grid or as row or column default, or if you have preloaded exceptional cells requiring protection, then the default setting of the Enable argument should be satisfactory.

If it is not, your onXCanCut event handler can override the Enable setting if necessary. But if your handler needs to have cells virtually loaded remember that doing so can make it too slow to be practical, especially when the selection block is large.

XCanPaste method:

1. If multiple blocks are selected this method returns zero (0) immediately without firing the onXCanPaste event.
2. No checking is done for cells in the selected range being protected or valid for pasting before firing the onXCanPaste event. It also does *not* check that the clipboard contains plain-text (CF_TEXT) format data that the XPaste method knows how to handle. These things are handled by the XPaste method and/or by your onXCanPaste event handler.
2. The onXCanPaste event is fired with the Enable argument initialized to one (1). Your handler can do whatever checking it needs to determine how to set the Enable result (`⎕wres[1]`) value. However, this should be done as rapidly as possible (see the remarks below).
3. If your handler leaves the Enable result (`⎕wres[1]`) set to one (1) or if you do not define a handler, the method returns one (1). Otherwise it returns zero (0).

Remarks:
The onXCanPaste event is fired *without* doing virtual loading of any cells in the selected block (see the “Managing Virtual Mode” section for details). However, virtual loading is done during the XPaste method and does enforce protection settings. If your handler needs to have cells virtually loaded remember that doing so can make the operation too slow to be useful for enabling of menu items and toolbar buttons.

XCopy method:

1. If multiple blocks are selected this method returns immediately without firing the onXCopy event.
2. The onXCopy event is fired with the DoDefault argument initialized to one (1). Your handler can override the operation by setting the DoDefault result (`⎕wres[1]`) value to zero (0) but if it does so it is completely responsible for handling the entire operation itself.
3. If your handler leaves the DoDefault result (`⎕wres[1]`) set to one (1) or you do not define a handler, the method continues with step (4) below. Otherwise, it returns immediately without doing any further processing.
4. As each cell in the selected block is about to be copied to the clipboard, the grid ensures that its value has been virtually loaded (cells are virtually loaded in groups of about 5000 cells at a time if virtual-mode is enabled for cut/copy operations) so that its data type and value are available. See the “Managing Virtual Mode” section for details.

5. The value of the xEditText property of each cell is copied to the clipboard. This may not be what you expected for some cell types. If you need a different behavior you have to override the default behavior by handling the clipboard yourself in your onXCopy event handler. When doing this, be sure to set the DoDefault result (`⎕wres[1]`) to zero (0).

Remarks:

If you have virtual mode enabled, you must be very careful when handling the copy operation on your own (i.e., when you return DoDefault=0 from the onXCopy). Virtual loading is *not* done before firing the onXCopy event (see the "Managing Virtual Mode" section for details). That happens later, incrementally, while cells are being copied to the clipboard (see step 4 above). Therefore the actual cell values in the selected block are not guaranteed to be available to your handler.

You will therefore have to refer to your data source rather than to the grid for data values. But you must take into consideration any changed values (see xChanged property) that have not been saved into your data source. These must take precedence over values from your data source. You can determine what values are changed by referencing either the xChanged or xChanges properties.

XCut method:

1. If multiple blocks are selected this method returns immediately without firing the onXCut event.
2. If any cell in the selected block is protected (xProtect property) or has a non-default cell type (i.e., if its xCellType property is explicitly or implicitly set to a non-zero value such as Combo=1 or Check=2) then the DoDefault argument of the onXCut event will be set to zero (0). Otherwise the DoDefault argument will be set to one (1).
3. The onXCut event is fired with the DoDefault argument initialized as described above. Your handler can override the operation by setting the DoDefault result (`⎕wres[1]`) value to zero (0) but if it does so it is completely responsible for handling the entire operation itself.
4. If your handler returns the DoDefault result (`⎕wres[1]`) as one (1) or you do not define a hander but the DoDefault argument computed above is one (1), then execution continues with step (5) below. Otherwise, the method returns without further processing.
5. As each cell in the selected block is about to be copied to the clipboard, the grid ensures that its value has been virtually loaded (cells are virtually loaded in groups of about 5000 cells at a time if virtual-mode is enabled for cut/copy operations) so that its data type and value are available. See the "Managing Virtual Mode" section for details.
6. The value of the xEditText property of each cell is copied to the clipboard. This may not be what you expected for some cell types. If you need a different behavior you have to override the default behavior by handling the clipboard yourself in your onXCut event handler. When doing this, be sure to set the DoDefault result (`⎕wres[1]`) to zero (0).
7. After each cell is copied to the clipboard its value is cleared. This is not the same as being deleted because deleting would create problems in a virtual grid context (deleted cells are automatically reloaded with their original value). Instead, its value is reset to an empty string. Before doing this however, several other tests must be passed as described in steps 8 and 9 below.
8. Any cell that is protected (xProtect property) or has a non-default cell type (i.e., if the xCellType property is explicitly or implicitly set to a non-zero value such as Combo=1 or Check=2) is not cleared. Instead, the Enable argument used in the next step (onXValueClear event) is set to zero (0).

9. Regardless of the test results from step (8) above, the onXValueClear event is fired and your handler can decline the clearing operation by setting the Enable result (`⎕wres[1]`) to zero (0). If step (8) set the default Enable argument to zero (0) then it does not matter what you return. No further clearing action will be taken unless you do that action in your handler. But this event does give you an opportunity to clear cells that the grid doesn't want to clear (such as those declined by step 8). If this test and that from step (8) are both passed, execution continues below with step 10. Otherwise process skips to the next cell in step (5) above.
10. Finally, the onXValueChange event is fired, the changed attribute is set (see xChanged property), and the value is cleared for any cell allowed by steps (8) and (9) above.

Remarks:

In item (2) above, the test is done *without* doing any virtual loading. However, this is usually OK if your grid is structured correctly. Otherwise you will have to handle the operation in your onXCut event handler. See the remarks for XCanCut method above.

In item (3) above, if you have virtual mode enabled, you must be very careful when handling the cut operation on your own in the onXCut event. Virtual loading is *not* done before firing the event (see the "Managing Virtual Mode" section for details). That happens later, incrementally, while cells are being copied to the clipboard (see step 5 above).

You will therefore have to refer to your data source rather than to the grid for data values. But you must take into consideration any changed values (see xChanged property) that have not been saved into your data source. These must take precedence over values from your data source. You can determine what values are changed by referencing either the xChanged or xChanges properties.

XPaste method:

1. If multiple blocks are selected this method returns immediately without firing the onXPaste event.
2. The onXPaste event is fired with the DoDefault argument initialized to one (1). Your handler can override the operation by setting the DoDefault result (`⎕wres[1]`) value to zero (0) but if it does so it is completely responsible for handling the entire operation itself.
3. If your handler leaves the DoDefault result (`⎕wres[1]`) set to one (1) or you do not define a handler, the method continues with step (4) below. Otherwise, it returns immediately without doing any further processing.
4. If there is plain-text (CF_TEXT) format data on the clipboard, it is pasted into the selected cells of the grid. If the selection is a single cell, that cell serves as the starting point for the paste and the data on the clipboard controls how many rows and columns of the grid are pasted into. If multiple cells are selected (in a single selection block), this block of cells defines the maximum extent that will be pasted. Any excess data (more rows or columns) on the clipboard will be ignored. If there are fewer rows or columns of data on the clipboard than in the selected region, the data will *not* be replicated to fill the selected block.
5. As each cell is about to be pasted the grid ensures that its value has been virtually loaded (cells are virtually loaded in groups of about 5000 cells at a time if virtual-mode is enabled for paste operations) so that its data type and protection attributes can be checked. See the "Managing Virtual Mode" section for details.
6. Any cell that is protected (xProtect property) or has a non-default cell type (i.e., if the xCellType property is explicitly or implicitly set to a non-zero value such as Combo=1 or Check=2) then it is skipped over (with the incoming clipboard value being ignored) and processing continues with the next cell.
7. For each cell that changes value the onXValueChange event is fired and its changed attribute is set (xChanged property).

8. When the paste operation is complete, the number of rows and columns actually pasted from the clipboard are selected and the method returns.

Remarks:

Virtual loading is *not* done before firing the onXPaste event in step (2) above (see the "Managing Virtual Mode" section for details). That happens later, incrementally, while cells are being pasted from the clipboard (see step 5 above).

You will therefore have to refer to your data source rather than to the grid for data values if you want to compare the incoming data from the clipboard to the data already in the grid (this is important for properly setting of the changed attribute). But you must take into consideration any already-changed values (see xChanged property) that have not been saved into your data source. These must take precedence over values from your data source. You can determine what values are changed by referencing either the xChanged or xChanges properties.

Because of rule (6) any non-default type cells such as Combo (1) or Check (2) are skipped over during paste operations. If your grid contains these types of cells and you want them to participate in paste operations you must handle the clipboard operation in your onXPaste event handler. When doing this, be sure to set the DoDefault result (`⎕wres[1]`) to zero (0).

Saving Content – Using XML

XML is an acronym for Extensible Markup Language. This is a standard endorsed by the World Wide Web Consortium for storing data with meaning. If you are going to work with the grid and JavaScript, for example, you need to know this standard and how the grid implements it. This is also a good tool for implementing features such as Copy and Paste. This document does not attempt to provide even an introduction to XML, but you can use most of the XML-related properties described below without knowing the underlying structure.

The xXML property references the entire APL Grid object; this includes all pages, formats, data and datatypes, and defaults. If you set the value of the xXML property with an XML string in APL.Grid format, the contents of the object completely replace the contents of the grid. In use, this property resembles the APL64 def property without a binary format. The XML string returned by the property or used to set the property is a regular APL character vector.

The xXMLChanges property refers to all pages in the grid. When referenced, it returns an XML string which reports all changes made to the grid by user interaction (not programmatic interaction) since the grid was created or the xXMLChanges property was reset. All of the state information present in xXML is not given; only cell values and value types are given in the result. If you reset xXMLChanges to an empty vector, all change flags are cleared; only subsequent changes made by the user are reflected in later references to the property. This property may be useful primarily when accessing the grid outside of APL.

The xXMLTable property allows you to reference a range of cells on the active page. You supply a vector of row arguments and a vector of column arguments to this property to specify the range. You can use the result of referencing this property to set xXMLRange in another location or on another instance of the grid.

The xXMLRange property allows you to reference or set a range of cells, but it returns only non-empty cells, that is, cells that have been changed from one or more of their default attributes or values. You can set this property with a character vector that is an XML string in APL.Grid format.

The xXMLValueRange property acts like the xXMLRange property, but it returns only the values and value types of non-empty cells in the specified range; other cell attributes are not included in the XML data vector.

APL64 uses XML to save the content of the grid in the content property of this ActiveX control. See the descriptions of the content and savecontent properties, and their relationship to the def and state properties in the APL+Win *Windows Reference Manual* if you need more information on saving an object in APL64.

Note that not every property or setting can be saved. For example, the xTrackingWindow property is a window handle that will not exist across a close and open or for reconstituting a grid in another session. You must reference its value, once the grid is open, perhaps in an event handler. Other properties may not be saved due to quirks in the interaction between APL64 and ActiveX in Windows.

Java Script Variants

Some properties are provided to simplify access to an APL Grid when using it outside of APL64, notably with the JavaScript or VBScript languages.

If you are using the grid under JavaScript (or any other non-APL container), you should always set the xUnfocusEndEdit property to `1`. If you do not, the grid losing focus will not terminate edit mode and the grid will appear to be frozen.

Each element of a selection is identified in a separate read-only property. The anchor cell is identified by the xSelectionRow and xSelectionCol properties, and the extents of the selection are identified by the xSelectionRows and xSelectionCols properties. You use the block number, not the row number of the xSelection matrix, as the argument to these selection identification properties; if you omit the argument, it defaults to `1`. The number of blocks selected is identified in the xSelectionCount property. Using these properties provides easier methods for dealing with selected cells from a script language. All of the functionality is accessible when working in APL by using the xSelection property.

The same statement holds true for setting the selection using the XSetSelection method. You can use this method when you are in a circumstance that you cannot easily create an array argument. In APL, you can set the xSelection property directly.

By default, the grid handles arrays of values. In fact, by default the grid returns arrays of values, two dimensional arrays for numeric values and three-dimensional arrays for text. Even a single number is a `1`-by-`1` matrix. The xArrayObjects property is a flag that by default is zero. If you set this property to `1`, the grid wraps array result values into an array access object. With this object, if you are using the grid outside of APL, you can address the grid using script-array notation. For example, you can address the third row and sixth column of the grid as: `array[2][5]` or `array.2.5`

Alternatively, the xConformingResultShape property changes the default behavior. If you set this property to `1`, the shape of the result conforms to the shape of the arguments used to reference it. Thus, if you reference a numeric cell with scalar arguments, the result will be a scalar, and you can access a cell value from a Java Script program without wrapping it by using the xArrayObjects property.

Therefore, either combination of xConformingResultShape =`0` and xArrayObjects = `1` or xConformingResultShape = `1` and xArrayObjects = `0` allows either of these expressions as valid Java Script:

```
num=grid.Number(1,2)[0][0]
num=grid.Number(1,2);
```

The rules for individual cells when xConformingResultShape is set to `1` are:

```
⎕wi 'xNumber' 1 2                    ⍝  Scalar result
⎕wi 'xNumber' (,1) 2                 ⍝  Vector result
⎕wi 'xNumber' 1 (,2)                 ⍝  Vector result
⎕wi 'xNumber' (,1) (,2)              ⍝  Matrix result
```

You may also find the xXMLChanges property useful as a way of capturing any user changes to the grid over a period of time without having to iterate over a range of cells.

Printing the Contents of a Grid Page

You can print the contents of the current grid page (including header cells) using the built-in APL64 printer object (or any other printer for which you can obtain a device context handle, HDC). There are a set of properties and methods that allow you to specify the size of the printer page on which to print and the position to start printing. The basic procedure consists of only a few steps. If you want to make a stylish display, you must generate and position all explanatory and title text on the printer page and exclude those areas from the grid page print window.

The following minimal steps are required to print with an APL64 printer object (not including the steps necessary to create the printer and initialize its attributes, such as margins, orientation, etc):

1. Set the grid's xPrinter property with the name of the printer object. The printer's margin settings are automatically transferred to the grid. You no longer have to set the grid's xPrintMargin property.
2. Call the grid's XPrintInit method to initialize the printer for all cells or a subset range of cells on the current grid page.
3. Call the grid's XPrintPages method specifying all the page numbers to be printed at once. You no longer need to invoke the printer's Draw or Print methods before printing each grid page. Initialization of each page happens automatically (including triggering of the printer's onPage event). And you no longer need to invoke the printer's Eject method after printing each page. That also happens automatically.

Printing multiple pages in one call with the XPrintPages method assumes that you will handle printing of any required page headers, footers, and other decorations via the printer's onPage event that is fired as each page begins. If you prefer to draw these elements without using the onPage event, you can use the more labor intensive XPrintPage method (in which case you will also have to invoke the printer's Eject method after each page) or you can use the XPrintPages method one page at a time (in which case the page eject is done for you automatically).

If you are printing many pages of data, whether running in virtual mode or not (see the "Managing Virtual Mode" section for details) it is highly desirable to use the XPrintPages method for all pages in one call rather than using the XPrintPage or XPrintPages method one page at a time. When you use XPrintPages the progress tracking arguments of the onXVirtualLoad event tell you the percent done, elapsed time, and estimated time remaining for all pages of the overall print job. When you use the XPrintPage method (or the XPrintPages method one page at a time), the progress tracking arguments of the onXVirtualLoad event tell you the progress metrics for only the page being printed. You have to correlate and scale them into overall progress metrics. If you want to use a progress reporting window this substantially complicates your task.

The examples shown in the rest of this section are taken from a `]DEMO` script that you can execute by loading the workspace and running the user command as shown below:

```
)LOAD C:\Users\johnw\AppData\Roaming\APLNowLLC\APL64\Examples\VDEMO.ws64
]DEMO pDemo
```

Printing via Printer Name

In the following example, an instance of the default printer is created, its margins are set to 1 inch (72 points), and the first 300 rows and 100 columns of grid are printed without borders at 75% zoom factor:

```
      'printer' ⎕wi 'Create' 'Printer'    ⍝ create instance of default printer
printer
      e ← "⎕←'*** Page ',⍕⎕warg"          ⍝ handler shows page number
      'printer' ⎕wi 'onPage' e            ⍝ set onPage handler
      'printer' ⎕wi 'scale' 3             ⍝ use "points" scaling
      'printer' ⎕wi 'margin' 72 72 72 72  ⍝ set 1 inch margins all around
      'fm.grid' ⎕wi 'xPrinter' 'printer'  ⍝ tell printer's name to grid
      'fm.grid' ⎕wi 'xPrintBorder' 0      ⍝ suppress outer borders
      'fm.grid' ⎕wi 'xPrintZoom' 0.75     ⍝ use 75% zoom factor
      r ← 1 1 300 100                     ⍝ range of cells to be printed
      p ← 'fm.grid' ⎕wi 'XPrintInit' 0,r  ⍝ prepare to print cell range
      p                                   ⍝ returned values are page numbers
  1  2  3  4  5  6  7  8  9               ⍝ transpose for vert. orientation
 10 11 12 13 14 15 16 17 18
 19 20 21 22 23 24 25 26 27
 28 29 30 31 32 33 34 35 36
 37 38 39 40 41 42 43 44 45
 46 47 48 49 50 51 52 53 54
       ok ← ⎕wi 'XPrintPages' p           ⍝ print all pages at once
*** Page 1                                ⍝ output of onPage handler
*** Page 2
    ...                                   ⍝ many lines of output omitted...
*** Page 54
      c ← 'printer' ⎕wi 'Close'           ⍝ Close to let job start/finish
```

Printing via Device Context

The original printing mechanism was considerably more cumbersome but could still be useful if you need to print to a printer that is not represented by an APL64 Printer object. The following minimal steps required to print via a device context:

1. Query the printer's hdc property for its device context handle.
2. Set the grid's xPrintDC with this value.
3. Query the printer's margin property to determine its margin settings. This had to be done when the printer was in actual-pixel scaling units or you could query them in other units and scale them into pixels yourself.
4. Set the grid's xPrintMargin property with these settings. However, the elements of the printer's margin property where in a different order than those of the grid's xPrintMargin property. So they had to be reordered.
5. Call the grid's XPrintInit method to initialize the printer for a selected range of cells.
6. For each page number returned by the XPrintInit method, you had to:

a. Call the printer's Draw or Print method to request that the printer start a new page. This causes the printer's onPage event to be triggered and also makes the page "dirty" so it will properly eject and/or be included in the print job when the printer was closed. Failing to do so creates problems because the grid cannot notify the printer its page had been drawn using only a device context handle.

b. Call the XPrintPage method to print the page.

c. Call the printer's Eject method.

The same output as the previous example can be produced using the older technique:

```
      'printer' ⎕wi 'Create' 'Printer'    ⍝ create instance of default printer
printer
      e ← "⎕←'*** Page ',⍕⎕warg"          ⍝ handler shows page number
      'printer' ⎕wi 'onPage' e            ⍝ set onPage handler
      'printer' ⎕wi 'scale' 3             ⍝ use "points" scaling
      'printer' ⎕wi 'margin' 72 72 72 72  ⍝ set 1 inch margins all around
      h ← 'printer' ⎕wi 'hdc'             ⍝ get printer's device context (hdc)
      'fm.grid' ⎕wi 'xPrintDC' h          ⍝ tell printer's DC to grid
      'printer' ⎕wi 'scale' 5             ⍝ change printer scaling to pixels
      m ← 'printer' ⎕wi 'margin'          ⍝ query margins in pixels
      'printer' ⎕wi 'scale' 3             ⍝ restore original printer scaling
      m ← m[3 1 4 2]                      ⍝ permute margins into grid order
      'fm.grid' ⎕wi 'xPrintMargin' m      ⍝ set grid margins
      'fm.grid' ⎕wi 'xPrintBorder' 0      ⍝ suppress outer borders
      'fm.grid' ⎕wi 'xPrintZoom' 0.75     ⍝ use 75% zoom factor
      r ← 1 1 300 100                     ⍝ range of cells to be printed
      p ← 'fm.grid' ⎕wi 'XPrintInit' 0,r  ⍝ prepare to print cell range
      p                                   ⍝ returned values are page numbers
  1  2  3  4  5  6  7  8  9               ⍝ transpose for vert. orientation
 10 11 12 13 14 15 16 17 18
 19 20 21 22 23 24 25 26 27
 28 29 30 31 32 33 34 35 36
 37 38 39 40 41 42 43 44 45
 46 47 48 49 50 51 52 53 54

      'printer' ⎕wi 'Print' ' '           ⍝ force printer to start page
*** Page 1                                ⍝ output of onPage handler
      'fm.grid' ⎕wi 'XPrintPage' 1        ⍝ print page
      'printer' ⎕wi 'Eject'               ⍝ eject page; done with this page

      'printer' ⎕wi 'Print' ' '           ⍝ force printer to start page
*** Page 2                                ⍝ output of onPage handler
      'fm.grid' ⎕wi 'XPrintPage' 2        ⍝ print page
      'printer' ⎕wi 'Eject'               ⍝ eject page; done with this page

      ...                                 ⍝ many iterations omitted here...

      'printer' ⎕wi 'Print' ' '           ⍝ force printer to start page
*** Page 54                               ⍝ output of onPage handler
      'fm.grid' ⎕wi 'XPrintPage' 54       ⍝ print page
      'printer' ⎕wi 'Eject'               ⍝ eject page; done with last page

      c ← 'printer' ⎕wi 'Close'           ⍝ Close to let job start/finish
```

Print Preview

It is relatively straightforward to do print preview of grid pages. You describe the printer to the grid as before (by setting the xPrinter or xPrintDC property). Then you tell the grid where the image of printed page should be rendered. This is done by setting the xPrintWindow property with the device context of the window to receive the image along with the coordinates within that window where the page image should be drawn. Finally, you draw the page image by invoking the XPrintPage method. The XPrintPages method does not function when the xPrintWindow property is set so it cannot be used for print preview. The XPrintPage method takes the page number argument zero (0) and small negative integers to draw the page background including a shadow around edge of page image. This makes rendering of the full page image a little easier.

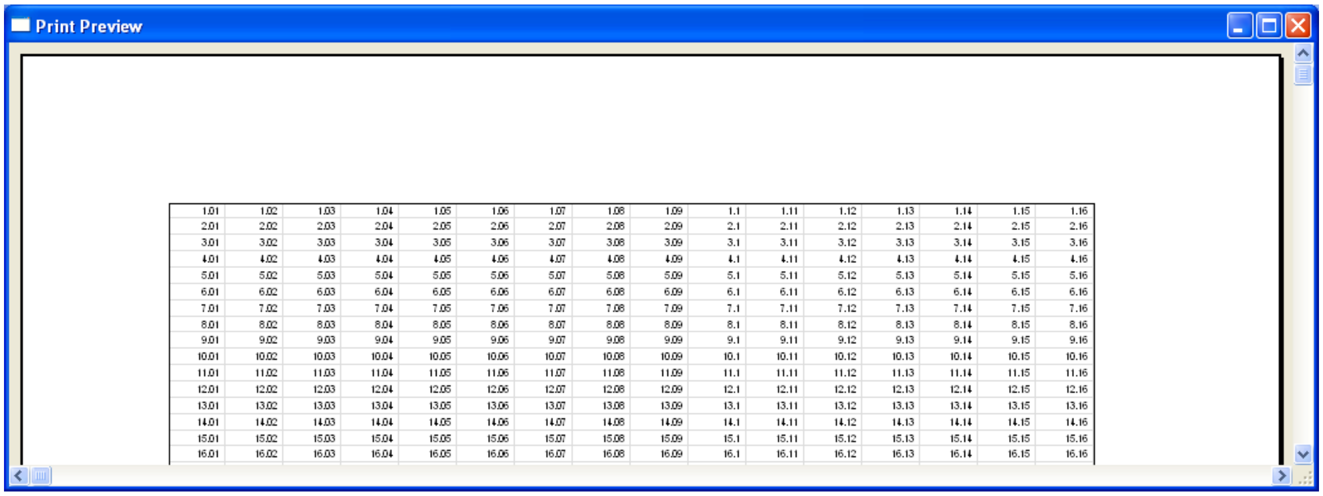
It can be challenging to set up the preview form and determine where to draw the page image in your preview window. This requires knowing the size of the printed page and can be determined by referencing the xPrintMetrics property with the “size” option. It returns the physical page size of the printer page in actual-pixels. You can also reference the xPrintMetrics property with the “inches” option to obtain the pixels per inch scaling for screen and printer. And from this information you can convert page size in actual-pixels into inches. You can also use it for determining the print preview zoom factor so you can display it to the user (this is different than the referencing the xPrintZoom property setting).

Headers and footers can be scaled and drawn into your preview window. Note that the onPage event on the printer is not fired when you use XPrintPage to draw a preview page. So your header, footer, and other decorations must be drawn outside of that event handler context. If you draw them to the printer they will be printed rather than being rendered into the preview window.

The first example below illustrates creating a preview form that contains a Picture object onto which is drawn the print preview output. The imagesize of the Picture is set to be proportional in size to the printer page plus a few pixels for margins around the page image. It is assumed that a printer object named “printer” already exists but we set it up with scaling in points and 1 inch margins for this example:

```
      'printer' ⎕wi 'scale' 3
      'printer' ⎕wi 'margin' 72 72 72 72
      'fm.grid' ⎕wi 'xPrinter' 'printer'
      'fm.grid' ⎕wi 'xPrintZoom' 0.50
      p←'fm.grid' ⎕wi 'xPrintInit' 0 1 1 100 50  ⍝ prepare for 100x50 cells
      p
 1 2 3 4
 5 6 7 8

      ⍝ Create Print-Preview Form including Picture object with scroll bars
      'pre' ⎕wi 'Create' 'Form'
pre
      'pre' ⎕wi 'caption' 'Print Preview'
      'pre' ⎕wi 'scale' 5
      'pre.pic' ⎕wi 'Create' 'Picture'
pre.pic
      'pre.pic' ⎕wi 'where' (0 0,'pre' ⎕wi 'size')
      'pre.pic' ⎕wi 'style' 16 64
      'pre.pic' ⎕wi 'border' 0
      'pre.pic' ⎕wi 'scale' 5

      ⍝ Margins around all sides of preview image (10 pixels)
      m←10

      ⍝ available width of preview picture
      w←('pre.pic' ⎕wi 'size')[2]-m×2

      ⍝ size of printed page (pixels)
      s←'fm.grid' ⎕wi 'PrintMetrics' 'size'

      ⍝ scaling factor
      z←w÷s[2]

      ⍝ set image size in picture object (this can be scrolled)
      'pre.pic' ⎕wi 'imagesize' (m+m+z×s)

      ⍝ Tell grid about preview window where it should draw as an alternative
      ⍝ to the actual printer.  The printer is only used for "reference"
      ⍝ purposes. Drawing takes place in preview window
      'fm.grid' ⎕wi 'PrintWindow' ('pre.pic' ⎕wi 'hdc'),m,m,z×s

      ⍝ Draw page image with 2-pixel wide frame and shadow
      ⍝ Only needs to be done once
      'fm.grid' ⎕wi 'PrintPage' ¯2

      ⍝ Draw page 1 and repaint to show it
      'fm.grid' ⎕wi 'PrintPage' 1
      'pre.pic' ⎕wi 'Paint'
```

This results in the following preview image:

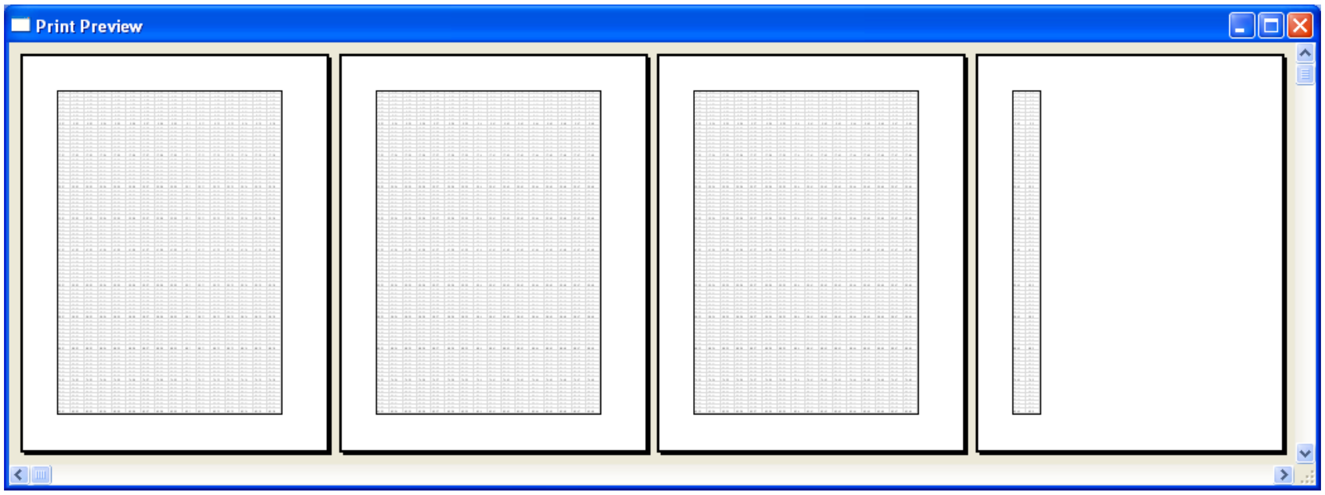


1.01	1.02	1.03	1.04	1.05	1.06	1.07	1.08	1.09	1.1	1.11	1.12	1.13	1.14	1.15	1.16
2.01	2.02	2.03	2.04	2.05	2.06	2.07	2.08	2.09	2.1	2.11	2.12	2.13	2.14	2.15	2.16
3.01	3.02	3.03	3.04	3.05	3.06	3.07	3.08	3.09	3.1	3.11	3.12	3.13	3.14	3.15	3.16
4.01	4.02	4.03	4.04	4.05	4.06	4.07	4.08	4.09	4.1	4.11	4.12	4.13	4.14	4.15	4.16
5.01	5.02	5.03	5.04	5.05	5.06	5.07	5.08	5.09	5.1	5.11	5.12	5.13	5.14	5.15	5.16
6.01	6.02	6.03	6.04	6.05	6.06	6.07	6.08	6.09	6.1	6.11	6.12	6.13	6.14	6.15	6.16
7.01	7.02	7.03	7.04	7.05	7.06	7.07	7.08	7.09	7.1	7.11	7.12	7.13	7.14	7.15	7.16
8.01	8.02	8.03	8.04	8.05	8.06	8.07	8.08	8.09	8.1	8.11	8.12	8.13	8.14	8.15	8.16
9.01	9.02	9.03	9.04	9.05	9.06	9.07	9.08	9.09	9.1	9.11	9.12	9.13	9.14	9.15	9.16
10.01	10.02	10.03	10.04	10.05	10.06	10.07	10.08	10.09	10.1	10.11	10.12	10.13	10.14	10.15	10.16
11.01	11.02	11.03	11.04	11.05	11.06	11.07	11.08	11.09	11.1	11.11	11.12	11.13	11.14	11.15	11.16
12.01	12.02	12.03	12.04	12.05	12.06	12.07	12.08	12.09	12.1	12.11	12.12	12.13	12.14	12.15	12.16
13.01	13.02	13.03	13.04	13.05	13.06	13.07	13.08	13.09	13.1	13.11	13.12	13.13	13.14	13.15	13.16
14.01	14.02	14.03	14.04	14.05	14.06	14.07	14.08	14.09	14.1	14.11	14.12	14.13	14.14	14.15	14.16
15.01	15.02	15.03	15.04	15.05	15.06	15.07	15.08	15.09	15.1	15.11	15.12	15.13	15.14	15.15	15.16
16.01	16.02	16.03	16.04	16.05	16.06	16.07	16.08	16.09	16.1	16.11	16.12	16.13	16.14	16.15	16.16

The next example displays four preview page images side by side in the preview window. Rather than fitting the page width to the available viewing area of the window, it instead fits the page height and then repeatedly sets the xPrintWindow property to point to a different area of the picture's image.

```
⍝ available height for preview page
h←('pre.pic' ⎕wi 'size')[1]-m×2

⍝ size of printer page (pixels)
s←'fm.grid' ⎕wi 'PrintMetrics' 'size'

⍝ scaling factor required to fit page in window height
z←h÷s[1]

⍝ height and width of preview rendering of page
(h w)←z×s

⍝ set imagesize big enough to display four pages across
s←m+(h,w+m+w+m+w+m+w)+m
'pre.pic' ⎕wi 'imagesize' s

⍝ Draw page 1
(x y)←m
'fm.grid' ⎕wi 'PrintWindow' ('pre.pic' ⎕wi 'hdc') y x h w
'fm.grid' ⎕wi 'PrintPage' ¯2
'fm.grid' ⎕wi 'PrintPage' 1
'pre.pic' ⎕wi 'Paint'

⍝ Draw page 2
x←x+m+w
'fm.grid' ⎕wi 'PrintWindow' ('pre.pic' ⎕wi 'hdc') y x h w
'fm.grid' ⎕wi 'PrintPage' ¯2
'fm.grid' ⎕wi 'PrintPage' 3
'pre.pic' ⎕wi 'Paint'

⍝ Draw page 3
x←x+m+w
'fm.grid' ⎕wi 'PrintWindow' ('pre.pic' ⎕wi 'hdc') y x h w
'fm.grid' ⎕wi 'PrintPage' ¯2
'fm.grid' ⎕wi 'PrintPage' 3
'pre.pic' ⎕wi 'Paint'

⍝ Draw page 4
x←x+m+w
'fm.grid' ⎕wi 'PrintWindow' ('pre.pic' ⎕wi 'hdc') y x h w
'fm.grid' ⎕wi 'PrintPage' ¯2
'fm.grid' ⎕wi 'PrintPage' 4
'pre.pic' ⎕wi 'Paint'
```

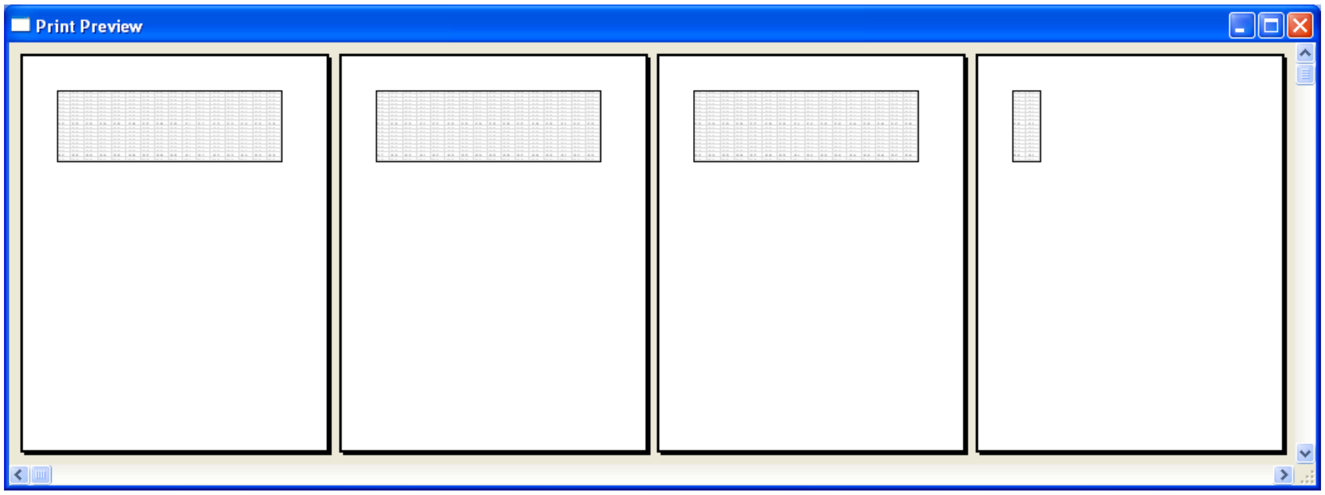
This results in the following preview image:



The next example advances to the last four pages:

```
      n←(ρ,p)-4
      x←m
      'fm.grid' ⎕wi 'PrintWindow' ('pre.pic' ⎕wi 'hdc') y x h w
      'fm.grid' ⎕wi 'PrintPage'   (n←n+1)
      'fm.grid' ⎕wi 'PrintWindow' ('pre.pic' ⎕wi 'hdc') y (x←x+m+w) h w
      'fm.grid' ⎕wi 'PrintPage'   (n←n+1)
      'fm.grid' ⎕wi 'PrintWindow' ('pre.pic' ⎕wi 'hdc') y (x←x+m+w) h w
      'fm.grid' ⎕wi 'PrintPage'   (n←n+1)
      'fm.grid' ⎕wi 'PrintWindow' ('pre.pic' ⎕wi 'hdc') y (x←x+m+w) h w
      'fm.grid' ⎕wi 'PrintPage'   (n←n+1)
      'pre.pic' ⎕wi 'Paint'
```

This results in the following preview image:



Using the XML Properties

The various properties whose names start with xXML provide means for saving the ActiveX content of the grid.

The xXML property stores every property value that the grid can save. This excludes ephemeral values such as the handle of a tracking window, but includes almost everything of each ActiveX property documented in the section above. This XML string does not include values that are stored in the built-in APL64 properties, such as the size of the grid or its placement on a form. If you want to use xXML to reconstitute a grid, you should create a grid object that is the same size; you do not have to specify any of the ActiveX settings on the grid itself, such as numbers of pages, rows, and columns.

The value of the xXML property is the basis for the value of the built-in content property (although they are saved in different representations). The content property in turn is contained in the def property of the grid or its parent form (unless you have turned off savecontent).

Note that all this information is saved by default each time the grid closes; if you don't need to save the state of the grid, you can speed up performance by setting the savecontent property to zero.

The other XML properties are substantially different. Most apply to a specified range of cells, and therefore they cannot include such information as the number of rows or columns on a page or settings of properties that apply to the entire grid. They also cannot include page defaults or column defaults, since these settings do, or could, apply to a greater range than you specify. However, what values are saved is modified by the setting of the xImplicitMode property (see below).

The xXMLRange property saves any applicable cell value, flag, or code for any cell in a specified range that has an explicit setting on that cell. If there is no property that has an explicit setting for the cell, then there is no entry for the cell in that position in the xXMLRange property when you reference it. This property is read-write; you can capture the settings from a range you specify and set a similarly sized range elsewhere on the same page, on another page, or on another instance of the grid. If you attempt to set it on a blank page, you see no values until you specify rows and columns. This property does not create the underlying structure of the grid in the same way as the xXML property.

The xXMLTable property saves any applicable cell value, flag, or code for every cell in a specified range. It contains an entry for each cell in the range, regardless of whether there is an explicit setting on the cell. This property is read-only, but you can use it to set the xXMLRange property on a similarly sized range elsewhere. Using xXMLTable to set the xXMLRange property may be useful if you are working outside of APL and doing an XSLT transformation; in this circumstance, having an entry for every cell eliminates the need for you to calculate individual coordinates of the cells you are setting. Also, using xXMLTable may provide different results when xImplicitMode is set to `1`. This may be useful or distressing, depending on your wishes (see below).

The xXMLValueRange property saves a lesser amount of information. This property saves only the values and ActiveX value types of cells in a specified range; in this case, the word "value" refers not to those properties designated "Cell value" but rather to any setting you could reference with the xValue property. This excludes cell attributes such as color properties; cell flags, such as xAutoEditSize, and cell codes, such as xLocale. It does include the data types such as Number, Boolean, Currency, and Date that determine how the content of the xValue property is interpreted. You can capture the settings of xXMLValueRange and copy or restore a similarly shaped and configured range of cells elsewhere on a grid or in another instance of the grid.

The xXMLChanges property saves even less information. This property records only the values that have been changed by the user since the property was last cleared. This allows the program to detect if the user has changed any values, and it provides a way to get just those values. The latter functionality is probably of use mostly to those working in a language such as JavaScript where the program would have to iterate through each cell; in APL, you could simply reference the xValue or xText properties over a range of cells.

XML properties are affected by the setting of the xVirtualMode property. See the “Managing Virtual Mode” section for details about virtual mode and how it affects the XML properties.

How xImplicitMode Affects XML

If you set xImplicitMode to 1, any property you reference returns an explicit value, if one is set; it also returns a column default, if one is set and there is no explicit value, or a grid default, if one is set and the others are not. If you reference xXMLRange or xXMLTable when implicit mode is on, they do not distinguish the origin of the setting; thus either property includes default settings where they exist on cells without explicit settings.

The difference between the two is that for xXMLRange to include the cell, there must be some (other) property on the cell that has an explicit setting when you reference the range. Since xXMLTable has an entry for every cell in the range, it captures any default setting on a cell for a property whether or not the cell has an explicit setting on another property.

For example, suppose you have two rows of cells on a page that has a page default setting for xColorBack, and you place text or numbers in the first row but not the second. Then, referencing xXMLRange captures the color value and places it in the first row of cells when you set the range with that XML string, along with the text or numbers that are values in each cell. Referencing xXMLTable not only does the same thing for the first row, when you use its XML string to set a new range, it colors the otherwise blank cells in the second row as well.

Specifying Missing Values

This enhancement was inspired by the old behavior of the grid which returned the conversion error value (controlled by the xConversionErrorValue property: default = ¯2147352572) when a cell containing a non-numeric value were referenced. For example, if a cell had been edited to contain a value such as "abc" or "" and you reference it with xNumber property, it would return the conversion error value.

Users requested that they be able to store error values back into the grid by specifying a numeric value matching the conversion error value. For example, if the conversion error value was 999.999 and you used the xNumber property to reference a cell containing "abc" then the value 999.999 would be returned. They wanted to be able to set the xNumber property with 999.999 and have an error value stored back into the cell in its place. However, since there is no way of knowing from the generic xConversionErrorValue what string had actually caused conversion error it is impossible to replace that string back into the grid.

For this reason it did not make sense to use the conversion error value as a signal for storing some error marker back into the grid. Instead we introduced the concept of "missing values" and introduced the xMissingValue and xMissing properties. A cell is considered to have a "missing value" when either of the following conditions is true:

1. The cell is undefined (no properties ever set for the cell or it has been deleted)

2. The cell contains an empty string value (i.e., the xText property would return an empty string value for that cell).

A cell can get into the second missing state (defined with an empty string value) in several ways:

1. Set the xText property with an empty string value ("")

2. Set the xNumber, xDate, or xCurrency property with an empty string value (''). Note that this used to give an error. However, trying to store a string that is not empty but contains all blanks is not considered a missing value and still causes an error.

3. Set the xNumber, xData, or xCurrency property with a numeric value that matches the xMissingValue state of the grid.

4. Paste an empty string value into a cell from the clipboard.

5. Edit cell value and store an empty string into cell.

Grid Properties Reference

The properties described below are those that are unique to the grid. The grid also has the standard collection of APL64 ⎕WI properties that apply to any ActiveX control.

The first element of each summary description below describes the scope to which a property applies and the nature of its content. If the scope is the grid, there is only one value at a time for the property; no matter where you are when you change it, the change applies to the entire grid. If the scope is a page, when you set the property, the value applies only to the active page (or, in one case, to a page whose index you specify as an argument). If you change pages, the property setting may have a different value without affecting the previously active page. If the scope is a cell, each cell may have any valid value; you must explicitly specify the coordinates of the cell to which you want the setting to apply. Most properties apply to one of these three: the entire grid, a page, or an individual cell. A few properties apply to a specific selection block, a row, a column, or a place (a location within in a cell).

For most of the properties described below that apply to each cell of the grid, there are actually three levels of settings. You can assign a page default that applies to every cell on a page; a column default, which overrides the page default, and applies to every cell in a specified column on the page; and an explicit setting, which overrides either or both defaults, and applies to a specific cell.

The nature of the setting is usually described as a value, a code, or a flag. A value can have a wide range of settings within the allowable domain; a value may be text or numeric in nature, depending on the property. A code usually has a limited set of possibilities, each of which implies a specific behavior or appearance. A flag can be a Boolean value implying an exclusive-or choice of behaviors (or appearance). For a cell, a flag can also be a ternary value (see below).

For properties that apply to a cell, you can assign an individual cell, a number of cells in a single row or column, or a rectangular array of cells in a number of not-necessarily-contiguous rows and columns. The default assumption for purposes of this description is that when you are assigning values, you supply vectors specifying rows and columns and an array of values that matches the shape defined by your row and column arguments.

To set a single cell, you specify a single row and a single column and a scalar numeric value or a character vector; in most cases, a one-element numeric vector also suffices as the value. You can supply a numeric scalar or a character vector to assign the same value to any array of cells. If you specify one row and multiple columns, you should supply a numeric vector or a nested vector of character vectors as the value argument. If you specify multiple rows and one column, you need to supply a one column matrix; supplying a vector does not put a different value in each specified cell. If you specify multiple rows and multiple columns, you should supply a matrix of values whose shape matches the number of rows and the number of columns specified. The behavior of the grid in tiling and extending assigned values when this is not the case is described above.

Many properties that apply to a cell have a default value of `¯1`; you can also assign this value to a property of the cell to clear its setting. Each possibility for the cell's behavior will have a different code or value setting. If you reference a property of a cell and the grid returns `¯1`, you do not know what the behavior of the cell will be relative to that property. A cell with this setting may exhibit a default behavior, but it also could inherit a setting from a page default or a column default.

Even some properties that are flags on a cell-by-cell basis can return `¯1` if the value has not been set explicitly. The descriptions of these flags use the term "ternary" (meaning three possibilities) to describe the value. When you set one of these flags, you can use a Boolean value to specify one of the alternate behaviors, or you can specify `¯1`, which clears either explicit setting and allows the cell to inherit a page or a column default. If the setting is `¯1`, there is a default behavior that matches one of the Boolean settings.

The properties are listed alphabetically, but note that the alphabetization is all uppercase before any lowercase, so the xColSize property precedes xColorBack, which precedes xCols.

xActiveCell

Description:
Page value: This property is a two-element integer vector, *Row Col*, that defines the cell that has the focus. This cell gets the next keystroke, either to initiate editing or as the starting point for a movement, such as the Tab or Enter key. When you move to a page on which there are no regular cells, referencing this property returns `¯1 ¯1`, but the corner cell cannot be edited, nor does it recognize the Tab or Enter key for movement purposes.

Syntax:
```
Value ← ⎕wi 'xActiveCell'
        ⎕wi 'xActiveCell' Value [EnsureVisible]
```

You invoke this method with zero, two or three arguments. When you invoke this property without an argument, it returns a two-element vector showing the coordinates of the currently active cell. When you set the active cell, the argument can be a simple two-element, positive integer vector, *Row Col*; it cannot be a one-row matrix nor nested. Optionally, you can include a third argument, EnsureVisible=0, that will supress scrolling the active cell into view.

Remarks:
Moving the active cell by keyboard action (Tab, Enter, arrows, PageUp, PageDown, Home, End) changes the active cell, as does setting the selection. Assigning a value to a cell under program control does not make that cell active. Starting an edit session on a cell by using the XEditStart method allows the cell to get the next keystroke, but does not make it the active cell, except that if the editing is terminated by the user pressing the Tab or Enter key, the focus moves as though the edited cell had been active. Canceling the edit session or invoking the XEditEnd method returns the focus to the previously active cell.

xAlign

Description:
Cell code: Scalar sum of two code values that define alignment for the content of a cell. The codes are:

Horizontal alignment:	`1` = left, `2` = right, or `4` = center
Vertical alignment:	`8` = top, `16` = bottom, or `32` = middle.

Syntax:
```
Value ← ⎕wi 'xAlign' Rows Cols
        ⎕wi 'xAlign' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of codes, one per cell. The default value returned for a regular cell is zero. You set the property with *Rows Cols IntegerMatrix*, where each value in the matrix is the sum of a horizontal and a vertical alignment code. If you omit a value, the setting is zero; the default behavior for normal cells is left and top. The default alignment for the caption of a Button cell or a Combo box item is also top left. The box portion of a Check box cell is left-justified but centered vertically; the caption appears at the top and to the right of the box. The arrow for an Arrow cell appears in the center of the cell regardless of the setting of this property.

The default setting for header cells is `36`; the display in a header cell is centered in the cell by default.

Remarks:
The primary purpose of this setting is to locate a cell's contents. You can also use this setting to locate an image in a cell. You assign a value of `5` for the place code (within a cell) when setting the xImage property, and the image appears according to this property's setting for that cell. Note that there are explicit xImage settings for each of the four corners, so this facility is primarily for centering images, either vertically, horizontally, or both.

xAllowSelection

Description:
Cell flag: The default value is `¯1`; the default behavior is the same as `1`, which allows the cell to be part of a selection. If you set it to zero, it forbids the cell from being selected. This has the effect of preventing the cursor from being dragged across the cell when the Shift key is pressed. It also prevents the cell from becoming the active cell, although you can initiate an editing session on the cell by invoking the XEditStart method with the cell coordinates as arguments.

Syntax:
```
Value ← ⎕wi 'xAllowSelection' Rows Cols
        ⎕wi 'xAllowSelection' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a ternary matrix of flag values, one per cell. You set it with *Rows Cols TernaryMatrix*. The value `¯1` clears either explicit value.

Remarks:
This flag has the expected effect for the default xSelectionMode or when only a single cell is allowed as the selection. When the selection mode is set to allow only a row (or rows) as the selection, disabling this property on a cell or a column does not prevent a row from being selected (for the nonce). However, the user still cannot Tab into or past the cell within the selected row, nor drag a selection to include it.

xArrayObjects

Description:
Grid flag: The default value is zero. If you set it to `1`, when you reference any cell properties, the grid wraps the array result values into an array-access ActiveX object.

Syntax:
```
Value ← ⎕wi 'xArrayObjects'
        ⎕wi 'xArrayObjects' Value
```

You reference this property without an argument; it returns a Boolean flag. You set it with a Boolean singleton.

Remarks:
The primary value of this facility is for using the grid outside of APL, for example in a Java Script application. With an array object, you can address the grid using script-array notation. For example, you can address the third row and sixth column of the grid as: `array[2][5]` or `array.2.5`

See also the description of the xConformingResultShape property.

xAutoEditStart

Description:
Grid code: This property contains one value that represents three flags specifying which user actions initiate an editing session in a normal cell. The default value is `7`, which allows any of the three. The codes are:
`1` = Pressing a character key, `2` = Pressing the `F2` key, `4` = Double-clicking the cell.

Syntax:
```
Value ← ⎕wi 'xAutoEditStart'
        ⎕wi 'xAutoEditStart' Value
```

You reference this property without an argument; it returns a single value that is the sum of the codes. You set it with a scalar integer that is the sum of the codes for the actions you want to allow.

xBorderStyle

Description:
Cell code: This property sets various style borders on any of four sides of a cell. The codes are:

	Thin	**×16 = Medium**	**×256 = Thick**	**Double** (Thick+Thin)
Left	1	16	256	257
Top	2	32	512	514
Right	4	64	1024	1032
Bottom	8	128	2048	2056

You can also assign either of these codes to a cell as a single value, but not in combination with any of the above: `4096` = **Recessed** or `8192` = **Raised**

Syntax:
```
Value ← ⎕wi 'xBorderStyle' Rows Cols
        ⎕wi 'xBorderStyle' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of codes, one per cell. You set it with *Rows Cols IntegerMatrix*, where each value is a single border style or a sum of one to four border styles, one per side of the cell. The default value, zero, specifies no borders for regular cells in the grid. The default value for headers is also zero, but there is a default header setting for each page of the grid (cell `¯2 ¯2`) of `8192`, which header cells inherit in the absence of an explicit setting, giving them the "raised" appearance.

Remarks:
Each cell can have a border on any or all four sides; however, there are few circumstances where you would want to set the border on adjacent cells on all four sides. A cell border is drawn within the area of the cell itself, so setting a border on the right side of column `1` and another on the left side of column `2` gives the appearance of a double border or one border of double thickness. You can use a border to highlight a single cell or place borders around the edges of a block of cells. If you want borders on a range of cells, it is useful to adopt a style of setting one vertical and one horizontal border on each cell, such as the right edge and the bottom. This is the technique that produces the effect of the raised or recessed cell.

xCellType

Description:
Cell code: The default value is `¯1`, which you can also assign to a cell to clear its cell type. The codes are:
`0` = normal, `1` = Combo box, `2` = Check box, `3` = Arrow, `4` = Button

Syntax:
```
Value ← ⎕wi 'xCellType' Rows Cols
        ⎕wi 'xCellType' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of codes, one per cell. You set it with *Rows Cols IntegerMatrix*, where each value is the cell type you want to specify.

Remarks:
Each cell type can be further defined: normal cells have different value types, and special cells have styles defined by the XCellTypeEx property. If no cell type is assigned, the cell behaves as a normal cell; you can assign numeric or text data, or the user can enter characters from the keyboard.

xCellTypeEx

Description:
Cell code: This property defines the sub-type of a cell; that is, it functions like a style property for cells whose basic character is defined by xCellType. For each of these cell types, the default value is ¯1; you can assign ¯1 to clear a value. The default behavior in each case is the first one listed. The values are:

Normal Cell (xCellType = 0)

The xCellTypeEx values apply primarily to text cells (xValueType = 0), but these settings have some effect on other value types.

xCellTypeEx = 0: Normal (extend right if possible)
In a normal normal cell, if the length of the text display exceeds the width of the column, the display extends over the cell to the right if that cell is empty (this applies to text and dates, but not numbers). If you are close to the edge of the control, a text display may wrap over the cell below, if it is empty (this applies to text but not to dates). Otherwise the visible display is truncated.

xCellTypeEx = 1: Constrained (do not extend)
In a normal constrained cell, the visible display of text or dates is unconditionally truncated. The entire value still exists in the xValue property and the property that corresponds to xValueType for the cell (for example xNumber); if you widen the column, more of it shows.

xCellTypeEx = 2: Multi-line (text only)
In a normal multi-line cell, text wraps at word breaks or at ⎕tcnl characters. If the length of the text exceeds the column width and row height of the cell, the visible text is truncated. This sub-type applies only to text cells.

xCellTypeEx = 4: Ellipsis
In a normal ellipsis cell, if the length of the display exceeds the width of the cell, the control displays an ellipsis at the end of the truncated visible display to indicate there is more (all value types).

xCellTypeEx = 8: Label Cell
In a normal label cell, the display comes from the xValueEx property (all value types). A user can type into the cell, but the results are not displayed. This sub-type is more like a special purpose cell type than it is like a normal cell. You may want to protect such a cell with either the xProtect property or an event handler to keep a user from thinking he can type into the cell. What the user types, or what you set in a label cell, exists in the xValue and other (for example, xCurrency) properties. If you query either, you get the hidden value, not the visible text. It is your responsibility to know which cells are labels, so you can query the xValueEx property if appropriate.

Combo Box Cell (xCellType = 1)

xCellTypeEx = 0: Drop-down button always visible; cell shows most recently selected value and button
xCellTypeEx = 1: Drop-down button visible only when cell is selected; otherwise, cell shows value only.

Check Box Cell (xCellType = 2)

xCellTypeEx = 0: Check box is flat box with **x** marker
xCellTypeEx = 1: Check box is recessed box with **x** marker
xCellTypeEx = 2: Check box is raised button with **x** marker
xCellTypeEx = 4: Check box is flat box with check mark.

Arrow Cell (xCellType = 3)

xCellTypeEx = 1: Display has single arrow (triangle) pointing to the right
xCellTypeEx = 2: Display has single arrow (triangle) pointing to the left
xCellTypeEx = 4: Display has double arrow (two triangles) pointing to the right
xCellTypeEx = 8: Display has double arrow (two triangles) pointing to the left.

Syntax:
```
Value ← ⎕wi 'xCellTypeEx' Rows Cols
        ⎕wi 'xCellTypeEx' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of sub-type codes. You set it with *Rows Cols IntegerMatrix*, where each value is the sub-type code you want for the cell.

xChanged

Description:
Cell flag: This property returns a Boolean matrix indicating whether the value of specified cells have changed due to user input. It is complimentary to the similarly named xChanges property that returns the coordinates of all cells that have changed due to user input.

Syntax:
```
Value ← ⎕wi 'xChanged' Rows Cols
        ⎕wi 'xChanged' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of boolean values. You set this property with *Rows Cols BooleanMatrix*, where each value is the changed state you want for the corresponding cell.

Remarks:
In order to get the most out of this property you should understand scatter-point indexing (see the “Using Scatter-Point Indexing” section). It allows you to access cell properties that are scattered across a grid page in a random, nonrectangular, noncontiguous pattern (such as you would expect from user editing).

The value of this property is 0 (default) if the value of the cell has not changed; 1 if the value of the cell has changed. If the user takes an action that changes the value, the system sets this property to 1. However, if the value of the cell is changed under program control, the system does not set this property to 1.

You may want to use this property to mark a cell that was changed under program control to make it appear it was changed by user input action in some cases. For example, if your have defined a custom edit window (via the xEditWindow property) then you would want to set the xChanged property after the editing interaction changes the cell value.

But carefully when setting the changed flag in a virtual grid. If you set the changed flag for a cell that has not yet been loaded this will inhibit virtual loading (see the “Managing Virtual Mode” section for details). If you want to restore a previous grid state including changed flags, you must load those cells first, before setting their changed flag.

You can clear the changed flag in all cells of the current page by setting the related xChanges property to an empty array. That is much more efficient that using this property with a possibly large set of arguments for the same purpose.

xChanges

Description:
Page value: This property returns the coordinates of all cells on the current page that have changed due to user input. It is complimentary to the similarly named xChanged property that returns a Boolean matrix indicating whether the value of specified cells have changed due to user input.

This property is described in detail and illustrated by several examples in the “Managing Virtual Mode” section.

Syntax:
```
Value ← ⎕wi 'xChanges'
        ⎕wi 'xChanges' Value
```

You reference this property without an argument; it returns a two column matrix of the coordinates of all cells that have their changed attribute set on the current page. You can set the changed attributes for all cells on the page by setting this property with the coordinates of all changed cells. Any cells not included in your argument will be cleared. Therefore you can use an empty matrix to clear all changed flag. That is useful immediately after saving changes.

Remarks:
This property is especially useful to determine what cells, if any, have changed during a user edit session. This enables you to save just the saved values and is especially important in a large virtual grid. You can also use this property to clear the changed flag on all cells by specifying an empty argument.

The changed flag is `0` if the value of a cell has not changed; `1` if the value has changed. If a user takes an action that changes the value, the system sets changed flag to `1`. However, if the value in the cell is changed under program control, the system does not set the changed flag to `1`.

If you use this property to restore a set of changed flags you must be careful to avoid to avoid setting it on any cells in a virtual grid that have not yet been loaded. Doing so will inhibit virtual loading (see the “Managing Virtual Mode” section for details). If you want to restore a previous grid state including changed flags, you must load those cells first, before setting their changed flag.

xCol

Description:
Page value: This property holds the column index of the active cell. This value is the same as xActiveCell`[2]`.

Syntax:
```
Value ← ⎕wi 'xCol'
        ⎕wi 'xCol' Value
```

You reference this property without an argument; it returns a scalar. When you set it, the argument must be an integer scalar or one-element vector specifying the column. When you assign a new value, the active cell changes within the same row.

xColSize

Description:
Column value: This property specifies the width in virtual-pixels of a column on the active page.

Syntax:
```
Value ← ⎕wi 'xColSize' Cols
        ⎕wi 'xColSize' Cols Value
```

You reference this property with a scalar or vector argument of column indices: *Cols*; it returns a vector of widths, measured in virtual-pixels. You set it with *Cols IntegerVector*, where each value is the non-negative width in virtual-pixels you want for the corresponding column.

Remarks:
You can set a default value for the page by specifying the first argument as zero. This setting does not apply to columns already on the page; if you set this property before you set xCols, it applies to the entire page.

xColorBack

Description:
Cell value: This property specifies the background color of a cell when it is not part of a selection.

Syntax:
```
Value ← ⎕wi 'xColorBack' Rows Cols
        ⎕wi 'xColorBack' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of color values. You set this property with *Rows Cols IntegerMatrix*, where each value is the background color you want for the cell when it is not selected.

Remarks:
Color values must be specified as a single integer calculated as: `256 ⊥` *blue green red*, where each of the three is a number for the respective color component ranging from zero (none, or black) to `255` (the maximum pigmentation). The maximum composite color value, white, is `16777215`. When you reference this property, if no color has been specified, it may return `¯1` or a Microsoft default determined by the host machine's control panel display settings.

xColorGrid

Description:
Grid value: This property specifies the background color of the area to the right of and below cells, when cells do not occupy all the active area of the grid. See the Remarks under xColorBack for specifying color values.

Syntax:
```
Value ← ⎕wi 'xColorGrid'
        ⎕wi 'xColorGrid' Value
```

You reference this property without an argument; it returns a scalar color value. You set it with a scalar color value.

xColorSelBack

Description:
Cell value: This property specifies the background color of a cell when it is part of a selection, but not the active cell. See the Remarks under xColorBack for specifying color values.

Syntax:
```
Value ← ⎕wi 'xColorSelBack' Rows Cols
        ⎕wi 'xColorSelBack' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of color values. You set this property with *Rows Cols IntegerMatrix*, where each value is the background color you want for the cell when it is selected but not active.

xColorSelText

Description:
Cell value: This property specifies the color of text in a cell when it is part of a selection, but not the active cell. See the Remarks under xColorBack for specifying color values.

Syntax:
```
Value ← ⎕wi 'xColorSelText' Rows Cols
        ⎕wi 'xColorSelText' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of color values. You set this property with *Rows Cols IntegerMatrix*, where each value is the color you want for text when the cell is selected but not active.

xColorText

Description:
Cell value: This property specifies the color of text in a cell when it is not part of a selection. See the Remarks under xColorBack for specifying color values.

Syntax:
```
Value ← ⎕wi 'xColorText' Rows Cols
        ⎕wi 'xColorText' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of color values. You set this property with *Rows Cols IntegerMatrix*, where each value is the color you want for text when the cell is not selected.

xCols

Description:
Page value: This property specifies the number of columns of cells on the current page. Initially, it is zero; the maximum number is 2147483647.

Syntax:
```
Value ← ⎕wi 'xCols'
        ⎕wi 'xCols' Value
```

You reference this property without an argument; it returns a scalar integer. You set it with an integer scalar or one-element vector specifying the number of columns of cells you want on the currently active page.

xConformingResultShape

Description:
Grid flag: The default value is zero. By default, when you reference a property value for a cell, the grid returns a matrix, regardless of the number of elements. If you set this flag to `1`, referencing a single cell returns a scalar and referencing values from a single row or column returns a vector.

Syntax:
```
Value ← ⎕wi 'xConformingResultShape'
        ⎕wi 'xConformingResultShape' Value
```

You reference this property without an argument; it returns a Boolean flag. You set it with a Boolean singleton.

Remarks:
The primary value of this facility is for using the grid outside of APL, for example in a Java Script application. If you set this flag to `1`, you do not have to create an array object to reference the value returned for a single cell. You may also find it useful in APL programming, particularly if you reference values from a single row or column, where the values returned are a vector and not a one-row matrix. See also the description of the xArrayObjects property.

xConversionErrorValue

Description:
Grid value: This value is returned when referencing a cell whose content is not valid for its value type, for example text in a number cell (xValueType `1`). The default value is a negative number of large magnitude.

Syntax:
```
Value ← ⎕wi 'xConversionErrorValue'
        ⎕wi 'xConversionErrorValue' Value
```

You reference this property without an argument. You set it with any APL array.

xCurrency

Description:
Cell value: This property contains numeric content associated with xValueType `4` cells. If you query the value of empty cells for which you have set xValueType to `4`, you get a conversion error indication, not a zero.

Syntax:
```
Value ← ⎕wi 'xCurrency' Rows Cols
        ⎕wi 'xCurrency' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of numeric values. You set this property with *Rows Cols NumericMatrix*, where each value is the numeric value you want formatted as currency.

Remarks:
This property holds the numeric content only. You must specify currency formatting in the xFormat and xFormatMode properties. Currency formatting is affected by the default system locale setting, which can be overridden by setting the xLocale property for the cell.

xDate

Description:
Cell value: This property contains numeric content associated with xValueType 3 cells. If you query the value of empty cells for which you have set xValueType to 3, you get a conversion error indication, not a zero.

Syntax:
```
Value ← ⎕wi 'xDate' Rows Cols
        ⎕wi 'xDate' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of numeric values. You set this property with *Rows Cols Matrix*, where each value is the numeric value you want formatted as a date and/or time or a text representation that the system can recognize as a date or time.

Remarks:
This property holds the numeric content only. You can set this property with an integer that represents a day, where day 1 is December 31, 1899, or you can assign a text vector that the grid can recognize as a date. Either type of entry is displayed in a format determined by the xFormat property or the default system settings on the host machine. Querying the cell with this property returns the numeric value.

You can also assign a fractional numeric value to represent a time. The decimal value is converted to a percent of 24 hours and displayed as hours, minutes, seconds according to the formatting for the host machine. If you assign a numeric decimal value greater than one, the grid interprets the integer portion as a date and the fractional portion as a time and displays both.

If the user types in a string that the grid recognizes as a date, the display shows what was typed, but querying the cell with xDate returns the numeric value that corresponds to the date. If the user types in a number, it is not converted to a date, and querying the cell returns the conversion error value.

In the cell, the display of values set by the program depends on the xFormat property, the xLocale property, and default system settings on the host machine. Similarly, the strings that the grid recognizes as a date are dependent on these settings. You can query the display with the xText property.

You can specify non-zero negative numbers to get dates earlier in the 19th century, but there are some anomalies around zero. If you specify zero, the grid assumes a time of midnight; if you type in zero, you get a conversion error. If you specify the xDate property as `'Dec 30, 1899'`, the grid may interpret it as zero and format it as midnight. You may be successful in setting the xText property to `'Dec 30, 1899'` but a zero returned by querying the xDate property is ambiguous. Otherwise the grid is accurate, including February 1900 (which was not a leap year).

xDragCell

Description:
Page value: This property is a two-element integer vector, *Row Col*, that defines the last cell in the selection. When you move to a page on which there are no regular cells, referencing this property returns ¯1 ¯1.

Syntax:
```
Value ← ⎕wi 'xDragCell'
        ⎕wi 'xDragCell' Value
```

You reference this property without an argument; it returns a two-element vector showing the coordinates of the endpoint of the selection after dragging. When there is no selection, the result is the same as the coordinates of the active cell. When you set xDragCell, the argument must be a simple two-element, positive integer vector, *Row Col*; it cannot be a one-row matrix nor nested; and in the range of cells in the selection.

xEditAutoSize

Description:
Cell flag: The default value is ¯1; the default behavior is the same as 1, which means, during an edit session on a cell, the edit window expands as the user types. If you set this flag to zero, the window fits the cell size and the entry scrolls.

Syntax:
```
Value ← ⎕wi 'xEditAutoSize' Rows Cols
        ⎕wi 'xEditAutoSize' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a ternary matrix of flag values, one per cell. You set it with *Rows Cols TernaryMatrix*. The value ¯1 clears either explicit value.

xEditText

Description:
Cell value: This read-only property holds the text as it would display in an edit control, if you initiate an edit session without specifying a starting character.

Syntax:
```
Value ← ⎕wi 'xEditText' Rows Cols
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a nested character matrix.

Remarks:
This property does not change while the user is editing a cell; it does, however, change if you set the value of the cell during an editing event handler. You can reference it during an event-handler to see how the session started or the value to be used if the edit session is canceled. This property can differ from the xText property particulaly when you have xFormat defined on a cell.

xEditWindow

Description:
Cell value: This property holds the window handle of a non-grid control. The default is zero; if non-zero, the specified control replaces the grid edit window for user input.

Syntax:
```
Value ← ⎕wi 'xEditWindow' Rows Cols
        ⎕wi 'xEditWindow' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of integer handles. You set it with *Rows Cols IntegerMatrix*, where each non-zero value is (*ctrl* `[]wi 'hwnd'`)

Remarks:
When you specify a special edit window, the events that occur for the editing session are the events of the defined control and not the events of the grid; for example, there will not be an onXEditKeyPress event. Nor are the XEditEnd, XEditCancel, or XValueChange events triggered. You are responsible for recognizing the end of the edit session (for example by capturing the Enter key), retrieving the value entered by the user, and assigning it to the appropriate cell. You are also responsible for sizing the control, if necessary; for example, if the input exceeds the width of the cell. You are also responsible for assigning a new active cell; if you move the focus, you may need to invoke XRedraw.

Note that a window handle will not exist across a close and open nor can it be saved in the def property for creating another instance of the grid. You must reference its value, once the grid is open, before setting this property.

xFixedCols

Description:
Page value: This property specifies the number of non-scrolling columns of cells on the current page. The default value is zero.

Syntax:
```
Value ← ⎕wi 'xFixedCols'
        ⎕wi 'xFixedCols' Value
```

You reference this property without an argument; it returns a scalar. When you set it, the argument must be an integer scalar or one-element vector specifying the number of columns of cells (starting with column `1`) that you want to remain visible when the rest of the grid scrolls horizontally.

xFixedRows

Description:
Page value: This property specifies the number of non-scrolling rows of cells on the current page. The default value is zero.

Syntax:
```
Value ← ⎕wi 'xFixedRows'
        ⎕wi 'xFixedRows' Value
```

You reference this property without an argument; it returns a scalar. When you set it, the argument must be an integer scalar or one-element vector specifying the number of rows of cells (starting with row `1`) that you want to remain visible when the rest of the grid scrolls vertically.

xFontName

Description:
Cell value: This property holds the name of the ANSI font you want to use for the display in a cell. The default value is a one-row, one-column empty character matrix. The default font is Arial; this is a default setting for each page.

Syntax:
```
Value ← ⎕wi 'xFontName' Rows Cols
        ⎕wi 'xFontName' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; if set, it returns a string. You set it with *Rows Cols NestedMatrix* where each value is an enclosed string specifying the name of a font.

xFontSize

Description:
Cell value: This property holds the size of the display font in virtual-pixels. The default setting is `¯1` to inherit the font height. Negative and positive heights have different interpretations (see below). For historical reasons, the default size is `19`; this is a default setting for each page. However, this default size is almost always too large and you should specify a smaller size for both the grid default cell, `0 0`, *and* the header default cell `¯2 ¯2` (*on each page*) before creating any rows or columns.

Syntax:
```
Value ← ⎕wi 'xFontSize' Rows Cols
        ⎕wi 'xFontSize' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a numeric matrix. You set it with *Rows Cols IntegerMatrix*, where each value is the font size in virtual-pixels you want for a cell. Use a positive value to specify the "cell height" of the font. Use a negative value (other than `¯1`) to specify the "character height" of the font. Use `¯1` to clear an explicit setting and allow the cell to inherit a default value.

xFontStyle

Description:
Cell code: This property holds a sum of codes, which, if non-zero, modifies the appearance of the display font in a cell. The default value is `¯1`, which you can also assign to a cell to clear its style code. The codes are:
`0` = plain, `1` = bold, `2` = italic, `4` = underline, `8` = strikeout.

Syntax:
```
Value ← ⎕wi 'xFontStyle' Rows Cols
        ⎕wi 'xFontStyle' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a numeric matrix. You set it with *Rows Cols IntegerMatrix*, where each value is the sum of codes you want to apply to a cell.

xFormat

Description:
Cell coding: This property holds strings that specify elaborate schemes for formatting the display in a numeric, currency, or date cell. The default value is a one-row, one-column empty character matrix.

Syntax:
```
Value ← ⎕wi 'xFormat' Rows Cols
        ⎕wi 'xFormat' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of strings. You set it with *Rows Cols CodingMatrix*, where each value specifies format rules appropriate to the cell.

Remarks:
This property is a character vector that specifies from one to four format phrases; the phrases are separated by seimicolons. The first phrase is required. For numeric and currency cells the phrases apply to:
Positive values; *Negative values*; *Zero value*; *Missing value*

You can omit either or both of the middle two phrases, but each semicolon is required if you want to specify a phrase beyond it. For date cells (which include the specification of time), the middle two phrases are not meaningful, and, therefore, ignored. You can set a first and a fourth phrase separated by three semicolons.

A format phrase can consist of selector codes, control codes, literals, and highlights. Selector codes are placeholder characters that indicate where and how a value should be displayed. Control codes have many functions including identifying literals, defining white space, acting as placeholders for decorations, and separating phrases. Literals are characters that appear in the display. Highlights are modifiers of the display, including color and font styles.

For numeric and currency cells, specifying certain control codes in the positive phrase effectively creates a meaningful negative phrase. Even without these codes, the grid generates a default negative phrase if you do not supply one. See the "Details on the Format Property" section later in this manual for details on formatting possibilities.

xFormatMode

Description:
Cell coding: This property controls how currency values are formatted for cells in a currency cell. The default value is `¯1`; the allowable values and their meaning are:
`¯1` = inherit (default), `0` = disable, `1` = enable for currency, `2` = enable for currency (with user overrides)

Syntax:
```
Value ← ⎕wi 'xFormatMode' Rows Cols
        ⎕wi 'xFormatMode' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of scalars. You set it with *Rows Cols CodingMatrix*, where each value specifies format rules appropriate to the currency cell.

Remarks:
The value 1 gives formatting according to international locale standards. The value 2 gives formatting that may have been customized by the user on their computer via the regional settings.

xGridLines

Description:
Page flag: The default value is `1`; if you set it to zero, the pale lines delineating cells disappear from the page.

Syntax:
```
Value ← ⎕wi 'xGridLines'
        ⎕wi 'xGridLines' Value
```

You reference this property without an argument; it returns a Boolean flag. You set it with a Boolean singleton.

Remarks:
If you disable this flag for a page, it provides a blank slate on which only the position of the contents of cells indicate their existence. This property appears to affect only regular grid cells, but it also affects the pale lines between header cells. For headers, you must first reset the page default setting (cell `¯2 ¯2`) for the xBorderStyle property, which makes header cells appear raised. Alternatively, you can set xBorderStyle explicitly to zero for the column headers.

xGridOptions

Description:
Page flag: This property controls several grid options. The default value is 0; the allowable values and their meanings are:
1 = keep selection rectangle displayed when grid loses focus
2 = use Windows GUI font
4 = add extra pixel to default body xRowSize (rows > 0)
8 = add extra pixel to default header xRowSize (rows < 0)
16 = suppress onXSelectionChange event handler when active cell changes without changing selection
32 = automatically clear combo cell value when Delete key is pressed
64 = suppress repl box on focus rectangle (if xReplMode not enabled)
128 = enable storing of transient properties
256 = enable the dropdown window for a combo style cell (xCellType 1) to extend (up and down directions) beyond the borders of the grid

Syntax:
```
Value ← ⎕wi 'xGridOptions'
        ⎕wi 'xGridOptions' Value
```
You reference this property without an argument; it returns an integer scalar value. You set it with an integer scalar value.

Remarks:
Setting options 2, 4, and/or 8 causes the default row heights to be recomputed. Changes in these options should only be done when the grid is empty before setting the xRows or xHeadRows properties. Otherwise the default heights will not be put into effect for the existing row or row-headers.

Setting option 128 results in the transient properties, xEditWindow and xImageList to be stored in the 'content/XML' state of the grid. In addition the static properties listed below were previously not saved are now saved: xAllowSelection, xChanged, xConformingResultShape, xImplicitMode, xMargin, xScrollMode, xUnfocusEndEdit, and xVirtualLoaded.

xHeadCols

Description:
Page value: This property specifies the number of columns of row headers on the page. The default value is 1.

Syntax:
```
Value ← ⎕wi 'xHeadCols'
        ⎕wi 'xHeadCols' Value
```
You reference this property without an argument; it returns a scalar. When you set it, the argument must be an integer scalar or one-element vector specifying the number of row-header columns you want on the current page.

xHeadRows

Description:
Page value: This property specifies the number of rows of column headers on the page. The default value is 1.

Syntax:
```
Value ← ⎕wi 'xHeadRows'
        ⎕wi 'xHeadRows'Value
```
You reference this property without an argument; it returns a scalar. When you set it, the argument must be an integer scalar or one-element vector specifying the number of column-header rows you want on the current page.

xHwnd

Description:
Grid value: This read-only property holds the window handle of the grid control. You can use it for low-level programming.

Syntax:
```
Value ← ⎕wi 'xHwnd'
```

You reference this property without an argument. It returns a scalar integer window handle.

xImage

Description:
Place value (within a cell): This property specifies an origin-one index to an image to display within a cell. You can have up to five images in a cell; the place codes, which you use as the first argument, are:
`1` = upper left corner, `2` = upper right corner, `3` = lower left corner, `4` = lower right corner, `5` = aligned (this location is taken from the setting of the xAlign property for the cell).

Syntax:
```
Value ← ⎕wi 'xImage' Places Rows Cols
        ⎕wi 'xImage' Places Rows Cols Value
```

You reference this property with three scalar or vector arguments: *Places Rows Cols*; it returns an integer array of indices in three dimensions; the number of elements of *Places* is the number of planes of the array. You set the property with *Places Rows Cols* `3`*-DIntegerArray*, where each element of the array is an integer index to the image you want to display at the specified location in the specified cell.

Remarks:
The images to which the indices apply are located in either of two containers: an APL64 Imagelist object, whose window handle (*imglist* `⎕wi 'himage'`) is contained in the xImageList property of the grid, or a bitmap file whose name is specified in the xImageFile property of the grid. You cannot have both of these specified at the same time; setting one property blanks the other. If you specify a bitmap file, you divide the bitmap into multiple images by specifying the width of a bitmap image in the xImageWidth property; if you do not specify the width, the default is to have square images.

When you specify indices, you specify all the images for one place code, for example the upper left corner, in one plane of the value array. The row and column positions for the index correspond to the row and column arguments specified for the cells; that is, the image index in the second row, third column applies to the cell whose coordinates are the second one specified in *Rows* and the third one specified in *Cols*. If you do not want an image in a specified place in a particular cell, you can use zero as the index for no image.

xImageFile

Description:
Grid value: This property holds the name of a bitmap file for images in cells. It is assumed that this bitmap has a number of images tiled together, which you will index to specify individual images. The default value is `''`.

Syntax:
```
Value ← ⎕wi 'xImageFile'
        ⎕wi 'xImageFile' Value
```

You reference this property without an argument. You set it with a character vector that specifies the bitmap file you want to use for the images.

Remarks:
The width of a bitmap image in this file is specified in the xImageWidth property. If you do not set that property, the system assumes you want square images; that is the width equals the height of the image in this file. The system divides the bitmap in this file by the width to determine the number of images you can index.

If you set this property, the system resets the xImageList property. See the Remarks under xImage for details on specifying the images.

xImageList

Description:
Grid value: This property holds the handle of an Imagelist object for images in cells. The default value is zero.

Syntax:
```
Value ← ⎕wi 'xImageList'
        ⎕wi 'xImageList' Value
```

You reference this property without an argument. You set it with (*imglist* `⎕wi 'himage'`).

Remarks:
If you set this property, the system resets the xImageFile property. See the Remarks under xImage for details on specifying the images.

xImageMask

Description:
Grid value: This property specifies the mask color of bitmaps in xImageFile for images in cells.

Syntax:
```
Value ← ⎕wi 'xImageMask'
        ⎕wi 'xImageMask' Value
```

You reference this property without an argument. You set it with a positive integer scalar or one-element vector color value; default is 16777215 (white).

xImageMetrics

Description:
Grid value: This read-only property returns values (height width count) for the Imagelist object in the cell that is associated with the xImageList property of the grid, or a bitmap file whose name is specified in the xImageFile property of the grid.

Syntax:
```
Value ← ⎕wi 'xImageMetrics'
```

You reference this property without an argument. It returns a three-element integer vector: *height width count* where *height* and *width* are in virtual-pixels. The height and width indicates the cell size of the image list (how large each image in the list is, not the overall size of all images in the list) and the count would indicate how many image cells there are in the image list. When the image list is not defined, `0 0 0` is returned.

xImageScale

Description:
Grid value: This property specifies the DPI scale of the xImageFile bitmap file (source scaling percentage).

Syntax:
```
Value ← ⎕wi 'xImageScale'
        ⎕wi 'xImageScale' Value
```

You reference this property without an argument. You set it with a positive integer scalar or one-element vector DPI scale; default is 100.

xImageWidth

Description:
Grid value: This property specifies the width of a bitmap image in the bitmap file specified in the xImageFile property to be used for images in cells. The default value is zero, which means this property is ignored.

Syntax:
```
Value ← ⎕wi 'xImageWidth'
        ⎕wi 'xImageWidth' Value
```

You reference this property without an argument. You set it with a positive integer scalar or one-element vector value in actual-pixels.

Remarks:
The system divides the bitmap specified in xImageFile by the width to determine the number of images you can index. If you do not set this property, the system assumes you want square images; that is the width equals the height of the image in the file. See the Remarks under xImage for details on specifying the images.

xImplicitMode

Description:
Grid flag: The default value is zero. By default, when you reference a property value, such as a color, for a cell when no explicit value has been set, the grid may return an empty result or a Microsoft default. If you set this flag to `1`, when you query the explicit values for a cell, the grid returns an explicit setting, if one exists; or the inherited value that applies if one or both of the page or column default settings exist. It returns the empty value, `¯1`, or Microsoft default only for a completely empty cell.

Syntax:
```
Value ← ⎕wi 'xImplicitMode'
        ⎕wi 'xImplicitMode' Value
```

You reference this property without an argument; it returns a Boolean flag. You set it with a Boolean singleton.

Remarks:
This property has a significant impact on the behavior of the XML properties that cover a range of cells.

xLocale

Description:
Cell code: This property holds an integer code that specifies to Windows a country/language pair; the setting influences date and currency displays. See the "Details on the Format Property" section later in this manual.

Syntax:
```
Value ← ⎕wi 'xLocale' Rows Cols
        ⎕wi 'xLocale' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of codes. You set it with *Rows Cols CodeMatrix*, where each value is a four- or five-digit country/language code.

Remarks:
Microsoft help says: "The available user locales are determined by the language groups you have installed." See the appendix at the end of this manual for a list of the codes you are most likely to use with languages using the Latin alphabet.

xMargin

Description:
Page value: Four-element numeric vector that specifies the top, left, bottom and right margins around the cells in virtual-pixels.

Syntax:
```
Value ← ⎕wi 'xMargin'
        ⎕wi 'xLocale' Value
```

You reference this property without an argument. It returns the values for the top, left, bottom and right margins. You set it with a `0`-, `2`-, or `4`-element vector specifying sizes as (leftRight) or (topRight leftRight) or (top left bottom right).

Remarks:
The default value for the top and bottom margin is `1` and the default value for the left and right margin is `4`.

xMergedCells

Description:
Page value: This read-only property is a `4`-column matrix, each row showing a block of cells that is joined into an anchor cell.

Syntax:
```
Value ← ⎕wi 'xMergedCells'
```

You reference this property without an argument. It returns a matrix (*Row Col RowExtent ColExtent*) of integers showing the coordinates of the upper left (anchor) cell and the number of rows and number of columns of cells merged into the block.

Remarks:
You reference a merged block by the coordinates of the anchor cell; the merged block acquires the property settings of the anchor cell. You merge cells or undo a block of merged cells by using the `XJoin` method.

xMissing

Description:
Cell flag: This property returns a Boolean matrix indicating whether the value of specified cells is identified as a missing value. This property allows you to query cells that are missing or to make cells missing. It specifies a set of rows and columns on the current page to be set or queried regarding their missing state.

Syntax:
```
Value ← ⎕wi 'xMissing' Rows Cols
        ⎕wi 'xMissing' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of Boolean values. You set this property with *Rows Cols BooleanMatrix*, where each value is the missing state you want for the corresponding cell.

Remarks:
The value of this property is `0` (default) for cells that contain non-empty string values (including values for numeric cells that are not valid numbers and would result in a conversion error); `1` for any cell that is considered missing (based on the rules outlined in section **Specifying Missing Values**).

Setting this property clears the string content of any cells you specify as one (`1`) to an empty state. It does not change cells you specify as zero (`0`).

xMissingValue

Description:
Cell value: This property returns the value specified when referencing a cell containing an empty string value using the xNumber, xDate, xCurrency or xValue properties.

Syntax:
```
Value ← ⎕wi 'xMissingValue'
        ⎕wi 'xMissingValue' Value
```

You reference this property without any arguments. You set it with a numeric scalar or empty string ("").

Remarks:
The value of this property is `0` (default), which means “not defined”. In this case, the missing value is not defined and no numeric value is recognized as “missing”. This value is not returned when you reference an empty cell with the xText property. In this case, an empty string ("") is always returned.

xNChanges

Description:
Page value: This property returns the number of all cells on the current page that have changed due to user input. This property is described in detail and illustrated by several examples in the “Managing Virtual Mode” section.

Syntax:
```
Value ← ⎕wi 'xNChanges'
```

You reference this property without an argument; it returns an integer value. Its value is reset to zero after clearing all changed flags with the xChanges property.

Remarks:
This property is especially useful to determine the number of cells, if any, have changed during a user edit session.

xNumber

Description:
Cell value: This property holds the numeric content of a non-text cell; that is, a cell with xValueType `> 0`.

Syntax:
```
Value ← ⎕wi 'xNumber' Rows Cols
        ⎕wi 'xNumber' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a numeric matrix. You set it with *Rows Cols ValueMatrix*, where each value is numeric and appropriate for the specified cell.

Remarks:
Setting a default regular cell with this property implicitly sets the cell's xValueType to `1`. If you reference this property for a cell with non-numeric content, the grid returns the conversion error value.

xPage

Description:
Grid value: This property holds the index of the currently active page. Initially it is `1`.

Syntax:
```
Value ← ⎕wi 'xPage'
        ⎕wi 'xPage' Value
```

You reference this property without an argument; it returns a positive scalar integer. You set it with a positive integer scalar or one-element vector whose value is the index to the page you want to be active.

xPageName

Description:
Page value: This property specifies the caption that appears on the tab corresponding to a page. Initially it is a one-element nested character vector with the word "Page" followed by a numeral corresponding to the order in which the page was created; for example: `1⍴⊂'Page5'`

Syntax:
```
Value ← ⎕wi 'xPageName' Pages
        ⎕wi 'xPageName' Pages Value
```

You reference this property with a scalar or vector argument of page indices: *Pages*; it returns a nested vector of character vectors, showing the captions for the pages. You set it with *Pages NestedVector*, where each value is the caption you want for the corresponding page tab.

xPageVisible

Description:
Page flag: The default value is `1`. If you set it to zero, the page and its tab are hidden, but the page's index number still applies.

Syntax:
```
Value ← ⎕wi 'xPageVisible' Pages
        ⎕wi 'xPageVisible' Pages Value
```

You reference this property with a scalar or vector argument of page indices: *Pages*; it returns a Boolean vector. You set it with *Pages BooleanVector*, where each value is zero if you want to hide the page, `1` if you want it available to the user.

xPages

Description:
Grid value: This property specifies the number of pages in the grid. Initially it is `1`.

Syntax:
```
Value ← ⎕wi 'xPages'
        ⎕wi 'xPages' Value
```

You reference this property without an argument; it returns a scalar integer. You set it with an integer scalar or one-element vector specifying the number of pages you want the grid to contain.

Remarks:
Each page is associated with a tab that displays beneath the active portion of the grid. The default captions are the word "Page" followed by a numeral sequentially assigned as pages are created.

xPrintBorder

Description:
Grid flag: This property controls whether the border is printed. The default value is `1`; if you set it to `0`, the contents of the current grid page are printed without a border when you use the XPrintPage method.

See the “Printing the Contents of a Grid Page” section for a detailed discussion of printing.

Syntax:
```
Value ← ⎕wi 'xPrintBorder'
        ⎕wi 'xPrintBorder' Value
```

You reference this property without an argument; it returns a Boolean flag. You set it with a Boolean singleton.

Remarks:
This border is drawn inside the grid's print window, nominally at the edges of the xPrintMargin property settings. However, if you use the printer's Draw method to place a rectangle, this may not match exactly. This border is smaller than the border printed by invoking XPrintPage with a negative argument, unless the grid's xPrintMargin property is all zeroes.

xPrintBorderThickness

Description:
Grid value: This property controls the thickness of "heavy weight" border elements on the Printer and in print preview. The default value is `.5`; positive values are in points; negative values are in actual-pixels; and if you set it to `0`, an error is reported.

Syntax:
```
Value ← ⎕wi 'xPrintBorderThickness'
        ⎕wi 'xPrintBorder' Value
```

You reference this property without an argument; it returns a floating point value. You set it with a floating point value.

xPrintDC

Description:
Grid value: This property associates the device context handle of a Printer object with the grid. Initially it is `0`. It has been superseded by the xPrinter property which supports association of a Printer object by name; making printing much simpler. See the “Printing the Contents of a Grid Page” section for details on using both xPrinter and xPrintDC properties.

See the “Printing the Contents of a Grid Page” section for a detailed discussion of printing.

Syntax:
```
Value ← ⎕wi 'xPrintDC'
        ⎕wi 'xPrintDC' Value
```

You reference this property without an argument; it returns a scalar integer. You set it with an integer scalar or one-element vector specifying the device context handle of a Printer object (the printer's hdc property).

Remarks:
This property creates the association between the printer object and the grid for purpose of drawing the contents of grid cells to one or more pages to be printed.

xPrintGridlineCadence

Description:
Grid value: This property controls spacing of dots in gridlines on the printer and in print preview. Initially it is `1.0`; positive values are in points; and negative values are in actual-pixels. Zero means solid line (dots only used for real printing projected print preview mode.)

Syntax:
```
Value ← ⎕wi 'xPrintGridlineCadence'
        ⎕wi 'xPrintGridlineCadence ' Value
```

You reference this property without an argument; it returns a floating point value. You set it with a floating point value.

xPrintMargin

Description:
Grid value: This property is a four-element vector that defines margins, relative to the edges of the printer page that has been associated with the grid via the xPrinter or xPrintDC property. These margins define the areas of the page upon which the grid is not allowed to draw when printing. Initially, all four values are zero when you set the printer association via the xPrintDC property. But when you use a named association with the printer via the xPrinter property, the margins are automatically set to match those of the printer's margin property (and therefore you usually don't need to set the xPrintMargin property at all). See the "Printing the Contents of a Grid Page" section for details about printing.

See the "Printing the Contents of a Grid Page" section for a detailed discussion of printing.

Syntax:
```
Value ← ⎕wi 'xPrintMargin'
        ⎕wi 'xPrintMargin' Value
```

You reference this property without an argument; it returns a four-element vector showing the margins counterclockwise from the top (top, left, bottom, right). You set it with a four-element vector specifying the size of the margins, in actual-pixels, that you want to allow.

Remarks:
The order of the elements differs from the margin property of a Printer in APL64 (which is left, right, top, bottom). The result of setting this property may not match pixel for pixel to the location you would expect if you were using the Draw method.

If you set the margins to non-zero values, the area for printing is reduced, the font size remains the same, and the amount of data is reduced. However, the XPrintInit method accounts for the reduced area, so no data are lost.

There was a bug in the previous release that caused the order of the last two elements of the result to be exchanged when you referenced the property. In other words, if you set the property with a value such as 100 200 300 400 the value returned when you referenced the property would be 100 200 400 300. This has been fixed but could create problems for anybody who had worked around the old bug.

xPrintMetrics

Description:
Grid value: This read-only property returns values for the printer object that is associated with the grid via the xPrinter or xPrintDC property. This property returns an error if the xPrinter or xPrintDC property has not been set.

See the "Printing the Contents of a Grid Page" section for a detailed discussion of printing.

Syntax:

```
Value ← ⎕wi 'xPrintMetrics' Option
```

You reference this property with a character-vector option name argument having one of the following values: `'size'`, `'usable'`, or `'inches'`.

The size option returns a two-element vector showing the full size of a printer page (for example, `6600 5100` for a normal page with a `600` dpi printer).

The usable option returns a four-element vector showing the location of the printable area of the page. These values are the same as the default top and left margins for the printer and the height and width as reported by the printer's size property.

The inches option returns the actual-pixels per logical inch ratio of the screen and printer as a two row by two column matrix. The first row specifies the vertical and horizontal pixel density for the screen and the second row returns these units for the printer. The inches metrics are helpful in calculating zoom factors used to set the xPrintZoom property.

This property also accept a list of page numbers and returns an array of row and column ranges corresponding to each page. The result is a vector if the argument is a scalar and a matrix if the argument is a vector. The columns of the vector/matrix result are:

```
 [;1] Page number (always constant as current page)
 [;2] Starting row number
 [;3] Starting column number
 [;4] Number of rows
 [;5] Number of columns
```

Remarks:
The first two elements of usable vector are not directly applicable to any calculation you probably might make because the zero origin for both the Draw method and the Print method on the printer and the xPrintWindow property of the grid are relative to the first printable pixel on the page. So these distances are already accounted for.

xPrintOptions

Description:
Grid value: This property is a scalar that can be used to enable printing of header rows and columns in the grid. The default value is `0`; the allowable values and their meanings are:
`0` = do not print column and row header
`1` = print column header
`2` = print row header
`4`[1] = preview in projected mode
`8`[2] = allow 3D checkbox printing
`16` = show fixed-rows
`32` = show fixed-cols

Syntax:

```
Value@Long ← ⎕WI 'xPrintOptions'
             ⎕WI 'xPrintOptions' Value@Long
```

You reference this property without an argument; it returns a scalar. When you set it, the argument must be an integer scalar.

Remarks:
Note 1: When normally doing print preview, printing is drawn directly to the target device (such as a bitmap in a Picture object) by scaling to that view but using standard pen thickness and color (same as for the screen). This often looks a little better on the screen than a real emulation of what printing would look like but isn't as accurate of a representation of printing. Option 4 (preview in projected mode) uses Windows projection techniques and draws as if it were really printing but lets Windows project these printed coordinates onto the target device. Projected mode looks better (as an emulation of the printed page) at some resolutions. But at other resolutions it doesn't look as good. It is, however, always a better emulation of what the printed page would look like and therefore may be a better choice for print preview. Some applications may decide to switch modes as a function of zoom factor or other considerations. Experimentation and individual preferences are the only way to decide.

Note 2: Option 8 (allow 3D checkbox printing) controls printing of 3D checkboxes. If this option is not specified (default) then 3D checkboxes are printed as 2D. In general they look better this way because 3D shading does not print very well unless you are using high-resolution photo quality paper.

xPrintScaleString

Description:
Grid value: This property is a character vector representative of the data in the grid to be printed. The default value is `''`; implies "`0000000000`" setting.

Syntax:
```
Value@String ← ⎕WI 'xPrintScaleString'
               ⎕WI 'xPrintScaleString ' Value@String
```

You reference this property without an argument; it returns a character string. When you set it, the argument must be a character string.

xPrintScaleString

Description:
Grid value: This property is a character vector representative of the data in the grid to be printed, used to help calculate the appropriate font size for printing. The default value is `''`; implies "`0000000000`" setting.

Syntax:
```
Value@String ← ⎕WI 'xPrintScaleString'
               ⎕WI 'xPrintScaleString ' Value@String
```

You reference this property without an argument; it returns a character string. When you set it, the argument must be a character string.

Remarks:
Note: When the grid prints print, it has to be decided on what font size to use. This is a tricky decision. Printed fonts are drawn at much larger sizes than screen fonts (because of the higher resolution of printers). At these larger sizes font characters have different aspect ratios than on the screen. Some are relatively wider while others are narrower. If the grid simply picks a font that has the same relative size on the printer as on the screen (scaled in the same way the grid scales the lines drawn around cells, for example) then the result is a font that is usually too wide to fit into the cells and this looks very bad. So the letter spacing has been corrected to use the same relative spacing as on the screen. This serves to make sure the printed cells show exactly the same information as on the screen without overflowing the cell boundaries. The only issue is that because of the different aspect ratios of printed characters this can sometimes lead to characters being "squished" together too closely. The xPrintScaleString property helps to address this. It allows the user to specify a representative text string that is appropriate to the kind of data being printed. The default uses the string "`0000000000`" and it appropriate if the grid contains mostly numeric data. But if the grid contains mostly text, another string may be more appropriate. The grid uses this string to compare the relative widths of printed characters versus screen characters and adjusts the font aspect ratio and size to make sure characters from this reference string will fit within the printed aspect ratio.

The workspace PTEST.ws64 (in C:\Users\johnw\AppData\Roaming\APLNowLLC\APL64\Examples folder) contains several scripts for testing the enhanced printing functionality. One of these scripts:

]demo printTest4

demonstrates various print preview modes including "projected mode". When normally doing print preview the grid draws directly to the target device (such as a bitmap in a Picture object) by scaling to that view but using standard pen thickness and color (same as for the screen). This often looks a little better on the screen than a real emulation of what printing would look like but isn't as accurate of a representation of printing. Projected mode uses Windows projection techniques and draws as if it were really printing but allows Windows project these printed coordinates onto the target device. Projected mode looks better (as an emulation of the printed page) at some resolutions. But at other resolutions it doesn't look as good. It is, however, always a better emulation of what the printed page would look like and therefore may be a better choice for print preview. Some applications may decide to switch modes as a function of zoom factor or other considerations. Experimentation and individual preferences are the only way to decide.

xPrintWindow

Description:
Grid value: This property is a five-element vector that can be used to define a device context handle upon which to draw, and a location and a size for a rectangular window into which to draw the text contents of the current page of the grid. Unless you have unusual printing requirements or are doing print preview you probably will not need to use this property.

See the “Printing the Contents of a Grid Page” section for a detailed discussion of printing.

Syntax:
```
Value ← ⎕wi 'xPrintWindow'
        ⎕wi 'xPrintWindow' Value
```

Before you can set or reference this property you must first associate a printer with the grid by setting the xPrinter or xPrintDC property.

You reference this property without an argument; it returns a five-element vector. You set it with a five-element numeric vector showing the following values:

1) The device context handle upon which you want to draw; this can be the device context handle of any printer object (the same or different than the printer you set via the xPrinter or xPrintDC property) or a window (such as a Picture object) upon which to draw the display to the screen. (The obvious use for the latter is to preview your printer output, since the current page is already displayed on the screen.)
2) The distance in actual-pixels from the top margin of the printer's page where you want the contents of the grid to be started.
3) The distance in actual-pixels from the left margin of the printer's page where you want the contents of the grid to be started.
4) The height in actual-pixels of the window into which to format the contents of the grid page.
5) The width in actual-pixels of the window into which to format the contents of the grid page.

Remarks:
Understanding the effects of setting this print window is not obvious. If you reduce the height and width of the window, you reduce the size of the printed display proportionally, and the amount of data remains the same. This allows you space to add other text above, below, or to the side of the grid cell text for explanatory purposes. Using non-zero values for the top and left arguments moves the location of the smaller window relative to the printable area on the printer's page, so that your extra space is on the side where you want it.

If you have a relatively small grid page, you can increase the size of the window to increase the size of the display. However, this is a dangerous technique, because the grid's printing mechanism is nominally formatting the contents of the grid page to the metrics of the printer you have defined. If you make the sum of the second and fourth elements (the height of the print window plus the distance from the top of the printable area) greater than the default height of the printer page, the grid will format the same number of rows to a greater size. If your grid page requires more than one page to show all the rows, this will result in lost information between your first and second vertical pages. Using the xPrintMargin property with the nominal page size does not have this effect.

xPrintZoom

Description:
Grid value: This property controls the relative scaling factor between printed pages and pages displayed on the screen.

See the “Printing the Contents of a Grid Page” section for a detailed discussion of printing.

Syntax:
```
Value ← ⎕wi 'xPrintZoom'
        ⎕wi 'xPrintZoom' Value
```

You reference this property without an argument; it returns a floating point value. You set it with a floating point singleton. Positive values are interpreted as scaling ratios (see remarks below). Negative values are codes that control “fit to page” behavior. The following control codes are defined:

`¯1`	Fit all columns of the XPrintInit selected range on one page;
`¯2`	Fit all rows of the XPrintInit selected range on one page;
`¯3`	Fit all rows and columns of the XPrintInit selected range on one page

It is easy to try squeezing too much content on a page to be readable with the options above. They usually work best when you know there is a good chance of the number of rows and/or columns being able to fit without being squeezed too small to read. On the other hand, you don’t really have to be too careful; fitting an entire large grid on one page is a great way to conserve paper while printing a solid black square.

Remarks:
By default, the grid uses a scaling factor for printing that displays one “logical inch” on the printer for every logical inch on the screen. This is defined as zoom factor 1.0 and is the default value for this property. If you specify a zoom factor of 0.5 you will get 0.5 logical inches on the printer per logical inch on the screen. In other words, you will get a smaller sized image (per cell) and that will result in more cells being printed per page. On the other hand, if you specify a zoom factor of 2.0 you will get 2.0 logical inches on the printer per logical inch on the screen. In other words, you will get a larger sized image (per cell) and that will result in fewer cells being printed per page.

You must set this value before calling the XPrintPage method and if you change this setting you must call XPrintPage method again.

xPrinter

Description:
Grid value: This property defines a named association between a Printer object and the grid. Initially this property has a value of `''`. You can set it to the name of a Printer object to begin preparations for printing or print-preview. See the “Printing the Contents of a Grid Page” section for details. It supersedes the xPrintDC property that can also be used to create a printer association.

See the “Printing the Contents of a Grid Page” section for a detailed discussion of printing.

Syntax:
```
Value ← ⎕wi 'xPrinter'
        ⎕wi 'xPrinter' Value
```

You reference this property without an argument; it returns a character vector. You set it with a character vector specifying the name of a Printer object.

Remarks:
This property creates the association between the printer object and the grid for purposes of drawing the contents of grid cells to one or more pages to be printed.

xProtect

Description:
Cell flag: This property controls whether cells are protected from user input. The default value is `¯1` (meaning to inherit); the default behavior is the same as `0`, which allows the cell to be edited, deleted, and fire mouse and keyboard events. You can set this property to protect the cell against such user input actions and associated events. If you set this flag to enable protection, an edit session cannot be initiated on the cell, either by user action or by invoking the XEditStart method. You can set values under program control.

Syntax:
```
Value ← ⎕wi 'xProtect' Rows Cols
        ⎕wi 'xProtect' Rows Cols Value
```

The table below lists the recognized protection codes:

¯1	Inherit from row, col, and grid defaults; same as `0` unless defaults are set to non-default value
0	Unprotected (allow input to the cell)
	Or any combination of the following codes (2, 4, 8, and 16 imply 1 automatically):
1	Forbid all user input except as allowed by the following codes:
2	Allow mouse events
4	Allow keyboard events
8	Allow DELETE key to function on adjacent unprotected cells in the same selection as *this* cell
16	Allow DELETE key to function on *this* cell as well as on adjacent unprotected cells

xRow

Description:
Page value: This property holds the row index of the active cell. This value is the same as xActiveCell`[1]`.

Syntax:
```
Value@Long ← ⎕wi 'xRow'
             ⎕wi 'xRow' Value@Long
```

You reference this property without an argument; it returns a scalar. When you set it, the argument must be an integer scalar or one-element vector specifying the row. When you assign a new value, the active cell changes within the same column.

xReplMode

Description:
Page value: This property enables or disables replication mode on the current page. See the “Managing Replication Mode” for a detailed discussion about using replication mode.

Syntax:
```
Value ← ⎕wi 'xReplMode'
        ⎕wi 'xReplMode' Value
```

You reference this property without an argument; it returns an integer scalar. When you set it, the argument must be an integer scalar or one-element vector specifying the replication mode.

The table below lists the recognized replication mode codes:

0	Disable replication mode (default)
	Or one of the following axis codes:
1	Allow extension of target range along 1 axis at a time
2	Allow extension of target range along 2 axes at a time
	Plus any combination of the following codes:
8	Ignore the xAllowSelection property when extending range
16	Ignore the xProtect property when extending range

xRowSize

Description:
Row value: This property specifies the height in virtual-pixels of a row on the active page.

Syntax:
```
Value ← ⎕wi 'xRowSize' Rows
        ⎕wi 'xRowSize' Rows Value
```

You reference this property with a scalar or vector argument of row indices: *Rows*; it returns a vector of heights, measured in virtual-pixels. You set it with *Rows IntegerVector*, where each value is the non-negative height in virtual-pixels you want for the corresponding row.

Remarks:
You can set a default value for the page by specifying the first argument as zero. This setting does not apply to rows already on the page; if you set this property before you set xRows, it applies to the entire page.

xRows

Description:
Page value: This property specifies the number of rows of cells on the current page. Initially, it is zero.

Syntax:
```
Value ← ⎕wi 'xRows'
        ⎕wi 'xRows' Value
```

You reference this property without an argument; it returns a scalar integer. You set it with an integer scalar or one-element vector specifying the number of rows of cells you want on the currently active page.

xScrollMode

Description:
Page code: This property specifies various codes that affect scroll bars and scrolling behavior for the page. The default value is zero; the possible values are:

1	Enable vertical scrolling hints	2	Enable horizontal scrolling hints
4	Enable vertical scrolling tracking	8	Enable horizontal scrolling tracking
16	Leave vertical scroll bar in place if not needed	32	Leave horizontal scroll bar in place if not needed
64	Hide vertical scroll bar	128	Hide horizontal scroll bar.

Syntax:
```
Value ← ⎕wi 'xScrollMode'
        ⎕wi 'xScrollMode' Value
```

You reference this property without an argument. You set it with an integer scalar or one-element vector that is the sum of the codes you want to apply on the current page.

Remarks:
Scrolling hints are tooltip-like boxes that appear when the user presses the left (or primary) mouse button on the scroll box (thumb). By default, these show "Row *n*" for vertical scrolling and "Column *n*" for horizontal scrolling where n is the topmost visible row or leftmost visible column. The numbers change as the user scrolls. You can change the text of the tip just before it appears by using the onXScrollHint event handler.

Tracking means the grid moves column by column or row by row as the user scrolls; without these settings, the grid does not reposition itself until the user releases the mouse from the scroll box. By default a scroll bar appears if needed and disappears if not needed; that is, if all the cells fit in the active area. You can have a disabled scroll bar, meaning the channel stays in place without a box to scroll if not needed. If you hide a scroll bar, it does not appear even if needed.

xSelection

Description:
Page value: This property is a `4`-column matrix, each row showing a block of selected cells.

Syntax:
```
Value ← ⎕wi 'xSelection'
        ⎕wi 'xSelection' Value
```

You reference this property without an argument. It returns a matrix (*Row Col RowExtent ColExtent*) of integers showing the coordinates of the upper left (anchor) cell and the number of rows and number of columns of cells in a block of selected cells. You set it with a valid `4`-column matrix of selection blocks. When setting this property, you do not have to specify the anchor cell; you can specify a cell at any corner of your desired selection and use positive or negative integers for the extents. Negative extents are up and to the left.

Remarks:
If a single cell is selected, the third and fourth arguments are both `1`. When multiple selections are allowed, as new selections are made, they are added at the top of the matrix, so the first block is at the bottom and the last block is at the top. Each selection has a "block number," which is an index to that block in the order of selection. Thus, the block number of the top row is equal to the number of rows in the matrix. The block number is not evident in this property, but it can be used as an argument to some properties and the XSetSelection method. The type of selection and whether multiple selections are allowed is controlled by the xSelectionMode property.

xSelectionCol

Description:
Block value: This read-only property is column index to the anchor cell of a selection block on the active page.

Syntax:
```
Value ← ⎕wi 'xSelectionCol' [Block]
```

You reference this property either without an argument or with an integer scalar or one-element *BlockNumber*. (See the description of the xSelection property for details; if you omit the argument, it defaults to `1`.) It returns a scalar column index of the anchor cell of the specified selection block.

Remarks:
This property is primarily for use with a script language; in APL, you can use the xSelection property directly.

xSelectionCols

Description:
Block value: This read-only property holds the column extent of a selection block on the current page.

Syntax:
```
Value ← ⎕wi 'xSelectionCols' [Block]
```

You reference this property either without an argument or with an integer scalar or one-element *BlockNumber*. (See the description of the xSelection property for details; if you omit the argument, it defaults to `1`.) It returns a scalar integer showing the number of columns in a specified selection block.

Remarks:
This property is primarily for use with a script language; in APL, you can use the xSelection property directly.

xSelectionCount

Description:
Page value: This read-only property shows the number of selection blocks on the current page.

Syntax:
```
Value ← ⎕wi 'xSelectionCount'
```

You reference this property either without an argument or with an integer scalar or one-element *BlockNumber*. (See the description of the xSelection property for details; if you omit the argument, it defaults to `1`.) It returns a scalar integer showing the number of blocks of selected cells.

Remarks:
This property is primarily for use with a script language; in APL, you can use the xSelection property directly.

xSelectionMode

Description:
Page flag: The default value is `1`; the allowable values and their meanings are:
`0` = current page limited to one single-cell selection at any given time
`1` = allows multiple selection blocks with either or both row and column extents greater than `1`
`2` = current page limited to one full-row selection block (that is, cells in all the columns are part of the block)
`3` = allows only full-row selections, but you can have multiple selections, and row extent can be greater than `1`.

Syntax:
```
Value ← ⎕wi 'xSelectionMode'
        ⎕wi 'xSelectionMode' Value
```

You reference this property without an argument; it returns a scalar code. You set it with an integer scalar or one-element vector.

Remarks:
Cells whose xAllowSelection flag is set to zero cannot be part of a selection when this property is `0` or `1`. Having such a cell prevents a user from dragging a selection over the row or column of that cell; it also prevents you from setting a row or column extent that includes the cell. The behavior may be anomalous in full-row modes.

xSelectionRow

Description:
Block value: This read-only property is a row index to the anchor cell of a selection block on the current page.

Syntax:
```
Value ← ⎕wi 'xSelectionRow' [Block]
```

You reference this property either without an argument or with an integer scalar or one-element *BlockNumber*. If you omit the argument, it defaults to `1`. It returns a scalar row index of the anchor cell of the specified block.

Remarks:
This property is primarily for use with a script language; in APL, you can use the xSelection property directly.

xSelectionRows

Description:
Block value: This read-only property holds the row extent of a selection block on the current page.

Syntax:
```
Value ← ⎕wi 'xSelectionRows' [Block]
```

You reference this property either without an argument or with an integer scalar or one-element *BlockNumber*. (See the description of the xSelection property for details; if you omit the argument, it defaults to `1`.) It returns a scalar integer showing the number of rows in a specified selection block.

Remarks:
This property is primarily for use with a script language; in APL, you can use the xSelection property directly.

xStayInSelection

Description:
Page flag: The default value is `1`, which means stay in the selection. If you set this property to zero, pressing Tab from the last column or Enter key from the last row of the selection moves out of the selection block.

Syntax:
```
Value ← ⎕wi 'xStayInSelection'
        ⎕wi 'xStayInSelection' Value
```

You reference this property without an argument; it returns a Boolean flag. You set it with a Boolean singleton.

Remarks:
When the user presses the Tab or Enter key, either to move or to end an edit session on a cell, the focus moves to a new active cell. If the active cell is part of a selection block, and this flag is on, the selection remains intact and the new active cell is in the same selection block. The movement is either to an adjacent cell or it wraps around to the beginning or end of the next row or column of the block, depending on which key is pressed and whether the Shift key is also down.

Arrow keys or other keys that cause movement of the focus invalidate the selection and create a new, single-cell selection that is the active cell. Note that these movement keys may have different behavior when there is an active edit session than if not. If this flag is off, the Tab and Enter keys move to the column or row next to the active cell, whether or not it is in the selection, and invalidate the selection.

xTabWidth

Description:
Grid value: This property specifies the width in virtual-pixels of the portion of the bar that holds page tabs. This space is shared with the horizontal grid scroll bar, when it exists.

Syntax:
```
Value ← ⎕wi 'xTabWidth'
        ⎕wi 'xTabWidth' Value
```

You reference this property without an argument; it returns a scalar. You set it with an integer scalar or one-element vector specifying the width in virtual-pixels for the portion of the grid allotted to page tabs.

Remarks:
If there is not enough room to display all the tabs, the pair of scroll buttons to the left of the tabs becomes active. You can set this property to zero, which hides the page tabs and prevents the user from changing pages.

xText

Description:
Cell value: This property holds the text content or the display of any normal cell. If you set a default normal cell with this property, it implicitly sets xValueType to zero.

Syntax:
```
Value ← ⎕wi 'xText' Rows Cols
        ⎕wi 'xText' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a nested character matrix. You set it with *Rows Cols CharacterMatrix*, where each value is a character string.

Remarks:
The text that displays in a cell can be very different from the value that sets it or what is returned in the xValue property. This property returns the formatted character representation, for example, of dates and times. You can also use this property to read the item selected from the list of a Combo box cell.

xTrackingWindow

Description:
Grid value: This property holds the window handle of a non-grid control. The default is zero; if non-zero, the specified control appears beside the cell whose coordinates are given to the XMoveTrackingWindow method.

Syntax:
```
Value ← ⎕wi 'xTrackingWindow'
        ⎕wi 'xTrackingWindow' Value
```

You reference this property without an argument; it returns an integer handle. You set it with (*ctrl* `⎕wi 'hwnd'`)

Remarks:
Although the window could be most anything, this facility is best used to provide information to the user rather than allowing interaction, since the tracking window is displayed beside, not over, the specified cell. Typically, you would use this facility as a super tooltip, triggering the display by defining an onXCellMouseEnter event handler and providing information to the user about the cell the mouse is over. If you invoke the method with a zero row or column argument, the window is again hidden. You can specify only one tracking window at a time.

Note that a window handle will not exist across a close and open nor can it be saved in the def property for creating another instance of the grid. You must reference its value, once the grid is open, before setting this property.

xUnfocusEndEdit

Description:
Grid flag: The default value is zero. By default, when the grid loses focus during an edit session on a cell, the editing is just suspended. When the focus returns to the grid, the user is returned to the edit session. If you set this flag to `1`, losing focus requests the end of the edit session and forces its completion. That is, the onXEditEnd event handler is triggered, but setting `⎕wres` to return to the edit session has the effect of canceling it.

Syntax:
```
Value ← ⎕wi 'xUnfocusEndEdit'
        ⎕wi 'xUnfocusEndEdit' Value
```

You reference this property without an argument; it returns a Boolean flag. You set it with a Boolean singleton.

Remarks:
If you are using the grid under JavaScript (or any other non-APL container), you should always set the xUnfocusEndEdit property to `1`. If you do not, the grid losing focus will not terminate edit mode and the grid will appear to be frozen.

xValue

Description:
Cell value: This property holds the content of any normal and some special cells. If you use this property to set a default normal cell with a character vector or a number, it implicitly sets xValueType to `0` or `1`, respectively; you can also use typed values (that is, using `'#' ⎕wi 'VT'`) to set dates, currency or Boolean cells.

Syntax:
```
Value ← ⎕wi 'xValue' Rows Cols
        ⎕wi 'xValue' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a matrix of values. You set it with *Rows Cols ValueMatrix*, where each value is appropriate to the cell specified by the arguments.

Remarks:
The value of a cell is what the grid recognizes as the value, not necessarily what is displayed; for example, in a date cell, referencing xValue returns the same value as referencing xDate. Similarly, xValue returns the same value as xNumber from numeric or currency cells, and the same string as xText from text cells. This is the primary property to use to reference Boolean cells. You can also use xValue to return a heterogeneous matrix from a range of cells of mixed value types.

If you reference this property using the @ suffix character, you can get values that include encoded ActiveX datatype information.

Example:
```
      ⎕wi 'xCellType' 1 (⍳4) 0
      ⎕wi 'xValueType' 1 (⍳4) (⍳4)
      ⎕wi 'xValue' 1 (⍳4) (⍳4)
      ⎕wi 'xValue@' 1 (⍳4)
 1      VT 11 1        VT 7 3        VT 6 4
```

xValueEx

Description:
Cell value: This property defines a supplementary display value for certain special cell types; these include the list for a Combo box, and the caption for a Check box or a label cell.

Syntax:
```
Value ← ⎕wi 'xValueEx' Rows Cols
        ⎕wi 'xValueEx' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns a character string matrix. You set it with *Rows Cols CharacterMatrix*, where each value is appropriate for the cell type.

Remarks:
A label cell is a normal text cell (xCellType and xValueType `0`) with xValueTypeEx set to `8`. Anything that is typed into the cell or any value assigned in xText (or xValue) is ignored; the content of xValueEx appears in the cell. The caption can be a simple or enclosed character string.

A Check box cell (xCellType `2`) also has a character-string caption. It appears to the right of the check box.

A Combo box cell (xCellType `1`) requires a list that is a character string with the list items separated by line feed (`⎕tclf`) or newline (`⎕tcnl`) characters. The string can be a simple or enclosed character string. The list appears when a user clicks the drop-down arrow.

xValueType

Description:
Cell code: This property defines nature of the content of a normal cell (xCellType `0`). The default value is `¯1`; you can use `¯1` to clear any setting. The valid value types are:
`0` = Text, `1` = Number, `2` = Boolean, `3` = Date, `4` = Currency

Syntax:
```
Value ← ⎕wi 'xValueType' Rows Cols
        ⎕wi 'xValueType' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns an integer matrix of codes. You set it with *Rows Cols IntegerMatrix*, where each value is the value type you want to specify.

Remarks:
You can set this property explicitly. It can also be set implicitly if you assign a value to a cell using the xValue, xNumber or xDate property. A user typing a value in a cell does not ordinarily cause the value of this property to change. However, if a user enters a value in a cell from the keyboard, and you then set, for example, a column default value of `1` (meaning you want it to be a number cell), those cells in that column that already have a keyboard-entered value become text cells (xValueType `0`) even if the characters entered are numerals. It is best to set column or page defaults before the user has a chance to edit any cells. You can set this property explicitly on a cell that already has a value; you may have to paint the cell for the effect of the change to be visible.

xView

Description:
Page value: This property is a `4`-element vector defining the block of cells (excluding fixed rows and columns) that are wholly or partially visible on the active page.

Syntax:
```
Value ← ⎕wi 'xView'
        ⎕wi 'xView' Value
```

You reference this property without an argument. It returns a matrix (*Row Col RowExtent ColExtent*) of integers showing the coordinates of the upper left visible, scrollable cell and the number of rows and number of columns of cells that are wholly or partially visible. You can set the property with two or four arguments; only the first two are meaningful: *Row Col* brings the specified cell into the visible area, positioning it at the upper left of the scrollable area of the grid (except that it does not overscroll; see the Remarks).

Remarks:
If you specify a cell near the right edge of the grid, it scrolls far enough to bring the edge of the grid into view but does not move the specified cell all the way to the left of the scrollable area. Similarly, if the bottom of the grid is visible, the specified cell may not be all the way to the top of the scrollable area.

Note that changing the view does not change the active cell; you can scroll the active cell off the screen.

xVirtualBlockSize

Description:
Cell code: This property defines the maximum block size (number of cells) in the onXVirtualLoad event.

See the “Managing Virtual Mode” section for a detailed discussion of virtual mode.

Syntax:
```
Value ← ⎕wi 'xVirtualBlockSize'
        ⎕wi 'xVirtualBlockSize' Value
```

You reference this property without any argument; it returns an integer scalar value. You set this property with an integer scalar. The default is 5000.

xVirtualLoaded

Description:
Cell code: This property indicates whether a particular cell was virtually loaded (1) or is either a normal cell or not loaded (0). You can set (1) or clear (0) this value to alter the behavior of a virtual cell. But normally, you should let this value be handled automatically by the grid and only reference its value.

See the “Managing Virtual Mode” section for a detailed discussion of virtual mode.

Syntax:
```
Value ← ⎕wi 'xVirtualLoaded' Rows Cols
        ⎕wi 'xVirtualLoaded' Rows Cols Value
```

You reference this property with two scalar or vector arguments: *Rows Cols*; it returns an integer matrix of codes. You set it with *Rows Cols IntegerMatrix*, where each value is the value type you want to specify.

xVirtualMode

Description:
Page value: This property enables or disables virtual mode for the current page. It also specifies if virtual mode should be done for print, clipboard, and xml operations and if automatic unloading of cells should be disabled.

See the “Managing Virtual Mode” section for a detailed discussion of virtual mode.

Syntax:
```
Value ← ⎕wi 'xVirtualMode'
        ⎕wi 'xVirtualMode' Value
```

You reference this property without any argument; it returns an integer scalar value. You set this property with an integer scalar or one-element vector specifying the codes listed in the table below:

`0`	Disable virtual mode
	Or any combination of the following (if code 1 is not specified, virtual mode is disabled):
`1`	Enable virtual mode
`2`	Disable virtual loading for print operations
`4`	Disable virtual loading for clipboard cut/copy operations
`8`	Disable virtual loading for clipboard paste operations
`16`	Disable virtual loading for xml operations
`32`	Do not automatically unload cells after they were loaded for print, clipboard, or xml operations

xVirtualParts

Description:
Page value: This property controls which parts of the grid take part in virtual loading. All parts of the grid (body, columns headers, row headers, corner button, column defaults, and row defaults) are enabled for virtual loading by default. But no parts are virtually loaded unless virtual mode is enabled via the xVirtualMode property.

See the “Managing Virtual Mode” section for a detailed discussion of virtual mode.

Syntax:
```
Value ← ⎕wi 'xVirtualParts'
        ⎕wi 'xVirtualParts' Value
```

You reference this property without any argument; it returns an integer scalar value. You set this property with an integer scalar or one-element vector specifying the any combination of codes listed in the table below:

1	Body cells	(`Row > 0`	and	`Col > 0`)
2	Column headers cells	(`Row < 0`	and	`Col > 0`)
4	Row headers cells	(`Col < 0`	and	`Row > 0`)
8	Corner button cell	(`Row = ¯1`	and	`Col = ¯1`)
16	Column defaults cells	(`Row = 0`	and	`Col > 0`)
32	Row defaults cells	(`Col = 0`	and	`Row > 0`)

The default value, `¯1`, selects all parts for loading. These are the same codes as used in the Part argument of the `onXVirtualLoad` event.

Note that the grid default cell (`0 0`) and header default cell (`¯2 ¯2`) are *not* virtualized.

xXML

Description:
Grid value: This property holds a representation of the entire grid as an XML string in APL.Grid format.

Syntax:
```
Value ← ⎕wi 'xXML'
        ⎕wi 'xXML' Value
```

You reference this property without an argument; it returns a character vector that is an XML string. You can set it with an XML string in APL.Grid format.

Remarks:
In order to use this and the other XML properties, you must have `MSXML 6.0` or later installed. The grid uses XML strings to store representations of all or part of the grid, for example, for the content property or when the grid object is closed but not deleted. You could use these strings to store grid values or to pass them to another spreadsheet. However, XML formats must be defined, and the APL.Grid format for the grid is not necessarily exactly compatible with any other format. You must know the XML format for the application you want to use in order to transfer values out of APL64.

You can use this property to re-create a grid, but this specifies the content of the grid only. You must create a new instance of the grid, establish its size, etc. This property, unlike xXMLRange and xXMLTable, does establish the grid and page settings, such as numbers of rows and columns and sizes of rows, columns, and fonts, etc.

xXMLChanges

Description:
Grid value: This property holds a representation, as an XML string in APL.Grid format, of changes in values made to the grid by user action since this property was last set.

Syntax:
```
Value ← ⎕wi 'xXMLChanges'
        ⎕wi 'xXMLChanges' Value
```

You reference this property without an argument; it returns a character vector that is an XML string. For the nonce, you can reset it only with an empty vector.

Remarks:
This property allows you to capture any changes made after the point when it was last reset. It should be particularly useful when working outside of APL; it obviates the need to iterate across all cells, as you would have to do, for example, when using JavaScript, but not when using APL, where you can simply reference an array.

xXMLMode

Description:
Grid value: This property controls how XML output is formatted. It allows you to balance the tradeoff between human readable output and performance. The default value of `0` gives default formatting. But you can use the sum of the following codes in do less formatting in exchange for faster performance. Code `1` suppresses line breaks and indentation. Code `2` only includes unformatted values rather than both unformatted and formatted values in the resulting XML (these are roughly the same as what you get when referencing the xEditText versus xText properties). Of the two codes, `2` is the most important for performance.

If the output you are generating is intended for consumption by another program you should definitely use code (`1`) and possibly both codes together (`3`). If the output is intended to be examined in raw format by the human eye (without an XML viewer) you should use code `0` for "prettier" formatting.

Syntax:
```
Value ← ⎕wi 'xXMLMode'
        ⎕wi 'xXMLMode' Value
```

You reference this property without an argument; it returns an integer scalar. You set this property with a integer scalar or one element vector specifying the code described above.

xXMLRange

Description:
Grid value: This property holds a representation, as an XML string in APL.Grid format, of the non-empty cells in a specified range on the active page.

Syntax:
```
Value ← ⎕wi 'xXMLRange' Selection
        ⎕wi 'xXMLRange' Selection Value
```

You reference this property with a four-element selection vector, *Row Col RowExtent ColExtent*; it returns a character vector that is an XML string. You can set it with two arguments: a two- or four-element selection vector (*Row Col RowExtent ColExtent*) and an XML string in APL.Grid format.

Remarks:
If you set this property on a blank page, you do not see anything! This property sets the attributes and values for the range that you specify, but you must create the rows and columns for the effect to be visible. Also note that you must enclose the row and column, and the extents if you specify them, into a single argument.

Example:
```
      ⎕wi 'xPage'
3
      saverng ← ⎕wi 'xXMLRange' 1 1 5 8
      ⎕wi 'XInsertPages' 3.5
      ⎕wi 'xPage' 4
      ⎕wi 'xXMLRange' (1 1 5 8) saverng      ⍝ Note the required parens
      ⎕wi 'xRows' 5
      ⎕wi 'xCols' 8
```

xXMLTable

Description:
Grid value: This read-only property holds a representation, as an XML string in APL.Grid format, of a range of cells on the active page.

Syntax:
```
Value ← ⎕wi 'xXMLTable' Selection
```

You reference this property with a four-element selection vector, *Row Col RowExtent ColExtent*; it returns a character vector that is an XML string.

Remarks:
If you use the result of referencing this property to set xXMLRange on a blank page, you do not see anything! This property sets the attributes and columns for the range that you specify, but you must create the rows and columns for the effect to be visible. Also note that you must enclose the row and column, and the extents if you specify them, into a single argument. See the example under xXMLRange above.

xXMLValueRange

Description:
Grid value: This property holds a representation, as an XML string in APL.Grid format, of the values and value types only of the non-empty cells in a specified range on the active page.

Syntax:
```
Value ← ⎕wi 'xXMLValueRange' Selection
        ⎕wi 'xXMLValueRange' Selection Value
```

You reference this property with a four-element selection vector, *Row Col RowExtent ColExtent*; it returns a character vector that is an XML string. You can set it with two arguments: a two- or four-element selection vector (*Row Col RowExtent ColExtent*) and an XML string in APL.Grid format.

Remarks:
This property sets the values and value types for the range that you specify, but you must create the rows and columns for the effect to be visible. Also note that you must enclose the row and column, and the extents if you specify them, into a single argument. See the example under xXMLRange above.

Grid Methods Reference

The methods described below are those that are unique to the grid. The grid also has the standard collection of APL64 WI methods that apply to any ActiveX control. Many of the methods described below allow you to create an effect by a program statement that would often be caused by a user action, for example, initiating an edit session on a cell. Others allow you to modify the basic parameters of a grid, for example by adding rows or columns. A few of the methods create an effect uniquely, for example, displaying a tracking window.

For most methods that apply to a cell, you can specify an individual cell, a number of cells in a single row or column, or a rectangular array of cells in a number of not-necessarily-contiguous rows and columns. The effect of the method applies to each of the specified cells individually.

If you need more information about the effect of a method, consult the description of the property that is affected or used by the method.

XAddSelection

Effect:
This method inserts a new row at the top of the xSelection matrix. The new row becomes the highest block number, which equals the number of rows in the matrix.

Syntax:
```
⎕wi 'XAddSelection' Row Col [RowExtent ColExtent]
```
You invoke this method with a two- or four-element selection vector: *Row Col RowExtent ColExtent*

Remarks:
The xSelectionMode property must be set to allow multiple selections (`1` or `3`). If the latter, and you are specifying multiple rows, you should specify column `1` as the second element of the argument.

XCanCopy

Effect:
This method tests whether the currently selected block of cells can be copied to the clipboard.

See the “Managing Clipboard Operations” section for a detailed discussion about using the clipboard.

Syntax:
```
Result ← ⎕wi 'XCanCopy'
```
You invoke this method without an argument. Its range of action is controlled by the current selection. It returns a code indicating if the operation is enabled (1) or disabled (0).

XCanCut

Effect:
This method tests whether the currently selected block of cells can be cut (copied and cleared) to the clipboard.

See the “Managing Clipboard Operations” section for a detailed discussion about using the clipboard.

Syntax:
```
Result ← ⎕wi 'XCanCut'
```

You invoke this method without an argument. Its range of action is controlled by the current selection. It returns a code indicating if the operation is enabled (1) or disabled (0).

XCanPaste

Effect:
This method tests whether the currently selected block of cells can be pasted into from the clipboard.

See the “Managing Clipboard Operations” section for a detailed discussion about using the clipboard.

Syntax:
```
Result ← ⎕wi 'XCanPaste'
```

You invoke this method without an argument. Its range of action is controlled by the current selection. It returns a code indicating if the operation is enabled (1) or disabled (0).

XCopy

Effect:
This method copies the currently selected block of cells to the clipboard.

See the “Managing Clipboard Operations” section for a detailed discussion about using the clipboard.

Syntax:
```
⎕wi 'XCopy'
```

You invoke this method without an argument. Its range of action is controlled by the current selection.

XCut

Effect:
This method cuts (copies and clears) the currently selected block of cells to the clipboard.

See the “Managing Clipboard Operations” section for a detailed discussion about using the clipboard.

Syntax:
```
⎕wi 'XCut'
```

You invoke this method without an argument. Its range of action is controlled by the current selection.

XDeleteCells

Effect:
This method, like setting values or attributes, can work at any of three levels: page default, column default, or specific cells. When you specify non-zero coordinates, this method deletes both the value, which may have been entered by a user, and any attributes, such as color and value type, which the program may have set explicitly or may have been set implicitly, from a specified cell or range of cells. This action returns a cell to a state that allows it to inherit default column or page settings.

If you specify zero for the row, you delete only the column default values, leaving any explicitly set values and any page defaults. If you use `0 0` as the arguments, you clear only the page defaults.

Syntax:

```
⎕wi 'XDeleteCells' Rows Cols
```

You invoke this method with two scalar or vector cell coordinate arguments: *Rows Cols*

Remarks:
When you delete cells using explicit coordinates, the cells still exist, and non-default settings that are part of a larger grouping, such as the size of a row or column or being part of a selection, are not affected.

XDeleteCols

Effect:
This method deletes one or more entire columns of cells. All the values are lost and cannot be recaptured.

Syntax:
```
⎕wi 'XDeleteCols' Cols
```

You invoke this method with a scalar or vector argument of integer column indices.

Remarks:
The columns remaining on the active page are re-indexed, starting with `1` for the leftmost column.

XDeletePages

Effect:
This method deletes one or more pages from the grid. All the values are lost and cannot be recaptured.

Syntax:
```
⎕wi 'XDeletePages' Pages
```

You invoke this method with a scalar or vector argument of integer page indices.

Remarks:
The pages remaining in the grid are re-indexed, starting with `1` for the page corresponding to the leftmost tab. Any captions, including default captions with a page number, are unaffected (and may now be misleading).

XDeleteRows

Effect:
This method deletes one or more entire rows of cells. All the values are lost and cannot be recaptured.

Syntax:
```
⎕wi 'XDeleteRows' Rows
```

You invoke this method with a scalar or vector argument of integer row indices.

Remarks:
The rows remaining on the active page are re-indexed, starting with `1` for the topmost row.

XEditCancel

Effect:
This method ends the current edit session, ignoring any characters that have been entered, and restores the existing value of the cell. This action is equivalent to the user pressing the Escape key during an edit session.

Syntax:
```
⎕wi 'XEditCancel'
```

You invoke this method without an argument.

Remarks:
Note that the value in the cell may have been set by an event handler during the edit session, so the existing value is not necessarily the same as the value of the cell before the edit session started.

XEditEnd

Effect:
This method attempts to end the current edit session; it invokes the onXEditEnd event handler. This action is equivalent to the user pressing the Enter or Tab key during an edit session, except the active cell does not change.

Syntax:
```
⎕wi 'XEditEnd' [Force]
```

This method takes an optional numeric argument. If you do not provide an argument, it is as though you specified zero.

Remarks:
If you specify zero as the argument, the onXEditEnd event handler controls the behavior. The possibilities include: accept the entered value or assign a new value to the cell, ending the edit session; reject the entered value, restoring the last value that was assigned to the cell (which could have occurred in another event handler during the edit session) and invoke either the onXEditComplete or onXEditCancel event handler; or, do not end the edit session and place the user back in edit mode with the entered value highlighted.

If you specify a non-zero argument, the last option is not available. The session either completes or is canceled.

XEditStart

Effect:
This method initiates an edit session, by default on the active cell. Alternatively, you can specify the cell, and if you do, you can specify an initial character with which the edit session commences.

Syntax:
```
⎕wi 'XEditStart' [Row [Col [CharCode]]]
```

You invoke this method with zero, two or three arguments. You can specify row and column indices as coordinates of the cell on which to start editing. If you specify the cell, you can also specify the ANSI character index to one character to place as the first character in the edit box of the session. If you do specify a character, any existing text is replaced; if you do not specify a character, the existing text is retained and selected.

Remarks:
This method is more likely to be useful when xAutoEditStart is disabled. Note that this method triggers the onXEditStart event handler, in which you can specify a starting character or string for the edit session.

Note: When the combo cell is the target of the XEditStart method, the list for the combo cell drops down. However, the onXEditStart event handler does not fire.

XEnsureVisible

Effect:
This method ensures that a cell is visible in the grid.

Syntax:
```
⎕wi 'XEnsureVisible' Row Col
```

Remarks:
This method does not change the selection or active cell.

XFitCol

Effect:
This method adjusts the width of a specified column to match the widest display value in any cell in the column.

Syntax:
```
⎕wi 'XFitCol' Cols
```

You invoke this method with a scalar or vector argument of integer column indices.

Remarks:
This method does **not** trigger either the onXColSizing or onXColSized event handler.

XFitRow

Effect:
This method adjusts the height of a specified row to match the tallest display value in any cell in the row.

Syntax:
```
⎕wi 'XFitRow' Rows
```

You invoke this method with a scalar or vector argument of integer row indices.

Remarks:
This method does **not** trigger either the onXRowSizing or onXRowSized event handler.

XGridVersion

Effect:
This method returns the four-part internal version number of the grid control (*major minor revision patch*).

Syntax:
```
Result ← ⎕wi 'XGridVersion'
```

You invoke this method without an argument.

Remarks:
If you request help from APLNow or send a message about the behavior of the grid, this is the version number to cite. This is not the same number that is returned by the version property.

XInsertCols

Effect:
This method inserts one or more columns on the active page of the grid and re-indexes the columns.

Syntax:
```
Result ← ⎕wi 'XInsertCols' Cols
```

You invoke this method with a scalar or vector numeric argument of column indices. The method returns the indices of the newly inserted columns after the re-indexing.

Remarks:
If the argument is a single integer, one new column is inserted before the specified column. You can also specify a floating point number to indicate the location for a new column; thus, using an argument of `4.5` inserts a column between current fourth and fifth columns. In either case, subsequent columns are re-indexed.

If the argument is a vector, new columns are inserted before each of the existing columns that are specified (or between existing columns, if decimal arguments are used). Then, all the columns are re-indexed. That is, if you specify an argument of `5 6`, the new columns will be the fifth and seventh columns, while the columns that were previously fifth and sixth will be sixth and eighth, respectively. To insert two new adjacent columns, duplicate an index in the argument.

XInsertPages

Effect:
This method inserts one or more pages in the grid and re-indexes the pages.

Syntax:
```
Result ← ⎕wi 'XInsertPages' Pages
```

You invoke this method with a scalar or vector numeric argument of page indices. The method returns the indices of the newly inserted pages after the re-indexing. See the Remarks under XInsertCols for the effect of various arguments.

Remarks:
The pages in the grid are re-indexed, starting with `1` for the page corresponding to the leftmost tab. The default captions for the new pages correspond to the order in which the pages were created. These captions, as well as default captions on previous pages, may now be misleading in that the page index may be different than the page number shown on the tab.

XInsertRows

Effect:
This method inserts one or more rows on the active page of the grid and re-indexes the rows.

Syntax:
```
Result ← ⎕wi 'XInsertRows' Rows
```

You invoke this method with a scalar or vector numeric argument of row indices. The method returns the indices of the newly inserted rows after the re-indexing.

Remarks:
See the Remarks under XInsertCols for the effect of various arguments.

XJoin

Effect:
This method creates one cell that occupies the space of a block of cells, or reverses that action.

Syntax:
```
⎕wi 'XJoin' Row Col [RowExtent ColExtent]
```

You invoke this method with a four-element selection argument vector (*Row Col RowExtent ColExtent*) to create a merged cell. You invoke it with a two-element cell coordinate argument (*Row Col*) to undo a merged cell and restore the individual cells.

Remarks:
When you join a block of cells, you may specify any corner cell and use positive (down and to the right) or negative (up and to the left) row and column extents. However, the upper left corner cell becomes the designated anchor cell in the xMergedCells property. The merged cell inherits the attributes and value of the anchor cell. You can specify any cell in the block to undo the merge.

XMoveTrackingWindow

Effect:
This method displays a tracking window adjacent to a specified cell or hides the tracking window.

Syntax:
```
⎕wi 'XMoveTrackingWindow' Row Col
```

You invoke this method with a two-element cell coordinate argument (*Row Col*).

Remarks:
A tracking window can be any Windows object that has a window handle; you specify the handle in the xTrackingWindow property. (You can specify only one tracking window at a time for the grid.) The grid displays the window adjacent to the cell whose coordinates you specify as arguments to this method. If you specify zero as either the row or column argument, it hides the window.

Typically, you would use this facility as a super tooltip, triggering the display by defining an onXCellMouseEnter event handler and providing information to the user about the cell the mouse is over. Although the window could be most anything, this facility is best used to provide information to the user rather than allowing interaction, since the tracking window is displayed beside, not over, the specified cell. If you want a special window for the user to enter input, you can define a special edit window for any particular cell.

XPaste

Effect:
This method pastes the clipboard content into the currently selected block of cells.

See the “Managing Clipboard Operations” section for a detailed discussion about using the clipboard.

Syntax:
```
⎕wi 'XPaste'
```

You invoke this method without an argument. Its range of action is controlled by the current selection.

XPrintInit

Effect:
This method prepares a range of cells on the current page to be printed on the printer specified by the xPrinter or xPrintDC property or for print preview in the window specified by the xPrintWindow property. It returns an array of page numbers representing the numbers of the pages it will take to print the selected portion of the current grid page and their order (left to right, top to bottom).

See the “Printing the Contents of a Grid Page” section for a detailed discussion of printing.

Syntax:
```
Result ← ⎕wi 'XPrintInit' [Page [Row [Col [RowExtent ColExtent]]]]
```

You invoke this method without an argument to print the entire grid. Otherwise you specify zero (0) for the Page argument and select a range of cells with the other arguments. In the future you may be able to specify an alternate page number other than the current page. But the current implementation only accepts zero (meaning the current page) as the Page argument.

Remarks:
After calling this method, you can invoke the XPrintPages method to print all pages in one batch or iteratively invoke the XPrintPage method to print pages one at a time (in which case you have several other tasks to worry about as described in the “Printing the Contents of a Grid Page” section).

XPrintPage

Effect:
This method prints the one printer page of the contents of the current grid page as formatted by the XPrintInit method.

See the “Printing the Contents of a Grid Page” section for a detailed discussion of printing.

Syntax:
```
⎕wi 'XPrintPage' PrinterPage
```

You invoke this method with an integer scalar or one-element vector specifying the page number you want to print. The value of the argument must be one of the values returned by the XPrintInit method.

You can also use the argument values zero (0) and small negative integers when doing print preview. A zero causes the page image area in the preview window to be cleared to the background color (white). Negative values do the same thing but also add a border and shadow around the edge of page. The absolute value of the argument controls the thickness of the border and shadow. Typical useful values are `¯1` or `¯2`. There are some examples of print preview in the “Printing the Contents of a Grid Page” section.

Remarks:
For this method to be successful, you must cause the printer page to open. If you place text or a figure on the page using the Print or Draw method on the printer object, this will suffice. If you are printing only the contents of the grid cells, you must invoke some action that opens the grid page, such as drawing a circle of zero radius. After printing each page you must also cause it to be ejected.

The XPrintPages method offers an easier way of printing that can handle multiple pages in a single call and takes care of opening and ejecting pages automatically.

XPrintPages

Effect:
This method prints multiple printer pages in a single call. It automatically handles opening each page before printing and ejecting each page after printing. This is substantially simpler than using the XPrintPage method. You can only use this method when the xPrinter property is used to specify the associated printer (it does not work when the printer is specified via the xPrintDC property).

You must call the XPrintInit method to prepare a set of cells for printing before calling this method.

See the “Printing the Contents of a Grid Page” section for a detailed discussion of printing.

Syntax:
```
⎕wi 'XPrintPages' PrinterPages
```

You invoke this method with an integer scalar, vector, or matrix specifying a set of printer page numbers to be printed. The elements of this argument must each be one of the printer page numbers result values returned by the XPrintInit method.

XRedraw

Effect:
This method causes Windows to paint the entire visible grid, a cell, or a specified block of cells.

Syntax:
```
⎕wi 'XRedraw' [Row [Col [RowExtent ColExtent]]]
```

You invoke this method with zero, two or four arguments. The arguments can be a two-element cell coordinate (*Row Col*) or a four-element selection vector (*Row Col RowExtent ColExtent*).

Remarks:
If you specify the method without an argument, Windows paints the entire visible grid. Otherwise, it paints the portion you specify. You must invoke this method after many program actions that affect the content or appearance of the grid before the effect of the program action is visible to the user.

It is generally more efficient and causes less flicker to use the XRedraw method with specific cell arguments than to invoke the Paint method or to omit the arguments. However, both techniques have a similar net effect.

XSetSelection

Effect:
This method defines one row of the xSelection matrix.

Syntax:
```
⎕wi 'XSetSelection' Row Col [RowExtent ColExtent [Block]]
```

You invoke this method with two, four, or five arguments. The arguments can be a two-element cell coordinate (*Row Col*), a four-element selection vector (*Row Col RowExtent ColExtent*) or a selection vector with an additional block argument.

Remarks:
The primary value of this method is for using the grid outside of APL, for example in a Java Script application, where creating an array argument to the xSelection property is not possible. APL users should generally set the property instead. The behavior of this method depends on the setting of the xSelectionMode property.

If xSelectionMode is set to zero, you may have only a single cell selected. You can invoke this method with two arguments or with the coordinates of any cell and the row and column extents both set to `1`.

If xSelectionMode is set to `1`, multiple rectangular blocks (including single cells) may be selected. You can select a single cell or specify a single block by specifying the coordinates of the upper left corner and the number of rows and number of columns in the block. If you have a multi-row selection matrix, you cannot replace a row, but you can add a row to the matrix by specifying the fifth argument as the next block number (one more than the number of existing rows in the matrix).

If xSelectionMode is set to `2`, you may have only a single full row selected. You can invoke this method with two arguments of a cell (the row containing that cell is selected) or with the coordinates of any cell and the row extent set to `1` with any value for the column extent that does not extend past the end of the grid from the specified cell (the full row is selected).

If xSelectionMode is set to 3, multiple full rows may be selected. You may select any single row just as when it is set to 2. You may select multiple adjacent rows in a single selection by specifying any cell and a row extent greater than one (with a valid column extent). If you have a multi-row selection matrix, you cannot replace a row, but you can add a row to the matrix by specifying the fifth argument as the next block number (one more than the number of existing rows in the matrix).

XTextSize

Effect:
This method returns a two element vector indicating the height and width (in virtual-pixels) required to display the text specified by the argument when formatted using the font defined for the specified row and column of the grid. This can be useful in deciding the proper initial size for the rows and columns of your grid.

Syntax:
```
Result ← ⎕wi 'XTextSize' Text [Row Col]
```

The Text argument is a character scalar or vector specifying a string of characters you want to “measure”. It can contain embedded newline character to specify line breaks in a multiple line display.

The Row Col arguments specify the row and column of the grid from which to get the font information used to compute the size of the text string. If you omit these arguments they default to the grid default cell (0 0)

The Result is a two element vector reporting the height and width (in virtual-pixels) required to display the entire string.

XVirtualLoad

Effect:
This method preloads virtual cells from the specified range on the current page of the grid (default is all cells). It scans all cells in the specified range to find cells that have their virtual loaded attribute set and their changed attribute clear. It preloads any such cells that it find during the time limit it is given to work with.

See the “Managing Virtual Mode” section for a detailed discussion of virtual mode.

Syntax:
```
Result ← ⎕wi 'XVirtualLoad' [Timeout [ResumeRow ResumeCol ...
                            [Row Col [RowExtent ColExtent]]]]
```

You invoke this using zero to seven arguments described:

The *Timeout* argument specified the timeout limit (in milliseconds) for the operation to run before returning. If you specify zero (0), the operation runs to completion, scanning and preloading all cells in the specified range. Otherwise it controls the maximum time the method will run before returning. If the method returns before it completes scanning of all cells, the result indicates where to resume on the next call. This makes it easy to do background processing that preloads a few thousand cells at a time, usually hooked to a Timer object.

The *ResumeRow* and *ResumeCol* arguments are used to resume a previous call that timed out before completing all its work. These values come from the first and second elements of the result value of the previous call. The values must specify a cell that lies somewhere in the range specified by the *Row Col RowExtent ColExtent* argument. The values (`0 0`) mean processing should start from the top-left corner of the specified scan range.

The *Row Col RowExtent ColExtent* arguments specify a range of cells to be scanned for preloading. If you omit these arguments they default to the entire grid page. If you omit only the last two argument they default to an extent of 1 row by 1 column (a single cell).

The first two elements of the *Result* indicate the resume row and column. If the method finishes scanning all cells before returning, these values will both be zero (`0`). Otherwise the values indicate the starting row and column position where the next scan should be resumed. You can pass these values as the *ResumeRow* and *ResumeCol* arguments on the next invocation of this method.

The third element of the result specifies the number of cells that were preloaded during the call. You can use this to determine to you should stop scanning. If you previous complete scan results in fewer than some threshold number of cells you can increase you timer interval between scans. Result[1]

Remarks:
The *Timeout* should either be zero (`0`) to clean the whole grid at once without interruption or a small value such as 50 milliseconds when running on a Timer to keep your user interface responsive. You might set up a timer to do the "next" block every few seconds until ResumeRow and ResumeCol return `0 0` indicating the while grid has been finished sweeping. Then you might restart from the top some time later (perhaps based on a counter of how many virtual cells you have loaded in your onXVirtualLoad event handler). If you use this loaded cell counting technique remember to only count cells loaded for screen display (onXVirtualLoad reason=`1`). Cells loaded for other reasons are automatically unloaded unless you specify the “no unload” option in the xVirtualMode property.

XVirtualUnload

Effect:
This method unload cells from the specified range on the current page of the grid (default is all cells). It scans all cells in the specified range to find cells that have their virtual loaded attribute set and their changed attribute clear. It unloads any such cells that it find during the time limit it is given to work with.

It is suited for running in the background on a Timer to keep the number of loaded cells to a minimum. This can be useful if users tend to browse the grid for long periods creating many “display” cells that are not unloaded automatically. This method provides an easy way to clean up such cells.

See the “Managing Virtual Mode” section for a detailed discussion of virtual mode.

Syntax:
```
Result ← ⎕wi 'XVirtualUnload' [Timeout [ResumeRow ResumeCol ...
                              [Row Col [RowExtent ColExtent]]]]
```

You invoke this using zero to seven arguments described:

The *Timeout* argument specified the timeout limit (in milliseconds) for the operation to run before returning. If you specify zero (`0`), the operation runs to completion, scanning and unloading all cells in the specified range. Otherwise it controls the maximum time the method will run before returning. If the method returns before it completes scanning of all cells, the result indicates where to resume on the next call. This makes it easy to do background processing that unloads a few thousand cells at a time, usually hooked to a Timer object.

The *ResumeRow* and *ResumeCol* arguments are used to resume a previous call that timed out before completing all its work. These values come from the first and second elements of the result value of the previous call. The values must specify a cell that lies somewhere in the range specified by the *Row Col RowExtent ColExtent* argument. The values (`0 0`) mean processing should start from the top-left corner of the specified scan range.

The *Row Col RowExtent ColExtent* arguments specify a range of cells to be scanned for unloading. If you omit these arguments they default to the entire grid page. If you omit only the last two argument they default to an extent of 1 row by 1 column (a single cell).

The first two elements of the *Result* indicate the resume row and column. If the method finishes scanning all cells before returning, these values will both be zero (`0`). Otherwise the values indicate the starting row and column position where the next scan should be resumed. You can pass these values as the *ResumeRow* and *ResumeCol* arguments on the next invocation of this method.

The third element of the result specifies the number of cells that were unloaded during the call. You can use this to determine to you should stop scanning. If you previous complete scan results in fewer than some threshold number of cells you can increase you timer interval between scans. Result[1]

Remarks:
The *Timeout* should either be zero (0) to clean the whole grid at once without interruption or a small value such as 50 milliseconds when running on a Timer to keep your user interface responsive. You might set up a timer to do the "next" block every few seconds until ResumeRow and ResumeCol return `0 0` indicating the while grid has been finished sweeping. Then you might restart from the top some time later (perhaps based on a counter of how many virtual cells you have loaded in your onXVirtualLoad event handler). If you use this loaded cell counting technique remember to only count cells loaded for screen display (onXVirtualLoad reason=`1`). Cells loaded for other reasons are automatically unloaded unless you specify the “no unload” option in the xVirtualMode property.

XWhereIs

Effect:
This method returns the location and size in virtual-pixels of specified cell or a range of cells relative to grid's client area.

Syntax:
```
Result ← ⎕wi 'XWhereIs' Row Col [RowExtent ColExtent]
```

You invoke this method with a two-element cell coordinate argument (*Row Col*) or a four-element selection vector (*Row Col RowExtent ColExtent*).

Remarks:
This method returns zero if the anchor cell you specify is not visible.

Grid Event Handler Reference

The event handler properties described below are those that are unique to the grid. The grid also has the standard collection of APL64 WI event handlers that apply to any ActiveX control. The event handlers described below allow you to react to a user manipulating the grid. As with other events in APL64, the event-handler name is the name of the event prefixed with "on"; the events property and the `⎕wevent` system variable show the event name. If you define an event handler, you must specify the name with the prefix.

It is relatively more important when using the grid to be aware of the shape of the system variables, `⎕warg` and `⎕wres`. For those cases when you can change the effect during an event handler by setting `⎕wres`, you should set the proper element, for example `⎕wres[2]`, rather than just trying to replace the value of `⎕wres`.

onXCanColSize

Trigger:
The cursor is in position to resize a column; that is, it is over the border between two column header cells; if not suppressed in the handler, the arrow transforms into a two-headed, east-west arrow.

Behavior:
Under default circumstances, if the user presses the primary mouse button and drags the border, the column to the left of the border being dragged changes size, becoming smaller if the user drags left and larger if to the right.

Syntax:
```
⎕wevent ←→ 'XCanColSize'
⎕warg   ←→ Col Enable
⎕wres[2] ←     Enable
```

`⎕warg` is a two-element vector showing `[1]` the index to the column that could be resized and `[2]` a Boolean flag that by default is `1`. You can set `⎕wres[2]` to zero to prevent resizing.

Remarks:
This event handler can prevent the user from being able to grab the border. There are also two other event handlers, onXColSizing and onXColSized that you can use to override or limit the change.

When dragging the edge of the corner cell, the column argument in `⎕warg` is `¯99999`.

onXCanCopy

Trigger:
This event is triggered when the user presses Ctrl+C or your application invokes the XCanCopy method. It allows you to override the default grid behavior for determining whether the currently selected block of cells should be enabled for coping to the clipboard.

Behavior:
See the “Managing Clipboard Operations” section for a detailed discussion about using the clipboard.

Syntax:
```
⎕wevent ←→ 'XCanCopy'
⎕warg   ←→ Enable Row Col RowExtent ColExtent
⎕wres[1] ← Enable
```

The argument specifies which cells are being considered for this clipboard action. You can set the Enable result (`⎕wres[1]`) to allow (`1`) or disallow (`0`) the operation.

onXCanCut

Trigger:
This event is triggered when the user presses Ctrl+X or your application invokes the XCanCut method. It allows you to override the default grid behavior for determining whether the currently selected block of cells should be enabled for cutting (copying and clearing) to the clipboard.

Behavior:
See the “Managing Clipboard Operations” section for a detailed discussion about using the clipboard.

Syntax:
```
⎕wevent ←→ 'XCanCut'
⎕warg   ←→ Enable Row Col RowExtent ColExtent
⎕wres[1] ← Enable
```
The argument specifies which cells are being considered for this clipboard action. You can set the Enable result (`⎕wres[1]`) to allow (`1`) or disallow (`0`) the operation.

onXCanPaste

Trigger:
This event is triggered when the user presses Ctrl+V or your application invokes the XCanPaste method. It allows you to override the default grid behavior for determining whether the currently selected block of cells can be pasted into from the current clipboard content.

Behavior:
See the “Managing Clipboard Operations” section for a detailed discussion about using the clipboard.

Syntax:
```
⎕wevent ←→ 'XCanPaste'
⎕warg   ←→ Enable Row Col RowExtent ColExtent
⎕wres[1] ← Enable
```
The argument specifies which cells are being considered for this clipboard action. You can set the Enable result (`⎕wres[1]`) to allow (`1`) or disallow (`0`) the operation.

onXCanRowSize

Trigger:
The cursor is in position to resize a row; that is, it is over the border between two row-header cells; if not suppressed in the handler, the arrow transforms into a two-headed, north-south arrow.

Behavior:
Under default circumstances, if the user presses the primary mouse button and drags the border, the row above the border being dragged changes size, becoming smaller if the user drags up and larger if down.

Syntax:
```
⎕wevent ←→ 'XCanRowSize'
⎕warg   ←→ Row Enable
⎕wres[2] ←     Enable
```
`⎕warg` is a two-element vector showing `[1]` the index to the row that could be resized and `[2]` a Boolean flag that by default is `1`. You can set `⎕wres[2]` to zero to prevent resizing.

Remarks:
This event handler can prevent the user from being able to grab the border. There are also two other event handlers, onXRowSizing and onXRowSized that you can use to override or limit the change.

When dragging the edge of the corner cell, the row argument in `⎕warg` is `¯99999`.

onXCellChange

Trigger:
The active cell has changed.

Behavior:
There is always one cell that is the active cell; this is the cell that would, for example, receive the next keystroke. Many user keyboard actions change the active cell, as does clicking a new cell with the mouse. In addition, you can set the xActiveCell property programmatically. Any of these actions triggers this event handler.

Syntax:
```
⎕wevent ←→ 'XCellChange'
⎕warg   ←→ NewRow NewCol OldRow OldCol
```

`⎕warg` is a four-element vector showing `[1 2]` the coordinates (*Row Col*) of the newly active cell and `[3 4]` the coordinates (*Row Col*) of the previously active cell.

Remarks:
This event is informational; when it is triggered, the active cell has already changed. You can use it to prepare the new cell or process the content of the previously active cell. You should not attempt to control the movement of the focus in this event handler; there are other ways of preventing cells from becoming active.

onXCellClick

Trigger:
The mouse was clicked (down/up) in a cell.

Behavior:
If the user double-clicks, by default it initiates an edit session in the cell; subsequent mouse actions in the same cell are not recognized until the edit session terminates.

Syntax:
```
⎕wevent ←→ 'XCellClick'
⎕warg   ←→ Row Col Button Buttons Keys Count
```

`⎕warg` is a six-element integer vector showing `[1 2]` the coordinates (*Row Col*) of the affected cell;
`[3]` the mouse button that caused the click: `1`=left, `2`=right;
`[4]` any mouse buttons that were pressed when the click occurred: sum of `1`=left, `2`=right, `4`=center;
`[5]` any modifier keys that were down when the click occurred: sum of `1`=Shift, `2`=Ctrl, `4`=Alt;
`[6]` the click count: `0`=normal (single) click, `2`=double-click.

Remarks:
If the click is not on the active cell, the first event is XCellChange. Subsequent events, XCellMouseDown, XCellMouseMove, and XCellMouseUp, all precede the first XCellClick. If the user pauses and clicks again, the order is XCellMouseMove, XCellMouseDown, XCellMouseMove, and XCellClick. However, if there is a double click, only a second XCellMouseMove separates the two XCellClick events.

onXCellDropDown

Trigger:
In a Combo box cell (xCellType 1), the list is poised to drop down as the result of a user action.

Behavior:
Once the list drops down, the user must take another action to select an item; it is possible to arrow down the list highlighting various items as you go, but still not make a selection.

Syntax:
```
⎕wevent ←→ 'XCellDropDown'
⎕warg   ←→ Row Col
```

⎕warg is a two-element vector showing the coordinates (*Row   Col*) of the Combo box cell.

Remarks:
This event occurs before the XCellMouseDown event that is triggered by the same user mouse action, but after the XCellKeyDown event, if the user triggered them by pressing Ctrl+down-arrow. The obvious uses for this handler are to populate the list before it drops down or to check the current value to be able to determine whether the user changes it; that is, if an item is already selected and the user selects the same item, it does trigger the onXValueChange event handler. You can set the list using the xValueEx property.

onXCellDropWindow

Trigger:
In a Combo box cell (xCellType 1), the list is poised to drop down as the result of a user action.

Behavior:
The user can take other actions on the drop down list before it is displayed.

Syntax:
```
⎕wevent ←→ 'XCellDropWindow'
⎕warg   ←→ Row ColHwnd
```

⎕warg is a three-element vector showing the coordinates (*Row   Col*) and window handle of the drop down list for the Combo box cell.

Remarks:
This event fires immediately before displaying the dropdown window. This is similar to the onXCellDropDown event except it fires immediately before the dropdown window and includes the window handle that is about to be displayed. The handler can reposition the window during this event via ⎕WCALL 'MoveWindow'. Or it can suppress the display of the window by calling ⎕WCALL 'DestroyWindow' on the Hwnd argument (⎕WARG[3]). If the window handle is destroyed, the cell does not try displaying a drop down list. But the application is free to display a surrogate dropdown list if it chooses to do so, in order to get more control over what is displayed.

onXCellKeyDown

Trigger:
A key was pressed while a cell is active but not in an edit session.

Behavior:
If the user presses a character key in a normal cell, by default it initiates an edit session in the cell; subsequent keystroke actions are recognized by the temporary edit control and not the cell until the edit session terminates.

Syntax:
```
⎕wevent ←→ 'XCellKeyDown'
⎕warg   ←→ Key Keys Row Col
```

`⎕warg` is a four-element vector showing `[1]` the virtual key code of the key that was pressed;
`[2]` any modifier keys that were down when the click occurred: sum of `1`=Shift, `2`=Ctrl, `4`=Alt;
`[3 4]` the coordinates (*Row Col*) of the cell where the keystroke occurred.

Remarks:
Note that there is no XCellKeyUp event; when a character key intiates an edit session, releasing the key triggers onXEditKeyUp. The virtual key code for the `F2` key, which also typically initiates an edit session, is `113`.

onXCellMouseDown

Trigger:
The left or right mouse button was pressed in a regular cell that was active but not in an edit session or in a header cell.

Behavior:
Pressing a mouse button while the cursor is over a cell that is not the active cell activates that cell; the XCellChange event occurs before XCellMouseDown, so the cell is already active when this event occurs.

Syntax:
```
⎕wevent ←→ 'XCellMouseDown'
⎕warg   ←→ Row Col Button Buttons Keys
```

`⎕warg` is a five-element vector showing `[1 2]` the coordinates (*Row Col*) of the affected cell;
`[3]` the mouse button that came down: `1`=left, `2`=right;
`[4]` any mouse buttons that were pressed when the event occurred: sum of `1`=left, `2`=right, `4`=center;
`[5]` any modifier keys that were down when the event occurred: sum of `1`=Shift, `2`=Ctrl, `4`=Alt.

Remarks:
You can receive two consecutive XCellMouseDown events if the user presses both left and right mouse buttons.

onXCellMouseEnter

Trigger:
The mouse crossed a boundary of a regular cell (this includes "crossing" from a regular cell to an edit session and returning from an edit window to the grid).

Behavior:
The movement of the mouse over the grid generates events regardless of whether a mouse button is pressed or not. Each time the mouse crosses a regular cell boundary, it triggers an XCellMouseEnter event. If the mouse was already on a regular cell, the XCellMouseEnter event is preceded by an XCellMouseLeave event. If the move is onto a regular cell the XCellMouseEnter event is followed by an XCellMouseMove event.

Syntax:
```
⎕wevent ←→ 'XCellMouseEnter'
⎕warg   ←→ Row Col RowLeave ColLeave
```

`⎕warg` is a four-element vector showing `[1 2]` the coordinates (*Row Col*) of the cell where the mouse now is and `[3 4]` the coordinates of the cell where the mouse was. Either of these two pairs can be `0 0` if the boundary line that was crossed is at the edge of the body of the grid (that is, the portion occupied by regular, not header, cells).

Remarks:
If the mouse is moved off the body of the grid, the first pair of coordinates in `⎕warg` for onXCellMouseEnter is `0 0`. If you move onto a cell from a row or column header or from off an edge of the grid, the second pair of coordinates is `0 0`.

onXCellMouseLeave

Trigger:
The mouse crossed a boundary out of a regular cell (this includes "leaving" a regular cell to begin an edit session).

Behavior:
The movement of the mouse over the grid generates events regardless of whether a mouse button is pressed or not. Each time the mouse moves out of a regular cell, it triggers an XCellMouseLeave event. An XCellMouseLeave event is always followed by an XCellMouseEnter event.

Syntax:
```
⎕wevent ←→ 'XCellMouseLeave'
⎕warg   ←→ Row Col RowEnter ColEnter
```

`⎕warg` is a four-element vector showing `[1 2]` the coordinates (*Row Col*) of the cell the mouse left and `[3 4]` the coordinates of the cell where the mouse now is. If the mouse is moved off the body of the grid, the second pair of coordinates is `0 0`.

Remarks:
There is not an XCellMouseLeave event for movement from off the grid, so if the user moves the mouse onto and off the grid, there will be one more XCellMouseEnter event than XCellMouseLeave.

onXCellMouseMove

Trigger:
This event follows other events; it is not triggered by the actual movement of the mouse:
it follows the XCellClick event when this event occurs for a single click in a regular cell;
it follows the XCellMouseEnter event when the mouse moves into a regular cell;
it follows the XCellMouseDown event in a regular cell, if the mouse button is not immediately released;
it follows the XCellMouseUp event in a regular cell, if there is no XCellClick event (which implies that the mouse was moved to a different cell after it was pressed, even if it was returned to the same cell).

Behavior:
Although this event can occur frequently, it occurs singly; moving the mouse does not generate a string of these events as it does MouseMove events over a typical built-in control.

Syntax:
```
⎕wevent ←→ 'XCellMouseMove'
⎕warg   ←→ Row Col Button Buttons Keys
```

`⎕warg` is a five-element vector showing `[1 2]` the coordinates (*Row Col*) of the affected cell;
`[3]` zero, since a mouse button does not actually trigger this event;
`[4]` any mouse buttons that were pressed when the event occurred: sum of `1`=left, `2`=right, `4`=center;
`[5]` any modifier keys that were down when the event occurred: sum of `1`=Shift, `2`=Ctrl, `4`=Alt.

Remarks:
If the focus left the grid during an edit session, and the first action upon returning is to click a different cell, the order of the events is slightly different. The XCellMouseDown event can occur before the XCellMouseEnter event, and the XCellMouseMove event does not follow XCellMouseDown, even if the button is held down; XCellMouseMove follows XCellMouseEnter, regardless.

onXCellMouseUp

Trigger:
The left or right mouse button was released while over any cell (header or regular) or open grid area.

Behavior:
You can see XCellMouseUp events even when an XCellMouseDown event did not occur. It is even possible to get an XCellMouseUp event over a cell while another cell is in an edit session, without breaking the edit session.

Syntax:
```
⎕wevent ←→ 'XCellMouseUp'
⎕warg   ←→ Row Col Button Buttons Keys
```

`⎕warg` is a five-element vector showing `[1 2]` the coordinates (*Row Col*) of the affected cell;
`[3]` the mouse button that was released: `1`=left, `2`=right;
`[4]` any mouse buttons that were pressed when the event occurred: sum of `1`=left, `2`=right, `4`=center;
`[5]` any modifier keys that were down when the event occurred: sum of `1`=Shift, `2`=Ctrl, `4`=Alt.

Remarks:
You can receive two consecutive XCellMouseUp events if the user releases both left and right mouse buttons.

onXColSized

Trigger:
A column has been resized; that is, the right border of a column header cell was grabbed by the user and then the mouse was released.

Behavior:
Under default circumstances, if the user drags the border of a column header cell, the column to the left of the border being dragged changes size, becoming smaller if the user drags left and larger if to the right.

Syntax:
```
⎕wevent ←→ 'XColSized'
⎕warg   ←→ Col Size
⎕wres[2] ←     Size
```

`⎕warg` is a two-element vector showing `[1]` the index to the column that was resized and `[2]` the current width in virtual-pixels. You can set `⎕wres[2]` to a desired virtual-pixel width to override or limit the change.

Remarks:
This event occurs only once when the mouse button is released; if the user grabbed the border but did not actually move the mouse, this event still occurs, even if no XColSizing events occurred.

When dragging the edge of the corner cell, the column argument in `⎕warg` is `¯99999`.

onXColSizing

Trigger:
A column is being resized; that is, the right border of a column header cell is being dragged by the user.

Behavior:
Under default circumstances, if the user drags the border of a column header cell, the column to the left of the border being dragged changes size, becoming smaller if the user drags left and larger if to the right.

Syntax:
```
⎕wevent ←→ 'XColSizing'
⎕warg   ←→ Col Size
⎕wres[2] ←     Size
```

`⎕warg` is a two-element vector showing `[1]` the index to the column that is being resized and `[2]` the current width in virtual-pixels. You can set `⎕wres[2]` to a desired virtual-pixel width to override or limit the change.

Remarks:
This event occurs repeatedly, as often as once for each pixel if the drag is slow enough. You can also use the onXColSized event handler, which occurs once when the border is released, to override or limit the change.

When dragging the edge of the corner cell, the column argument in `⎕warg` is `¯99999`.

onXCopy

Trigger:
This event is triggered when the user presses Ctrl+C or your application invokes the XCopy method. It allows you to override the default grid behavior and handle the clipboard yourself.

Behavior:
See the “Managing Clipboard Operations” section for a detailed discussion about using the clipboard.

Syntax:
```
⎕wevent ←→ 'XCopy'
⎕warg   ←→ DoDefault Row Col RowExtent ColExtent
⎕wres[1] ← DoDefault
```

The arguments specify which cells are to be used in the clipboard operation. If your application handles the clipboard operation itself you must set the DoDefault result (`⎕wres[1]`) to zero (`0`) to prevent the grid from taking its default action.

onXCut

Trigger:
This event is triggered when the user presses Ctrl+X or your application invokes the XCut method. It allows you to override the default grid behavior and handle the clipboard yourself.

Behavior:
See the “Managing Clipboard Operations” section for a detailed discussion about using the clipboard.

Syntax:
```
⎕wevent ←→ 'XCut'
⎕warg   ←→ DoDefault Row Col RowExtent ColExtent
⎕wres[1] ← DoDefault
```

The arguments specify which cells are to be used in the clipboard operation. If your application handles the clipboard operation itself you must set the DoDefault result (`⎕wres[1]`) to zero (`0`) to prevent the grid from taking its default action.

onXEditCancel

Trigger:
An edit session on a cell was canceled by the user pressing Esc or by a program action.

Behavior:
When an edit session is canceled, the entry that exists in the temporary edit control is ignored and the last value that was assigned to the cell, which could have occurred in another event handler during the edit session, is restored to the cell.

Syntax:
```
⎕wevent ←→ 'XEditCancel'
⎕warg   ←→ Row Col
```

`⎕warg` is a two-element vector showing the coordinates (*Row Col*) of the cell that was being edited.

Remarks:
This event is akin to a notification that something (that is, the editing) didn't happen.

Note: The onXEditCancel event fires when editing is cancelled in grid versions 7.2 and later.

onXEditComplete

Trigger:
An edit session on a cell has completed and a value has been assigned.

Behavior:
When an edit session on a cell is active and the user presses either the Enter or the Tab key, it is a signal to try to end the edit session. You can also end an edit session by programmatic action. Additionally, when certain non-default conditions are set, moving the focus off the cell being edited can signal the end of an edit session. The onXEditEnd event fires before XEditComplete; within an onXEditEnd event handler you can control how (and sometimes if) you want the session to end. If certain options are selected, including the default of simply allowing the edit session to end normally, the XEditComplete event is triggered.

Syntax:
```
⎕wevent ←→ 'XEditComplete'
⎕warg   ←→ Row Col
```

`⎕warg` is a two-element vector showing the coordinates (*Row Col*) of the cell that was edited.

Remarks:
During the onXEditComplete event handler, the value of the cell has already been established, so you cannot meaningfully set `⎕wres`. You can use this event handler to adjust other cells that may be dependent on the most recently edited cell, for example, recalculating a total that includes the just-edited cell.

onXEditEnd

Trigger:
An edit session on a cell is ending; it is time to save a value.

Behavior:
When an edit session on a cell is active and the user presses either the Enter or the Tab key, it is a signal to try to end the edit session. You can also end an edit session by programmatic action. Additionally, when certain non-default conditions are set, moving the focus off the cell being edited can signal the end of an edit session.

Syntax:
```
⎕wevent ←→ 'XEditEnd'
⎕warg   ←→ Status Text Row Col
⎕wres[1] ← Status
⎕wres[2] ←        Text
```

`⎕warg` is a four-element vector showing `[1]` a status control code that is initially `1`;
`[2]` the text of the temporary edit control that is the value to be saved by default;
`[3 4]` the coordinates (*Row Col*) of the cell that was edited.

You can set `⎕wres[2]` to any string to specify a value to be saved (leaving `⎕wres[1]` set to `1`);
You can set `⎕wres[1]` to define how to end the session: `¯1`=cancel, `0`=re-edit; `2`=leave cell unchanged.

Remarks:
The default status control code value of `1` means save the value, either the value in the temporary edit box (which is already in `⎕wres[2]`) or a different value you assign to `⎕wres[2]`.

If you set `⎕wres[1]` to either `¯1` or `2`, it means close the temporary edit box and restore the last assigned value of the cell. This value could be the one that was there before the edit session started, or it could be a value that was assigned by another event handler during the edit session; for example, the user might enter a state code that your event handler converted to a full spelled-out name and assigned to the cell. You now want to ignore the abbreviation. The difference between these two values is that `2` means to complete the edit session successfully, triggering the XEditComplete event (in which you might use the state name), while `¯1` means to cancel the edit session, as though the user had pressed Esc, triggering the XEditCancel event.

If you assign a status control code of zero, the edit session does not end, the focus returns to the temporary edit box with the existing value highlighted. Note that this behavior may surprise a user; it is useful to provide some indication as to why the entered value was unsatisfactory.

Under certain circumstances, for example if the signal to end the edit session was given by invoking the XEditEnd method with a non-zero argument, or if the xUnfocusEndEdit property is set to `1`, the edit session must end; assigning zero to `⎕wres[1]` in the onXEditEnd event handler then cancels the session, leaves the cell unchanged and invokes XEditCancel.

onXEditKeyDown

Trigger:
A key was pressed while a cell is in an edit session.

Behavior:
The first keystroke in a cell under default circumstances triggers the onXCellKeyDown event handler as well as initiating the edit session. Subsequent keystrokes, including the Enter or Tab key that signals the end of the session, or the Esc key that aborts it, trigger this event handler.

Syntax:
```
⎕wevent ←→ 'XEditKeyDown'
⎕warg   ←→ Key Keys Row Col
```

`⎕warg` is a four-element vector showing `[1]` the virtual key code of the key that was pressed;
`[2]` any modifier keys that were down when the click occurred: sum of `1`=Shift, `2`=Ctrl, `4`=Alt;
`[3 4]` the coordinates (*Row Col*) of the cell being edited.

Remarks:
Note that the code in `⎕warg[1]` is a virtual key code that corresponds to a physical keyboard position. If you want to do something with the character that a key generates, use the onXEditKeyPress event handler.

onXEditKeyPress

Trigger:
A key or keystroke combination that generates a character was pressed while a cell is in an edit session.

Behavior:
This event follows XEditKeyDown; if the user holds a character key down, it triggers multiple pairs of XEditKeyDown and XEditKeyPress events without triggering XEditKeyUp.

Syntax:
```
⎕wevent ←→ 'XEditKeyPress'
⎕warg   ←→ CharCode Keys Row Col
⎕wres[1] ← CharCode
```

`⎕warg` is a four-element vector showing `[1]` the ANSI character code of the key that was pressed;
`[2]` any modifier keys that were down when the click occurred: sum of `1`=Shift, `2`=Ctrl, `4`=Alt;
`[3 4]` the coordinates (*Row Col*) of the cell being edited.

You can set `⎕wres[1]` to zero to ignore the character or to any valid ANSI character value to substitute a character.

Remarks:
You can find the ANSI character codes in the *System Functions Manual*, Appendix SFA, Atomic Vector/ANSI Sequence. You can find the virtual key codes in the *Windows Reference Manual*, Appendix WA.

onXEditKeyUp

Trigger:
A key was released while a cell is in an edit session.

Behavior:
The first keystroke in a cell, which under default circumstances initiates the edit session, triggers this event. However, keystrokes such as Enter or Tab that signal the end of the session, or Esc that aborts it, do not.

Syntax:
```
⎕wevent ←→ 'XEditKeyUp'
⎕warg   ←→ Key Keys Row Col
```

`⎕warg` is a four-element vector showing `[1]` the virtual key code of the key that was pressed;
`[2]` any modifier keys that were down when the click occurred: sum of `1`=Shift, `2`=Ctrl, `4`=Alt;
`[3 4]` the coordinates (*Row Col*) of the cell being edited.

Remarks:
Note that the code in `⎕warg[1]` is a virtual key code that corresponds to a physical keyboard position. If you want to do something with the character that a key generates, use the onXEditKeyPress event handler.

onXEditStart

Trigger:
An edit session on a cell is starting.

Behavior:
Typically a user can initiate an edit session in any of three ways: pressing a character key; pressing the `F2` key, or double-clicking a cell. (You can limit these actions by setting the xAutoEditStart property.) If the user presses a character key, that character appears in `⎕warg[2]` and, unless suppressed, in the temporary edit box. You can also start an edit session by invoking the XEditStart method, optionally assigning an initial character.

Syntax:
```
⎕wevent ←→ 'XEditStart'
⎕warg   ←→ Status Text Row Col CharCode EditWindow
⎕wres[1] ← Status
⎕wres[2] ← Text
```

`⎕warg` is a six-element vector showing `[1]` a status control code that is initially `1`;
`[2]` the text with which to start the edit session;
`[3 4]` the coordinates (*Row Col*) of the cell to be edited
`[5]` a character code showing which character initiated the edit session, or zero (for example, for `F2`)
`[6]` the Windows handle of the edit control, either the one created by the grid or equal to xEditWindow.

You can set `⎕wres[1]` to zero to cancel the edit session; you can leave `⎕wres[1]` set to `1` and set `⎕wres[2]` to any enclosed character string to specify a value with which to start editing. Note that `⎕warg[5]` is an ANSI character code and not a virtual key code.

Remarks:
Even if the cell is numeric (xValueType `1`), you can assign numerals (that is, a text representation of the number you want to use) to initiate the session. When the value is stored, the grid treats the entry as a number.

If you are using a special edit window, that is if xEditWindow is not equal to zero, any character that the user presses to initiate an edit session is not automatically sent to the special edit control. You are responsible for handling the user's character in an appropriate manner.

If the edit session is initiated without generating a character, by pressing `F2`, by double-clicking, or by invoking the XEditStart method without specifying a starting character, the character code in `⎕warg[5]` is zero, but `⎕warg[2]` contains any existing text. The text is selected in the edit window.

If you specify a starting character with the method, or if the user presses a character key, the existing text is replaced. However, you can still query the previous text in the xEditText property until you explicitly change the text or complete the edit session. If the user presses a key that generates an action rather than an explicit, visible character, for example, the Backspace key, `⎕warg[5]` still contains the character code for that "character." The content of `⎕warg[2]` is the character string that exists in the built-in edit control after the application of the non-printing key, or, if the edit window is a custom control, the string that would exist in that edit window. If the action key is the Backspace, the result is an empty character vector, since the back space action deletes the whole block of highlighted text.

The window handle in `⎕warg[6]` allows you to make low-level calls to the default grid edit window.

Note: The onXEditStart event handler does not fire for a combo cell.

Example:
To set initial text for an edit session using the character the user generates, use a statement similar to:

```
⎕wi 'onXEditStart' '⎕wres[2]←(⊂("prefix",⊃⎕warg[2]))'
```

onXKeyMove

Trigger:
The active cell is changing by virtue of the user pressing the Tab key or the Enter key.

Behavior:
By default, the Tab key moves the active cell one to the right (Shift+Tab to the left) unless it is the last one in a row, in which case it moves to the next row. If the active cell is part of a selection, this behavior may apply to the extents of a selection rather than the whole page. Similarly, pressing the Enter key moves the active cell down (or up, for Shift+Enter).

Syntax:

```
⎕wevent ←→ 'XKeyMove'
⎕warg   ←→ ToRow ToCol FromRow FromCol Key Keys
⎕wres[1] ← ToRow
⎕wres[2] ←       ToCol
```

`⎕warg` is a six-element vector showing `[1 2]` the coordinates (*Row Col*) of the newly active cell;
`[3 4]` the coordinates of the previously active cell (*Row Col*);
`[5]` the virtual key code of the key that was pressed (`9`=Tab, `13`=Enter);
`[6]` any modifier keys that were down when the click occurred: sum of `1`=Shift, `2`=Ctrl, `4`=Alt.

You can set `⎕wres[1]` and/or `⎕wres[2]` to assign the next active cell.

Remarks:
This event fires whether or not an edit session is active on the cell; it comes after the XCellKeyDown or XEditKeyDown event and, if in an edit session, before XEditEnd. Note that this event may fire for combinations of keys involving Ctrl or Alt only when you are in an edit session. When you are not in an edit session the effect of such a combination keystroke may be different and may not cause the active cell to change.

onXPageChanged

Trigger:
The active page has changed.

Behavior:
Typically, this occurs when the user clicks the tab for a different page. You can also set the xPage property.

Syntax:
```
⎕wevent ←→ 'XPageChanged'
⎕warg   ←→ NewPage OldPage Forced
```

`⎕warg` is a three-element vector showing `[1]` the index to the page that is newly active;
`[2]` the index to the page that was active;
`[3]` a reason code that is zero if the user triggered the change, or `1` if it was forced programmatically.

onXPageChanging

Trigger:
The active page is being changed.

Behavior:
Typically, this occurs when the user clicks the tab for a different page. You can also set the xPage property.

Syntax:
```
⎕wevent ←→ 'XPageChanging'
⎕warg   ←→ NewPage OldPage Forced
⎕wres[1] ← NewPage
```

`⎕warg` is a three-element vector showing `[1]` the index to the page that will become active;
`[2]` the index to the page that was active;
`[3]` a reason code that is zero if the user triggered the change, or `1` if it was forced programmatically.

If the user triggered the change (`⎕warg[3]=0`), you can set `⎕wres[1]` to zero to suppress the change.

onXPageOpen

Trigger:
A page is poised to be shown for the first time.

Behavior:
This event occurs only once for each page, except that it does not occur for the first page when the grid is created.

Syntax:
```
⎕wevent ←→ 'XPageOpen'
⎕warg   ←→ Page
```

`⎕warg` contains the index of the page that is about to become active for the first time.

Remarks:
This event occurs after the XPageChanging event and before XPageChanged. You can use it to establish the initial setup of the cells on a page before the page actually appears.

onXPaste

Trigger:
This event is triggered when the user presses Ctrl+V or your application invokes the XPaste method. It allows you to override the default grid behavior and handle the clipboard yourself.

Behavior:
See the “Managing Clipboard Operations” section for a detailed discussion about using the clipboard.

Syntax:
```
⎕wevent ←→ 'XPaste'
⎕warg   ←→ DoDefault Row Col RowExtent ColExtent
⎕wres[1] ← DoDefault
```

The arguments specify which cells are to be used in the clipboard operation. If your application handles the clipboard operation itself you must set the DoDefault result (`⎕wres[1]`) to zero (`0`) to prevent the grid from taking its default action.

onXReplCancel

Trigger:
This event is triggered when the user presses the ESCAPE key to cancel replication mode. It is one of two ways in which replication mode can be ended (the other is via the onXReplEnd event).

Behavior:
See the “Managing Replication Mode” section for a detailed discussion about using replication mode.

Syntax:
```
⎕wevent ←→ 'XReplCancel'
```

This event does not have any arguments.

onXReplEnd

Trigger:
This event is triggered when the user ends replication mode by releasing the left mouse button. It is one of two ways in which replication mode can be ended (the other is via the onXReplCancel event).

Behavior:
See the “Managing Replication Mode” section for a detailed discussion about using replication mode.

Syntax:
```
⎕wevent ←→ 'XReplEnd'
⎕warg   ←→ DoDefault State TargetRange SourceRange Keys
⎕wres[1] ← DoDefault
```

The State argument (`2⊃⎕warg`) describes the ending state of the operation:

	One of the following codes:
0	TargetRange matches SourceRange (degenerate selection)
1	TargetRange is extended from SourceRange along two axes (above, below, left, or right)
2	TargetRange is extended from SourceRange along two axes (one vertical and one horizontal)
	Plus any combination of the following:
4	Reserved for future use: TargetRange smaller than SourceRange
8	TargetRange was restricted from expanding by xAllowSelection setting of some cell(s)
16	TargetRange was restricted from expanding by xProtect setting of some cell(s)

The TargetRange argument (`3⊃⎕warg`) is a four element vector (*Row Col RowExtent ColExtent*) specifying the bounds of the target range where the replication rectangle was located when the mouse button was released.

The SourceRange argument (`4⊃⎕warg`) is a four element vector (*Row Col RowExtent ColExtent*) specifying the bounds of the source range where the replication rectangle was located when the mouse button was pressed to begin dragging. This is where the selection was located at the beginning of the operation. The selection will still be at this location unless your application has changed it during the replication mode dragging (that would probably be a bad idea).

The Keys argument (`5⊃⎕warg`) is an integer scalar indicating which shift keys were pressed at the time the mouse button was releases. It is zero (0) or the sum of the following codes: 1=Shift, 2=Ctrl, and 4=Alt.

The DoDefault result (`⎕wres[1]`) is currently unused but is reserved for possible future use. You should not modify it.

Programming Alert: At the present time you cannot drag the target range to be smaller than the source range. But we plan to enhance the grid to support this in the future. Therefore you must program now in anticipation that the target may eventually be allowed inside the original selection.

onXReplStart

Trigger:
This event is triggered when the user presses the left mouse button down on the replication handle to start replication mode.

Behavior:
See the “Managing Replication Mode” section for a detailed discussion about using replication mode.

Syntax:
```
⎕wevent ←→ 'XReplStart'
⎕warg   ←→ Enable SourceRange Keys
⎕wres[1] ← Enable
```

The SourceRange argument (`2⊃⎕warg`) is a four element vector (*Row Col RowExtent ColExtent*) specifying the bounds of the source range where the replication rectangle will be located as dragging begins. This is also where the selection is located at the present time.

The Keys argument (`3⊃⎕warg`) is an integer scalar indicating which shift keys are currently pressed. It is zero (0) or the sum of the following codes: 1=Shift, 2=Ctrl, and 4=Alt.

The Enable result (`⎕wres[1]`) can be set to zero (0) to disallow replication mode from starting.

Remarks:
At the time this event is fired, replication mode has not yet started. You can set the result to allow or disallow replication mode from starting. This is sometimes appropriate if enabling depends upon which cells are selected. But in most cases it is better to set xReplMode property to control this as a static setting. Otherwise your users can be confused by the cursor that appears when they are above the replication handle but fails to work to start replication mode.

onXReplTrack

Trigger:
This event is triggered whenever the position of the replication mode tracking rectangle changes or when the state of one of the three shift keys changes.

Behavior:
See the "Managing Replication Mode" section for a detailed discussion about using replication mode.

Syntax:
```
⎕wevent ←→ 'XReplTrack'
⎕warg   ←→ State TargetRange SourceRange Keys
```
The arguments describe the current state of replication tracking.

The State argument (1⊃⎕warg) describes the current state of replication tracking:

	One of the following codes:
0	TargetRange matches SourceRange (degenerate selection)
1	TargetRange is extended from SourceRange along two axes (above, below, left, or right)
2	TargetRange is extended from SourceRange along two axes (one vertical and one horizontal)
	Plus any combination of the following:
4	Reserved for future use: TargetRange smaller than SourceRange
8	TargetRange was restricted from expanding by xAllowSelection setting of some cell(s)
16	TargetRange was restricted from expanding by xProtect setting of some cell(s)

The TargetRange argument (2⊃⎕warg) is a four element vector (*Row Col RowExtent ColExtent*) specifying the bounds of the target range where the replication rectangle is currently located.

The SourceRange argument (3⊃⎕warg) is a four element vector (*Row Col RowExtent ColExtent*) specifying the bounds of the source range where the replication rectangle was located when the mouse button was pressed to begin dragging. This is where the selection was located at the beginning of the operation. The selection will still be at this location unless your application has changed it during the replication mode dragging (that would probably be a bad idea).

The Keys argument (4⊃⎕warg) is an integer scalar indicating which shift keys are currently pressed. It is zero (0) or the sum of the following codes: 1=Shift, 2=Ctrl, and 4=Alt.

Programming Alert: At the present time you cannot drag the target range to be smaller than the source range. But we plan to enhance the grid to support this in the future. Therefore you must program now in anticipation that the target may eventually be allowed inside the original selection.

onXRowSized

Trigger:
A row has been resized; that is, the bottom border of a row header cell was grabbed by the user and then the mouse was released.

Behavior:
Under default circumstances, if the user drags the border of a row header cell, the row above the border being dragged changes size, becoming smaller if the user drags up and larger if down.

Syntax:
```
⎕wevent ←→ 'XRowSized'
⎕warg   ←→ Row Size
⎕wres[2] ←     Size
```

`⎕warg` is a two-element vector showing the index to the row that was resized and the current height in virtual-pixels. You can set `⎕wres[2]` to a desired virtual-pixel height to override or limit the change.

Remarks:
This event occurs only once when the mouse button is released; if the user grabbed the border but did not actually move the mouse, this event still occurs, even if no XRowSizing events occurred.

When dragging the edge of the corner cell, the row argument in `⎕warg` is `¯99999`.

onXRowSizing

Trigger:
A row is being resized; that is, the bottom border of a row header cell is being dragged by the user.

Behavior:
Under default circumstances, if the user drags the border of a row header cell, the column above the border being dragged changes size, becoming smaller if the user drags up and larger if down.

Syntax:
```
⎕wevent ←→ 'XRowSizing'
⎕warg   ←→ Row Size
⎕wres[2] ←     Size
```

`⎕warg` is a two-element vector showing the index to the row that is being resized and the current height in virtual-pixels. You can set `⎕wres[2]` to a desired virtual-pixel height to override or limit the change.

Remarks:
This event occurs repeatedly, as often as once for each pixel if the drag is slow enough. You can also use the onXRowSized event handler, which occurs once when the border is released, to override or limit the change.

When dragging the edge of the corner cell, the row argument in `⎕warg` is `¯99999`.

onXScrollHint

Trigger:
A scroll hint is about to appear. For this to happen, the grid must be large enough to be able to scroll, scroll hints are enabled, and the user has pressed the primary mouse button on the thumb of one of the grid's scroll bars (that is vertical or horizontal, not the arrows that scroll page tabs) or has moved the thumb sufficiently that it has caused the grid to scroll or would change the view of the grid if the mouse were released at this point.

Behavior:
The xScrollMode property affects the scrolling behavior on a page; horizontal and vertical settings are independent. Three of the four possible settings for each direction are relevant to this event. 1) You must not have hidden the scroll bar (it appears, when needed, by default), but 2) you must have activated scrolling hints (which do not appear by default) for the event to be able to occur. In addition, the grid must be large enough on the active page so that at least one row or column is not completely visible (hence the grid can scroll). The third relevant setting determines whether the grid scrolls one row or column at a time or moves by larger amounts when the mouse is released. Moving a row or column at a time is called tracking; by default it is off.

The first hint appears when the user presses the primary mouse button while the cursor is over the thumb of the scroll bar. By default, the hint shows the index number of the row or column (depending on which scroll bar) that is currently the anchor cell of the view. (The anchor cell of the view is the uppermost and leftmost cell that is visible and scrollable; this excludes any rows or columns that are fixed by setting xFixedRows or xFixedCols.)

The purpose of this event handler is to allow you to set the text of the hint, which appears in a small tooltip-like window, before the hint appears. The row or column index, whichever is relevant, is in `⎕warg`. Note that this event does not inhibit the actual scrolling; it only controls the appearance of the hint.

Subsequently, this event is triggered each time the user moves the thumb far enough to change the view of the grid. The default hint is the index to the row or column of the cell that is becoming or would become the anchor cell. If tracking is on, the grid's movement and the hints are virtually simultaneous. If tracking is not enabled, the view does not change until the user releases the primary mouse button. At that time, the grid jumps, rather than scrolls, to the new position, as though the xView property had been set.

Syntax:
```
⎕wevent ←→ 'XScrollHint'
⎕warg   ←→ Hint Row Col
⎕wres[1] ← Hint
```

`⎕warg` is a three-element vector showing `[1]` the text of the default hint;
`[2]` the index to the row that will become the anchor row, if the mouse is on the vertical scroll bar;
`[3]` the index to the column that will become the anchor column, if scrolling horizontally.

Whichever of `⎕warg[2]` and `⎕warg[3]` is not relevant, is zero.

The default scroll hint in `⎕wres[1]` is the row or column number formatted as a character vector. You can set `⎕wres[1]` to a different value change the scroll hint.

Remarks:
Note that it is only the mouse dragging the scroll thumb that enables this event. If the user scrolls by keyboard action or by clicking in the channel of the scroll bar or on one of the scroll arrows, this event does not occur.

onXSelectionChange

Trigger:
The current selection has changed, or the active cell within the current selection has changed.

Behavior:
As a user creates or modifies the selection by dragging the mouse or arrowing while holding down the Shift and/or Ctrl keys, this event fires for each action that changes the selection. In addition, if a multi-cell selection is current, moving within the selection also triggers this event; however, in the latter case, the values in `⎕warg` remain constant.

Syntax:
```
⎕wevent ←→ 'XSelectionChange'
⎕warg   ←→ Row Col RowExtent ColExtent Block
```

`⎕warg` is a five-element vector showing `[1 2]` the coordinates (*Row Col*) of the anchor (upper left) cell;
`[3 4]` the number of rows and columns in the current (block of the) selection (*RowExtent ColExtent*);
`[5]` the block number; this will generally be the most recently created block, equal to the number of rows in the xSelection matrix.

Remarks:
The anchor cell, whose coordinates are in `⎕warg`, is always at the upper left of the selection, even if the user starts in a cell and drags the mouse up and/or to the left. The anchor cell is not necessarily the active cell, even for a newly created selection.

onXTabMouseDouble

Trigger:
The left or right mouse button was pressed on a page tab twice in succession.

Behavior:
This event occurs on the second click; if the first click caused the page to change, this event occurs after the XPageChanged event.

Syntax:
```
⎕wevent ←→ 'XTabMouseDouble'
⎕warg   ←→ Tab Button Buttons Keys
```

`⎕warg` is a four-element vector showing `[1]` the page number of the tab that was clicked ;
`[2]` the mouse button that came down: `1`=left, `2`=right;
`[3]` any mouse buttons that were pressed when the event occurred: sum of `1`=left, `2`=right, `4`=center;
`[4]` any modifier keys that were down when the event occurred: sum of `1`=Shift, `2`=Ctrl, `4`=Alt.

Remarks:
You can use this event, for example, to allow a user to change the name of the page.

onXTabMouseDown

Trigger:
The left or right mouse button was pressed on a page tab.

Behavior:
Pressing a mouse button while the cursor is over a tab that does not represent the current page attempts to change the page; the XTabMouseDown event occurs before XPageChanging.

Syntax:
```
⎕wevent ←→ 'XTabMouseDown'
⎕warg   ←→ Tab Button Buttons Keys
```

`⎕warg` is a four-element vector showing `[1]` the page number of the tab that was clicked ;
`[2]` the mouse button that came down: `1`=left, `2`=right;
`[3]` any mouse buttons that were pressed when the event occurred: sum of `1`=left, `2`=right, `4`=center;
`[4]` any modifier keys that were down when the event occurred: sum of `1`=Shift, `2`=Ctrl, `4`=Alt.

Remarks:
You can use this event, for example to pop up a menu for the user after a right mouse click.

onXValueChange

Trigger:
The user changes the value of a cell by an action such as completing an edit session, clicking a Check box cell, selecting an item from the list of a Combo box cell, pressing the Enter key or the spacebar on a Button cell, or pressing the delete key while the cell was selected. This event fires for each cell if the delete key is pressed while a multi-cell selection is current.

Behavior:
Program actions such as setting a cell's value or invoking the XDeleteCells method do not trigger this event. The XValueChange event has particular utility for the Combo box, Check box, and Button cell types.

For a Combo box cell, clicking the cell triggers the XCellClick event and not this one; dropping the menu down and clicking an item triggers XValueChange but not XCellClick. Pressing the spacebar has no effect, but pressing the Enter key drops the list down, triggering the XCellDropDown event.

For a Check box cell, clicking the box to select or unselect it triggers both XCellClick and XValueChange. Pressing the space bar has the same effect as clicking the box.

For a Button cell, clicking the mouse on the cell triggers only XCellClick. Pressing the Enter key or the spacebar triggers only XValueChange. In addition, pressing Enter causes the active cell to change.

Syntax:

```
⎕wevent ←→ 'XValueChange'
⎕warg   ←→ Row Col
```

`⎕warg` is a two-element vector showing the coordinates (*Row Col*) of the cell whose value was changed.

Remarks:
For normal cells that a user can edit, the onXEditComplete event is often more appropriate than this event.

onXValueClear

Trigger:
The user pressed the Delete key while a cell was selected, but not in an edit session. If a multi-cell selection is current, the event fires for each cell in each block of the selection.

Behavior:
This event does not fire if you invoke the XDeleteCells method or reset a cell's default value, nor if the user erases the value of a cell in an edit session. It precedes XValueChange on regular cells. If you do not suppress the action in the event handler, any value in a regular cell is erased. This event also fires on special cell types, but there is no default action that the system takes; for example, a check box is not cleared.

Syntax:

```
⎕wevent ←→ 'XValueClear'
⎕warg   ←→ Enable Row Col
⎕wres[1] ← Enable
```

`⎕warg` is a three-element vector showing `[1]` a status control code
`[2 3]` the coordinates (*Row Col*) of the cell that is about to be cleared.

The status control code is initially `1` for regular cells, but zero for special cell types. For regular cells, you can set `⎕wres[1]` to zero to suppress the deletion and retain the existing value. Setting `⎕wres[1]` has no meaning for special cell types, but you can take any action deemed desirable in the handler.

onXViewChange

Trigger:
The current view (the range of visible cells to the right and below any fixed columns and rows) has changed.

Behavior:
When the grid has more rows or columns than are visible at one time, those rows that are not fixed can be scrolled into view, either by using the default scroll bar, or by using the keyboard to move to a cell that is all or partially outside the visible area.

Syntax:

```
⎕wevent ←→ 'XViewChange'
⎕warg   ←→ Row Col RowExtent ColExtent
```

`⎕warg` is a four-element vector showing `[1 2]` the coordinates (*Row Col*) of the top left visible, scrollable cell; `[3 4]` the number of rows and columns (*RowExtent ColExtent*) that are wholly or partially visible.

Remarks:
If you specify a cell near the right edge of the grid in the xView property, the grid scrolls far enough to bring the edge into view but does not move the specified cell all the way to the left of the scrollable area. Similarly, if the bottom of the grid is visible, a specified cell may not be moved all the way to the top of the scrollable area.

Note that changing the view does not change the active cell; you can scroll the active cell off the screen.

onXVirtualLoad

Trigger:
One or more virtual cells need to be loaded for display on the screen, printing, cut, copy, paste, or xml operation when virtual mode is enabled on a selected part of the grid (see the xVirtualMode and xVirtualParts properties). Alternatively, regardless of whether virtual mode is enabled or not, the event is fired periodically as a progress report for potentially long running operations (printing, cut, copy, paste, and xml).

See the "Managing Virtual Mode" section for a detailed discussion of virtual mode.

Behavior:
Before displaying a cell on the screen, or using it for printing, cut, copy, paste, or xml operations, the grid fires this event to give your application an opportunity to load any cells that are currently not defined (i.e., that have no properties set). This is done via a series of onXVirtualLoad events, one for each part of the grid that isn't already loaded from a previous call and was not set statically before virtualization was enabled. A cell is a candidate for virtual loading only if it meets all of the following conditions:

1. Virtual mode is enabled and has not been disabled for the particular operation type being performed by the grid (see the xVirtualMode property).
2. The part of the grid containing the cell is enabled for virtualization (see the xVirtualParts property).
3. The cell must be **completely** clear. This means it cannot have **any** properties set. Once you set any property for a cell it will not be virtually loaded again unless you invoke the XDeleteCells or XVirtualUnload methods or it is automatically unloaded after being consumed in certain operations (see below). Note that cells can also become "set" and ineligible for virtual loading due to certain operations such as editing or pasting from the clipboard.

After your handler returns, the grid checks the list of cells that it requested you to load. Any cells it finds on this list that have been loaded during the event get their virtually-loaded attribute set. You can query the virtually-loaded attribute (or set or clear it) via the xVirtualLoaded property.

Programming Alert: Any cells you load during your onXVirtualLoad handler that are **not** part of the requested cells list will **not** get their virtually-loaded attribute set. Thus they will become permanently loaded, perhaps by accident! See the "Loading and Unloading of Cell" topic in the "Managing Virtual Mode" section for details.

Any cells that were requested for loading but were not loaded will be requested again if needed (since they continue to meet the qualifications listed above).

Any cells you load when the event is requesting them for display on the screen become "sticky" and are never unloaded automatically. This is an optimization since users often revisit cells they have seen before and it makes scrolling a little faster. They will only be unloaded if you invoke the XDeleteCells or XVirtualUnload methods.

Cells loaded for other reasons, such as printing, clipboard, or xml operations, are transient. An event is fired to load a few thousand of them at a time, then their values are used, and when the operation moves on to another part of the grid (such as the next part of the page when printing) the previously loaded cells are automatically unloaded so long as their changed flag has not become set.

Syntax:

```
⎕wevent ←→ 'XVirtualLoad'
⎕warg   ←→ Reason Cells Range Part PercentDone ElapsedTime RemainingTime Extra
⎕wres[1] ← Reason
```

`⎕warg` is an eight element vector containing the following arguments:

`⎕warg[1]`	Reason	Reason event was fired (see table below)
`⎕warg[2]`	Cells	Matrix of cell coordinates to be loaded
`⎕warg[3]`	Range	Minimum and maximum range of coordinates in Cells (see below)
`⎕warg[4]`	Part	Part of grid being loaded (see below)
`⎕warg[5]`	PercentDone	Percentage of work completed (always zero for screen)
`⎕warg[6]`	ElapsedTime	Actual elapsed time (always zero for screen)
`⎕warg[7]`	RemainingTime	Estimated time remaining or `¯1` for screen or if not yet estimated
`⎕warg[8]`	Extra	Reserved for future use

Your application should respond to the request by setting the values of the requested cells. For example, it can set the `xText` property, `xValue` property, `xNumber` property, etc. As a general rule it will be more efficient to avoid setting attributes such as `xCellType`, `xAlign`, etc. for individual cells. When possible, you should instead let them be inherited from column and row defaults.

During long running print, cut, copy, paste, xml and loading operations, you can set `⎕wres[1]` to `0` to cancel processing. This will stop the operation and suppress any future events that would have occurred for that instance of the operation. You will not get a final 100% completion event in this case.

The Reason argument indicates why the event is being fired. The following codes are possible:

1	Loading cells for display on the screen
2	Loading cells for printing
4	Loading cells for clipboard cut/copy operation
8	Loading cells for clipboard paste operation
16	Loading cells for xml generation
128	Loading cells for preloading

The Cells argument is a two column matrix specifying the cell coordinates (row and column) of each cell that needs to be loaded. These coordinates will all be in the same part of the grid (body, row header, column header, corner button, row default, or column default) for any one event. Multiple events will be fired if more than one part of the grid needs cells to be loaded.

The Range argument describes the smallest rectangle containing all the cells of the Cells argument as a standard grid range: Row Col RowExtent ColExtent. If the Cells argument is empty the Range argument will have a value of 0 0 0 0. This occurs only for progress reporting events that don't load any cells and just report progress tracking arguments.

The Part argument indicates which part of the grid is being requested for loading. The following codes are possible:

1	Body cells	(`Row > 0` and `Col > 0`)
2	Column headers cells	(`Row < 0` and `Col > 0`)
4	Row headers cells	(`Col < 0` and `Row > 0`)
8	Corner button cell	(`Row = ¯1` and `Col = ¯1`)
16	Column defaults cells	(`Row = 0` and `Col > 0`)
32	Row defaults cells	(`Col = 0` and `Row > 0`)
0	Progress report	(No cells are being loaded)

These are the same as the codes you can specify for grid parts with the `xVirtualParts` property except for the addition of zero which means no part is being loaded and is used for progress reporting events. When the Part code is zero, the Cells argument will be empty and the Range argument will have the value 0 0 0 0.

The PercentDone argument is an integer value specifying the estimated percentage of work completed in the operation that caused the virtual load event to be fired. It is always zero when the Reason code is 1 (loading data for display on the screen). In other contexts, this percentage can be useful when displaying progress information to users. The value counts from 0 to 100 but never reaches 100 until the task is fully complete. Therefore you can use 100 as a signal to hide any progress gauge you might have displayed.

The ElapsedTime argument is an integer value indicating the number of seconds that have elapsed since the operation started running. This will always be zero for display operations (Reason=1). In other contexts it can be used as a signal to display the progress gauge. For example, you might delay the display of a progress form until at least 2 seconds have elapsed.

The RemainingTime argument is an integer value indicating the estimated number of seconds remaining before the operation completes. It has the value ¯1 for the first few seconds of operation until the grid has enough data to begin estimating the time remaining to completion. Then it starts counting downward toward zero as the operation draws toward completion (however, it might backtrack to a higher value at times if progress slows). The value never reaches zero until the operation is complete.

The Extra code is just that, extra. It is not used in this release but may be in a future version.

Remarks:
The “Managing Virtual Mode” section contains a detailed explanation of virtual mode and describes how the xVirtualMode and xVirtualParts properties and the onXVirtualMode event interoperate.

The VDEMO workspace provides an in depth set of virtualization examples. Load the workspace and execute the]DEMO user command as shown below:

```
      )LOAD C:\APLWIN10\EXAMPLES\VDEMO
      ]DEMO vDemo
```

Printing and clipboard cut and copy operations generate events to load cells before they are printed or placed on the clipboard. This makes sense. But it might be surprising that clipboard paste operations are also virtualized. After all, the values of these cells are going to be overwritten by whatever data is coming from the clipboard. So why preload them before doing the paste operation? The reason is so we know the cell type being pasted into (this is important for how the data is handled). We also need to know if the incoming value is different than the current value in the cell. This is necessary in order to set the changed flag properly (see the xChanged and xChanges properties). Otherwise all pasted cells would be marked as changed even when they match the original value in the cell. If you don’t care about the accuracy of the changed flag in this case, you can disable clipboard paste virtualization via the xVirtualParts property.

Even though the clipboard is fully virtualized automatically, you might prefer to do clipboard operations directly by handling the onXCut, onXCopy, and onXPaste events. This gives you more complete control and might offer better performance if you want to handle multiple clipboard formats or load large blocks from your database directly onto clipboard without having to load them into cells incrementally. It is a lot more work than handling the clipboard automatically. But you should consider it if you find the built-in methods are slower than you would like.

The XML properties do virtualization in an asymmetric way. All of the properties fire the onXVirtualLoad during output generation. But they do not fire the event during XML input. XML input operations don’t set the changed flag on the cells they load into the grid and therefore don’t need to know the previous value in the way clipboard paste operations require. In addition, they are difficult to gauge in terms of progress toward completion because there isn’t any way to guess how many cells they contain in advance of actually processing through them completely. And that would mean processing the XML twice which could be slow.

When generating XML output each property behaves differently. The xXMLRange, xXMLTable, xXMLValueRange, and xXMLText request cells to be loaded while running. However, the xXMLChanges, xXML, and xXMLPage do **not** load cells. These properties only fire progress reporting events.

The first set of XML properties is able to do virtual loading because they operate on the current page only. At the present time, we are restricted to only firing events for the current page. It is therefore not possible for the second set of properties to load cells because they span multiple pages.

But even if we could fire cross-page events, it doesn't make sense to do so for two of these three properties. The xXML property is supposed to capture the current state of the grid and is used internally to save the state when you close the grid (when savecontent is enabled). Loading cells in this case would change the state and unnecessarily bloat the xml document size. Similarly, the xXMLChanges property only contains cells that have changed and by definition those cells cannot be virtually loaded (since they already have at least the changed attribute set).

Therefore, of the three properties in the second set, the only possible future candidate for virtual loading is the xXMLPage property. If that does happen some other flag will have to be added to enable it to be fired and the Extra argument might be used to indicate which page is being requested for loading.

Details on the xFormat Property

The xFormat property allows you to set formatting rules for numbers, currency, dates, and times. This applies to normal (xCellType `0`) cells with xValueType set to `1`, `4`, or `3`. If you set xFormat for any other cell, it will either be ignored or cause you problems. It is important to set xCellType and xValueType before the user has a chance to edit a cell. Setting these properties when values are already present can have unexpected results. Also note that setting xValueType as a column default is not as strong as setting the entire column explicitly.

The discussion below addresses the formatting of numbers first; formatting currency cells is the same as formatting numeric cells with some additional decorations. Formatting dates and times follow. Dates and times are both cells with xValueType set to `3`; the value of a date is an integer-valued number, while a time is the fractional part of a decimal number. You can have both a date and a time in the same cell.

When you have a value in any of these three types of cells, it is numeric. That is, if you query the xValue property, or any of the xNumber, xCurrency, or xDate properties, as appropriate, the grid returns a number, or perhaps the conversion error value, if the entry is inappropriate for the cell type. What the user sees in the cell, which is what the grid returns when you query xText, may appear to be very different. Setting the xFormat property allows you to specify how the text displays in the cell. It does not change the underlying value. You can also see the text that is available for editing, which may differ from both xValue and xText, by querying the read-only xEditText property.

The xFormat property is a character vector, holding one to four formatting phrases; phrases are separated by one to three semicolons. A format phrase can consist of selector codes, control codes, literals, and highlights.

> Selector codes are placeholder characters that indicate where and how (or whether) a value should be displayed; for example # indicates to display a significant numeric digit (that is, other than a leading or trailing zero) and MM indicates to identify a month by a two-digit number, using a leading zero if necessary.
>
> Control codes have many functions including identifying literals, defining white space, acting as placeholders for decorations, and separating phrases.
>
> Literals are characters that appear in the display.
>
> Highlights are modifiers of the display, including color and font styles.

Although the format phrases apply to different values, the specification of the first phrase can affect the meanings of other phrases; in addition, if you do not specify other phrases, the system may generate a relevant display that is affected by the phrase that is specified. The implications of this statement are explained throughout. Displays are also affected by the settings of the xLocale property for each cell and the Control Panel / Regional Options settings on the host machine.

When a person chooses a regional setting, the basic locale choice is a country/language combination. Some countries have only one language choice (United States/English) but other countries have more; for example, if you are in or select Switzerland, you may choose French, Italian, or German. The choice of language affects the representation of months of the year and days of the week, if you choose to spell them out, for example. The regional setting also determines what characters are used as the decimal mark and the thousands separator, as well as the order of day, month, and year in date settings. An individual may also tailor these choices with personal preferences.

Setting the xLocale property to a value other than the local default specifies that country/language choice for a cell and overrides the non-default personal preferences on the host machine. Setting xLocale to `1024` specifically chooses the personal preferences on the host machine. The local system default setting is a synonym for `1024`, the user preferences (see below for the implications of this statement).

Setting xFormat to an empty character vector clears any explicit formatting for the cell and returns it to default formatting. As with other cell attributes, you can set page defaults and/or column defaults that take precedence over the default behavior of the grid.

Numeric and Currency Formatting

A numeric cell has xCellType `0` and xValueType `1`; you can set it with the xNumber property or the xValue property. By default, a number cell right aligns its values. You can type in a number in any of several formats. If you type in letter characters, the grid does not reject the entry; it displays it as typed, but if you reference the value, the grid returns the conversion error value. If you set a numeric cell with text, it changes the value type.

When you set the xFormat property, you can use up to four format phrases. They apply, in order, to positive values, negative values, zero, and missing values.

Alert: When discussing numeric formatting, it is useful to keep in mind that a "number" is an abstract idea. In a computer, a number is represented by a bit pattern; or it may have several possible bit patterns. For example, the number that is represented in the session by the numeral `9` has a different bit pattern if generated by (`10-1`) than it does if generated by (`10.5-1.5`), but the values match in APL.

It is difficult to write about a number without representing it in some fashion. In the discussion below, numbers are represented in the traditional English language and APL convention using a decimal point (the period character). In countries whose primary language is other than English, numbers may comprise the same numerals but use a comma (a decimal squirt) to separate the integral portion of a number from the fractional portion. This difference may be important (and confusing) in the grid if your default machine settings are for a country that uses the comma. In those cases, when setting the value of a cell with an APL statement, use the period; when typing a number into a cell, use the comma. If you type in the wrong symbol, the grid may ignore it and treat all the digits as an integer, or the input may be invalid.

Numeric Formatting – Positive Values

The basic format phrase for a positive numeric value consists of two types of digit selectors and one or two control codes. A digit selector can be a zero (`0`), which indicates that a numeral should appear in the corresponding space, or an octothorpe (`#`), also called number sign among other names, which indicates that a numeral should appear if it is non-zero, but a non-significant digit may be omitted; that is, if there are only zeroes before it (to the left of the decimal symbol) or only zeroes after it (to the right of the decimal). Note however, that these digit selectors apply differently to the integral part of a number and the fractional part.

The control codes are period (`.`) or exclamation point (`!`) for the decimal mark, one of which is mandatory unless you want only integers, and comma (`,`) for the thousands separator. Note that these are, in fact, codes and not literals. Positioning a period in the format phrase determines the position of the decimal mark in the display of the number. When the number displays in the grid, the decimal mark may, in fact, be a comma.

Similarly, the separator comma is a code and not a literal. If you specify a comma in the format phrase, the number displays in a grid cell with groups of (usually) three numbers separated by a mark which may be a space (generated by the ANSI code for a non-breaking space), a period, an apostrophe, or, literally, a comma.

Note that there may be many more possibilities for the actual display characters, if there are bizarre individual settings on a host machine, or if a machine has support for language groups that use non-Latinate alphabets.

Basic Format

The default format for a number cell is that all the significant digits of an assigned or calculated number are displayed (assuming the cell is large enough that the left portion of the number is not obscured). The number is rounded, if necessary, with a print precision of `15` (although if you query the value, the grid returns the number with print precision of `⎕pp` for the workspace). The visible display is right justified. The decimal mark shows only if there is a fractional portion of the number; there is a leading zero if the number is less than one. A zero value is represented by a single zero with no decimal mark. The thousands separator does not appear, no matter how large the number. If the number is typed in, it displays as typed.

If you set the xFormat property, the rules are significantly different. If you want any fractional part of a number to display, you must place one or more digit selectors to the right of the decimal control code. If you use zeroes, which must go immediately to the right of the period, the decimal symbol and the same number of digits display. If you use the octothorpe, the digits display only when they are non-zero or are followed by a non-zero digit. However, the grid aligns the number in the cell so that there is room for the number of digits that is equal to the number of digit selectors. The grid rounds values to the last displayed digit.

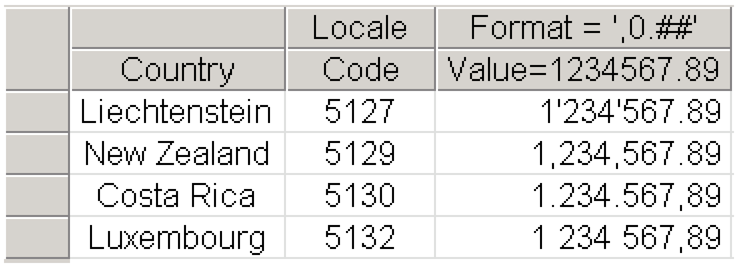
Example

```
      ⎕wi 'xFormat' (⍳5) (1) '.0##'
      ⎕wi 'xValue' (⍳5) (1) (5 1⍴1÷1 2 4 8 16)
      ⎕wi 'xValue' (⍳5) (1)   ⍝   GRID DISPLAY
 1                            ⍝   |      1.0  |
 0.5                          ⍝   |       .5  |
 0.25                         ⍝   |       .25 |
 0.125                        ⍝   |       .125|
 0.0625                       ⍝   |       .063|
```

One of the great advantages of using such a format is that a column of related numbers all align on the decimal. Even if the format were `'.###'` so that the integer displayed without a decimal mark, the first two cells would be aligned like this:

```
                              ⍝   _____________
                              ⍝   |      1    |
                              ⍝   |       .5  |
```

On the left side of the decimal code, the digit selectors do not have as much effect. Specifying a zero guarantees a numeral in the units position. Specifying an octothorpe leaves a blank in the units position when the magnitude of the value is less than `1`. It is prudent to specify one zero selector code, so that you do not end up with a blank cell when the value, is in fact, zero. Alternatively, you can be careful to specify a meaningful third format phrase. In addition to a digit selector, you can add the thousands separator control code. In the format phrase, this code is a comma; how the number displays depends on the locale.



		Locale	Format = ',0.##'
	Country	Code	Value=1234567.89
	Liechtenstein	5127	1'234'567.89
	New Zealand	5129	1,234,567.89
	Costa Rica	5130	1.234.567,89
	Luxembourg	5132	1 234 567,89

Using the default Arial font, even with the same format phrase, numbers do not necessarily align perfectly. If your application is used in Liechtenstein or Switzerland (three locale codes, varying with language) as well as other places, you may want to use a monospace font. For locale codes listed in the appendix, the first pattern above (apostrophe, period) is used in the two countries mentioned. The second pattern (comma, period) is typical of English speaking countries (including Philippines) and Spanish speaking locales in Central America and the Caribbean (including Mexico), plus Peru and Kenya. The third pattern (period, comma) is used in most of continental Europe and South America, plus Brunei, Indonesia, and Maylasia. The fourth pattern (space, comma) applies to the geographically large Scandinavian countries (Sweden, Norway, Finland) and some French-speaking areas (including Quebec). The grid formats these with a non-breaking space `⎕av[160+⎕io]`.

Mandatory Decimal Mark

When you use the period as the decimal control code, the decimal mark appears only when there is a fractional portion to the number. If you want the decimal mark to appear at the end of integers, you can use the exclamation point for the decimal control code, followed by one or more octothorpes or nothing. In this case the decimal mark appears regardless of the value. Note that if you use only octothorpe digit selectors with the exclamation point, you may, when the value is zero, end up with only the decimal mark in an otherwise empty cell.

Negative Control Codes Specified in the Positive Format String

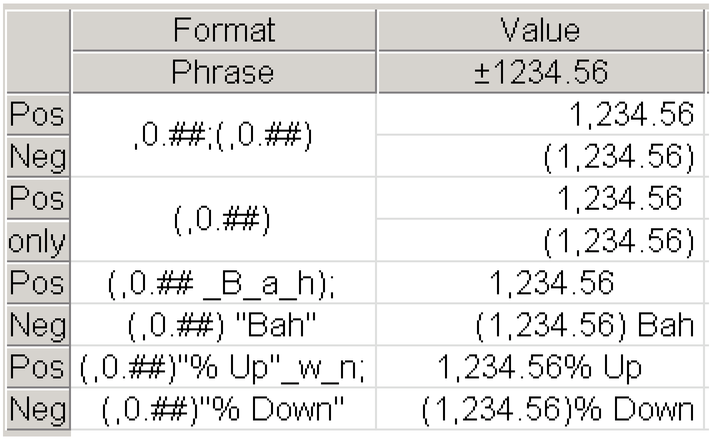
Although you can specify a second format phrase that applies to negative values, the positive phrase can be used to define typical default negative formats and coincidentally align positive and negative numbers. If, for example, you want to identify negative numbers by enclosing them in parentheses or by placing the minus sign after the number (as is done in some accounting reports), these characters in the display change the right-hand alignment. If you specify either the parentheses or the minus sign in the positive format string, it leaves space for that character, but does not display it. It also defines the default negative format. This allows you to have both positive and negative numbers aligned with the decimal point at the same position in the cell.

Other Spacing

In addition, on the right hand side of a numeric format phrase, you can leave space that matches any character. For example, if you want to place the word "Bah" after a negative number, but you want the positive and negative numbers to align with each other, you can use the underscore as a control character followed by the character whose width you want to match with white space. In this case, you would have to explicitly code the "Bah" in the negative (second) format phrase. This positive format phrase would be coded as: `'0.##_B_a_h'`

Note that white space generated by the negative or underscore control codes works only at the right side of the cell. It has no meaning to place it to the left of the number, and it does not locate the white space between the number and a literal. You can see that the percent signs in rows `7` and `8` in the picture below do not line up exactly, but the numbers do; the white space for the right parenthesis is to the right of the literals.

Note also that the space is a literal character in a format phrase. If you leave a space before the semicolon that separates the positive phrase from the next phrase, it affects spacing in the cell. Note the difference in the juxtaposition of the percent sign in rows `7` and `8` from the "Bah" in row `6`. (See below for specifying literals.)



	Format	Value
	Phrase	±1234.56
Pos	,0.##;(,0.##)	1,234.56
Neg		(1,234.56)
Pos	(,0.##)	1,234.56
only		(1,234.56)
Pos	(,0.## _B_a_h);	1,234.56
Neg	(,0.##) "Bah"	(1,234.56) Bah
Pos	(,0.##)"% Up"_w_n;	1,234.56% Up
Neg	(,0.##)"% Down"	(1,234.56)% Down

Literals

As prematurely demonstrated in the example above, you can specify a literal phrase after or before a number simply by enclosing the characters that you want to appear in quotes. Alternatively, you can specify a single character as a literal by preceding it with the backslash. In addition, certain characters themselves are treated as literals in a format phrase. These include the space ( ), colon (`:`), forward slash (`/`), and Euro (`€`). Literals must be to the left or right in a numeric cell; you cannot intersperse them among numeric digits. You can, however, place them between the number and the negative control code.

There is also one character that functions as a translated literal. If you include the character `⎕av[160+⎕io]` in a format string, it functions as a literal non-breaking space. This character appears in the APL session as a lowercase a with an acute accent (á). You can generate the character in APL by typing Alt+`0225`.

This character and the Euro demonstrate a problem you may face if you want to use characters other than those in the basic English alphabet, numerals, and the common punctuation marks and symbols. If you want to use the Euro symbol as a quoted literal in a format phrase, you must enter `⎕av[128+⎕io]`. For example,

```
      ⎕wi 'xFormat' 6 3  '(,0.00 "Ç")'
```

Note that the symbol produced by `⎕av[128+⎕io]` in the APL session is not the Euro. This occurs because there is no translation between APL and the grid, so that symbols that are not in the same position in `⎕av` and the ANSI font do not display the same in APL and the grid. If you are not concerned with european characters, skip the next section.

Digression: Setting Text Values for non-English Characters

When you set a grid property with a text string, the grid is using the zero-origin `⎕av` indices of the characters in APL as indices to its font. For the English alphabet, numerals, most punctuation marks, and common symbols below font positions `128`, the indices match. For many characters, such as european vowels with diacritical marks, the font positions are not the same. If you want to generate these characters in the grid using an APL statement that contains the characters, you must perform an AV to ANSI translation.

You can do this using an existing utility function, or you can do it in the session using a `⎕wcall` intrinsic.

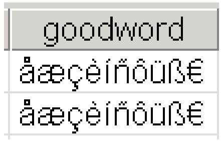
1) From the `WINDOWS` workspace in your Tools directory, copy `AV2ANSI` and `av2ansi`. The former is a function, that latter a variable.
2) Generate the text string you want to use in a variable, named, for example: `goodword`
3) Assign the text with a statement like: `⎕wi 'xText' 1 4 (AV2ANSI goodword)`

or

1) Capture the two-row, `256`-column variable that is generated by: `ttbl ← ⎕wcall W_AvANSI`
2) Assign the text with a statement like: `⎕wi 'xText' 2 4 (ttbl[1;(⎕av ⍳ goodword)])`

For example:

```
goodword ← 'åæçèíñôüß€' ⍝ ⎕av[17 18 136 139 162 165 148 130 226 3] (⎕io←1)
```



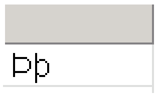
goodword
åæçèíñôüß€
åæçèíñôüß€

Alternatively, you can generate characters in the grid by assigning the APL characters whose positions in `⎕av` match the positions in an ANSI font of the characters you want. Using this technique, you do not use either of the translation schemes demonstrated above. This is demonstrated by assigning the character `⎕av[128+⎕io]`, which appears in the APL session as uppercase C cedilla (Ç), to generate the Euro.

Conveniently, when you know the ANSI index value, you can frequently, but not always, generate the appropriate character in an APL session by typing the code on the numeric keypad while holding down the Alt key. Using the above example, type Alt+`0128`, highlight the resulting character, and paste it into your APL statement, enclosed in quotes.

You can also generate characters in the grid that are in the ANSI font but not in `⎕av` by assigning as a literal the APL character whose position in the APL font corresponds to the position in the ANSI font; for example, the APL statement below, which uses a `⎕av` position occupied by a former line-drawing character, generates the thorns shown in the picture.

```
⎕wi 'xText' 1 1 (⎕av[222 254+⎕io])
```



Þþ

Highlights

You can also apply highlights to the display in a cell. Highlights come in two flavors; font styles and coloring. You specify a highlight by placing its name or value in square brackets at the beginning of the format phrase, for example `'[bold],0.00'` would generate bold numbers with the thousands separated and two digits after the decimal mark. In some cases, you must prefix the name, and in all cases a value, with a *keyword*=.

Font Styles

You can specify any of these style highlights to the text in a formatted cell: `[bold]`, `[italic]`, `[underline]`, or `[strikethrough]`. In addition, you can specify `[normal]`; this highlight specification has the effect of neutralizing any non-default settings for the cell in the xFontStyle property.

Colors

You can specify as a highlight the background or foreground (text) color in a cell. The eight colors that are specified by combinations of all-or-nothing for the three (*blue*, *green*, *red*) components of a color are named. Thus, you can use any of these names at the beginning of a format phrase to color the text: `[black]`, `[red]`, `[green]`, `[yellow]`, `[blue]`, `[magenta]`, `[cyan]`, `[white]`. Equivalently, you could specify these as `[text=`*colorname*`]`. You can specify the background color of a cell as a highlight using `[back=`*colorname*`]`.

You can also specify these or any other valid color by using the *keyword*= syntax and defining the color value, but color values for grid highlights are specified differently than for the color properties.

Color Values

You specify a formatting highlight color using three pairs of hexadecimal digits, each pair defining the valid range for the components *blue*, *green*, *red*, in that order. Note that this is the same order you use when calculating a single number composite value, but the opposite order that you use to specify general APL64 color values when you specify the three components separately. The range for two hexadecimal digits, `00` through `FF`, is the same as the decimal range `0-255`.

The named colors correspond to these values: `[black=000000]`, `[red=0000FF]`, `[green=00FF00]`, `[yellow=00FFFF]`, `[blue=FF0000]`, `[magenta=FF00FF]`, `[cyan=FFFF00]`, `[white=FFFFFF]`. You can calculate any other color value by modifying the statements below that use the default background color of a header cell as the example:

If you query the color:

```
          ,⍉'0123456789ABCDEF'[1+16 16 ⊤ 256 256 256 ⊤ (⎕wi 'xColorBack' ¯1 1)
      ]
      C8D0D4
```

If you know the composite color value:

```
          ,⍉'0123456789ABCDEF'[1+16 16 ⊤ (256 256 256 ⊤ 13160660)]
      C8D0D4
```

If you know the individual color component values (*blue*, *green*, *red*):

```
          ,⍉'0123456789ABCDEF'[1+16 16 ⊤ 200 208 212]
      C8D0D4
```

Numeric Formatting – Negative Values

The default format for a negative number when the xFormat property is not set is a single minus sign (hyphen) to the left of the numerals. If you set xFormat with a single format phrase for positive numbers that does not include either of the negative whitespace/control codes (minus sign or parentheses), the default for negative numbers is to use the same format pattern as for the positive with the minus sign preceding the numerals. If the positive format phrase does include either or both minus sign and parenthesis (note that the parentheses do not have to match), then the grid generates the negative format using the negative control characters as literals (while creating white space in the corresponding location for positive numbers).

If you specify anything in the second format phrase, that is, if you have anything after a semicolon and before a second semicolon, that phrase determines the negative format.

Programming Alert: A space is a literal character in a format phrase. If you want a special zero value without specifying a negative format phrase, thus requiring a format phrase in the third position, you must use two semicolons, and they must be adjacent. The format specification `'(0.0);;"nada"'` formats any negative value inside parentheses with the decimal and a value for tenths. However, the format specification `'(0.0); ;"nada"'` leaves any cell with a negative value apparently blank (the text would be a single space).

All of the discussion in the section above on the positive format phrase, including digit selectors, control codes, literals, spacing, and highlights apply to the negative format phrase, except that the minus sign and any parentheses appear as literals rather than white space. Note that if you specify a negative format phrase, it must include something to indicate the numbers are, in fact, negative. If you do not use one of the control code/literals, you should use some word, symbol or highlight. The format specification `'(0.000);0.000'` displays the absolute value of any number to three decimal places, but gives no indication other than a slightly offset spacing, whether a number is positive or negative.

Numeric Formatting – Zero Values

The default format for a zero when the xFormat property is not set is a single numeral zero with no decimal mark. If you set xFormat, the default for a zero value is to use the same format pattern as for positive numbers. If you have only octothorpe codes to the right of the decimal code, the decimal character does not appear if there is no significant fractional part of the number. If you have any zero selector codes to the right of the decimal, the decimal mark and the same number of zeroes display as there are codes. If you have a zero digit selector to the left of the decimal control code, a single zero shows in the units position. If you have zero to the left of the exclamation point decimal control code and blank or octothorpes to the right, the decimal mark displays after a numeral. Any literals or spacing specifications are honored.

If you specify anything in the third format phrase, that is, if you have anything after a second semicolon and before a third semicolon, that phrase determines the zero format. Note that the octothorpe digit selector is essentially meaningless in a zero format phrase.

If you include parentheses or a minus sign in the zero format phrase, they display. If you want to apply the same spacing in the zero format phrase, you must precede each parenthesis or minus sign with an underscore, which creates an equivalent white space but does not display the character following it.

The more likely use for a zero format phrase is to display a literal word or series of characters to make the non-positive, non-negative condition more obvious. You can, of course, use highlights. The following format phrases produce the same result: `'0.0;;("not"-"one")'` and `'0.0;;"(not-one)"'`

Numeric Formatting – Missing Values

By default, there is nothing to format when no value exists. If you want to draw attention to the absence of a value, you specify a format phrase after a third semicolon, whether or not there is any content to the second and third format phrases. You can supply a literal value, spaced as you desire, and you can apply highlights. If you desire, the missing value format phrase could merely set the background color of the cell. The example below shows some of the possibilities discussed above, including some meaningful and some silly spacing.

```
⎕wi 'xFormat' (⍳4) 2 '"★★"(0.000) "★";[red]-0.000_) _★;"none"   _);[back=cyan]'
```

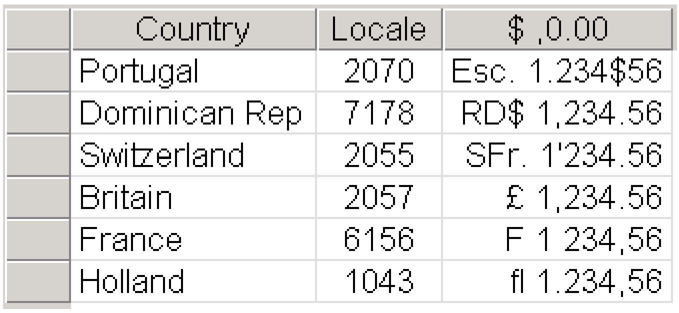
Currency Formatting

A numeric cell has xCellType `0` and xValueType `4`; you can set it with the xCurrency property or the xValue property. By default, a currency cell acts exactly like a numeric cell. You indicate the meaning of currency with any of a single, double, or triple currency control code. As with the period for the decimal control code, the grid uses `$` or `$$` or `$$$` as its currency control codes, but the symbol that appears depends on the locale.

Single Currency Control Code

If you use a single currency control code in a format phrase, it can go before or after the number. You can use other codes or literals to space the symbol. It can go inside or outside the parentheses for a negative value. When a currency value displays, the grid places one or more characters before or after the number. The currency symbol can add a lot of space to the display in a cell, depending on the locale.

For example, Switzerland uses SFr, the Dominican Republic RD$, and Portugal Esc., as well as $ for the decimal mark. These are considerably wider than a single dollar sign, pound sign, uppercase F, or lowercase fl ($ £ F fl) used by other countries.



	Country	Locale	$,0.00
	Portugal	2070	Esc. 1.234$56
	Dominican Rep	7178	RD$ 1,234.56
	Switzerland	2055	SFr. 1'234.56
	Britain	2057	£ 1,234.56
	France	6156	F 1 234,56
	Holland	1043	fl 1.234,56

Using the Euro Symbol

As of the time this is written, none of the locale codes specify the Euro symbol as a country's currency code. However, for the grid, APL has made the Euro symbol a literal character. You can place it in a format phrase either before or after the numeric value, without quotes or control code, spaced appropriate to your wishes.

The Euro character is `⎕av[2+⎕io]` but it is in ANSI font position `128`. You can generate the character in APL by typing Alt+`0128`. If you include the Euro as a literal, you may not want to use the currency control code.

Double Currency Control Code

While many of the currency symbols are unique, or use letters plus a symbol to distinguish them, there are duplications. France and Luxembourg both use a single uppercase F, four countries with six locale codes denote their kronor as kr (although Iceland uses a period in addition), and there are a dozen locales that use an unadorned dollar sign as their currency symbol, even though the names of these currencies are not all dollars. If your application serves multiple countries or you need to distinguish currencies from multiple countries, you can use the `$$` currency control code. This generates a code, usually three letters long, that specifies the currency in use in that locale; the code is generally two letters designating the country issuing the currency and the first letter of their currency name.

You use this control code in the same manner as the single currency control code. Its location in the format phrase determines the location and spacing of the currency designator characters in the cell.

Triple Currency Control Code

You can specify a `$$$` currency control code; this has the effect of spacing the value as currency, and using the appropriate symbol for the decimal mark (which differs from the simple numeric decimal mark in at least one locale), but displays no currency symbol. You may want to use this in columns of currency figures where you specify only the first line and total lines with the currency symbol.

Negative, Zero, and Missing Currency Values

The same considerations for negative, zero, and missing currency values apply as explained above for numeric formatting. The currency symbol, as with other literals, does not appear if there is no value assigned.

Date and Time Formatting

A date cell has xCellType `0` and xValueType `3`; it is fundamentally a numeric cell that has a stylized, left justfied text display. By default, a date cell displays whatever is typed in, whether it is a valid date designation or not. (What constitutes a valid entry depends, by default, on system settings.) If the entry can be interpreted as a valid date, the grid converts it to an integer, which it returns if you query the xDate property or the xValue property. If it is invalid, the grid returns the conversion error value.

You can also set a date cell with a time, which a user can type in as hours, minutes, and conceivably, seconds. The grid converts time to a fractional number measured as a percent of `24` hours. You can have both a date and a time in the same cell; when queried, the grid returns a floating point number with the integer value representing the day and the fractional value the time.

You can set a date cell with either the xDate property or the xValue property. You can assign a numeric value (day `1` is December `31, 1899)`, which the grid converts to a date (and/or time), or you can assign a text string that can be interpreted as a date. In either case, the grid formats the cell as a valid date according to the Regional Options settings on the host machine. You can query the xText property to see what is displayed.

The variations in default displays include the order of the elements of a date: month-day-year, day-month-year, or year-month-day; how the elements are separated: slashes, dashes, or periods; whether the leading zero appears for single digit days and months; whether time is represented as a `24`-hour clock or `12`-hour clock, and if the latter, the form of distinguishing before noon from after noon.

For a typical US setting, you can type in or specify dates as Jan `31, 99`; `2/29/00`; or March `31, 2001`, for example. If you specify a non-zero two-digit year that is less than `30` (this number is a host machine system setting), the grid assumes this century. If you specify a number greater than `29`, it assumes the `1900`s. Note that you cannot type in a numeric value and have the grid convert it to a date. If you want to assign the numeric value to the cell, you must do it under program control.

You can assign negative numbers to represent days before December 30, 1899, but note that time always works forward; that is the time represented by ¯1.4 is two hours and 24 minutes later on December 29, 1899, than the time represented by ¯1.3. This has the disquieting effect of making ¯0.5 and +0.5 both equal to noon on December 30, 1899.

What it Means to Format a Date Cell

Dates are represented in a wide variety of patterns. The month can be a number, with or without a leading zero for single digit months; a month can also be a character abbreviation or a spelled out name. The day can come before or after the month; the year can come first or last, and be represented by two or four digits. The day of the week can be included. Similarly, times can be represented with or without seconds, as 12 or 24 hour cycles, and with leading zeroes. The symbols that separate hours from minutes, or days from months and months from years vary according to local custom and personal preference.

If you leave date cells with their default formatting, you have little control over the size of the cell's content or what a user might enter. If you set a locale code (other than the default locale for the host machine), the Windows defaults for that locale prevail, and any non-standard user settings are ignored. For example, if you set the locale code to one of the values for Belgium, and the default date format for Belgium is a one- or two-digit day, followed by a two-digit month and a four-digit year, no one outside of Belgium who is running your program will see dates with the months spelled out. This also means that if you specify a date as 4/02 under program control, the grid will interpret it correctly as February 4 rather than April 2. However, if your user is running the program in Belgium, his particular preferences apply. If you set the xFormat property, you can specify the exact format pattern you want your users to see. In this case, no matter how the date is entered, the grid formats it according to your specified pattern, but how the grid interprets a typed string may depend on the user's preferences.

Date and Time Format Phrases

The xFormat property for a date cell has positions for four format phrases. As for numeric and currency cells, the first one is mandatory; however, the grid ignores anything in the second and third positions (since negative values and zero represent days, they are formatted by the pattern of the first phrase). If you want to specify a literal or highlight the cell with a background color when the value is missing, specify an appropriate format phrase after three semicolons, following the first phrase.

For the primary date/time format phrase, you can use font style and coloring highlights just as you do for numeric and currency cells. You can also use literals; however, in a date format phrase, you are not limited to the beginning and end. You can place literal values between components of the date or time. You can specify any literal by enclosing it in quotation marks, and you can use the literal control characters mentioned above. Specifically, parentheses and the minus sign are not just spaceholders, they display; so you could for example, enclose the time in parentheses following a date. You can insert spaces for white space, but the underscore control code has no visible effect, unless you switch the alignment to right justification.

Date and Time Selector Codes

Programming Alert: Date and time selector codes are case sensitive.

Date and time selector codes are alphabetic characters; for each letter that is a code, the meaning varies with the number of times you specify it consecutively. You can have both date and time codes in the same cell, and you can have the time either before or after the date, but you cannot intersperse the codes. Note that if you use a code for the day of the week, you should also pick a code for the day of the month (same selector character, different number).

Date codes you can use include:

Day of the week

dddd	Day of the week spelled out with its full name
ddd	Day of the week as a three-letter abbreviation

Day of the month

dd	Day of the month with leading zero for single digit days
d	Day of the month as one or two digits, as appropriate (no leading zero)

Month

MMMM	Month spelled out with its full name
MMM	Month as a three-letter abbreviation
MM	Month as a two-digit number, using leading zero when necessary
M	Month as a one- or two-digit number as appropriate (no leading zero)

Year

yyyy	Year represented as four digits
yy	Year represented as two digits with leading zero when necessary
y	Year represented as one or two digits as appropriate (no leading zero).

Time codes you can use include:

24-Hour Clock

HH	Hour represented as two-digit number on a `24`-hour clock (`00-23`)
H	Hour represented as one-or two digit number on a `24` hour clock (`0`, `1`, ... `23`)

12-Hour Clock

hh	Hour represented as two-digit number on a `12`-hour clock (`00-12`)
h	Hour represented as one-or two digit number on a `12` hour clock (`0`, `1`, ... `12`)
t	One character time-marker string, such as A or P
tt	Multi-character time-marker string, such as AM or PM; a.m. or p.m.; blank or nm

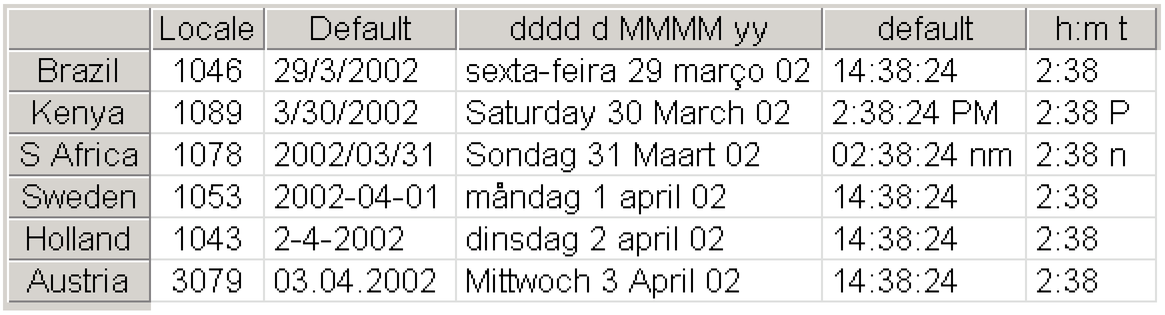
Minutes

mm	Minutes represented as two-digit number (`00-59`)
m	Minutes represented as one- or two-digit number (`0`, `1`, ...`59`)

Seconds

ss	Seconds represented as two-digit number (`00-59`)
s	Seconds represented as one- or two-digit number (`0`, `1`, ...`59`).

Examples:
Note that it is dangerous to be careless. If you specify a `12`-hour clock for a locale that uses a `24`-hour cycle by default, there is no system for distinguishing which half of the day you are representing; you can, however, use a `24`-hour clock for any locale.



	Locale	Default	dddd d MMMM yy	default	h:m t
Brazil	1046	29/3/2002	sexta-feira 29 março 02	14:38:24	2:38
Kenya	1089	3/30/2002	Saturday 30 March 02	2:38:24 PM	2:38 P
S Africa	1078	2002/03/31	Sondag 31 Maart 02	02:38:24 nm	2:38 n
Sweden	1053	2002-04-01	måndag 1 april 02	14:38:24	2:38
Holland	1043	2-4-2002	dinsdag 2 april 02	14:38:24	2:38
Austria	3079	03.04.2002	Mittwoch 3 April 02	14:38:24	2:38

Appendix – Locale Codes

By Country

The table below provides the locale ID codes for various countries/languages; you can use the numeric code to set the xLocale property. Note that these codes are valid as of 2001, when European countries had individual currencies. You can specify the Euro symbol as the currency identifier by using an appropriate literal string in the xFormat property.

Selected Locale Codes for Formatting Currency and Dates

Country	Language	Code	Country	Language	Code
Argentina	Spanish	11274	Liechtenstein	German	5127
Austria	German	3079	Luxembourg	French	5132
Australia	English	3081	Luxembourg	German	4103
Belgium	French	2060	Malaysia	Malay(sian)	1086
Belgium	Dutch	2067	Mexico	Spanish	2058
Belize	English	10249	Monaco	French	6156
Bolivia	Spanish	16394	Netherlands	Dutch	1043
Brazil	Portuguese	1046	New Zealand	English	5129
Brunei	Malay	2110	Nicaragua	Spanish	19466
Canada	English	4105	Norway	Norwegian-Bokmål	1044
Canada	French	3084	Norway	Norwegian-Nynorsk	2068
Carribbean	English	9225	Panama	Spanish	6154
Chile	Spanish	13322	Paraguay	Spanish	15370
Colombia	Spanish	9226	Peru	Spanish	10250
Costa Rica	Spanish	5130	Philippines	English	13321
Denmark	Danish	1030	Portugal	Portuguese	2070
Dominican Rep	Spanish	7178	Puerto Rico	Spanish	20490
Ecuador	Spanish	12298	South Africa	Afrikaans	1078
El Salvador	Spanish	17418	South Africa	English	7177
England (see UK)			Spain	Basque	1069
Faeroe Islands	Faeroese	1080	Spain	Catalan	1027
Finland	Finnish	1035	Spain	Spanish (traditional)	1034
Finland	Swedish	2077	Spain	Spanish (modern sort)	3082
France	French	1036	Sweden	Swedish	1053
Germany	German	1031	Switzerland	German	2055
Guatemala	Spanish	4106	Switzerland	French	4108
Holland (see Netherlands)			Switzerland	Italian	2064
Honduras	Spanish	18442	Trinidad	English	11273
Iceland	Icelandic	1039	United Kingdom	English	2057
Indonesia	Indonesian	1057	United States	English	1033
Ireland	English	6153	Uruguay	Spanish	14346
Italy	Italian	1040	Venezuela	Spanish	8202
Jamaica	English	8201	Zimbabwe	English	12297
Kenya	Swahili	1089	(default to user's host machine setting)		1024

By Code

Selected Locale Codes for Formatting Currency and Dates

Code	Country	Language	Code	Country	Language
1024	(default to user's host machine setting)*		3084	Canada	French
1027	Spain	Catalan	4103	Luxembourg	German
1030	Denmark	Danish	4105	Canada	English
1031	Germany	German	4106	Guatemala	Spanish
1033	United States	English	4108	Switzerland	French
1034	Spain	Spanish (traditional)	5127	Liechtenstein	German
1035	Finland	Finnish	5129	New Zealand	English
1036	France	French	5130	Costa Rica	Spanish
1039	Iceland	Icelandic	5132	Luxembourg	French
1040	Italy	Italian	6153	Ireland	English
1043	Netherlands	Dutch	6154	Panama	Spanish
1044	Norway	Norwegian-Bokmål	6156	Monaco	French
1046	Brazil	Portuguese	7177	South Africa	English
1053	Sweden	Swedish	7178	Dominican Rep	Spanish
1057	Indonesia	Indonesian	8201	Jamaica	English
1069	Spain	Basque	8202	Venezuela	Spanish
1078	South Africa	Afrikaans	9225	Carribbean	English
1080	Faeroe Islands	Faeroese	9226	Colombia	Spanish
1086	Malaysia	Malay(sian)	10249	Belize	English
1089	Kenya	Swahili	10250	Peru	Spanish
2055	Switzerland	German	11273	Trinidad	English
2057	United Kingdom	English	11274	Argentina	Spanish
2058	Mexico	Spanish	12297	Zimbabwe	English
2060	Belgium	French	12298	Ecuador	Spanish
2064	Switzerland	Italian	13321	Philippines	English
2067	Belgium	Dutch	13322	Chile	Spanish
2068	Norway	Norwegian-Nynorsk	14346	Uruguay	Spanish
2070	Portugal	Portuguese	15370	Paraguay	Spanish
2077	Finland	Swedish	16394	Bolivia	Spanish
2110	Brunei	Malay	17418	El Salvador	Spanish
3079	Austria	German	18442	Honduras	Spanish
3081	Australia	English	19466	Nicaragua	Spanish
3082	Spain	Spanish (modern sort)	20490	Puerto Rico	Spanish

***Note:** If there is more than one user profile for a machine, or if the user has changed his location, there may be two different "defaults" on the machine. The user default locale is the current setting in the Regional Options location box, and there is also a system default locale, which corresponds to the setting when Windows was installed. The code for the latter is `2048`, but it is not clear that you can use this setting to any advantage in the grid.

Glossary

The terms listed below are used in this document to describe the APL Grid. They are either terms that are used in a general sense with a particular application to the grid, or terms that have multiple meanings with regard to the grid. This list does not include all the specific terms used for grid features, which are defined when described; for example, to see the definition for a merged cell, see the description of the xMergedCells property.

active area
The space within the client area of the grid below and to the right of header cells, where regular cells can be seen. This area is above the bar that holds the page tabs and, if needed, the horizontal scroll bar. It is the area colored by the xColorGrid property when there are no rows and columns.

active cell
The cell, which often has a heavy black border on all four sides, that usually has the selection cursor, when the focus is on the grid. This cell usually is, or would be, the location of any user activity. For example, under most circumstances, the active cell receives the next keystroke, is the one identified in `⎕warg` during many event handlers, and is the base from which a movement would originate when transversing or scrolling the grid.

active page
The currently visible page, to which many property settings, and most methods and events, apply. Also referred to as the current page.

anchor cell
The upper left corner cell in a contiguous range of cells. This term is used for selection, view, and merged (or joined) cells. See also "selection" below for defining a range.

attribute
A setting for a cell that affects its behavior or appearance, but is not the value (contents) of the cell.

block (number)
In a general sense, a block can simply be a rectangular area of contiguous cells; the same as a range. More specifically, when discussing the xSelection property, and its associated methods and events, a block is one such range of cells when there are multiple blocks selected simultaneously. The xSelection property is a two-dimensional array, where each row is a block of selected cells. The blocks are numbered (conceptually) in the order in which they are selected, and each row is added at the top of the matrix. Block number one is the last row of the current xSelection matrix, and the highest block number, which corresponds to the first row of the matrix, is equal to the number of rows in the matrix. The block number is sometimes needed as an argument.

Boolean cell
A normal (xCellType `0`) cell with xValueType set to `2`. The value of such a cell is numeric; however, non-zero values are converted to `1`. You can also use the named constants `True` and `False`.

corner cell
The cell that exists at the junction of the row(s) of column headers and the column(s) of row headers. This cell has coordinates of `¯1 ¯1`. If the user clicks it, it selects all the regular cells in the grid.

currency cell
A normal (xCellType `0`) cell with xValueType set to `4`. The value of such a cell is numeric; you can use the xFormat property with a currency control code in the format phrases to display the numeric value as currency.

current page
The currently visible page to which most actions apply; the active page.

date cell
A normal (xCellType `0`) cell with xValueType set to `3`. The value of such a cell is numeric; however, the display is text, formatted according to default Control Panel/Regional Options settings on the host machine, according to the country/language patterns determined by the xLocale property, or by an explicit format phrase in the xFormat property. A date cell can hold a value that represents just a date, just a time of day, or a combination of both.

extents
The number of rows and number of columns of a rectangular array of contiguous cells. These are generally the third and fourth arguments specifying such an array, the first two arguments being the cell coordinates of one corner of the range. This form of specification is used for selections, merged cells, view, some XML properties, and methods such as XRedraw; in the text it is sometimes referred to as a selection vector.

focus
Whenever you are working on the grid, the focus, in its normal Windows sense is on the grid itself; that is, if you query the focus property of the system object in a grid event handler, it will return the name of the grid. However, in this document, the term is also used to identify the movement when the active cell is changing.

header cell
A cell that is not part of the regular grid, appearing above a column or beside a row. By default there is one row of column headers and one column of row headers, but you can specify more. You identify headers with negative numbers. You can set attributes of header cells and assign text values (that usually serve as labels for the corresponding row or column), but a user cannot edit a header cell. Header cells have some defaults that differ from regular cells.

highlights
The general meaning of highlight in a computer interface refers to the technique of giving selected text a dark background and a light foreground (color of the text itself); the term is used in this document with that meaning. It also has a special meaning when using the xFormat property, where it refers to control codes in format phrases that assign background or foreground colors to the cell being formatted or produce font styles to the text therein.

modifier keys
Keyboard keys whose status of being pressed changes the behavior of mouse or keyboard actions; the status is returned in `⎕warg` for certain events. The value is usually the sum of `1`=Shift, `2`=Ctrl, and `4`=Alt.

normal cell
A cell that can hold text or numbers or specially formatted numbers, such as dates and currency. Such a cell has xCellType zero, if it is other than the default (`¯1`). A normal cell is distinguished from special cell types whose xCellType `>0`; these include Combo box, Check box, Arrow, and Button cells. See also regular cell.

numeric cell
A normal (xCellType `0`) cell with xValueType set to `1`. The value of such a cell is numeric; you can use the xFormat property to specify the number of digits after the decimal, to format numbers with a thousands separator, to create custom negative number formats, and to display unique text for missing values or zero.

place code
A value in the range `1-5` that defines a location within a cell for an image. This code functions as a coordinate argument preceding the row and column when referencing or assigning the xImage property. The corresponding value for the specified place in the specified cell is an index to the image you want to appear. The first four values specify the four corners: upper left, upper right, lower left, lower right. The fifth value takes its meaning from the current setting of the xAlign property; this allows you to position an image in the center of the cell or centered on any of the four edges.

range
A rectangular area of contiguous cells. A range is generally defined by specifying the coordinates of one cell and the row extent (height, or number of rows including the specified cell) and column extent (width, or number of columns including the specified cell). This differs from an array of cells that is identified by a vector of row indices and a vector of column indices. An array of cells, which you specify, for example, when setting cell attributes and values, is not necessarily contiguous, nor are consecutively specified cells in the range necessarily in the same order as they appear on the screen.

A range may also be called a block, but block has a more specific meaning as well.

regular cell
A cell within the active area of the grid, identified by row and column indices that are greater than zero. A regular cell is distinguished from a header cell. Regular cells include both normal and special cell types.

selection
A selection has a distinct meaning of being one or more cells that are currently highlighted; the selection is identified in the xSelection property. Selection also has the meaning of identifying a contiguous range of cells in a manner that is like the where property of a regular control with cells as the scaling units (unit of measure). That is, the range of cells is identified by row and column indices of the upper left (anchor) cell and the number of rows and number of columns that the range comprises. These two values, which correspond to the height and width of the range are also known as row and column extents. This form of specification is used not only for each row of the xSelection property matrix, but also for the xView, xMergedCells, xXMLRange, xXMLTable, and xXMLValueRange properties, as well as arguments to a number of methods and identification in `⎕warg` during certain event handlers.

special cell
A special cell (special purpose) has xCellType `>0`; these include Combo box, Check box, Arrow, and Button cells. The adjective special is also applied to alternate windows you can cause to appear in place of default grid facilities and to cells where a normal value has a non-default display format, such as date or currency.

ternary
Three possibilities: Some properties that are flags on a cell-by-cell basis return `¯1` if the value has not been set explicitly. The descriptions of these flags use the term "ternary" to describe the value. You can use a Boolean value (two possibilities) to set one of these flags, or you can use `¯1` (the third possibility) to clear an explicit setting; this causes the cell to inherit either a column default or a page default, if one exists. Lacking a default setting, the cell exhibits its default behavior, which may match the behavior of one of the Boolean settings.

time cell
There is no such thing as a time cell; you can specify a time and format its display in a date cell.

Index

Properties and methods are indexed minus their leading "x" or "X"; event-handler properties are indexed under the event name, that is, without "on" and minus the leading "X". Alphabetization in the index is each letter lowercase before uppercase; therefore, the order of terms differs from the order in the properties, methods, and event-handlers reference sections.