DYALOG APL

The tool of thought for expert programming

Dyalog Release Notes

Version 14.1

Contents

Chapter 1:

Introduction

Key Features

Dyalog APL Version 14.1 provides the following new features, enhancements and changes:

Performance Improvements

Version 14.1 includes a considerable amount of research and development work designed to substantially improve speed of execution. See *Performance Improvements* on page 31.

Language Enhancements

New Language Features

- New `:Disposable ... :EndDisposable` control structure for the automatic disposal of unwanted .NET objects and resources. See *Disposable Statement* on page 45.

New I-Beam Features

- A new I-Beam is provided to execute expressions under program control, which handles shy results differently than the primitive function monadic Execute. See *Execute Expression* on page 59.
- A new I-Beam is provided to overwrite unused memory in the workspace in order to erase potentially secure data. See *Overwrite Free Pockets* on page 60.
- A new I-Beam is provided to determine whether or not the current function was called monadically. See *Called Monadically* on page 61.
- A new I-Beam is provided to report the names of dynamic link libraries loaded by ⎕NA. See *Loaded Libraries* on page 62.
- A new I-Beam is provided to make a thread uninterruptible. See *Mark Thread as Uninterruptible* on page 66.

- A new I-Beam is provided to spawn a .NET thread from APL thread 0. See *Use Separate Thread For .NET* on page 67.
- A new I-Beam function provides a faster way to reference early elements of ⎕LC. See *Line Count* on page 81.
- Four new I-Beams are provided to import and export text in JavaScript Object Notation (JSON) Data Interchange Format[1]. See *I-Beam Changes* on page 57.

Extensions

- There are 2 new options for Edit Object (⎕ED). See *Variants of Edit Object* on page 52.
- The arbitrary input and output system functions have been simplified and re-engineered, primarily for use in non-Windows environments. See *Arbitrary Input* on page 48 and *Arbitrary Output* on page 50.

Component File System Improvements

- File properties S and U have been added to the properties that may be set when the file is created using a function derived from ⎕FCREATE and the Variant operator ⎕. See *File Create and Variant* on page 35.

IDE Enhancements

- The editor can now be used to view the values of α, αα, ω and ωω. See *Tracing dfn Arguments* on page 37.
- *Align Comments* has been extended to scripted objects. See *Aligning Comments in Scripts* on page 37*Aligning Comments in Scripts* on page 37.

GUI Enhancements

- Version 14.1 provides a new coordinate system that automatically scales GUI windows and controls created by ⎕WC according to the scale factor set by the user's Desktop Windows Manager. See *High DPI Support* on page 3.
- Version 14.1 provides support for Windows Gestures. A workspace gesturedemo.dws is provided to illustrate how to use this new feature (touch-screen required). See *Gestures* on page 8.
- The Masked property has been extended to provide greater portability for ImageLists. See *Masked* on page 109.
- WPF Data Binding now supports matrices. See *WPF Data Binding Using Matrices* on page 12.

---

[1]IETF RFC 7159

High DPI Support

Modern high resolution screens present some practical challenges to a Graphical User Interface that was designed when lower-resolution screens were the norm. When you increase resolution you inherently decrease the size of each pixel (assuming same display size). By decreasing the size of each pixel the content shown on the display appears smaller. When display Dots-Per-Inch (DPI) gets sufficiently dense this shrinking effect can make content hard to see and user interface components such as menus and buttons, difficult to click/tap.

Also, people have different preferences and Windows enables the user to change the DPI setting.
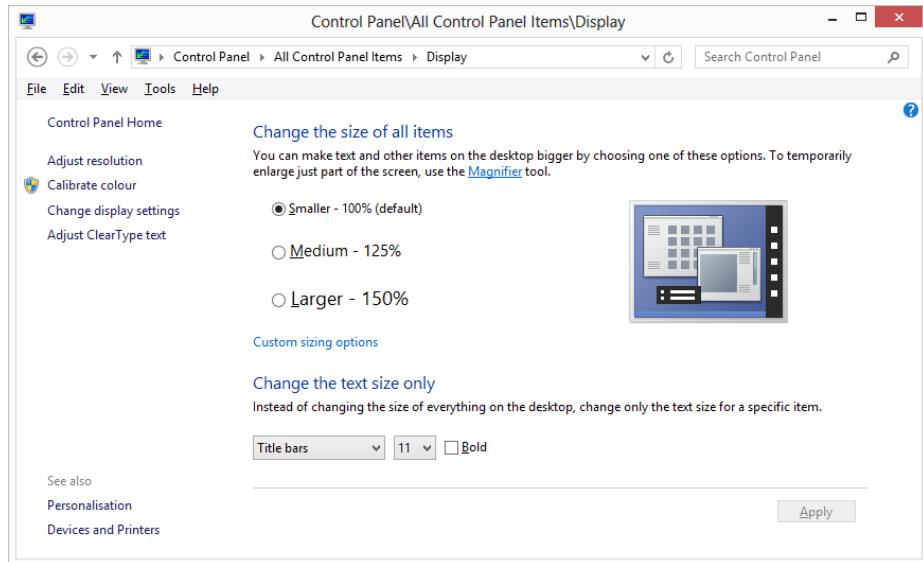
To address this issue, the Desktop Window Manager, which is enabled in Windows Vista and above, automatically scales up windows and their content to match the current DPI setting.

The problem with this approach is that, because the scaling is implemented by bitmap stretching, application user-interfaces windows tend to look fuzzy and/or distorted.

DPI-Awareness

To prevent the DWM from stretching its user-interface, a Windows application can declare itself to be DPI-Aware. If so, the application is expected to handle DPI issues itself. Whether it does so by scaling GUI components according to the DPI in use, or not, is up to the application itself. If an application chooses to register itself as DPI-aware, but fails to scale its GUI components on a high DPI device, they will simply appear physically smaller on the screen.

An application can declare itself as being DPI-Aware by making a system call or by making a declaration in the optional XML manifest file that may be associated with its `.exe`.

Dyalog APL will register itself as being DPI-Aware on startup if the value of the **AUTODPI** parameter is 1. This is the default, so in a standard developer installation, Dyalog APL itself and all Dyalog applications driven by the development and run-time versions of Dyalog are by default registered as DPI-Aware, so DPI scaling by the DWM is disabled.

This can be changed by setting the **AUTODPI** parameter to 0 (or by removing it) or using a declaration in a manifest file. See .

Coord Property

Dyalog APL includes a mechanism to automatically scale a pixel coordinate user-interface according to the DPI setting. This works by introducing two new coordinate types named *ScaledPixel* and *RealPixel* and by changing the way that the existing *Pixel* coordinate type is interpreted.

ScaledPixel means that the number of pixels specified will be automatically scaled by Dyalog APL according to the user's chosen display scaling factor. ScaledPixel also means that Dyalog will automatically de-scale coordinate values reported by `⎕WG` and coordinate values in event messages.

RealPixel means that Dyalog APL will precisely honour the number of pixels you specify and will apply no scaling. GUI windows and components will simply appear physically smaller on higher DPI devices.

The Dyalog Session uses Coord `'ScaledPixel'` and all the GUI components of the Session are therefore DPI-scaled by Dyalog itself.

Pixel Coordinates and DYALOG_PIXEL_TYPE

Dyalog Versions prior to Version 14.1 did not support ScaledPixel and RealPixel options; just Pixel. Rather than force users to change all pixel coordinate types in legacy applications, Dyalog provides a parameter named **DYALOG_PIXEL_TYPE** whose value is either ScaledPixel or RealPixel. If the value of the Coord property is `'Pixel'` this is interpreted as meaning whichever value is specified by **DYALOG_PIXEL_TYPE**.

If the **DYALOG_PIXEL_TYPE** parameter is not specified (the default), it defaults to RealPixel. So by default, Coord `'Pixel'`, will be treated as RealPixel and your Dyalog APL GUI application will simply appear physically smaller on higher DPI devices.

DYALOG_PIXEL_TYPE may be set to ScaledPixel by ticking the check-box on the General Tab the Configuration Dialog box labelled *Enable DPI Scaling of GUI application*.

If this check-box is cleared the **DYALOG_PIXEL_TYPE** parameter will be removed from the current user's registry.

Using ScaledPixel coordinates, if you specify an Edit object to be 80 units wide and 20 units high, and the user's scaling factor is 150%, Dyalog will automatically draw it 120 pixels wide and 30 pixels high. You won't have to change any of your code that handles the Edit, it will just appear larger on the screen than if it hadn't been scaled. Similarly, if you use the ScaledPixel coordinate type for the Font object, the font used to draw text in the object will automatically be scaled for you.

Font Object

The Font object has a Coord property which may be set to `'Pixel'`, `'ScaledPixel'` or `'RealPixel'` when the object is created, but may not subsequently be changed. Note that the Font object does not support other Coord values. `'Pixel'` is treated as `'ScaledPixel'` or `'RealPixel'` as discussed above.

If you are using `'ScaledPixel'`, this means that your fonts will also be scaled up automatically, as well as the sizes of the controls in which they are used.

Set Dialog Pixel Type (`2035ɪ`)

This function provides the means to set the meaning of `Coord 'Pixel'` programmatically and dynamically. This function affects the way that Pixel coordinates are subsequently treated. For further information, see *Language Reference: Set Dyalog Pixel Type*.

Enabling DWM Scaling

The DPI-Aware scaling features provided by Dyalog APL are designed to allow you to deploy GUI applications that look attractive in most situations, whatever the screen resolution and scaling factor is in use.

However, if you wish to ignore these facilities and fall back on Windows DWM scaling, you may do so as follows.

If you wish to *enable* DWM scaling in your application, you can either remove or set to zero the **AUTODPI** parameter. For example, the command line to start a run-time application might be:

```
dyalogrt.exe myruntime.dws AUTODPI=0
```

This will prevent Dyalog from registering your application as DPI-Aware in start-up.

Another way to enable DWM scaling is to use a manifest file. Note that if you disable DWM scaling for the development version of Dyalog APL, the appearance of the Session window may be imperfect.

Using a Manifest

A Windows application can declare itself to be DPI-aware or not using a declaration in the optional XML manifest file associated with its .exe. If you want your Dyalog APL application to be automatically scaled by the DWM, you may use a manifest file to override the call that Dyalog itself makes to register itself as being DPI-Aware.

This is done by setting the XML entilty *dpiAware* to the value *false* as illustrated by the skeleton manifest file listed below.

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersio
n="1.0" xmlns:asmv3="urn:schemas-microsoft-com:asm.v3" >
 <asmv3:application>
   <asmv3:windowsSettings xmlns="http://schemas.microsoft.com/SMI
/2005/WindowsSettings">
     <dpiAware>false</dpiAware>
   </asmv3:windowsSettings>
 </asmv3:application>
</assembly>
```

If *dpiAware* appears in the manifest file, its value take precedence over the value of the **AUTODPI** parameter, whether it is specified implicitly by omission (it defaults to 1) or is specified in the registry or on the command line.

Naming a Manifest File

The name of the manifest file is the full name of the application file followed by an optional resource id (if omitted, the default is 1) and the extension `.manifest`. If your application runs courtesy of the dyalogrt.exe or dyalogrt.dll, the name of the manifest file should be one of:

```
dyalogrt.exe.<resource ID>.manifest
dyalogrt.dll.<resource ID>.manifest
```

If you have exported your application as an executable called example.exe or or as a dll called example.dll, it should be one of:

```
example.exe.<resource ID>.manifest
example.dll.<resource ID>.manifest
```

Gestures

Introduction

Gestures are user interactions that are most commonly generated by touching and moving fingers on the screen, although other means (e.g. touch pad, stylus) may be used to perform the same operations.

The following Gestures are supported by Dyalog APL:

Gesture	Description
Pan	The user touches one or two fingers on the screen and drags or swipes them.
Zoom	The user touches two fingers on the screen and moves them towards each other (zoom out) or away from each other (zoom in).
Rotate	The user touches two fingers on the screen and then twists them as if turning a knob.
Tap	The user taps (touches briefly) one or two fingers on the screen.
Press and Tap	The user presses one finger on the screen, then taps the screen with a second finger, while the first finger remains in contact with the screen.

Gesture Events

Gestures generate GUI events, which are summarised in the table below.

Apart from GestureTwoFingerTap, which generates a single event, Gestures generate a series of events of the same type. The first of these is flagged as a starting event. Then there are a series of one or more continuation events, followed finally by one that is flagged as the final event in the series. Each series consists of events of the same type and no other type of event will be reported between the start and end of that series.

Event	Description
GesturePan	One or more of these events are generated by a Pan Gesture.
GestureRotate	One or more of these events are generated by a Rotate Gesture.
GestureZoom	One or more of these events are generated by a Zoom Gesture.
GesturePressAndTap	This event is generated by a Press and Tap Gesture.
GestureTwoFingerTap	This event is generated by a Tap Gesture using two fingers.

Handling Gestures

Gestures do not in themselves *do anthing* by default, but if left unhandled, may generate other events which do *do something* by default. For example, an unhandled Pan Gesture on a scrollbar will, by default, cause it to scroll.

An application can choose to handle Gesture events in a specifically application-dependent manner or choose to ignore them, relying on objects to respond to mouse or scroll events (which are generated by unhandled Gesture events) as appropriate.

In Dyalog APL, a Gesture event is handled by attaching a callback function which responds to the event in some way. The result of the callback function is important. The value 0 tells the Operating System that the application has handled (consumed) the event and instructs Windows NOT to take any further action. The value 1 means that the application has not taken action and instructs Windows to do whatever it would normally do in response to the Gesture; for example, to treat it as a mouse or scroll operation.

If you attach a callback to the GesturePan event which responds by, for example, moving the object, the callback should return 0. See also: .

Inertia

When a Pan gesture is made, the operating system may generate additional GesturePan events depending upon the speed with which the Gesture has been made. For example, if the user makes a short but rapid swiping motion with a finger, inertia generates GesturePan events over a greater distance than the finger was actually in contact with the screeen. Information as to whether or not the event was generated by inertia is provided in the event message.

High-Priority Callback Functions

A high-priority callback function is one that is invoked by a high-priority event which demands that Dyalog must return a result to the operating system before it may process any other event. Such high-priority events include Configure, ExitWindows, DateTimeChange, DockStart, DockCancel, DropDown, GetTipText, GesturePan, GestureZoom, GestureRotate, GestureTwoFingerTap, GesturePressAndTap.

If a high-priority callback function is traced or stops for any reason, the system is *in limbo* until the windows notification has been actioned. This will occur only when the callback exits. During this time, it is not possible to reset the state indicator or save the workspace. In the following example, there is a deliberate error on `GenCB` `[2]` which is assigned as the callback function for the GesturePan event on object `f.s1`.

```
      f.s1.onGesturePan←'GenCB'

    ∇ GenCB m
[1]   m
[2]   ∘
    ∇
```

[user drags finger in object]

```
 #.f.s1  GesturePan  1  84 103  0 0
SYNTAX ERROR
GenCB[2] ∘
         ^
      )si
#.GenCB[2]*
⎕DQ
      →
DOMAIN ERROR: Operation cannot be completed with an "exte
rnal" call on the stack
      →
    ^
      )reset
Can't )RESET with external call on the stack.
      )clear
Can't )CLEAR with external call on the stack.
      )save
Cannot perform operation with calls to or from external f
unctions or certain callbacks.
```

The only way to restore the situation to normal is to force the callback function to exit. For example:

```
      →0
      )si
```

Furthermore, it is therefore not permitted to use a `:Hold` control structure in a high-priority callback function and Dyalog cannot perform thread-switching during the execution of a high-priority callback.

WPF Data Binding Using Matrices

WPF Data Binding has been extended to allow you to bind APL matrices. The example shown later in this section, Example 8, is an addition to the existing documentation on WPF Data Binding. See *.NET Interface Guide: Windows Presentation Foundation*.

Binding a Matrix

Binding a matrix is like binding a vector of namespaces. Each row of `Y` represents one of a collection of instances of an object, which exports a particular set of properties for binding purposes. Each column of `Y` represents one of these properties.

Every row in the datasource will be of the same type (which might not be the case with an array of namespaces), and so the collection is a collection of specific things. The *specific thing* is a .Net type that is created dynamically and has a unique name.

Unlike variables in namespaces, the columns of an APL matrix do not have names which can be exported as properties, so this information must be provided in the left argument to `(2015⍳)` which also specifies their data types. If the left argument is omitted, the default names are `Column1`, `Column2`, ... and so forth and the default data type is System.Object.

So if the right argument of `(2015⍳) Y` is the name of a matrix, the left argument `X` is a matrix with as many rows as there are columns in `Y`. `X[;1]` contains the names by which each of the columns of `Y` will be exported as a property, and `X[;2]` their data types.

Values in the matrix may be scalar numbers, character scalars or vectors, or nested vectors, but each column in the matrix must be uniform.

The result `R` is a specific type that is created dynamically and assigned a unique name of the form Dyalog.Data.DyalogCollectionNotifyHandler`1 [Dyalog.Data.DataBoundRow_nnnnnnnn]. This is suitable for binding to a WPF property that requires an IEnumerable implementation, such as the ItemsSource property of the DataGrid.

Example

`mat` is a matrix of numbers and is bound with default property/column names `Column1`, `Column2`, … `Column10` and the default data type of System.Object.

```
      mat←?20 10ρ100
      bindSource←2015ɪ'mat'
```

Example

`winelist` is a matrix whose first column contains a list of wines, and whose second column their prices. The left argument is a matrix. Its first column specifies the property names by which the columns will be exported (`'Name'` and `'Price'`) and its second column , their data types (System.Object)

```
 winelist←Wines,[1.5]0.01×10000+?(ρWines)ρ10000
 info←(⍪'Name' 'Price'),⊂Object

 bindSource←info(2015ɪ)'winelist'
```

Example

`emp` is a 3-column matrix which contains names, numbers and addresses. Each address is made up of two character vectors containing street and town

`emp`

John Smith	Mary White	T.W. Penk
1	2	3
2 East Rd | Headley	42 High St | Alton	23 West St | Farnham

`schema`
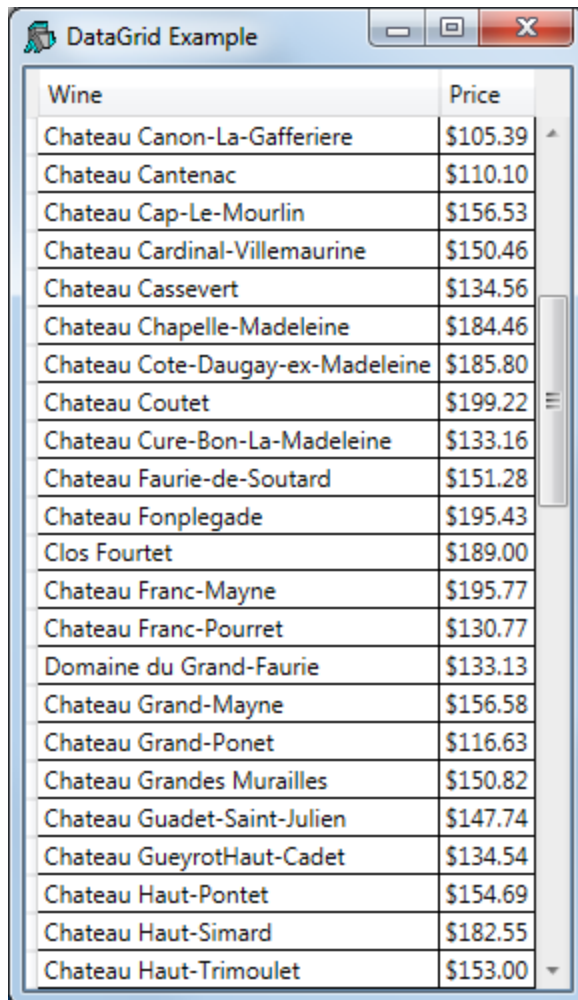
Name	(System.Object)	
Number	(System.Object)	
Address	Street	(System.Object)
	Town	(System.Object)

```
      bindSource←schema(2015ɪ)'emp'
```

Example 8

This example illustrates data binding using a matrix and is practically identical to Example 7 except that it uses a matrix instead of a vector of namespaces.

Each row in the WPF `DataGrid` control is represented by an object, and each column as a property of that object. Each row in the `DataGrid` is bound to an object in the data source, and each column in the data grid is bound to a property of the data object.

The XAML

The XAML shown below, describes a Window containing a DockPanel, inside which is a DataGrid. The XAML is identical to the XAML in Example 7, except for the window caption.

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentat
ion"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DataGrid Matrix Example" Height="500"
    SizeToContent="Width"
    Topmost="true">
    <DockPanel>
        <DataGrid Name="DG1" ItemsSource="{Binding}"
                  AutoGenerateColumns="False" >
            <DataGrid.Columns>
                <DataGridTextColumn Header="Wine"
                 Binding="{Binding Name}"/>
                <DataGridTextColumn Header="Price"
                 Binding="{Binding Price, StringFormat=C}" />
            </DataGrid.Columns>
        </DataGrid>
    </DockPanel>
</Window>
```

The phrase ItemsSource="{Binding}" states that the content of the DataGrid is bound to a data source, which in this case will be inherited from the DataContext property of the parent Window.

Binding="{Binding Name}" specifies that the contents of the first column are bound to a Path named *Name* in the data source.

Similarly, Binding="{Binding Price, StringFormat=C}" specifies that the Path for the second column is *Price* (StringFormat=C merely specifies the default currency format).

The APL Code

The function `Grid` is shown below.

```
      ∇ Grid;⎕USING;MySource;win;info
[1]    ⎕USING←'System'
[2]    ⎕EX'winelist'
[3]    winelist←Wines,[1.5]0.01×10000+?(ρWines)ρ10000
[4]    win←LoadXAML XAML
[5]    info←(,'Name' 'Price'),⊂Object
[6]    win.DataContext←info(2015⍳)'winelist'
[7]    win.Show
      ∇
```

As in Example 7, the global variable `Wines` contains a vector of character vectors, each of which is the name of a wine.
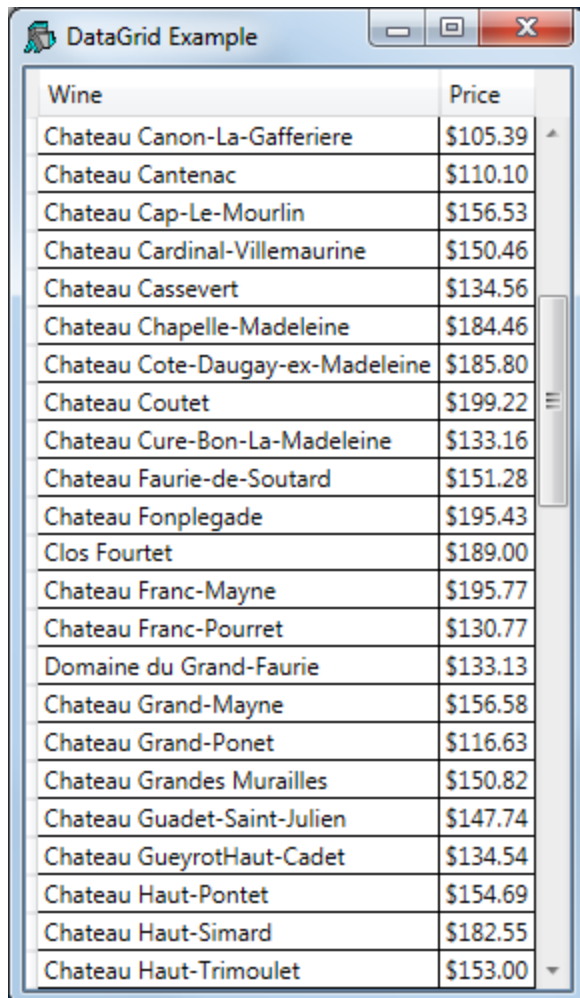
`Grid[2-4]` creates a matrix `winelist`, whose first column contains the names of the wines, and whose second column their (randomly generated) prices. As this is a global variable, the variable is expunged before being used in order to remove any previous data binding information that was associated with it.

`Grid[5]` creates the left argument for `(2015I)` which defines the names and data types of the properties which the columns of the matrix `winelist` will be exposed as. In this case, the names of the paths are `Name` and `Price`, and therir data types are both System.Object. So the first column will be exposed as Name and the second as Price, matching the path names specified in the XAML:

```
<DataGridTextColumn Header="Wine"
Binding="{Binding Name}"/>
<DataGridTextColumn Header="Price"
Binding="{Binding Price, StringFormat=C}" />
```

Testing the Data Binding

```
)LOAD wpfintro
)CS DataBinding.DataGridMatrix
Grid
```
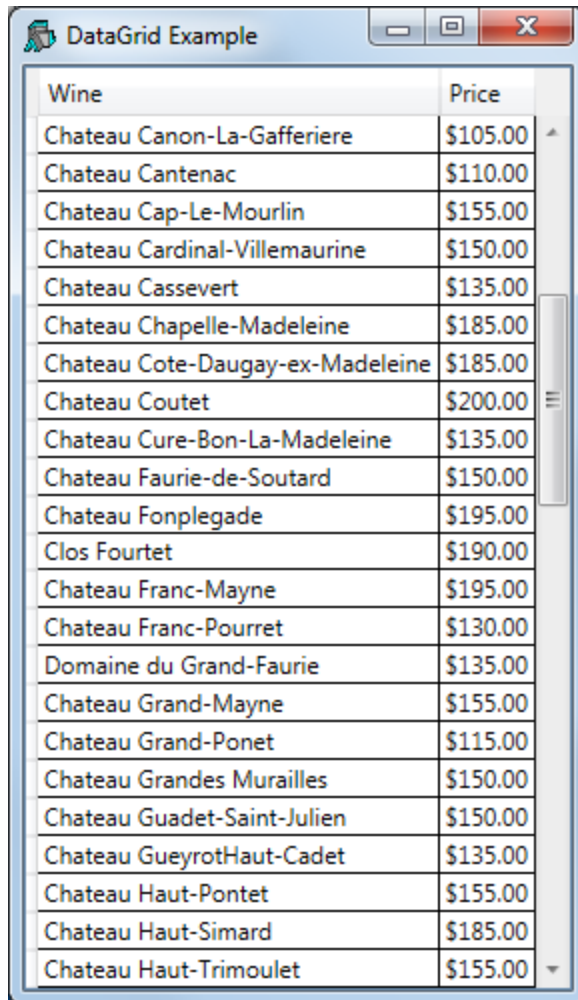
Wine	Price
DataGrid Example	
Chateau Canon-La-Gafferiere	$105.39
Chateau Cantenac	$110.10
Chateau Cap-Le-Mourlin	$156.53
Chateau Cardinal-Villemaurine	$150.46
Chateau Cassevert	$134.56
Chateau Chapelle-Madeleine	$184.46
Chateau Cote-Daugay-ex-Madeleine	$185.80
Chateau Coutet	$199.22
Chateau Cure-Bon-La-Madeleine	$133.16
Chateau Faurie-de-Soutard	$151.28
Chateau Fonplegade	$195.43
Clos Fourtet	$189.00
Chateau Franc-Mayne	$195.77
Chateau Franc-Pourret	$130.77
Domaine du Grand-Faurie	$133.13
Chateau Grand-Mayne	$156.58
Chateau Grand-Ponet	$116.63
Chateau Grandes Murailles	$150.82
Chateau Guadet-Saint-Julien	$147.74
Chateau GueyrotHaut-Cadet	$134.54
Chateau Haut-Pontet	$154.69
Chateau Haut-Simard	$182.55
Chateau Haut-Trimoulet	$153.00

Let's round the prices to the nearest $5.

```
winelist[;2]←5×⌊0.5+winelist[;2]÷5
```

Wine	Price
Chateau Canon-La-Gafferiere	$105.00
Chateau Cantenac	$110.00
Chateau Cap-Le-Mourlin	$155.00
Chateau Cardinal-Villemaurine	$150.00
Chateau Cassevert	$135.00
Chateau Chapelle-Madeleine	$185.00
Chateau Cote-Daugay-ex-Madeleine	$185.00
Chateau Coutet	$200.00
Chateau Cure-Bon-La-Madeleine	$135.00
Chateau Faurie-de-Soutard	$150.00
Chateau Fonplegade	$195.00
Clos Fourtet	$190.00
Chateau Franc-Mayne	$195.00
Chateau Franc-Pourret	$130.00
Domaine du Grand-Faurie	$135.00
Chateau Grand-Mayne	$155.00
Chateau Grand-Ponet	$115.00
Chateau Grandes Murailles	$150.00
Chateau Guadet-Saint-Julien	$150.00
Chateau GueyrotHaut-Cadet	$135.00
Chateau Haut-Pontet	$155.00
Chateau Haut-Simard	$185.00
Chateau Haut-Trimoulet	$155.00

Using Code
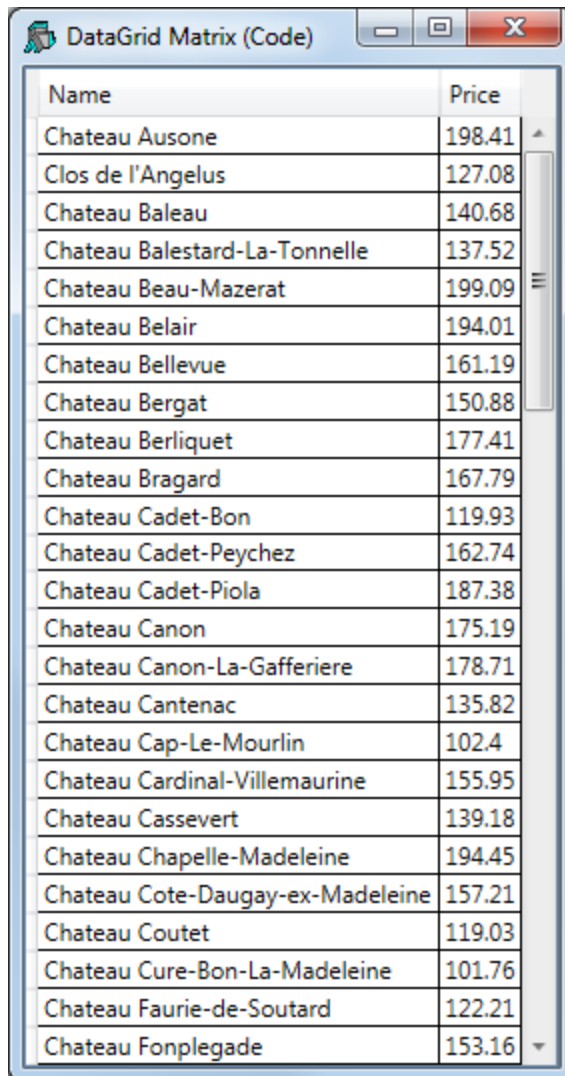
The same result can be achieved using code instead of XAML as illustrated by the function `GridCodeNoFmt`. The function is so-named because this code is insufficient to display the second column in currency format.

```
     ∇ GridCodeNoFmt;⎕USING;MySource;win;info;fmt
[1]    ⎕USING←'System'
[2]    ⎕USING,←,⊂'System.Windows.Controls,WPF/PresentationFramewo
rk.dll'
[3]    ⎕USING,←⊂'System.Windows.Controls.Primitives,WPF/Presentat
ionFramework.dll'
[4]    ⎕USING,←⊂'System.Windows,WPF/PresentationFramework.dll'
[5]    ⎕USING,←⊂'System.Windows,WPF/PresentationCore.dll'
[6]
[7]    ⎕EX'winelist'
[8]    winelist←Wines,[1.5]0.01×10000+?(⍴Wines)⍴10000
[9]    win←⎕NEW Window
[10]   win.Title←'DataGrid Matrix (Code)'
[11]   win.grid←⎕NEW DataGrid
[12]   info←(,'Name' 'Price'),⊂Object
[13]   win.grid.ItemsSource←info(2015⍳)'winelist'
[14]   win.grid.Height←500
[15]   win.Content←win.grid
[16]   win.SizeToContent←SizeToContent.WidthAndHeight
[17]   win.Show
     ∇
```

This is because by default the DataGrid generates its columns autommnatically with default formatting.

In order to apply special formatting to one or more columns, it is necessary to set the `AutoGenerateColumns` property to 0, and to generate the columns pro-gramatically as is shown in the second version of the function, `GridCode`.

```
      ∇ GridCode;⎕USING;MySource;win;info;fmt
[1]    ⎕USING←'System'
[2]    ⎕USING,←,⊂'System.Windows.Controls,WPF/Presentatio
nFramework.dll'
[3]    ⎕USING,←⊂'System.Windows.Controls.Primitives,WPF/P
resentationFramework.dll'
[4]    ⎕USING,←⊂'System.Windows,WPF/PresentationFramework
.dll'
[5]    ⎕USING,←⊂'System.Windows,WPF/PresentationCore.dll'
[6]
[7]    ⎕EX'winelist'
[8]    winelist←Wines,[1.5]0.01×10000+?(⍴Wines)⍴10000
[9]    win←⎕NEW Window
[10]   win.Title←'DataGrid Matrix (Code with Formatting)'
[11]   win.grid←⎕NEW DataGrid
[12]   info←(,'Name' 'Price'),⊂Object
[13]   win.grid.ItemsSource←info(2015⍳)'winelist'
[14]   win.grid.Height←500
[15]   win.grid.AutoGenerateColumns←0
[16]   win.Content←win.grid
[17]   win.SizeToContent←SizeToContent.WidthAndHeight
[18]   ⍝ Add columns and set format
[19]   win.grid.Columns.Add¨'' 'C'{
[20]       col←⎕NEW DataGridTextColumn
[21]       col.Header←⍵
[22]       col.Binding←⎕NEW Data.Binding(⊂⍵)
[23]       col.Binding.StringFormat←,⍺
[24]       col
[25]   }¨'Name' 'Price'
[26]
[27]   win.Show
      ∇
```

In this version of the function, lines `[19-25]` create the two columns `Name` and `Price`, applying currency format to the `Price` column.

DataGrid Matrix (Code+Fmt)	
Name	**Price**
Chateau Ausone	$159.62
Clos de l'Angelus	$102.14
Chateau Baleau	$110.31
Chateau Balestard-La-Tonnelle	$115.72
Chateau Beau-Mazerat	$137.46
Chateau Belair	$129.27
Chateau Bellevue	$106.04
Chateau Bergat	$189.44
Chateau Berliquet	$101.26
Chateau Bragard	$172.14
Chateau Cadet-Bon	$143.27
Chateau Cadet-Peychez	$116.36
Chateau Cadet-Piola	$152.97
Chateau Canon	$174.25
Chateau Canon-La-Gafferiere	$155.95
Chateau Cantenac	$107.56
Chateau Cap-Le-Mourlin	$180.33
Chateau Cardinal-Villemaurine	$138.34
Chateau Cassevert	$190.48
Chateau Chapelle-Madeleine	$119.01
Chateau Cote-Daugay-ex-Madeleine	$179.26
Chateau Coutet	$166.12
Chateau Cure-Bon-La-Madeleine	$114.33
Chateau Faurie-de-Soutard	$159.73
Chateau Fonplegade	$182.68

System Requirements

Microsoft Windows

Dyalog APL Version 14.1 supports versions of Windows from Microsoft Windows XP up to and including Microsoft Windows 8.1 and Microsoft Windows Server 2012. Dyalog APL Version 14.1 will not run on earlier versions.

Microsoft .NET Interface

Dyalog APL Version 14.1 .NET Interface requires Version 2.x or greater of the Microsoft .NET Framework. It does *not* operate with .NET Version 1.0.

For Windows Presentation Foundation (WPF) and basic Data Binding, Version 14.1 requires .NET Version 4.0.

For full Data Binding support (including support for the `INotifyCollectionChanged` interface[1]), and Syncfusion, Version 14.1 requires .NET Version 4.5.

AIX and Linux

For AIX, Version 14.1 requires AIX 6.1 or higher, and a POWER5 chip or higher.

Version 14.1 is built on RedHat 5, and runs on all recent distributions, including Ubuntu 12.04 and openSUSE 12.3. Contact Dyalog for information about other platforms.

OS X

For OS X Version 14.1 requires OS X Yosemite (10.10.x) or higher.

---

[1]This interface is used by Dyalog to notify a data consumer when the contents of a variable, that is data bound as a list of items, changes.

Inter-operability

Introduction

Workspaces and component files are stored on disk in a binary format (illegible to text editors). This format differs between machine architectures and among versions of Dyalog. For example, a file component written by a PC may well have an internal format that is different from one written by a UNIX machine. Similarly, a workspace saved from Dyalog Version 14.1 will differ internally from one saved by a previous version of Dyalog APL.

It is convenient for versions of Dyalog APL running on different platforms to be able to *interoperate* by sharing workspaces and component files. From Version 11.0, component files and workspaces can generally be shared between Dyalog interpreters running on different platforms. However, this is not always possible, for example:

- Component files created by Version 10.1 can often not be shared across platforms, even when used by later versions.
- *Small-span* (32-bit) component files become read-only when opened on a different architecture from that on which they were created.

Note however that the system function ⎕FCOPY can be used to make a logically identical copy of an old file, but the copy will be fully inter-operable.

The following sections describe other limitations in inter-operability:

Code

Code that is saved in workspaces, or embedded within ⎕ORs stored in component files, can generally only be read by the Dyalog version which saved them and later versions of the interpreter. In the case of workspaces, an load (or copy) into an older version would fail with the message:

```
this WS requires a later version of the interpreter.
```

Every time a ⎕OR object is read by a version later than that which created it, time may be spent in converting the internal representation into the latest form. Dyalog recommends that ⎕OR should not be used as a mechanism for sharing code or objects between different versions of APL

"Ordinary" Arrays

With the exception of the Unicode restrictions described in the following para-graphs, Dyalog APL provides inter-operability for arrays that only contain (nested) character and numeric data. Such arrays can be stored in component files - or trans-mitted using `TCPSocket` objects and Conga connections, and shared between all versions and across all platforms.

As mentioned in the introduction, full cross-platform interoperability of component files is only available for large-span component files.

Null Items (`⎕NULL`)

In Version 13.0 and earlier, an attempt to `⎕FREAD` a component containing a `⎕NULL` that was created by a Version 13.1 or later Dyalog APL will generate `DOMAIN ERROR`. `⎕NULL`s can be shared between Version 14.1 and Versions 13.1, 13.2 and 14.0 provided that the interpreters have been patched to revision 23705 or higher.

Object Representations (`⎕OR`)

From Version 13.2 onwards, an attempt to `⎕FREAD` a component containing a `⎕OR` that was created by a later version of Dyalog APL will generate `DOMAIN ERROR: Array is from a later version of APL`.

32 vs. 64-bit Component Files

Large-span (64-bit-addressing) component files are inaccessible to versions of the interpreter that pre-dated their introduction (versions earlier than 10.1).

From version 14.0 onwards it is no longer possible to create small-span (32-bit) files; Version 14.0 and 14.1 are still able to read and write to small span files. Setting the second item of the right argument of `⎕FCREATE` to anything other than 64 will gen-erate a `DOMAIN ERROR`.

Note that *small-span* (32-bit-addressing) component files cannot contain Unicode data. Unicode editions of Dyalog APL can only write character data which would be readable by a Classic edition (consisting of elements of `⎕AV`).

External Variables

External variables are implemented as small-span (32-bit-addressing) component files, and are subject to the same restrictions as these files. External variables are unlikely to be developed further; Dyalog recommends that applications which use them should switch to using mapped files or traditional component files. Please con-tact Dyalog if you need further advice on this topic.

32 vs. 64-bit Interpreters

From Dyalog APL Version 11.0 onwards, there are two separate versions of programs for 32-bit and 64-bit machine architectures (the 32-bit versions will also run on 64-bit machines running 64-bit operating systems). There is complete inter-operability between 32- and 64-bit interpreters, except that 32-bit interpreters are unable to work with arrays or workspaces greater than 2GB in size.

Note however that under Windows a 32-bit version of Dyalog APL may only access 32-bit DLLs, and a 64-bit version of Dyalog APL may only access 64-bit DLLs. This is a Windows restriction.

Unicode vs. Classic Editions

From Version 12.0 onwards, a Unicode edition is available, which is able to work with the entire Unicode character set. Classic editions (a term which includes versions prior to 12.0) are limited to the 256 characters defined in the atomic vector, `⎕AV`).

Component files have a Unicode property. When this is enabled, all characters will be written as Unicode data to the file. The Unicode property is always off for small-span (32-bit addressing) files, as these cannot contain Unicode data. For large-span (64-bit addressing) component files, the Unicode property is set *on* by Unicode Editions and *off* by Classic Editions, by default. The Unicode property can subsequently be toggled on and off using `⎕FPROPS`.

When a Unicode edition writes to a component file that cannot contain Unicode data, character data is mapped using `⎕AVU`; it can therefore be read without problems by Classic editions.

A `TRANSLATION ERROR` will occur if a Unicode edition writes to a non-Unicode component file (that is either a 32-bit file, or a 64-bit file when the Unicode property is currently off) if the data being written contains characters that are not in `⎕AVU`.

Likewise, a Classic edition (Version 12.0 or later) will issue a `TRANSLATION ERROR` if it attempts to read a component containing Unicode data that is not in `⎕AVU` from a component file. Version 11.0 cannot read components containing Unicode data and issues a `NONCE ERROR`.

A `TRANSLATION ERROR` will also be issued when a Classic edition `)LOAD`s or `)COPY`s a workspace containing Unicode data that cannot be mapped to `⎕AV` using the `⎕AVU` in the recipient workspace.

`TCPSocket` objects have an `APL` property that corresponds to the Unicode property of a file, if this is set to `Classic` (the default) the data in the socket will be restricted to `⎕AV`, if Unicode it will contain Unicode character data. As a result, `TRANSLATION ERROR`s can occur on transmission or reception in the same way as when updating or reading a file component.

The symbols `⍤`, `⍠` and `⌸` used for the Rank, Variant and Key operators respectively are available only in the Unicode edition. In the Classic edition, these symbols are replaced by `⎕U2364`, `⎕U2360` and `⎕U2338` respectively. In both Unicode and Classic editions Variant may be represented by `⎕OPT`.

AVU changes

The implementation of the function `Right` in Version 13.0 led to the discovery that `⎕AVU` incorrectly defined `⎕AV[59+⎕IO]` as `¤` (`⎕UCS 164`) rather than `⊢` (Right Tack, `⎕UCS 8866`). This error has been corrected in the default `⎕AVU` and in workspace `AVU.dws`. If you are operating in a mixed Unicode/Classic environment, this error will have caused earlier Classic editions to map `⎕AV[59+⎕IO]` to the wrong Unicode character (`¤`). This may cause `TRANSLATION ERROR`s when a Version 13.0 Classic system attempts to read the data, as it will not be able to represent `¤` in the Atomic Vector.

DECFs and Complex numbers

Version 13.0 introduced two new data types; DECFs and Complex numbers. Attempts to read components of these types in earlier interpreters will result in a `DOMAIN ERROR`.

Very large array components

The maximum size (in bytes) of a component written by Version 12.1 and prior is 2GB. This is the size of the component as held on disk which may be different than the size reported by `⎕SIZE`. In Version 13.0 the maximum size of a component written by a 64-bit interpreter is 4GB. From Version 13.2 onwards, the limit on the size of arrays or components is so large that for most practical purposes, there is effectively no limit.

An attempt to read a component greater than 2GB in 32-bit interpreters will result in a `WS FULL`. An attempt to read such a component in 64-bit Versions 12.0 and 12.1 patched after 1st April 2011 will result in a `NONCE ERROR`; earlier patches generate a `FILE COMPONENT DAMAGED` error.

File Journaling

Version 12.0 introduced File Journaling (level 1), and 12.1 added journaling levels 2 and 3 and checksumming. Versions earlier than 12.0 cannot tie files that have any form of journaling or checksumming enabled. Version 12.0 cannot tie files with journaling levels greater than 1, or checksumming enabled. Attempting to tie such files will result in a `FILE NAME ERROR`. Files can be shared with earlier versions by using `⎕FPROPS` to amend the journaling and checksumming levels.

File Component Compression

Version 14.0 introduced File Component Compression; earlier versions will be able to perform all file operation on such files with the exception of being able to `⎕FREAD` any compressed component. In particular, it is possible for any earlier version to `⎕FREPLACE` a compressed component with a non-compressed one.

Attempting to read a compressed component using earlier versions of Dyalog APL will generate an error:

- All 13.2 and 13.1.14842 and later:
  `DOMAIN ERROR: Array is from a later version of APL`
- 13.1 before revision 14842:
  `FILE COMPONENT DAMAGED: Incoming array is invalid`
- 13.0 and 12.1 after revision 11154:
  `DOMAIN ERROR`
- 13.0 and 12.1 before revision 11154:
  `FILE COMPONENT DAMAGED`

TCPSockets

TCPSockets used to communicate between differing versions of Dyalog APL are subject to similar limitations to those described above for component files. In particular TCPSockets with `'Style' 'APL'` will only be able to pass arrays that are supported by both versions.

Auxiliary Processors

A Dyalog APL process is restricted to starting an AP of exactly the same architecture. In other words, the AP must share the same word-width and byte-ordering as its interpreter process.

Session Files

Session (.dse) files can only be used on the platform on which they were created and saved.

Announcements

Withdrawal of Support for Version 13.1

The supported Versions of Dyalog APL are now Version 14.1, Version 14.0 and Version 13.2. Version 13.1 and earlier are no longer supported.

.NET Support

Support for Microsoft .NET Version 2 will be dropped in the next release of Dyalog.

Planned Operating System Requirements for the next version

Dyalog expects that the next version of Dyalog will require the following minimum platform requirements:

Operating System	Version
Microsoft Windows	Vista or Server 2008
AIX	6.1 on POWER 6
Linux	RedHat/Centos 6 or equivalent
OS X	OS X Yosemite 10.10.x

Further information will appear on the Forums as and when available.

Planned Hardware Requirements for next version

The next version of Dyalog running on Intel or AMD processors will require at lesat SSE2 or equivalent. This affects 32 bit interpreters only.

New Method

In Version 14.0 and earlier, Dyalog APL artificially added a method named *New* to all .NET objects which did not have such a member. This allowed the APL programmer to create an instance of an object (such as in this example a DateTime object) by executing the statements:

```
⎕USING←'System'
dt←DateTime.New 1949 4 30
```

This feature has been removed in Version 14.1. This mechanism was made redundant by the introduction of ⎕NEW, and the following syntax should be adopted:

```
dt←⎕NEW DateTime (1949 4 30)
```

Uppercase Property

The UpperCase property of the Root object has been removed. This was previously used to force property, method and event names to be reported in upper-case.

Thread Synchronisation Method (`1113⌶`)

`1113⌶` has been removed. All operating systems supported by Dyalog APL have semaphores, so this feature is no longer needed.

XPLookAndFeelDocker parameter

This parameter and the related check-box on the General tab of the Configuration dialog box have been removed.

Performance Improvements

The following primitives (or sub-cases thereof) are faster in Version 14.1.

`∧.= ∨.≠ +.= +.≠ ○.≡ ○.≢`
`⎕dr`
`⊂ ⊂[⎕io]`
`ρ`
`⍋ {ω[⍋ω]}`
`⍒ {ω[⍒ω]}`
`=\`
`+/ ⊢/ ⊣/ ⌈/ ⌊/`
`⍉`
`, b[i]←x`
`⍳ ∊ ∪ 8I (≢∪)`
`≡¨r ⎕¨O 15`
`{αω}⊟ {α(≢∪ω)}⊟ {⊂ω}⊟ {α(≢ω)}⊟ {α,≢ω}⊟ {α}⊟ ⊣⊟`

Highlights:

The interpreter now exploits some special instructions available on modern CPUs (SSE2, SSE4, BMI2). Primitives that benefit from such exploitation include `+/`, `⌈/`, `⌊/`, `⍋`, `{ω[⍋ω]}`, `⍳`, `∊`, `∪`, `⍉`.

Some primitives execute faster by exploiting the following insight: `x⍳x` are like ID numbers; questions of identity on `x` can often be answered more efficiently on `x⍳x` than on `x` itself. For example, `x∧.=y ↔ (x⍳x)○.=x⍳⍉y` and `x○.≡y ↔ (x⍳x)○.=x⍳y`; the longer expressions used to be faster and in Version 14.1 the short and longer expressions are equally fast.

`x⍳y` has special code for relational tables, matrices in which the items of a column have the same type and same rank. (If the type is floating point, the special code is invoked only if `0=⎕CT`.)

The following table lists the cases for which performance improvements have been achieved in Version 14.1.

Expression	Factor	Comments
`x∧.=y`	1.5-	`(xιx)∘.=xιⵁy` for matrix `x`
`⎕dr`	1-3.8	AKA squeeze
`(≢∪)x`	2.8-∞	when `x` is floating point, optimized code is used only when `0=⎕ct`
`b∊[⎕io]x`	1.3-42	underlies the general case of `⊟`, among other things
`x{⊂ω}⊟y`	1.2	
`sρx`	2	for non-pointer `x` which is 1, 2, 4, or 8 bytes
`=\b`	1.1	now as fast as `≠\b`
`⍋b` and `⍒b`	1-1.7	for boolean vector `b`
`⍋x` and `⍒x`	1-1.6	for `x` with 1-byte items
`x∧.=v`	7	for non-tolerant `=` and vector `v` with 1,2,4, or 8 bytes
`x≡⍤r⊢y`	7	for non-tolerant `≡` and one of `x` or `y` is a single cell with 1,2,4, or 8 bytes
`⊢/` and `⊣/`	3-28	the largest factors are for boolean arguments
`⌈/` and `⌊/`	2-20	on POWER6 onwards AIX systems, vectors
`⌈/` and `⌊/`	1-30	on Windows and Linux systems, vectors
`+/`	2-4	on POWER6 and POWER7 AIX systems, 1-byte and 2-byte integer vectors
`+/`	2-5	on Windows and Linux systems, integer vectors
`x∘.≡y` and `x∘.≢y`	6-64	non-simple arrays `x` and `y` not requiring tolerant comparison

Expression	Factor	Comments
`⌽y`	10	on Intel CPUs with BMI2, boolean matrices with both dimensions a multiple of 8
`999,b` and `b[1]` `←999` and similar	2	any expression which blows up an array from boolean to 16-bit integer
`99,b` and `b[1]` `←99` and similar	1.7	on Intel CPUs with BMI2, any expression which blows up an array from boolean to 8-bit integer
`x⍳y`	3-6	on 4-byte major cells
`y∊x`	4-18	on 4-byte major cells
`∪x`	4-12	on 4-byte major cells
`x⍳y`	2-4	on k-byte major cells
`x⍳y`	2.5-6	on relational matrices
`x(8I)y`	1-1.5	exploit special cases for the inverted table index-of
`i⎕⍳0 15⊢x`	2-250	selecting major cells
`⍋x` and `{ω[⍋ω]}` `x`	1-1.5	grade/sort on small-range 4-byte integers
`{α}⌸` and `⊣⌸`	5-60	special code; equivalent to extended `∪x`

New Idioms

The following new idioms are recognised:

Expression	Description
`≠⍴`	Rank. Note that `≠⍴` returns a scalar, whereas `⍴⍴` returns a 1-element vector.

Bug Fixes

A number of bug fixes implemented in Version 14.1 may change the way that existing code operates and are therefore documented in this section.

Change to the File Library System Function

There was an inconsistency in the result of the expression `⎕FLIB '.'`. Under UNIX it returned relative path names but under Windows it returned full path names. From 14.1 it returns relative path names under all operating systems.

```
      'dummy1' ⎕FCREATE 1
      'dummy2' ⎕FCREATE 2
      'dummy3' ⎕FCREATE 3
      ⎕FUNTIE ⎕FNUMS


      ⎕FLIB '.'
.\dummy1
.\dummy2
.\dummy3
```

If the right argument specifies a full pathname, the results include the pathname as before:

```
      ⎕FLIB 'c:\users\pete\desktop'
c:\users\pete\desktop\dummy1
c:\users\pete\desktop\dummy2
c:\users\pete\desktop\dummy3
```

Change to Scripted Objects

In previous versions of Dyalog APL it was possible to specify multiple definitions of the same one-lined named dfn in a script, whereas multiple definitions of the same multi-lined dfn or tradfn would generate an error. In version 14.1 the behaviour for one-lined dfns has been brought into sync with other attempts to redefine functions and now too generates an error.

In previous versions, the expression:

```
   ⎕fix ':class c' 'foo←{' '1}' 'foo←{' '2}' ':EndClass'
```

would cause an error, whereas:

```
   ⎕FIX':class c' 'foo←{1}' 'foo←{2}' ':EndClass'
```

would work.

In Version 14.1, both expressions correctly generate an error.

Chapter 2:

Component File Improvements

File Create and Variant

The following properties have been added to those that may be set using the Variant Operator:

- `'U'` - Unicode; 0 or 1
- `'S'` - File Size (span); 64

The principal option remains as follows; in all cases `U` and `S` are not changed.

- 0 - sets `('J' 0) ('C' 0)`
- 1 - sets `('J' 1) ('C' 1)`
- 2 - sets `('J' 2) ('C' 1)`
- 3 - sets `('J' 3) ('C' 1)`

For example,

```
      'newfile' (⎕FCREATE ⍠3) 0
1
      'SEUJCZ' ⎕FPROPS 1
64 0 1 3 1 0
```

Alternatively:

```
      JFCREATE←⎕FCREATE ⍠ 3
```

will name a variant of `⎕FCREATE` which will create component file with level 3 journaling, and checksum enabled. Then:

```
      'newfile'JFCREATE 0
1
```

File Copy and Variant

⎕FCOPY now allows you to specify properties for the new file via the variant operator ⍠ used with the following options:

- 'J' - journaling level; a numeric value.
- 'C' - checksum level; 0 or 1.
- 'Z' - compression; 0 or 1.
- 'U' - Unicode; 0 or 1
- 'S' - File Size (span); 64

The Principal Option is as follows:

- 0 - sets ('J' 0) ('C' 0)
- 1 - sets ('J' 1) ('C' 1)
- 2 - sets ('J' 2) ('C' 1)
- 3 - sets ('J' 3) ('C' 1)

Examples

```
      newfid←'newfile' (⎕FCOPY ⍠3) 1

      'SEUJCZ' ⎕FPROPS newfid
64 0 1 3 1 0
```

Alternatively:

```
      JFCOPY←⎕FCOPY ⍠ 3
```

will name a variant of ⎕FCREATE which will create component file with level 3 journaling, and checksum enabled. Then:

```
      newfid←'newfile' JFCOPY 1
```

Note: Setting ('U' 0) (no Unicode support) is discouraged as it may cause the copy to fail with a TRANSLATION ERROR.

Chapter 3: Miscellaneous

IDE Enhancements

Aligning Comments in Scripts

The alignment of comments has been extended to scripted objects. If the user is editing a script, selects one or more lines in the script, and chooses *Align Comments* from the Session pop-up menu, the comments in the selected lines are aligned.

If no lines are selected, and the user chooses *Align Comments*:

- If the cursor is in a line of a function or operator, all of the comments within that function are aligned. If the function is a dfn, all of the comments within the same *capsule*[1] are aligned.
- If the cursor is in a line that is not in a function, all of the comments between the end of the preceding function (or the start of the script) and the beginning of the next function (or the end of the script) are aligned.

Tracing dfn Arguments

It is now possible to view the value of α, αα, ω and ωω when tracing a dfn, using the mouse or using `)ED` or `⎕ED`. If you open an edit window on them it will be read-only.

Tracing Blank Lines and Comments

It is now possible to separately specify whether or not the Tracer should skip blank lines and comments.

See *User's Guide:* Configuration Dialog: Trace/Edit and *Installation & Configuration Guide:* SkipBlankLines parameter.

---

[1]A capsule is defined as the outermost dfn in a set of nested dfns.

Workspaces without File Extensions

When locating a workspace where no file name extension has been explicitly provided, the new **WSEXT** parameter is used to identify the corresponding file.

This parameter specifies workspace filename extensions. It complements the **WSPATH** parameter in that together they determine the file search order to satisfy `)LOAD` or `)COPY`; it also specifies the filename extension to add on `)SAVE` if none is explicitly provided.

WSEXT is a string that specifies a colon-separated list of one or more extensions, including any period (".") which separates the extension from its basename

If undefined, WSEXT defaults to `.dws:` on Windows and `:.dws:.DWS` on all other platforms.

Loading a Workspace

If the workspace name specified for `)LOAD`, `)XLOAD`, `)COPY`, `)PCOPY`, `⎕LOAD` and `⎕CY`, and the workspace name given on the Dyalog command line, does not end with a file extension, the corresponding file is located as follows:

The name is appended with each extension in **WSEXT** in turn.

- If the resulting name is a full or relative pathname, the system tries to open the named file in the appropriate directory.
- If the resulting name contains no directory information, the system looks in each of the directories specified by **WSPATH**.

This operation is repeated (the outer loop is the extensions specified by **WSEXT**, the inner loop is the list of directories specified by **WSPATH**) until a file matching the name is found or until the list of extensions is exhausted.

Saving a Workspace

If the workspace name specified for `)SAVE` or `)CONTINUE` is elided, the file will be saved with the name contained in `⎕WSID`. If an extension was specified in the name assigned to `⎕WSID` or `)WSID`, that extension will be used. If not, the corresponding file name is constructed by adding the first extension in **WSEXT**.

If the workspace name specified for `)SAVE`, `)CONTINUE` or `⎕SAVE` does not end with a file extension, the corresponding file name is constructed by adding the first extension in **WSEXT**.

Dropping a Workspace

If the workspace name specified for `)DROP` does not end with a file extension, the system identifies all the files in the specified directory whose names match the workspace name for all extensions in **WSEXT**

If this identifies a single file match, that file is deleted. If it identifies more than one file match, the list of matching files is displayed but no file is erased.

Example:

```
      )lib .
pete.dws        pete1.dws

      )CMD copy pete.dws pete
1 file(s) copied.

      )lib .
pete.   pete.dws        pete1.dws

      )drop pete
Multiple possibilities:
  pete.dws
  pete.
```

Consequential Changes

As a result of the implementation of **WSEXT**, certain changes have been made to the output generated by `)WSID` and `)LIB`. No changes have been made to the behaviour of `⎕WSID`, because this might have affected existing code.

Trailing Dot for Workspace Names Without File Extensions

Under Windows only, all system commands that report workspace names add a dot to any workspace name whose file name does not have an extension (see previous example).

Workspace Identification

`)WSID` now always reports the file extension whether it was assigned explicitly, or implicitly by the application of **WSEXT**.

Improved Thread Support for WPF

When an APL **thread** first makes a .NET call, it creates a unique system thread in which that and subsequent .NET calls are made. If a .NET call results in the creation of a message queue, that queue is associated with that same system thread. So each message queue is also unique. This strategy successfully maintains separation between multiple Windows message queues being executed in different APL threads.

By default, the base APL thread (thread 0) runs .NET code in the same system thread as itself. This is a different system thread to that used to run .NET code from other APL threads, so the separation between message queues associated with APL thread 0 and those associated with other APL threads is maintained. However, in certain circumstances, messages generated by .NET objects interfere with APL's internal message processing (and vice-versa), for example when handling exceptions.

For this reason, Dyalog recommends that APL code that creates instances of .NET objects that generate events (such as Windows Presentation Foundation objects) are run in a separate APL thread.

Where this is not possible, `2520⌶1` may be used to force Dyalog to use a unique system thread for .NET that is associated with APL thread 0. If so, it is recommended that `2520⌶1` is called at application start-up time.

For further information, see *Use Separate Thread For .NET* on page 67.

Removal of HelpURL from DMX

HelpURL has been removed from the display form of ⎕DMX, although it remains as a property.

Version 14.0

```
      1÷0
DOMAIN ERROR: Divide by zero
      1÷0
      ^
      ⎕DMX
 EM        DOMAIN ERROR
 Message   Divide by zero
 HelpURL   http://help.dyalog.com/dmx/14.0/General/1
```

Version 14.1

```
      1÷0
DOMAIN ERROR: Divide by zero
      1÷0
      ^
      ⎕DMX
 EM        DOMAIN ERROR
 Message   Divide by zero
```

New Parameters

ExternalHelpURL Parameter

The URL for help on external objects is now specified by the **ExternalHelpURL** configuration parameter and not by the **DefaultHelpCollection** parameter which has been removed.

RIDE_INIT

This parameter determines how the interpreter should behave with respect to the RIDE protocol. Setting this configuration parameter on the machine that hosts the interpreter enables the interpreter-RIDE connection.

The format of the value is:

```
<setting> : <address> : <port>
```

setting is the action the interpreter should take. Valid values are:

- serve – listen for incoming connections
- connect – connect to the specified RIDE and end the session if this fails
- poll – try to connect to the specified RIDE at regular intervals and reconnect if the connection is lost

address is the machine on which to listen for a connection (if **setting** is serve) or connect to (if **setting** is connect/poll). Valid values are:

- the name of the machine
- the IPv4 address of the machine
- the IPv6 of the machine
- empty – if **setting** is serve then the interpreter listens to everything on every network interface, if **setting** is connect/poll then the interpreter only listens for local connections (127.0.0.1).

port is the TCP port to listen on

The RIDE_INIT configuration parameter is set automatically when launching a new Dyalog Session from the RIDE.

RIDE_SPAWNED

If non-zero, this parameter disables `⎕SR` and `)SH` which instead generate `DOMAIN ERROR`. This parameter is used to prevent certain user-interfaces from being executed from a RIDE session which does not support them, and which would otherwise cause the RIDE session to become unresponsive. See *RIDE Reference Guide*.

Chapter 4:

Language Reference Changes

Disposable Statement   `:Disposable`

The Dyalog interface to .NET involves the creation and removal of .NET objects. Many such objects are *managed* in that the .NET Common Language RunTime (CLR) automatically releases the memory allocated to the object when that object is no longer used. However, it is not possible to predict when the CLR garbage collection will occur. Furthermore, the garbage collector has no knowledge of *unmanaged* resources such as window handles, or open files and streams.

Typically, .NET classes implement a special interface called `IDisposable` which provides a standard way for applications to release memory and other resources when an instance is removed. Furthermore, the C# language has the `using` keyword, which "Provides a convenient syntax that ensures the correct use of `IDisposable` objects."

The `:Disposable array` statement in Dyalog APL provides a similar facility to C#'s `using`. `array` may be a scalar or vector of namespace references.

When the block is exited, any .Net objects in `array` that implement `IDisposable` will have `IDisposable.Dispose` called on them.

Note that exit includes normal exit as the code drops through `:EndDisposable`,
or if an error occurs and is trapped, or if branch (→) is used to exit the block, or any-
thing else.

See also: *.NET Interface Guide: .Disposing of .NET Objects*.

Example (Normal Exit)

```
:Disposable f←⎕NEW Font
.
.
:EndDisposable
```

In the above example, when the `:EndDisposable` statement is reached, the system
disposes of the Font object `f` (and all the resources associated with it) by calling
`(IDisposable)f.Dispose()`. A subsequent reference to `f` would generate
`VALUE ERROR`.

Example (Normal Exit)

```
:Disposable fonts←⎕NEW ¨Font Font
.
.
:EndDisposable
```

In the above example, Dispose() is called on **each** of the Font objects in `fonts`
during the processing of `:EndDisposable`.

Example (Branch Exit)

```
:Disposable fonts←⎕NEW ¨Font Font
.
→0
.
:EndDisposable
```

In this example, Dispose() is called on the Font objects in `fonts` during the
processing of the branch statement `→0`.

Example (TrapExit)

```
:trap 0

   :Disposable fonts←⎕NEW ¨Font Font
   .
   ÷0
   .
   :EndDisposable

:else

   ⎕←'failed'

:endif
```

Here, the objects are disposed of when the DOMAIN ERROR generated by the expression ÷0 causes the stack to be cut back to the :Else clause. At this point (just before the execution of the :Else clause) the name class of fonts becomes 0.

:Disposable Statement

```
|
:Disposable array
|
code
|
:End[Disposable]
|
```

Arbitrary Input   R←{X}⎕ARBIN Y

This transmits a stream of 8-bit codes in `Y` to an output device specified by `X` prior to reading from an input device specified by `X`.

`Y` may be a scalar or a simple vector of integer numbers in the range 0-255.

`X` may take several forms:

```
terminate (input output) ⎕ARBIN   codes
terminate input          ⎕ARBIN   codes
```

terminate

This is a simple numeric scalar that specifies how the read operation should be terminated.

- If it is a numeric scalar, it defines the number of bytes to be read.
- If it is a numeric vector, it defines a set of terminating bytes.
- If it is the null vector, the read terminates on Newline (10).

input

This is a simple numeric scalar that specifies the input device.

- If it is positive or zero, it represents a file descriptor that must have been associated by the command that started Dyalog APL.
- If it is negative, it represents the tie number of a file opened by `⎕NTIE` or `⎕NCREATE`.

output

If specified, this is a simple numeric integer that identifies the output device.

- If it is positive or zero, it represents a file descriptor that must have been associated by the command that started Dyalog APL.
- If it is negative, it represents the tie number of a file opened by `⎕NTIE` or `⎕NCREATE`.

The result `R` is a simple numeric vector. Each item of `R` is the numeric representation of an 8-bit code in the range 0 to 255 received from the input device. The meaning of the code is dependent on the characteristics of the input device. If a set of delimiters was defined by `terminate`, the last code returned will belong to that set.

⎕RTL (Response Time Limit) is an implicit argument of ⎕ARBIN. This allows a time limit to be imposed on input. If the time limit is reached, ⎕ARBIN returns with the codes read up to that point. This does not apply under Windows.

The operation will fail with a DOMAIN ERROR if Y contains anything other than numbers in the range 0-255, or if the current process does not have permission to read from or write to the specified device(s).

Examples (UNIX)

```
      )sh mkfifo ./fifo

      in←'./fifo'⎕NTIE 0
      out←'./fifo'⎕NTIE 0

      (10 (in out))⎕ARBIN ⎕UCS ⎕D
48 49 50 51 52 53 54 55 56 57

      (0 (in out))⎕ARBIN 10
10

⍝ cope with parity on line ending 10
      ((10+0 128) (in out))⎕ARBIN 10
10
```

Arbitrary Output   {X}⎕ARBOUT Y

This transmits a stream of 8-bit codes in Y to an output device specified by X.

Y may be a scalar or a simple vector of integer numbers in the range 0-255.

X is a simple numeric integer that specifies the output device.

- If X is positive or zero, it represents a file descriptor that must have been associated by the command that started Dyalog APL.
- If X is negative, it represents the tie number of a file opened by ⎕NTIE or ⎕NCREATE.

If Y is an empty vector, no codes are sent to the output device.

The operation will fail with a DOMAIN ERROR if Y contains anything other than numbers in the range 0-255, or if the current process does not have permission to write to the specified device.

Examples

Write ASCII digits '123' to stream 9:

```
      9 ⎕ARBOUT 49 50 51
```

Write ASCII characters 'ABC' to MYFILE:

```
      'MYFILE' ⎕NCREATE ¯1
      ¯1 ⎕ARBOUT 65 66 67
```

Append the string 'Κάλο Πάσχα' to the same file, and close it:

```
      ¯1 ⎕ARBOUT 'UTF-8' ⎕UCS'Κάλο Πάσχα'
      ⎕NUNTIE ¯1
```

Edit Object   `{R}←{X}⎕ED Y`

`⎕ED` invokes the Editor. `Y` is a simple character vector, a simple character matrix, or a vector of character vectors, containing the name(s) of objects to be edited. The optional left argument `X` is a character scalar or character vector with as many elements as there are names in `Y`. Each element of `X` specifies the type of the corresponding (new) object named in `Y`, where:

`∇`	function/operator
`→`	simple character vector
`∊`	vector of character vectors
`−`	character matrix
`⊛`	Namespace script
`○`	Class script
`∘`	Interface

If an object named in `Y` already exists, the corresponding type specification in `X` is ignored.

If `⎕ED` is called from the Session, it opens Edit windows for the object(s) named in `Y` and returns a null result. The cursor is positioned in the first of the Edit windows opened by `⎕ED`, but may be moved to the Session or to any other window which is currently open. The effect is almost identical to using `)ED`.

If `⎕ED` is called from a defined function or operator, its behaviour is different. On asynchronous terminals, the Edit windows are automatically displayed in "full-screen" mode (ZOOMED). In all implementations, the user is restricted to those windows named in `Y`. The user may not skip to the Session even though the Session may be visible.

`⎕ED` terminates and returns a result ONLY when the user explicitly closes all the windows for the named objects. In this case the result contains the names of any objects which have been newly (re)fixed in the workspace as a result of the `⎕ED`, and has the same structure as `Y`.

Objects named in `Y` that cannot be edited are silently ignored. Objects qualified with a namespace path are (e.g. `a.b.c.foo`) are silently ignored if the namespace does not exist.

Variants of Edit Object

The behaviour of `⎕ED` may be modified using the variant operator `⍠` with the following options:

- `'ReadOnly'` - 0 or 1
- `'EditName'` - `'Default'`, `'Allow'` or `'Disallow'`.

If `ReadOnly` is set to 1, the edit window and all edit windows opened from it will be read-only. Note that setting `ReadOnly` to 0 will have no effect if the edit window is inherently read-only due to the nature of its content.

The `'EditName'` option determines whether or not the user may open another edit window by clicking a name, and its values are interpreted as follows:

EditName	`⎕ED` called from session	`⎕ED` called from function
`'Default'`	Allow	Disallow
`'Allow'`	Allow	Allow
`'Disallow'`	Disallow	Disallow

There is no Principal Option.

Examples

```
A←3 11ρ'Hello World'
```

In the first example, `⎕ED` will display the contents of `A` as an editable character array which the user may change. The user can double-click on *Hello* to open an edit window on an object named `Hello` (which will be a new function if `Hello` is currently undefined). Furthermore, the user can enter any arbitrary name and double-click to edit it. This may be undesirable in an application.

```
⎕ED 'A'
```

In the second example, the Edit window will display the contents of `A` as a Readonly Character array. The user can still open a new edit by double-clicking *Hello* or *World* but nothing else.

```
(⎕ED ⍠ 'ReadOnly' 1) 'A'
```

In the final example, the Edit window will display the contents of `A` as a Readonly Character array and the user cannot open a new edit window.

```
(⎕ED ⍠('ReadOnly' 1)('EditName' 'Disallow'))'A'
```

File Create   {R}←X ⎕FCREATE Y

Y must be a simple integer scalar or a 1 or 2 element vector. The first element is the *file tie number*. The second element, if specified, must be 64[1].

The *file tie number* must not be the tie number associated with another tied file.

X must be either

    a. a simple character scalar or vector which specifies the name of the file to be created. See *User Guide* for file naming conventions under UNIX and Windows.
    b. a vector of length 1 or 2 whose items are:

        i. a simple character scalar or vector as above.
        ii. an integer scalar specifying the file size limit in bytes.

The newly created file is tied for exclusive use.

The shy result of ⎕FCREATE is the tie number of the new file.

Automatic Tie Number Allocation

A tie number of 0 as argument to a create or tie operation, allocates, and returns as an explicit result, the first (closest to zero) available tie number. This allows you to simplify code. For example:

from:

```
tie←1+⌈/0,⎕FNUMS    ⍝ With next available number,
file ⎕FCREATE tie   ⍝ ... create file.
```

to:

```
tie←file ⎕FCREATE 0 ⍝ Create with first available..
```

Examples

```
'..\BUDGET\SALES'    ⎕FCREATE 2     ⍝ Windows
'../budget/SALES.85' ⎕FCREATE 2     ⍝ UNIX

'COSTS' 200000 ⎕FCREATE 4           ⍝ max size 20000
0
```

---

[1]This element sets the *span* of the file which in earlier Versions of Dyalog APL could be 32 or 64. Small-span (32-bit) component files may no longer be created and this element is retained only for backwards compatibility of code.

File Properties

⎕FCREATE allows you to specify properties for the newly created file via the variant operator ⍠ used with the following options:

- 'J' - journaling level; a numeric value.
- 'C' - checksum level; 0 or 1.
- 'Z' - compression; 0 or 1.
- 'U' - Unicode; 0 or 1
- 'S' - File Size (span); 64

The Principal Option is as follows:

- 0 - sets ('J' 0) ('C' 0)
- 1 - sets ('J' 1) ('C' 1)
- 2 - sets ('J' 2) ('C' 1)
- 3 - sets ('J' 3) ('C' 1)

Examples

```
      'newfile' (⎕FCREATE⍠3) 0
1
      'SEUJCZ' ⎕FPROPS 1
64 0 1 3 1 0
```

Alternatively:

```
      JFCREATE←⎕FCREATE ⍠ 3
```

will name a variant of ⎕FCREATE which will create component file with level 3 journaling, and checksum enabled. Then:

```
      'newfile'JFCREATE 0
1
```

File Copy   R←X ⎕FCOPY Y

Access Code: 4609

`Y` must be a simple integer scalar or 1 or 2-element vector containing the file tie number and optional passnumber. The file need not be tied exclusively.

`X` is a character vector containing the name of a new file to be copied to.

`⎕FCOPY` creates a copy of the tied file specified by `Y`, named `X`.

The new file `X` will have the same component level information, including the user number and update time as the original. The operating system file creation, modification and access times will be set to the time at which the copy occurred.

Unless otherwise specified (see File Properties below) the new file `X` will have the same file properties as the original, except that it will be a large-span file regardless of the span of the original.

The result `R` is the file tie number associated with the new file `X`.

Note that the Access Code is 4609, which is the sum of the Access Codes for `⎕FREAD` (1), `⎕FRDCI` (512) and `⎕FRDAC` (4096).

Note also that although the file need not be tied exclusively, the `⎕FCOPY` function will not yield the file to other APL processes while it is running, and it may take some considerable time to run in the case of a large component file.

Example

```
      told←'oldfile32'⎕FTIE 0
      'S' ⎕FPROPS told
32
      tnew←'newfile64' ⎕FCOPY told

      'S' ⎕FPROPS tnew
64
```

If `X` specifies the name of an existing file, the operation fails with a `FILE NAME ERROR`.

Note: This operation is atomic. If an error occurs during the copy operation (such as disk full) or if a strong interrupt is issued, the copy will be aborted and the new file `X` will not be created.

File Properties

⎕FCOPY now allows you to specify properties for the new file via the variant operator ⍠ used with the following options:

- 'J' - journaling level; a numeric value.
- 'C' - checksum level; 0 or 1.
- 'Z' - compression; 0 or 1.
- 'U' - Unicode; 0 or 1
- 'S' - File Size (span); 64

The Principal Option is as follows:

- 0 - sets ('J' 0) ('C' 0)
- 1 - sets ('J' 1) ('C' 1)
- 2 - sets ('J' 2) ('C' 1)
- 3 - sets ('J' 3) ('C' 1)

Examples

```
      newfid←'newfile' (⎕FCOPY ⍠3) 1

      'SEUJCZ' ⎕FPROPS newfid
64 0 1 3 1 0
```

Alternatively:

```
      JFCOPY←⎕FCOPY ⍠ 3
```

will name a variant of ⎕FCREATE which will create component file with level 3 journaling, and checksum enabled. Then:

```
      newfid←'newfile' JFCOPY 1
```

Note: Setting ('U' 0) (no Unicode support) is discouraged as it may cause the copy to fail with a TRANSLATION ERROR.

Chapter 5:

I-Beam Reference Changes

I-Beam Changes

In the following tables, `A` is an integer that specifies the type of operation to be performed.

I-Beam functionality added to Version 14.1.

A	Derived Function
85	Execute Expression
127	Overwrite Free Pockets
900	Called Monadically
950	Loaded Libraries
2017	Identify .NET Type
2035	Set Dyalog Pixel Type
2101	Close .NET AppDomain
2503	Mark Thread as Uninterruptible
2520	Spawn .NET Thread
3500	Send Text to RIDE-embedded Browser
3501	Connected to the RIDE
3502	Enable RIDE in Run-time Interpreter
7159	JSON Import
7160	JSON Export
7161	JSON TrueFalse
7162	JSON Translate Name
50100	Line Count

Experimental Features-related I-Beams

Dyalog APL includes a number of I-Beams which exist in order to support experimental features, and features which are associated with the interpreter.

The following table lists those I-Beams, together with the document which contains a description of them:

Table 1: Experimental and other I-Beams

A	Purpose	Where documented
400	Compiler	Dyalog APL Experimental Features - Compiler
8659	External Workspaces	Dyalog APL Experimental Features - External Workspaces
8666		
8667		

Execute Expression   R←X(85I)Y

Executes an expression.

`Y` is a character vector containing an APL expression.

The function executes the expression in `Y` exactly as it would be executed by the monadic Execute primitive function `⍎`, but handles shy results of the execution rather differently.

The left argument `X` determines how a shy result from the execution of `Y` is treated and is either 0 or 1.

If `X` is 1, and the expression in `Y` returns an explicit result, `R` is that result. If the expression in `Y` returns no result or returns a shy result, the function signals `ERROR 85`. Effectively, a shy result is discarded.

If `X` is 0, and the expression in `Y` returns an explicit result or a shy result, `R` is that result (but is no longer shy). If the expression in `Y` returns no result, the function signals `ERROR 85`.

Examples

```
      ⍎'a←42'
      ⎕← ⍎'a←42'      ⍝ shy result
42
      0 (85I) 'a←42' ⍝ not shy
42
      1 (85I) 'a←42'
ERROR 85
      1(85I)'a←42'
    ∧
```

Overwrite Free Pockets   R←127ɪY

Overwrites all free pockets in the workspace.

Some applications (cryptography for example) make use of secure data during execution. The nature of the APL workspace is such that remnants of this secure data may persist in the workspace (and thus the process memory) even after the relevant APL variables have been expunged. This function overwrites all unused data pockets in the workspace so that any potentially secure data is removed.

Y is any empty array, preferably zilde θ, R is always 1.

It is the responsibility of the programmer to ensure that there are no USED pockets in the workspace that reference the data.

Example

```
      ∇ foo;a
[1]    a←'my secure data'
[2]    ⎕EX'a'
[3]    ⍝ 'my secure data' is now in an
[4]    ⍝ UNUSED pocket in the workspace
[5]    a←127ɪ0 ⍝ all unused pockets are overwritten,
[6]            ⍝ 'my secure data' is no longer present
      ∇
```

Whereas

```
      ∇ foo;a;b
[1]    a←'my secure data'
[2]    b←a
[3]    ⎕EX'a'
[4]    ⍝ 'my secure data' is now in an
[5]    ⍝ UNUSED pocket in the workspace
[6]    a←127ɪ0 ⍝ all unused pockets are overwritten,
[7]            ⍝ but 'my secure data' is still present
[8]            ⍝ because it is referenced by b
      ∇
```

Called Monadically   R←900IY

Identifies how the current function was called. `900I` applies only when called from within a variadic defined function (not a dfn).

`Y` may be any array.

The result `R` is boolean. 1 means that the current function was called monadically; 0 means that it wasn't. If there is no function on the stack, the result is 0.

Example

```
      ∇ r←{left}foo right
[1]    r←900I0
      ∇
       foo 10
1
       0 foo 10
0
```

Loaded Libraries   R←950IY

Reports the names of the dynamic link libraries that are currently loaded as a result of executing ⎕NA.

Y is an empty vector.

The result R is a vector of character vectors containing the names of all the DLLs or shared libraries that have been **explicitly** loaded by ⎕NA and are still loaded by virtue of the presence of at least one external function reference.

Examples

```
      )clear
clear ws
      'Aloc'⎕NA'P kernel32|GlobalAlloc U4 P'
      'Free'⎕NA'P kernel32|GlobalFree P'
      'Lock'⎕NA'P kernel32|GlobalLock P'
      'Ulok'⎕NA'U4 kernel32|GlobalUnlock P'
      'Valu'⎕NA'U4 version|VerQueryValue* P <0T >U4 >U4'
      'copy'⎕NA'P msvcrt|memcpy >U4[] P U4'

      950I0
 KERNEL32.DLL   VERSION.DLL   MSVCRT.DLL
      )fns
Aloc    Free    Lock    Ulok    Valu    copy

      )erase Aloc   Free    Lock    Valu
      950I0
 KERNEL32.DLL   MSVCRT.DLL
      )fns
Ulok    copy

      )erase Ulok
      950I0
 MSVCRT.DLL

      )clear
clear ws
      950I0
```

Identify NET Type   `R←2017IY`

Windows only.

Returns the .NET Type of a named .NET class that is loaded in the current AppDomain. Note that `System.Type.GetType` requires the fully qualified name, i.e. prefixed by the assembly name, whereas (`2017I`) does not.

`Y` is a character string containing the name of a .NET object. Unless the fully qualified name is given, the namespaces in the current AppDomain are searched in the order they are specified by `⎕USING` or `:Using`.

If the object is identified in the current AppDomain, the result `R` is its Type. If not, the function generates `DOMAIN ERROR`.

Example

```
      ⎕USING←'System'
      2017I'DateTime'
System.DateTime
```

Set Dyalog Pixel Type   R←2035IY

Determines how Coord `'Pixel'` is interpreted. This is determined initially by the value of the DYALOG_PIXEL_TYPE parameter and, when altered by this function, applies to all subsequent GUI operations.

`Y` is a character vector that is either `'ScaledPixel'` or `'RealPixel'`. Any other value will cause a `DOMAIN ERROR`.

The result `R` is the previous value.

Example

```
      2035I'ScaledPixel'
RealPixel
      2035I'RealPixel'
ScaledPixel

      2035I'realpixel'
DOMAIN ERROR
      2035I'realpixel'
      ∧
```

Close .NET AppDomain   R←2101⌶Y

Windows only.

This function closes the current .NET AppDomain.

Y may be any array and is ignored.

The result R is 0 if the operation succeeded or a non-zero integer if it failed.

This I-Beam is very likely to be changed in future.

Mark Thread as Uninterruptible   R←2503ɪY

This function marks the current thread (the thread in which it is called) as uninterruptible, and/or determines whether or not any child threads, subsequently created by the current thread, will be uninterruptible.

The right argument Y is an integer whose value is the sum of the following (bit-wise) values:

- 1 : mark thread as uninterruptible
- 2 : mark its children as uninterruptible

The result R is an integer value that indicates the previous state of the thread.

In many multi-threaded applications a large proportion of the threads are used for communication mechanisms (⎕DQ on TCPsockets, Conga, isolates); but most of the "real work" is done in thread zero.

It is undesirable that a weak interrupt interrupts a seemingly random thread. The mechanism to prevent a thread from being (weak) interrupted allows an application to be configured so that only specific threads would respond to a weak interrupt.

Use Separate Thread For .NET   R←2520IY

This function determines the way that .NET calls are executed in APL thread 0.

The right argument Y is a boolean value:

- 1 : run .NET calls in a separate system thread
- 0 : run .NET calls in the same system thread

The result R is a boolean value which indicates the previous behaviour.

When an APL **thread** first makes a .NET call, it creates a unique system thread in which that and subsequent .NET calls are made. If a .NET call results in the creation of a message queue, that queue is associated with that same system thread. So each message queue is also unique. This strategy successfully maintains separation between multiple Windows message queues being executed in different APL threads.

By default, the base APL thread (thread 0) runs .NET code in the same system thread as itself. This is a different system thread to that used to run .NET code from other APL threads, so the separation between message queues associated with APL thread 0 and those associated with other APL threads is maintained. However, in certain circumstances, messages generated by .NET objects interfere with APL's internal message processing (and vice-versa), for example when handling exceptions.

For this reason, Dyalog recommends that APL code that creates instances of .NET objects that generate events (such as Windows Presentation Foundation objects) are run in a separate APL thread.

Where this is not possible, 2520I1 may be used to force Dyalog to use a unique system thread for .NET that is associated with APL thread 0. If so, it is recommended that 2520I1 is called at application start-up time.

Send Text to RIDE-embedded Browser   `R←{X}(3500ɪ)Y`

Optionally, `X` is a simple character vector or scalar, the contents of which are used as the caption for the tab in the RIDE client that contains the embedded browser. If omitted, then the caption defaults to "`3500ɪ`".

`Y` is a simple character vector the contents of which are displayed in the embedded browser tab.

To include SVG content, the HTML text in `Y` must include the following:

```
 <meta http-equiv="X-UA-Compatible" content="IE=9" >.
```

The result `R` identifies whether the write to the RIDE was successful. Possible values are:

- `0` : the write to the RIDE client was successful
- `¯1` : the write to the RIDE client was not successful

Connected to the RIDE   `R←{X}(3501ɪ)Y`

`X` and `Y` can be any value and are ignored.

The result `R` identifies whether the Dyalog Session is running through the RIDE. Possible values are:

- `0` : the write to the RIDE client was successful
- `¯1` : the write to the RIDE client was not successful

Enable RIDE in Run-time Interpreter   `3502⌶0`

By default, the RIDE is not enabled on run-time executables.

For security reasons, enabling the RIDE is a two-step process rather than using (for example) a single environment variable. To enable the RIDE, two steps must be taken:

1. Set the **RIDE_INIT** configuration parameter on the machine on which the run-time interpreter is running to an appropriate value.
2. Execute `3502⌶0` in your application code

The run-time interpreter can then attempt to connect to a RIDE client.

Note:

Enabling the RIDE to access applications that use the run-time interpreter means that the APL code of those applications can be accessed. The I-Beam mechanism described above means that the APL code itself must grant the right for a RIDE client to connect to the run-time interpreter. Although Dyalog Ltd might change the details of this mechanism, the APL code will always need to grant connection rights. In particular, no mechanism that is only dependent on configuration parameters will be implemented.

JSON Import   R←X(7159I)Y

Imports text in JavaScript Object Notation (JSON) Data Interchange Format[1].

`Y` is a character vector or matrix in JSON format. There is an implied newline character between each row of a matrix.

The optional left argument `X` specifies the conversion format and is a scalar or a vector singleton 0, 1 or 2 as follows. If not specified, the default value is 0.

Import as Object (X is 0)

- JSON objects are created as APL namespaces.
- JSON null is converted to `⎕NULL`.
- JSON true is converted to the enclosed character vector `⊂'true'`
- JSON false is converted to the enclosed character vector `⊂'false'`. The values for true and false can be obtained using `7161I`.
- If the JSON source contains object names which are not valid APL names they are converted to APL objects with mangled names. See *JSON Name Mangling* on page 80 . `7162I` can be used to obtain the original name.

Note: It is highly likely that the JSON-related I-beams will be superseded by a JSON system function in a future release of Dyalog APL. It is strongly recommended that you place all code that references the JSON I-beams in cover functions.

---

[1]IETF RFC 7159

Example

```
      ρJSON
18 19
      JSON
{
  "a": {
    "b": [
      "string 1",
      "string 2"
    ],
    "c": true,
    "d": {
      "e": false,
      "fα": [
        "string 3",
        123,
        1000.2,
        null
      ]
    }
  }
}
      j←0 (7159⌶) JSON ⍝ Import JSON as namespace
      j.⎕NL 9
a
      j.a.⎕NL 2
b
c

      j.a.b
```

string 1	string 2

```
      j.a.c
```

true

```
      j.a.⎕NL 9
d
      j.a.d.⎕NL 2 ⍝ Note that fα is an invalid APL name
e
Δ_102_9082

      j.a.d.e
```

false

```
      j.a.d.Δ_102_9082
```

string 3	123	1000.2	[Null]

Import as a 4-col matrix (X is 1)

The columns contain the following:

`[;1]`	depth
`[;2]`	name (for JSON object members)
`[;3]`	value
`[;4]`	JSON type (integer: see below)

- The representation of null, true and false are the same as for conversion format 0.
- Object names are reported as specified in the JSON text; they are not mangled as they are for conversion format 0.

JSON types are as follows:

Type	Description
1	Object
2	Array
3	Numeric
4	String
5	Null
6	Boolean (true / false)

Table 2: JSON data types

Example (JSON is as before)

```
      1 (7159⌶) JSON
```

0			1
1	a		1
2	b		2
3		string 1	4
3		string 2	4
2	c	true	6
2	d		1
3	e	false	6
3	fα		2
4		string 3	4
4		123	3
4		1000.2	3
4		[Null]	5

Import as a 3-col matrix (X is 2)

The columns contain the following:

[;1]	name (for JSON object members)
[;2]	value
[;3]	JSON type (integer: see above)

Example:

```
2 (7159⌶) JSON
```
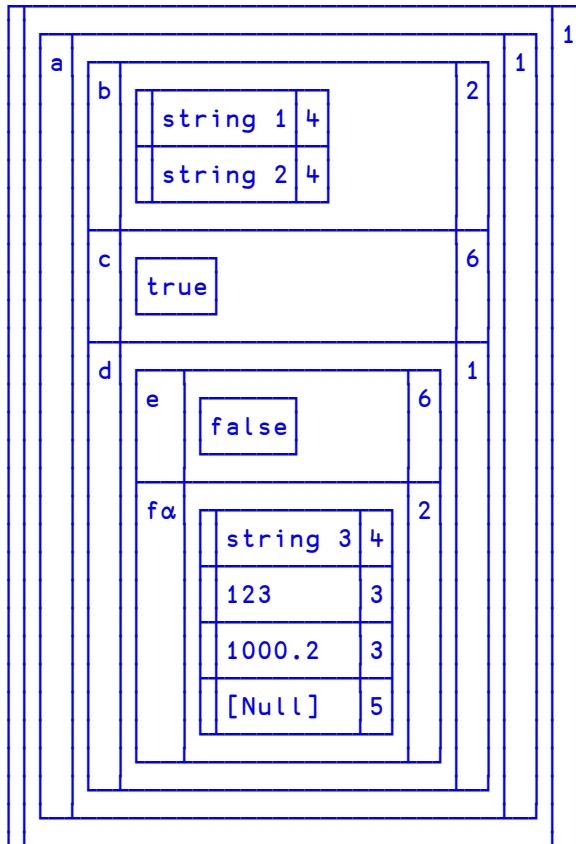

Duplicate Names

The JSON standard says that members of a JSON object should have unique names and that different implementations behave differently when there are duplicates. Dyalog handles duplicate names as follows:

- No error is generated
- For format 0, the last member encountered is used and all previous members with the same name are discarded
- For other formats all duplicate members are recorded in the result matrix

The JavaScript Object Notation

IETF RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format - is a widely supported, text based data interchange format for the portable representation of structured data; any application which conforms to the standard may exchange data with any other. Note, however, that the standard describes a limited set of data types and JSON does not provide a general APL import/export mechanism. In particular:

Not all APL arrays are representable in JSON.

For example, arrays with more than one dimension cannot be represented in JSON. Of course, this does mean that applications using JSON are unlikely to use such objects; you probably will need rearrange your data into the format that is expected by the receiving application. In the case of a 2-dimensional matrix, a split will give you a vector of tuples that a JSON application is likely to expect:

```
      (7160I) 3 4⍴⍳12
DOMAIN ERROR: Array unsupported by JSON
      (7160I) 3 4⍴⍳12
     ∧
      (7160I) ↓3 4⍴⍳12
[[1,2,3,4],[5,6,7,8],[9,10,11,12]]
```

Not all APL object names are representable in JSON.

Objects with names which cannot be represented in JSON are renamed on both import and export as described in JSON Name Mangling on page XX.

JSON true and false

The JSON standard includes Boolean values true and false which are distinct from numeric values 1 and 0, and have no direct APL equivalent.

To represent JSON true and false types this implementation adopts the convention of using APL arrays `⊂'true'` and `⊂'false'` respectively. These arrays cannot otherwise be represented in JSON and allow true and false to be uniquely identified. See also: .

JSON Export   R←X(7160I)Y

Exports text in JavaScript Object Notation (JSON) Data Interchange Format[1].

Y is the data to be exported as JSON, in one of the formats imported by `7159I`.

If X is specified it must be a scalar or a vector containing at most 3 elements as follows:

1. Conversion format - an integer between 0 and 2 inclusive; defaults to 0 if omitted
2. Generate compact JSON if 0; generate formatted JSON if non-zero; defaults to 0 if omitted
3. Tolerate data which cannot be exported as JSON if zero (only valid if conversion format is 0); error if data cannot be exported if non-zero; defaults to 1 if omitted.

The result R is a character vector.

If invalid JSON data is tolerated it is represented in R as an asterisk. This allows invalid data to be located within the array.

The name of any namespace member that begins with Δ and otherwise conforms to the conversion format used for JSON object names will be included.

Note: It is highly likely that the JSON-related I-beams will be superseded by a JSON system function in a future release of Dyalog APL. It is strongly recommended that you place all code that references the JSON I-beams in cover functions.

---

[1]IETF RFC 7159

Example

```
      j
#.[JSON object]

      ⍴JS←0(7160⌶)j
94
      JS
{"a":{"b":["string 1","string 2"],"c":true,"d":{"e":false
,"fα":["string 3",123,1000.2,null]}}}


      0 1(7160⌶)j
{
  "a": {
    "b": [
      "string 1",
      "string 2"
    ],
    "c": true,
    "d": {
      "e": false,
      "fα": [
        "string 3",
        123,
        1000.2,
        null
      ]
    }
  }
}
```

Note:

If an error is detected in the matrix input in conversion formats 1 and 2 the error
description identifies the relevant row number. This is ⎕IO sensitive. Conversion
format 2 allows nesting of matrices within matrices and the row may be identified by
multiple comma separated numbers: the first number is the row number of the out-
ermost matrix, the second is the row number of the next level matrix, and so on. If the
matrices are nested very deeply this will be truncated to the innermost matrix row
number and depth.

See also: *The JavaScript Object Notation* on page 75.

JSON TrueFalse   R←7161IY

This function returns the value of the APL array to which the JSON values of true and false are converted by *JSON Import* (7159I). It is also the value from which the JSON values of true and false are converted by *JSON Export* (7160I).

Y is a scalar or 1-element vector with the value 0 or 1.

If Y is 1, R is the APL equivalent of JSON true. If Y is 0, R is the APL equivalent of JSON false.

This function is provided to permit the programmer to avoid hard-coding the current values used to represent JSON true and false, as these may change in the future. It is also slightly faster to use this function for comparison than to hard-code the values.

See also: *The JavaScript Object Notation* on page 75.

Note: It is highly likely that the JSON-related I-beams will be superseded by a JSON system function in a future release of Dyalog APL. It is strongly recommended that you place all code that references the JSON I-beams in cover functions.

Examples

```
      7161I1
 true

      7161I0
 false
```

JSON Translate Name   R←X(7162I)Y

Converts between JSON names and APL names.

When `0(7159I)` imports an entity from JSON text whose name would be an invalid APL name, the function converts the invalid name into a valid APL name using a *name mangling* algorithm. For details, see *JSON Name Mangling* on page 80. When `0(7160I)` exports an APL namespace as JSON text, the process is reversed.

This function performs the same *name mangling* allowing the programmer to identify JSON entities as APL names, and vice-versa.

`Y` is a character vector or scalar.

`X` is a scalar numeric value which must be 1 or 0.

When `X` is 0, `R` is the name in `Y` converted, if necessary, so that it is a valid APL name. It performs the same translation of JSON object names to APL names that is performed when importing JSON.

When `X` is 1, `R` is the name in `Y` which, if mangled, is converted back to the original form.. It performs the same translation of APL names to JSON object names that is performed when exporting JSON.

Note: It is highly likely that the JSON-related I-beams will be superseded by a JSON system function in a future release of Dyalog APL. It is strongly recommended that you place all code that references the JSON I-beams in cover functions.

Examples:

```
      0(7162I)'2a'
∆_50_97
      1(7162I)'∆_50_97'
2a
      0(7162I)'foo'
foo
      1(7162I)'foo'
foo
      1(7162I)'∆_97_'
∆_97_
```

Note that the algorithm can be applied, even when mangling is not required. So:

```
      1(7162I)'∆_97'
a
```

For further details, see *JSON Name Mangling* on page 80.

JSON Name Mangling

When Dyalog converts from JSON to APL data, and a member of a JSON object has a name which is not a valid APL name, it is renamed.

Example:

In this example, the JSON describes an object containing two numeric items, one named *a* (which is a valid APL name) and the other named *2a* (which is not):

```
{"a": 1, "2a": 2}
```

When this JSON is imported as an APL namespace using `0(7159I)`, Dyalog converts the name *2a* to a valid APL name. The *name mangling* algorithm creates a name beginning with `∆`, followed by the `⎕UCS` of each of the characters in the JSON name preceded by an underscore (`_`).

```
      (0(7159I)'{"a": 1, "2a": 2}').⎕NL 2
a
∆_50_97
```

When Dyalog exports JSON it performs the reverse *name mangling*, so:

```
      0 1(7160I)0(7159I)'{"a": 1, "2a": 2}'
{
  "a": 1,
  "2a": 2
}
```

Should you need to create and decode these names directly, `7162I` provides the same name mangling and un-mangling operations.

```
      0(7162I)'2a'
∆_50_97
      1(7162I)'∆_50_97'
2a
```

Line Count   R←50100IY

This function is a compact version of the system function `⎕LC`. If an expression requires only the most recent line(s) in the function calling stack, this is a more efficient alternative to using `⎕LC`.

`Y` may be an integer specifying the depth of the function calling stack that is required in the result.

The result R is the same as `⎕LC`, but truncated to the number of stack levels specified by `Y`.

Example

```
          ∇ Foo
[1]     :If 4=ρ⎕LC
[2]         50100I0
[3]         50100I1
[4]         50100I2
[5]         50100I3
[6]         50100I4
[7]         50100I5
[8]         →
[9]     :Else
[10]        Foo
[11]    :EndIf
      ∇

      Foo

3
4 10
5 10 10
6 10 10 10
7 10 10 10
```

Chapter 6:

Object Reference Changes

Coord	Property

Applies To: ActiveXControl, Animation, Bitmap, Button, ButtonEdit, Calendar, Circle, ColorButton, Combo, ComboEx, DateTimePicker, Edit, Ellipse, Font, Form, Grid, Group, Image, Label, List, ListView, Locator, Marker, MDIClient, Menu, Metafile, Poly, Printer, ProgressBar, PropertyPage, PropertySheet, Rect, RichEdit, Root, Scroll, SM, Spinner, Splitter, Static, StatusBar, StatusField, SubForm, TabBar, Text, ToolBar, TrackBar, TreeView, UpDown

Description

This property defines an object's co-ordinate system. It is a character string with one of the following values; `'Inherit'`, `'Prop'`, `'Pixel'`, `'RealPixel'`, `'ScaledPixel'`, `'User'` or `'Cell'` (graphics children of a Grid only).

If Coord is `'Inherit'`, the co-ordinate system for the object is **inherited** from its parent. Note that the default value of Coord for the system object `'.'` is `'Prop'`, so by default all objects created by ⎕WC inherit `'Prop'`.

If Coord is `'Prop'`, the origin of the object's parent is deemed to be at its top left interior corner, and the scale along its x- and y-axes is 100. The object's position and size (Posn and Size properties) are therefore specified and reported as a percentage of the dimensions of the parent object, or, for a Form, of the screen.

If Coord is `'RealPixel'`, the origin of the object's parent is deemed to be at its top left interior corner, and the scale along its x- and y-axes is measured in physical pixel units. The object's position and size (Posn and Size properties) are therefore reported and set in physical pixel units. If you set Coord on the system object to `'Pixel'`, the value of its Size property gives you the resolution of your screen. Note that pixels are numbered from 0 to (Size -1).

If Coord is `'ScaledPixel'` the number of pixels specified for Posn, Size, and other such properties will be automatically scaled by Dyalog APL according to the user's chosen display scaling factor. So if you specify an Edit object to be 80 pixels wide and 20 pixels high, and the user's scaling factor is 150%, Dyalog will automatically draw it 120 pixels wide and 30 pixels high. Dyalog will also de-scale coordinate values reported by `⎕WG` and event messages.

If Coord is `'Pixel'`, it is interpreted as either `'RealPixel'` or `'ScaledPixel'` according to the value of the **DYALOG_PIXEL_TYPE** parameter, which is either ScaledPixel or RealPixel. See *Installation & Configuration Guide: DYALOG_PIXEL_TYPE parameter*.

If this parameter is not specified, the default is RealPixel. So by default, when you set Coord to Pixel, it will be treated as RealPixel.

If Coord is `'User'`, the origin and scale of the co-ordinate system are defined by the values of the YRange and XRange properties **of the parent object**. Each of these is a 2-element numeric vector whose elements define the co-ordinates of top left and bottom right interior corners of the (parent) object respectively.

Note that if Coord is `'User'` and you change the values of YRange and/or XRange of the parent, the object (and all its siblings with Coord `'User'`) are redrawn (and clipped) according to the new origin and scale defined for the parent. The values of their Posn, Size and Points properties are unaffected. Changing YRange and/or XRange therefore provides a convenient and efficient means to "**pan and zoom**".

The Coord property for graphic objects created as children of a Grid may also be set to Cell. Apart from being easier to compute, a graphic drawn using cell coordinates will expand and contract when the grid rows and columns are resized.

Example:

This statement creates a button 10 pixels high, 20 pixels wide, and 5 pixels down and along from the top-left corner of the parent Form T.

```
'T.B1'⎕WC'Button' 'OK'(5 5)(10 20)('Coord' 'Pixel')
```

If you set Coord to `'RealPixel'` in the Root object `'.'`, then query its Size, you get the dimensions of the screen in pixels, i.e.

```
'.' ⎕WS 'Coord' 'RealPixel'
'.' ⎕WG 'Size'
480 640
```

If you set Coord to `'ScaledPixel'` in the Root object `'.'`, then query its Size, you get the virtual resolution of the screen, i.e.

```
'.'⎕WS 'Coord' 'ScaledPixel'
'.'⎕WG'Size'
1080 1920
```

GesturePan   Event 494

Applies To: ActiveXControl, Animation, Button, ButtonEdit, Calendar,
ColorButton, Combo, ComboEx, DateTimePicker, Edit, Form,
Group, List, ListView, MDIClient, ProgressBar, PropertyPage,
RichEdit, Scroll, Spinner, SubForm, TreeView

Description

This event is reported when the user touches one or two fingers on an object and
drags them .

The event message reported as the result of ⎕DQ, or supplied as the right argument to
your callback function, is a 5-element vector as follows :

[1]	Object	ref or character vector
[2]	Event	'GesturePan' or 494
[3]	Flags	integer which reports the state of the gesture
[4]	Location	2-element integer vector containing the y and x-position respectively of the point at which the gesture applies. These are reported in pixel coordinates relative to the origin (top-left corner) of the object reporting the event.
[5]	Distance	2-element integer vector containing the high and low parts (words) of a 64-bit integer that indicates the distance between the two fingers. This will be (0 0) if only one finger is used.

The Flags parameter [3] which reports the state of the Gesture, is an integer with the
value 0, 1 (*GF_BEGIN*), 2 (GF_INERTIA), 4 (*GF_END*) or 6 (*GF_END+GF_
INERTIA*) with the following meanings:

Name	Value	Description
	0	A gesture is in progress
GF_BEGIN	1	A gesture is starting.
GF_INERTIA	2	A gesture has triggered inertia.
GF_END	4	A gesture has finished.

The term *inertia* refers to built-in Windows processing which provides a standardised
user-interface including smooth acceleration and de-acceleration of an object.

When the user first touches an object and begins to drag his finger(s), the object generates a GesturePan event with a `Flags` parameter of 1 (GF_BEGIN). Subsequently, if the user drags the object steadily it generates a series of GesturePan events with a `Flags` parameter of 0. When the user lifts his finger(s) away, the object generates a final GesturePan event, with a `Flags` parameter of 4 (GF_END).

If the user *flicks* an object, the system typically continues to generate GesturePan events after the user has ceased to touch the object. These events are generated in response to the acceleration and deceleration imparted by the *flick*, and the `Flags` parameter for these generated events will be 2 (GF_INERTIA) followed (for the last GesturePan event) by 6 (GF_END+GF_INERTIA).

No other event will be reported between the start and end of a series of GesturePan events.

The associated callback is run **immediately** while the windows notification is still on the stack. See *Interface Guide: High-Priority Callback Functions*.

Returning zero from the callback disables any default handling by the operating system.

GesturePressAndTap   Event 497

Applies To: ActiveXControl, Animation, Button, ButtonEdit, Calendar,
ColorButton, Combo, ComboEx, DateTimePicker, Edit, Form,
Group, List, ListView, MDIClient, ProgressBar, PropertyPage,
RichEdit, Scroll, Spinner, SubForm, TreeView

Description

This event is reported when the presses one finger on an object and then taps it with a
second finger.

The event message reported as the result of ⎕DQ, or supplied as the right argument to
your callback function, is a 5-element vector as follows :

[1]	Object	ref or character vector
[2]	Event	`'GesturePressAndTap'` or 497
[3]	Flags	integer which reports the state of the gesture
[4]	Location	2-element integer vector containing the y and x-position respectively of the point midway between the two fingers. These are reported in pixel coordinates relative to the origin (top-left corner) of the object reporting the event..
[5]	Offset	3-element integer vector whose first element is (currently) 0 and whose second and third elements contain the (y,x) offset of the second finger relative to the first.

The Flags parameter [3] which reports the state of the Gesture, is an integer with the
value 0, 1 (*GF_BEGIN*), or 4 (*GF_END*):

Name	Value	Description
	0	A gesture is in progress
GF_BEGIN	1	A gesture is starting.
GF_END	4	A gesture has finished.

When the user taps with his second finger, the object generates a Ges-
turePressAndTap event with a `Flags` parameter of 1 (GF_BEGIN). Subsequently,
until the user removes his first finger, it generates a series of GesturePressAndTap
events with a `Flags` parameter of 0. When the user lifts his first finger away, the
object generates a final GesturePressAndTap event, with a `Flags` parameter of 4
(GF_END)

No other event will be reported between the start and end of a series of Ges-
turePressAndTap events.

The associated callback is run **immediately** while the windows notification is still on the stack. See *Interface Guide: High-Priority Callback Functions*.

Returning zero from the callback disables any default handling by the operating system.

GestureRotate   Event 495

Applies To: ActiveXControl, Animation, Button, ButtonEdit, Calendar,
ColorButton, Combo, ComboEx, DateTimePicker, Edit, Form,
Group, List, ListView, MDIClient, ProgressBar, PropertyPage,
RichEdit, Scroll, Spinner, SubForm, TreeView

Description

This event is reported when the user touches two fingers on an object and twists them
clockwise or anticlockwise.

The event message reported as the result of ⎕DQ, or supplied as the right argument to
your callback function, is a 5-element vector as follows :

[1]	Object	ref or character vector
[2]	Event	`'GestureRotate'` or 495
[3]	Flags	integer which reports the state of the gesture
[4]	Location	2-element integer vector containing the y and x-position respectively of the point midway between the two fingers. These are reported in pixel coordinates relative to the origin of the screen.
[5]	Angle	a scalar number which represents the angle of rotation of the twist measured in radians ($0 \rightarrow \circ 2$) from the x-axis in a counter-clockwise direction.

The Flags parameter [3] which reports the state of the Gesture, is an integer with the
value 0, 1 (*GF_BEGIN*),or 4 (*GF_END*) with the following meanings:

Name	Value	Description
	0	A gesture is in progress
GF_BEGIN	1	A gesture is starting.
GF_END	4	A gesture has finished.

When the user first touches two fingers on an object and begins to twist, the object generates a GestureRotate event with a `Flags` parameter of 1 (GF_BEGIN). As the user continues to twist his fingers, the object generates a series of GestureRotate events with a `Flags` parameter of 0. When the user lifts one or both fingers away, the object generates a final GestureRotate event, with a `Flags` parameter of 4 (GF_END).

No other event will be reported between the start and end of a series of GestureRotate events.

The associated callback is run **immediately** while the windows notification is still on the stack. See *Interface Guide: High-Priority Callback Functions*.

Returning zero from the callback disables any default handling by the operating system.

GestureTwoFingerTap   Event 496

Applies To: ActiveXControl, Animation, Button, ButtonEdit, Calendar, ColorButton, Combo, ComboEx, DateTimePicker, Edit, Form, Group, List, ListView, MDIClient, ProgressBar, PropertyPage, RichEdit, Scroll, Spinner, SubForm, TreeView

Description

This event is reported when the user taps two fingers at the same time on an object

The event message reported as the result of □DQ, or supplied as the right argument to your callback function, is a 5-element vector as follows :

[1]	Object	ref or character vector
[2]	Event	`'GestureTwoFingerTapn'` or 496
[3]	Flags	integer which reports the state of the gesture
[4]	Location	2-element integer vector containing the y and x-position respectively of the point midway between the two fingers. These are reported in pixel coordinates relative to the origin (top-left corner) of the object reporting the event..
[5]	Distance	2-element integer vector containing the high and low parts (words) of a 64-bit integer that indicates the distance between the two fingers.

The Flags parameter [3] which reports the state of the Gesture, is always an integer with the value 5 (*GF_BEGIN+GF_END*).

Name	Value	Description
GF_BEGIN	1	A gesture is starting.
GF_END	4	A gesture has finished.

The associated callback is run **immediately** while the windows notification is still on the stack. See *Interface Guide: High-Priority Callback Functions*.

Returning zero from the callback disables any default handling by the operating system.

GestureZoom   Event 493

Applies To: ActiveXControl, Animation, Button, ButtonEdit, Calendar,
ColorButton, Combo, ComboEx, DateTimePicker, Edit, Form,
Group, List, ListView, MDIClient, ProgressBar, PropertyPage,
RichEdit, Scroll, Spinner, SubForm, TreeView

Description

This event is reported when the user touches two fingers on an object and moves
them apart or towards each other.

The event message reported as the result of ⎕DQ, or supplied as the right argument to
your callback function, is a 5-element vector as follows :

[1]	Object	ref or character vector
[2]	Event	'GestureZoom' or 493
[3]	Flags	integer which reports the state of the gesture
[4]	Location	2-element integer vector containing the y and x-position respectively of the centre point of the zoom (the point midway between the two fingers). These are reported in pixel coordinates relative to the origin (top-left corner) of the object reporting the event.
[5]	Distance	2-element integer vector containing the high and low parts (words) of a 64-bit integer that indicates the distance between the two fingers.

The Flags parameter [3] which reports the state of the Gesture, is an integer with the
value 0, 1 (*GF_BEGIN*),or 4 (*GF_END*) with the following meanings:

Name	Value	Description
	0	A gesture is in progress
GF_BEGIN	1	A gesture is starting.
GF_END	4	A gesture has finished.

When the user first touches two fingers on an object and begins to move them apart or towards each other, the object generates a GestureZoom event with a `Flags` parameter of 1 (`GF_BEGIN`). As the user continues to moves the fingers apart or towards each other, the object generates a series of GestureZoom events with a `Flags` parameter of 0. When the user lifts one or both fingers away, the object generates it generates a final GestureZoom event, with a `Flags` parameter of 4 (`GF_END`).

No other event will be reported between the start and end of a series of GestureZoom events.

The associated callback is run **immediately** while the windows notification is still on the stack. See *Interface Guide: High-Priority Callback Functions*.

Returning zero from the callback disables any default handling by the operating system.

Event   Property

Applies To: ActiveXContainer, ActiveXControl, Animation, Bitmap, BrowseBox, Button, ButtonEdit, Calendar, Circle, Clipboard, ColorButton, Combo, ComboEx, CoolBand, CoolBar, Cursor, DateTimePicker, Edit, Ellipse, FileBox, Form, Grid, Group, Icon, Image, ImageList, Label, List, ListView, Locator, Marker, MDIClient, Menu, MenuBar, MenuItem, Metafile, MsgBox, OCXClass, OLEClient, OLEServer, Poly, Printer, ProgressBar, PropertyPage, PropertySheet, Rect, RichEdit, Root, Scroll, Separator, SM, Spinner, Splitter, Static, StatusBar, StatusField, SubForm, SysTrayItem, TabBar, TabBtn, TabButton, TabControl, TCPSocket, Text, Timer, TipField, ToolBar, ToolButton, ToolControl, TrackBar, TreeView, UpDown

Description

This property defines how an object responds to user actions. Unlike other properties which only have a single value, this property has a value corresponding to each of the different types of event that may be generated by a particular object. Consequently the syntax for setting the Event property differs from the general syntax that applies to other properties.

Two syntactic forms are allowed:

- A 3 or 4-item vector containing the property name `'Event'`, followed by the Event Type(s), a value which determines the action to be taken, and an optional array that will be supplied as a left argument to the callback function
- A composite vector whose first element contains the property name `'Event'`, followed by a series of 2 or 3-element vectors, each defining the action to be taken for a different Event Type (or types).

Examples

```
'Event' 'MouseUp' 'foo' 88
'Event' ('MouseUp' 'MouseDown') 'foo' 88
'Event' ('MouseUp' 'foo' 88)('MouseDown' 'goo')
```

Like any other property, the Event property can be set using assignment. However, certain special considerations apply which are discussed later.

When you specify the Event property using `⎕WC` or `⎕WS`, the action to be taken for an event type or types is specified by a 2 or 3-element vector containing:

Element	Item	Description
[1]	Type(s)	see below
[2]	Action	numeric scalar or character vector ¯1 inhibit (ignore) event 0 handle event, do not report to APL 1 handle event, then report to APL fn name of callback function fn& name of callback function to be executed *asynchronously* &expr expression to be executed
[3]	Arg	any array (optional)

Event Types

The first element, *Type(s)* may be one of the following:

- A character vector containing an event name (for example `'MouseUp'`)
- A numeric scalar containing an event number (for example `2`). If the number is not one of the built-in event numbers generated by the object, it is assumed to be a user-defined event which can (only) be generated anagrammatically using `⎕NQ`
- A vector of character vectors containing a list of event names, for example (`'MouseDown'` `'MouseUp'`). This may be used as a shortcut to associate several different types of events with the same action
- An Event name preceded by the string `'on'` (for example `'onMouseUp'`)
- An event number preceded by the string `'on'` (for example `'on99'`). This syntax is intended for user-defined events although it can be used with regular events too.

The *onEvent* syntax causes all objects reported in the *event message* (see below) to be identified by a *ref*. Otherwise, objects reported in the event message are identified by name.

Action

Inhibit (¯1)

If *Action* is set to ¯1, the event is inhibited (if possible) by APL. If, for example, you set the action on a KeyPress event to ¯1, all keystrokes for the object in question will be ignored. Similarly, if you set the *action* on a Close event for a Form to ¯1, the user will be unable to close the Form. This is possible because APL intercepts most events before Windows itself takes any action. However, certain events (e.g. focus change events) are not notified to APL until **after** the event has occurred and **after** Windows has itself responded in some way. In these circumstances it is not always practical for APL to undo what Windows has already done, and an action code of ¯1 is treated as if it were 0. For further details, see the individual entries for each event type in this chapter.

Default Processing (0)

If *Action* is set to 0 (the default), the event is processed by APL and Windows in the normal way (this is referred to herein as the *default processing*) but your program is not notified in any way that the event has occurred. For example, the default processing for a keystroke is to action it and either echo a character in the object or perform some other appropriate function.

Terminate ⎕DQ (1)

If *Action* is set to 1, the event is first processed by APL (and Windows) in the normal way, then ⎕DQ terminates, returning an *event message* as its result. The format of the event message is given under the description of each event type.

Callback (function name)

If *Action* is set to a character vector that specifies the name of a function, this function (termed a *callback*) will be executed automatically by ⎕DQ every time the event occurs. The function may be a traditional defined function or a dfn.

A traditional defined function may be monadic, dyadic, or niladic. If dyadic, the left argument may be optional. A niladic callback may be appropriate if the function can perform its task without needing to interrogate the event message.

Unless the callback function is niladic, it will be supplied a right argument (ω for a dfn) containing the *event message* and a left argument (α for a dfn) of the value of the array *Arg* (if specified).

The function may be defined to return no result, a result, or a shy result. The result determines how the event is handled.

The *default processing* of the event is deferred until **after** the callback has been run, and may be inhibited or modified by its result. If the callback function returns no result, or returns a scalar 1, normal processing of the event is allowed to continue as soon as the callback completes. If the callback returns a scalar 0, normal processing of the event is inhibited and the effect is identical to setting *Action* to ¯1. A callback function may also return an event message as its result. If so, ⎕DQ will action *this* event rather than the original one that fired the callback.

If a callback function does not exist at the instant it is invoked, ⎕DQ terminates with a **VALUE ERROR**. However, the name of the missing function is reported in the Status Window.

Asynchronous Callback (function name followed by &)

If *Action* is set to a character vector that specifies the name of a callback function, followed by the character &, the callback function will be executed asynchronously in a new thread when the event occurs.

For example, the event specification:

```
'Event' 'onSelect' 'DoIt&'
```

tells ⎕DQ to execute the callback function DoIt asynchronously as a thread when a Select event occurs on the object. Note that a callback function executed in this way should not return a result (because ⎕DQ does not wait for it) and any result will be displayed in the Session window.

Execute

If action code is set to a character vector whose first element is the execute symbol (⍎) the remaining string will be executed automatically whenever the event occurs. The default processing for the event is performed first and may not be changed or inhibited in any way.

Notice that when you specify the action to be taken on the occurrence of an event there is a great difference between `'FOO'` and `'⍎FOO'`. The former causes APL to invoke the function `FOO` as a *callback function*. If the function takes an argument, APL will supply it with the event message. Secondly, the result (if any) of the function `FOO` will be used by APL and may cause the event to be disabled or changed in some way. In the second case, APL will perform the default processing for the event and then execute `FOO` without supplying an argument. If the function returns a result, it will be displayed in the Session.

Optional Left Argument (Arg)

If specified, *Arg* is an array whose value will be passed as the left argument to a callback function when that particular event (or events) is generated. Note that this is a constant defined when the value is assigned to the Event property.

If the callback function is defined to take an explicit left argument and *Arg* was not specified, the call will fail with the error message:

```
SYNTAX ERROR: The function requires a left argument
```

If the callback function is defined to take an optional left argment and *Arg* was not specified, a reference to the left argument (α for a dfn) will generate `VALUE ERROR`.

Event Message

When a callback function is invoked by `⎕DQ`, the corresponding event message is supplied as its right argument. The event message is a vector whose first 2 elements identify the object that generated the event and the type of the event. Additional elements may be provided, depending upon the type of the event.

The same event message is returned as a (shy) result by `⎕DQ` when it is terminated by an event whose Action is set to 1.

Object(s)

The first element of the event message always identifies the object that generated the event. Other elements may identify other objects associated with the event. For example, a DragDrop event reports both the object being dropped, and the object on which it is being dropped.

Objects are identified by *names* or *refs*. If the Event property was set using the *onEvent* syntax (whereby the event name or number is prefixed by the string `'on'`), for example, `'onSelect'` or `'on99'`, objects are identified by *refs*. This is also true if the object which generated the event has no name (i.e. was created by `⎕NEW`). Otherwise, objects are identified by ther names.

Event Type

If, when the event type was specified it was identified by its name, the second element of the event message will be a character vector containing that name. If it was identified by its number, the second element of the event message will be an integer containing that number. If the event type was identified using the *onEvent* syntax, the second element of the event message will be a character vector containing the prefix `'on'` followed by the event name, even if it had been specified by number. The exception is that if the event is a user-defined event, the second element of the event message will be a character vector containing the prefix `'on'` followed by the character representation of the user-defined event number.

Specifying the Event property using Assignment

There are two ways to specify the Event property using assignment; you can specify the entire set of events, or you can set events one by one (see below).

To specify the entire set of events, you assign an array to the Event property. The array must contain one or more nested vectors, each containing 2 or 3 elements (*Type*, *Action* and optionally *Arg*) as described above.

Example (`F1` is a Form)

```
F1.Event ← 'onMouseDown' 'FOO'
```

Means: invoke callback function `FOO` on MouseDown, the first element of the right argument to `FOO` will contain a *namespace reference* to `F1`. All other events perform their default actions.

Example

```
F1.Event ← 'MouseDown' 'FOO'
```

Means: invoke callback function `FOO` on MouseDown, the first element of the right argument to `FOO` will contain the *character vector* `'F1'`. All other events perform their default actions.

Example

```
F1.Event ← ('onMouseDown' 'FOO')('onMouseUp' 'FOO')
```

Means: invoke callback function `FOO` on MouseDown and MouseUp. All other events perform their default actions.

Example

```
F1.Event, ← ⊂ 'onMouseMove' 'FOO' ('THIS' 1)
```

Means: add a callback function `FOO` on the MouseMove event. The function will receive the array `('THIS' 1)` as its left argument. All other events perform their default actions.

Specifying Individual Event types using Assignment

To define the action to be taken for individual events, one by one, you use the *onEvent* syntax and make the assignment to the event name prefixed by the string `'on'`.

Example

```
F1.onMouseDown ← 'FOO'
```

Means: invoke callback function `FOO` on MouseDown.

Example

```
F1.onMouseUp ← 'FOO'
```

Means: add the same callback for MouseUp.

Example

```
F1.onMouseMove ← 'FOO' ('THIS' 1)
```

Means: add the same callback function `FOO` for the MouseMove event. The function will receive the array `('THIS' 1)` as its left-argument.

Notice that you must use the `'on'` prefix; you cannot assign to the Event name itself. This would cause an error:

```
     F1.MouseUp←'foo'
SYNTAX ERROR: Invalid modified assignment, or an attempt
was made to change nameclass on assignment
     F1.MouseUp←'foo'
    ^
```

Specifying the Event property using ⎕WC and ⎕WS

When you set the Event property using ⎕WC and ⎕WS you define the actions for the event types that you specify in the argument, leaving the actions for all other event types unchanged. When you create an object with ⎕WC, all unspecified event types will be unhandled; i.e. those events will perform the default processing. However, when you specify the action for a new event type using ⎕WS, any actions previously defined for other event types will remain as they were.

Examples using Event Names

Ignore MouseDown (1) event (APL will perform the default processing for you)

```
'F1' ⎕WS 'Event' 'MouseDown' 0
```

Terminate ⎕DQ on MouseDown

```
'F1' ⎕WS 'Event' 'MouseDown' 1
```

Invoke callback function FOO on MouseDown, the first element of the right argument to FOO will contain a *namespace reference* to F1

```
'F1' ⎕WS 'Event' 'onMouseDown' 'FOO'
```

Invoke callback function FOO on MouseDown, the first element of the right argument to FOO will contain the *character vector* 'F1'

```
'F1' ⎕WS 'Event' 'MouseDown' 'FOO'
```

Invoke callback function FOO on MouseDown and MouseUp

```
'F1' ⎕WS 'Event' ('onMouseDown' 'onMouseUp') 'FOO'
```

Invoke callback function FOO with ('THIS' 1) as its left-argument on MouseDown

```
'F1' ⎕WS 'Event' 'onMouseDown' 'FOO' ('THIS' 1)
```

Invoke callback function FOO with ('THIS' 1) as its left-argument on MouseDown, MouseUp and MouseMove

```
EV ← 'onMouseDown' 'onMouseUp' 'onMouseMove'
'F1' ⎕WS 'Event' EV 'FOO' ('THIS' 1)
```

Execute the expression COUNT+←1 on MouseDown

```
'F1' ⎕WS 'Event' 'MouseDown' '⍎COUNT+←1'
```

Execute the expression COUNT+←1 on MouseDown, MouseUp and MouseMove

```
EV ← 'MouseDown' 'MouseUp' 'MouseMove'
'F1' ⎕WS 'Event' EV '⍎COUNT+←1'
```

Examples using Event Numbers

Ignore MouseDown (1) event (APL will perform the default processing for you)

```
'F1' ⎕WS 'Event' (1 0)
'F1' ⎕WS 'Event' 1 0     ⍝ Ditto
```

Terminate ⎕DQ on MouseDown

```
'F1' ⎕WS 'Event' (1 1)
'F1' ⎕WS 'Event' 1 1     ⍝ Ditto
```

Call function FOO on MouseDown

```
'F1' ⎕WS 'Event' (1 'FOO')
'F1' ⎕WS 'Event' 1 'FOO'     ⍝ Ditto
```

Call function FOO on MouseDown and MouseUp

```
'F1' ⎕WS 'Event' ((1 2) 'FOO')
'F1' ⎕WS 'Event' (1 2) 'FOO'              ⍝ Ditto
'F1' ⎕WS 'Event' 1 2 'FOO'                ⍝ Ditto
'F1' ⎕WS 'Event' (1 'FOO')(2 'FOO')       ⍝ Ditto
```

Call function FOO with ('THIS' 1) as its left-argument on MouseDown

```
'F1' ⎕WS 'Event' (1 'FOO' ('THIS' 1))
'F1' ⎕WS 'Event' 1 'FOO' ('THIS' 1)       ⍝ Ditto
```

Call function FOO with ('THIS' 1) as its left-argument on MouseDown and
MouseUp

```
'F1' ⎕WS 'Event' ((1 2) 'FOO' ('THIS' 1))
'F1' ⎕WS 'Event' (1 2) 'FOO' ('THIS' 1)   ⍝ Ditto
'F1' ⎕WS 'Event' 1 2 'FOO' ('THIS' 1)     ⍝ Ditto
'F1' ⎕WS 'Event' 1 2 'FOO' ('THIS' 1)     ⍝ Ditto
```

Execute the expression COUNT+←1 on MouseDown

```
'F1' ⎕WS 'Event' 1 '⍎COUNT+←1'
```

Execute the expression COUNT+←1 on MouseDown, MouseUp and MouseMove

```
'F1' ⎕WS 'Event' (1 2 3) '⍎COUNT+←1'
'F1' ⎕WS 'Event' 1 2 3 '⍎COUNT+←1'        ⍝ Ditto
```

User defined Events

In addition to the standard events supported directly by Dyalog APL, you may specify your own events. For these, you **must** use event numbers; user-defined event names are not allowed.

You may use any numbers not already defined, but it is strongly recommended that you choose numbers greater than 1000 to avoid potential conflict with future releases of Dyalog APL.

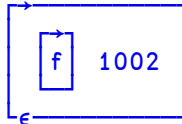You can only **generate** user-defined events under program control with ⎕NQ.

Examples

```
      ∇ foo m
[1]    ⎕SE.UCMD'display m'
      ∇


      'f'⎕WC'Form' ('Event' 1001 'foo')
      f.Event
 1001  #.foo
      ⎕NQ 'f' 1001
```
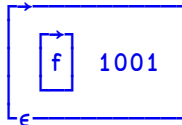


```
      'f' ⎕WS 'Event' 1002 'foo'
      f.Event
 1001  #.foo    1002  #.foo
      ⎕NQ 'f' 1002
```


Notice that if you use the *onEvent* syntax, the event property reports the event type as you specified, but the callback function receives just the number as before.

```
      f.on1003←'foo'
      f.Event
 1001  #.foo    1002  #.foo     on1003  #.foo
      ⎕NQ 'f' 1003
```
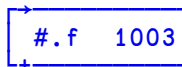
Notes
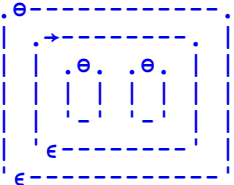
Resetting (clearing) the Event Property

If no events are set, the result obtained by `⎕WG` and the result obtained by referencing Event directly are different:

```
      'F'⎕WC'Form'
      DISPLAY 'F'⎕WG'Event'
.→--.
|0 0|
'~--'
      DISPLAY F.Event
.θ------------.
| .→--------. |
| | .θ. .θ. | |
| | | | | | | |
| | '-' '-' | |
| '∊--------' |
'∊------------'
```

To reset the Event property, the same (different) values must be used accordingly:

```
      f.Event←0⍴⊂'' ''
```

or
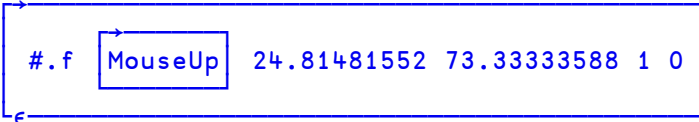
```
      'f'⎕ws'Event' 0 0
```

onEvent Syntax with Event Numbers

If you use the *onEvent* syntax with *built-in* event numbers, the effect is the same as if you had used the event name. This does not apply to user-defined events.

Example

```
      'f'⎕WC'Form'
      f.on2←'foo'
      f.Event
  onMouseUp  #.foo
      ∇foo∇
    ∇ foo m
[1]   ⎕SE.UCMD'display m'
    ∇
```

```
┌→────────────────────────────────────────────────────┐
│     ┌→──────┐                                         │
│ #.f │MouseUp│ 24.81481552 73.33333588 1 0             │
│     └───────┘                                         │
└∊──────────────────────────────────────────────────────┘
```

This differs from the behaviour when you use event number normally:

Example

```
      'f'⎕WC'Form'
      'f' ⎕ws 'Event' 2 'foo'
      f.Event
 2  #.foo
```

```
┌→─────────────────────────────────────┐
│ ┌→┐                                   │
│ │f│ 2 52.77777863 13.22916698 1 0     │
│ └─┘                                   │
└∊──────────────────────────────────────┘
```
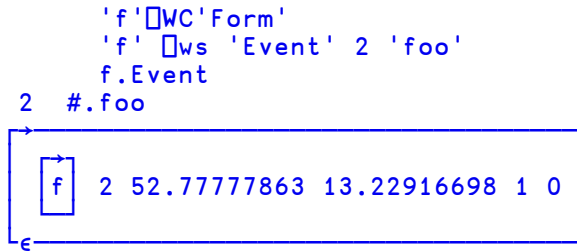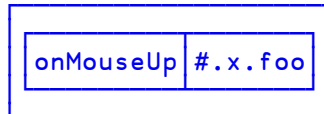
Callback Names

When you query the Event property using ⎕WG, names of callbacks associated with events are reported exactly as they were set. When you reference the Event property, the names are reported as absolute pathnames.

```
      )ns x
#.x
      )cs x
#.x
      'f'⎕WC'form'
      f.onMouseUp←'foo'
      f.Event
```

```
┌────────────────────────┐
│ onMouseUp │ #.x.foo     │
│           │             │
└────────────────────────┘
```

```
      'f'⎕wg'event'
```

```
┌────────────────────────┐
│ onMouseUp │ foo         │
│           │             │
└────────────────────────┘
```

```
      )cs
#
      #.x.f.Event
```

```
┌────────────────────────┐
│ onMouseUp │ #.x.foo     │
│           │             │
└────────────────────────┘
```

```
      '#.x.f'⎕wg'Event'
```

```
┌────────────────────────┐
│ onMouseUp │ x.foo       │
│           │             │
└────────────────────────┘
```
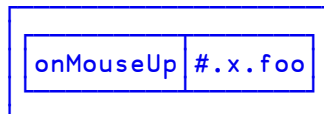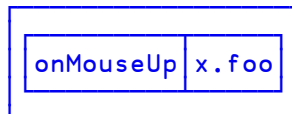
Spelling Event Names

When using regular event names, case is unimportant. For example, the system will accept `'MouseUp'`, `'MOUSEUP'` or even `'mOuSeUp'`. When using the *onEvent* syntax, case is critical. The `'on'` must be in lower-case and the case of the event name must be spelled exactly as documented. In all cases, the event name will be reported using the documented spelling.

Special Case for All events

The event number 0 and the event name `'All'`, are convenient shorthands to specify all built-in events supported by an object.

Example

```
      'f'⎕WC'Form' ('Event' 'All' 1)
      f.Event
 All   1
      ⎕←⎕DQ 'f'
 f   Create   1
```

Masked   Property

Applies To: ImageList

Description

The Masked property specifies whether or not the ImageList will contain opaque or transparent images. It may be 0, 1(the default) or 2.

Masked must be established when the ImageList is created by ⎕WC and may not subsequently be altered. An inappropriate value of Masked will cause the images to be drawn incorrectly.

If Masked is 0, the ImageList expects opaque BitMap objects.

If Masked is 1, the ImageList expects low-colour (4-bit or 8-bit) Icon objects whose transparency is defined by their Mask property.

If Masked is 2, the ImageList expects BitMap or Icon objects whose alpha channel (the degree of transparency of each pixel) is encoded in their CBits property, along with the colours.

If Masked is 3 and Native Look and Feel is enabled, the behaviour is the same as if Masked were 2. If Native Look and Feel is not enabled, it behaves as if Masked were 1. This setting provides the greatest degree of portability for applications whose users may or may not have Native Look and Feel enabled. This value is used for the ImageLists on the Dyalog Session CoolBars.

Native Look and Feel

Windows *Native Look and Feel* is an optional feature of Windows from Windows XP onwards.

If *Native Look and Feel* is enabled, user-interface controls such as Buttons take on a different appearance and certain controls (such as the ListView) provide enhanced features.

Dyalog Session

During development, both the Dyalog Session and the Dyalog APL GUI will display native style buttons, combo boxes, and other GUI components if *Native Look and Feel* is enabled. The option is provided in the *General* tab of the *Configuration* dialog.

Applications

There are two ways to enable *Native Look and Feel* in end-user applications.

If you use the *File/Export…* menu item on the Session MenuBar to create a bound executable, an OLE Server (in-process or out-of-process), an ActiveX Control or a .NET Assembly, check the option box labelled *Enable Native Look and Feel* in the *create bound file* dialog box. See User Guide.

If not, set the **XPLookandFeel** parameter to 1, when you run the program. For example:

```
dyalogrt.exe XPLookAndFeel=1 myws.dws
```

Note that to have effect, *Native Look and Feel* must also be enabled at the Windows level.

Chapter 7:

UNIX Specific Features

Summary

This section summarises the change specific to Dyalog APL Version 14.1 on UNIX-based platforms. This list currently consists of:

- AIX
- Linux (including the Raspberry Pi)
- OS X

Trace on Error

From Dyalog 14.1 onwards UNIX-based versions of Dyalog now open the tracer windows when an error occurs in the application. This brings UNIX-based versions in line with the Windows version.

This is controlled via the environment variable **TRACE_ON_ERROR**, which is set in the startup script. The previous behaviour can be configured by setting this environment variable to **0** or unsetting the environment variable.

The LoadData workspace

The loaddata workspace is included with Unicode editions of Dyalog APL Version 14.1. Note that the LoadXL and SaveXL functions will not run on UNIX-based versions of Dyalog since they require OLEControls. Note also that the SQL functions (and indeed to SQAPL workspace (where included)) require ODBC drivers to be installed before they will function correctly.

RIDE_SPAWNED environment variable

On certain platforms the Dyalog RIDE spawns a Dyalog interpreter. In such cases `)SH` on its own, and calls to `⎕SR` will apparently hang the interpreter because it is

waiting for input to the old-style session. If RIDE_SPAWNED is set to 1, in these two instances an error is generated instead.

Index