DYALOG APL

The tool of thought for expert programming

Dyalog Release Notes

Version 14.0

DYALOG

The tool of thought for expert programming

Dyalog™ for Windows

Release Notes

Version: 14.0

Dyalog Limited

email: support@dyalog.com

http://www.dyalog.com

Contents

Chapter 1:

Introduction

Key Features

Dyalog APL Version 14.0 provides the following new features, enhancements and changes:

Performance Improvements

Version 14.0 includes a considerable amount of research and development work designed to substantially improve speed of execution. See *Performance Improvements* on page 14.

Language Enhancements

New Language Features

- New terminology is introduced in Version 14.0 to describe sub-arrays of an array as *cells*. This terminology is used to describe the workings of the new Rank (⍤) and Key (⌸) operators, the function Tally (≢), and the extension to dyadic Iota (⍳). See *Cells and Sub-arrays* on page 55.
- New Rank operator ⍤. See *Rank* on page 69
- New Key operator ⌸. See *Key Operator* on page 20.
- New function Tally ≢. See *Tally* on page 57
- Function Trains. See *Function Trains* on page 19.

New I-Beam Features

- A function is provided to support WPF data binding. See *Data Binding* on page 112.
- A function is provided to perform fast index-of operations on inverted table structures. See *Inverted Table Index Of* on page 103.
- A function is provided to obtain the current type of an array without it being *squeezed* first. See *Unsqueezed Type* on page 105.
- A function is provided to compress and decompress vectors of short integers. See *Compress Vector of Short Integers* on page 106.
- A function is provided to serialise and de-serialise an array. See *Serialise/Deserialise Array* on page 108.
- A function is provided to update the function timestanp and user. See *Update Function Time Stamp* on page 110.
- A function is provided to affect fine control over memory allocated to the workspace. See *Specify Workspace Available* on page 111.
- A function is provided to flush the Session caption. See *Flush Session Caption* on page 118.
- A function is provided to close all windows. See *Close All Windows* on page 122.
- A function is provided to set workspace save options. See *Set Workspace Save Options* on page 120.

Extensions

- Dyadic ⍳ has been extended to matrices and higher-rank arrays. See *Index Of* on page 21.
- Mix (↑⍵ if ⎕ml<2; ⊃⍵ if ⎕ml≥2) has been extended. See *Mix* on page 22.
- The APL2-compatible version of Mix (⊃⍵ when ⎕ml≥2) has been extended. See *Mix With Axis* on page 23.
- Monadic operators may now be named and assigned. See *Rationalisation of Monadic Operators* on page 24.
- Dyadic operators may now be bound with their right arguments to form monadic operators. See *Right Operand Currying* on page 25.
- Options for ⎕XML may now be specified using the Variant operator ⍠. This becomes the recommended approach, although the use of the optional left argument will be retained for backwards compatibility. See *XML Convert* on page 86.
- Roll can now generate random numbers in the range 0 - 1. See *Roll* on page 100.
- Table (monadic ⍪) may now be used in selective assignment.
- ⎕RL has been extended to initialise the random number generator with a random seed. See *Random Link Extension* on page 26.

Component File System Improvements

- Data in component files may now be compressed to improve performance when file access is slow. See *Compressed Components* on page 30.
- File properties may now be set when the file is created using a function derived from `⎕FCREATE` and the Variant operator `⍠`. See *File Create and Variant* on page 31.
- `⎕FREAD` can now read multiple components. See *File Read* on page 29.
- Options for `⎕FCHK` may now be specified using the Variant operator (recommended) instead of using its optional left argument. See *File Check and Variant* on page 32.

IDE Enhancements

- The system now generates a SessionPrint event when a value is about to be displayed in the Session. This event is used by the two new user commands described below. For further details, see *SessionPrint* on page 123.
- `]boxing` user command optionally causes arrays displayed in the Session to be drawn in boxes to display their structure. See *]boxing* on page 35.
- `]rows` user command controls how rows in multi-row output to the Session are displayed. See *]rows* on page 36.
- New Editor feature to align comments. See *Aligning Comments* on page 36.
- New Editor toolbar buttons. See *New Editor Toolbar Buttons* on page 37.
- New Editor options *Allow search to wrap* and *Skip blank lines when tracing*. See *New Editor Options* on page 37.
- New Tracer extension to allow the Editor to be called when the cursor is on any whitespace.
- New Editor option *Remember previous window position* (Classic Mode only)
- New Tracer option *Limit tracer display to function in script*.
- Facility to customise the window captions for the IDE. See *Window Captions* on page 42.
- New function *Set Workspace Save Options* determines whether or not the system will reset Trace, Stop and Monitor settings whenever a workspace is saved. See *Set Workspace Save Options* on page 120.
- Value tips can now be used to investigate the syntax of external functions.

.Net Interface

- Under a licensing agreement with Syncfusion, Dyalog includes the Syncfusion library of WPF controls. These may be used by Dyalog APL users to develop applications, and may be distributed with Dyalog APL run-time applications.The Syncfusion libraries comprise a set of .NET assemblies which are supplied in the *Syncfusion/4.5* sub-directory of the main Dyalog APL installation directory (for example: *c:\Program Files\Dyalog\Dyalog APL-64 14.0 Unicode\Syncfusion\4.5*.. See *Syncfusion Libraries* on page 169.
- Version 14.0 includes support for Data Binding. See *Example 1* on page 147.
- The .Net interface includes a new DLL. The `dyalogdata4.5.dll` provides advanced support for Data Binding and Syncfusion. In particular it provides the `INotifyCollectionChanged` interface which is required to support data binding of collections and lists. This DLL requires .NET Version 4.5 and is not used unless .NET 4.5 is enabled.
- It is now possible to specify overloads and casts for calling .Net functions.

APL Applications as Windows Services

Features to allow the implementation of APL applications as Windows Services are now integrated into the Dyalog system.

System Requirements

Microsoft Windows

Dyalog APL Version 14.0 supports versions of Windows from Microsoft Windows XP up to and including Microsoft Windows 8.1 and Microsoft Windows Server 2012. Dyalog APL Version 14.0 will not run on earlier versions.

Microsoft .Net Interface

Dyalog APL Version 14.0 .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 14.0 requires .NET Version 4.0.

For full Data Binding support (including support for the `INotifyCollectionChanged` interface[1]), and Syncfusion, Version 14.0 requires .NET Version 4.5.

AIX and Linux

For AIX, Version 14.0 requires AIX 6.1 or higher, and a POWER5 chip or higher.

Version 14.0 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.

[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.

Interoperability

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 14.0 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, which is fully inter-operable.

The following sections describe other limitations in inter-operability:

Code

Code which is saved in workspaces, or embedded within `⎕OR`s stored in component files, can generally only be read by the version which saved them and later versions of the interpreter. In the case of workspaces, a 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 paragraphs, Dyalog APL provides inter-operability for arrays which only contain (nested) character and numeric data. Such arrays can be stored in component files - or transmitted 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 (see the following section), and for small-span component files created by Version 11.0 or later.

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 is still able to read and write to small span files. Setting the second item of the right argument of `⎕FCREATE` will generate 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 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 contact 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.

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, which may not 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 which may not contain Unicode data, character data is mapped using `⎕AVU`, and can therefore be read without problems by Classic editions.

A `TRANSLATION ERROR` will occur if a Unicode edition writes to a non-Unicode component (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 which 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 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 which cannot be mapped to `⎕AV` using the `⎕AVU` in the recipient workspace.

`TCPSocket` objects have an `APL` property which 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.

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 which 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 may only be used on the platform on which they were created and saved.

Announcements

Withdrawal of Support for Version 12.1 and 13.0

The supported Versions of Dyalog APL are now Version 14.0, Version 13.2 and Version 13.1. Versions 13.0, 12.1 and earlier are no longer supported.

Migration Level (⎕ML)

In Version 14.0, the default value of ⎕ML in a `CLEAR WS` has been changed from 0 to 1. This means that monadic ∈ is interpreted as Enlist and not as Type. Note that the value of ⎕ML in saved workspaces is unaffected by this change. Note too that this default value may be overridden using the **default_ml** parameter. See User Guide for details.

Deprecation of small-span component files

Since Version 10.1, Dyalog APL has supported large-span (64-bit) component files, and since Version 12.0 ⎕FCREATE has created these by default.

From Version 14.0 onwards it is not possible to create small-span component files, although you may continue to read and write components on existing small-span component files.

Dyalog recommends that you convert any existing small-span component files to large-span files using ⎕FCOPY. ⎕FCOPY will create a large-span copy even if the file being copied is small-span. You may use the user command `]Files.tolarge` to locate existing small-span files and convert them to the large-span architecture.

Dyalog APL now ignores the *-F32* argument, as well as the APL_FCREATE_PROPS_S environment variable.

Auxiliary Processors

In Version 14.0, the Dyalog-supplied Auxiliary Processor `qfsck` has been removed from the product. Anybody still using `fsck` is advised to switch to ⎕FCHK instead.

The auxiliary processors `strand` and `xutils` are still included with Version 14.0, but the intention remains to remove them as soon as possible.

The interfaces for user-written Auxiliary Processors will continue to be supported.

Random Number Generator

In Versions 13.1 and 13.2 the default random number generator in a `CLEAR WS` was 0 (Lehmer linear congruential).

In Version 14.0 the default is 1 (Mersenne Twister).

Note: the change to the default will only impact applications if they are rebuilt from a clear workspace; saved workspaces will be unaffected.

Recommendation concerning Component Files

Dyalog strongly recommends that Component files should be protected by Journalling and have Checksum enabled. See *File Properties* on page 76.

FontObj Property

The FontObj property should no longer be used directly to specify the properties of a font. It should only be used to specify the Font object to be used, which in turn specifies the characteristics of a font.

Carriage Return Change (UNIX only)

Previously, the expression `⎕UCS ⎕AV[3+⎕IO]` returned 133 (NEL) in Unicode UNIX versions. In all other versions it returned 13 (CR).

From Version 14.0, all versions of Dyalog APL return 13.

Note: the change to the default will only impact applications if they are rebuilt from a clear workspace; saved workspaces will be unaffected.

.NET Support

Support for Microsoft .Net Version 2 will cease in Dyalog APL Version 15.0.

New Method

Currently Dyalog APL adds a method named *New* to all .Net objects which do not have such a member. This allows 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
```

Dyalog intends to remove this feature from future versions of Dyalog APL. 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

Dyalog intends to remove the Uppercase Property as a property of Root in future versions of Dyalog APL. Since Version 12.0 its value has had no effect.

Dyalog-added .New method

In Version 14.0 and prior the interpreter would add a .New method to any .NET object which did not have one of its own. This was introduced when NEW was not part of the language. From Version 14.1 onwards the interperter will no longer add a .New method.

DWSIN/DWSOUT

The workspaces `DWSIN` and `DWSOUT` have been removed from version 14.0; `]in` and `]out` should be used instead. More details can be found in the *Dyalog APL Workspace Transfer Guide*.

DFNS

The dfns.dws workspace is now supplied in the `ws` sub-directory and not in the `samples` sub-directory as before.

TCPIP

The sample workspace `samples\tcpip.dws` is no longer supplied and is no longer supported.

aplserve

The sample web server which was previously provided in the `aplserve` sub-directory is no longer supplied nor supported.

Performance Improvements

Dyalog APL Version 14.0 provides a large number of performance improvements, including the following:

New Idioms

The following new idioms are recognised:

Expression	Description
`⌊0.5+NA`	Round to nearest integer

Inverted Table Index-of

An inverted table is a data structure that is commonly used in APL applications to handle relational data. The (`8⌶`) derived function provides a fast and efficient means to perform a table look-up of one inverted table in another. See *Inverted Table Index Of* on page 103.

Component File Operations

The performance of reading and writing APL components has been improved by better use of buffering and other changes.

Boolean Operations which are faster in 14.0 than in 13.2

- All structural functions such as `↑ ↓ ⍴ [;] ⌷ , ⍪ ⌽ ⊖` on Boolean arrays
- `BV/⍳⍴BV`
- `BV/⍳NS`
- `BV⍳1` and `BV⍳0`
- `BV1/BV2` and `BV1≠BV2` and `BV/[k]BV2`
- `+\[k]BV`
- `BA1=BA2`

Other operations which are faster in 14.0 than in 13.2

- Simple indexing including bracket indexing, bracket indexed assignment, and squad indexing
- `↑[⎕IO-0.5] PV`
- `NV1[NV2]+←1` when `NV2` is not in strictly ascending numerical order
- associative arithmetic scans `+\[k]NA` and `×\[k]NA` and `⌊\[k]NA` and `⌈\[k]NA`
- `⌊NA` and `⌈NA` when `⎕CT` is 0

Where:

BV	Boolean vector
BA	Boolean array
PV	nested vector
NV	numeric vector
NA	numeric array

Bug Fixes

A number of bug fixes implemented in Version 14.0 may change the way that existing code operates and are therefore documented in this section.

Matrix Inverse with scalar argument

If the argument to monadic ⌹ is scalar, the result should be scalar. In previous versions the result was a 1-element vector. This correction may change the way that existing code works, especially as the bug has been there since the earliest implementation of Dyalog.

```
      ⌹2
0.5

      ⍴⌹2 ⍝ Version 13.1 and earlier
1
      ⍴⌹2 ⍝ Version 14.0
0
```

Change Data Type

Previously, the Unicode Edition of Dyalog APL accepted a left argument of 82 (8-bits character) treating it as if it were 80. It now signals an error.

Version 13.2

```
      82 ⎕DR 65
A
```

Version 14.0

```
      82 ⎕DR 65
DOMAIN ERROR: Invalid conversion code
      82 ⎕DR 65
     ^
```

Threads and Error Trapping

If a function sets a global (*catch all*) trap, then spawns a thread with the intention that the trap should be in effect for the new thread, and subsequently introduces a new local `:Trap` or dfn error guard, it was possible that an error in the spawned thread would be caught by the second trap and not the first. This has now been resolved.

.Net Object Property Assignment

In version 14.0 the interpreter will no longer accept a one element vector as the value of a .Net object Property that is expected to be a single numeric value; only a simple scalar value will be accepted.

In 13.2 and prior, the interpreter would (incorrectly) pass a scalar rather than a single element vector as the value when assigning to a .Net Property that expected a vector, which lead to unexpected errors. In 14.0 the interpreter (correctly) insists that a vector Property is passed a vector and a scalar Property is passed a scalar value.

Example

```
      ⎕using←''
      s←⎕new System.IO.MemoryStream

      s.Capacity
0
      s.Capacity←100

      s.Capacity←,100
DOMAIN ERROR
      s.Capacity←,100
     ^
```

This last assignment will succeed in Version 13.2 and prior.

Chapter 2:

New Language Features

Function Trains

A *Train* is a sequence of 2 or 3 items in an expression which bind together to form a function. Each item in a train may be an array or a function but the right-most item must be a function.

Note that the right-most item of a function train (which is by definition a function) must be isolated from anything to its right, otherwise it will be bound to that rather than to the items to its left. This is done using parentheses.

For example, the following expression comprises a function train `-,÷` that is separated from its argument 2 by parentheses:

```
      (-,÷) 2
¯2 0.5
```

and means:

1. Calculate the reciprocal of 2
2. Calculate the negation of 2
3. Catenate these 2 results together

Whereas, without the parentheses to identify the function train, the expression means (as it did before):

1. Calculate the reciprocal of 2
2. Ravel the result of step 1
3. Negate the result of step 2

```
      -,÷ 2
¯0.5
```

For further information, see *Function Trains* on page 72.

Key Operator

In the expression `X f⌸ Y` the Key operator ⌸[1] applies the function `f` to the major cells of array `Y` *grouped by* a set of keys which are specified by the major cells of array `X`. For further details, see *Key* on page 65.

Consider a simple example where `FRUIT` and `QTY` are both vectors that represent sales transactions of fruit. Note that the major cells of a vector are its items.

```
      FRUIT←'oranges' 'pears' 'apples' 'pears' 'oranges'
      QTY←12 4 5 10 7
      FRUIT,[1.5]QTY
oranges  12
pears     4
apples    5
pears    10
oranges   7

      FRUIT {⍺,+/⍵} ⌸ QTY
oranges  19
pears    14
apples    5
```

In the expression `FRUIT{⍺,+/⍵}⌸QTY`, ⌸ first calculates the set of unique elements of `FRUIT`. It then groups together the elements of `QTY` which relate to the same unique element of `FRUIT`. In this case, `QTY[1 5]` relates to `'oranges'`, `QTY[2 4]` relates to `'pears'` and `QTY[3]` relates to `'apples'`.

It then pairs each unique `FRUIT` with each corresponding subset of `QTY` and applies the function `{⍺,+/⍵}` between each pair, with each unique `FRUIT` as its left argument and each subset of `QTY` as its right argument. Specifically, it applies function `{⍺,+/⍵}` three times with arguments as follows:

```
(⊂'oranges') {⍺,+/⍵} (12 7)
(⊂'pears')   {⍺,+/⍵} (4 10)
(⊂'apples')  {⍺,+/⍵} (,5)
```

[1]The symbol ⌸ is not available in Classic Edition, and the Key operator is instead represented by ⎕U2338

Index Of

Dyadic ⍳ has been extended to matrices and higher-rank arrays, so that sub-arrays specified by the right argument may be located in the left argument.

Specifically, this applies to major cells of the left argument `⍺`, where a major cell is a sub-array on the leading dimension of `⍺` with shape `1↓⍴⍺`. The major cells of a matrix are its rows.

Until now, the idiom `{(↓⍺)⍳↓⍵}` was the best way to look up rows of one matrix in another (often called the `matiota` idiom). This extension to dyadic ⍳ provides a more concise solution that is equally as fast, so renders the idiom redundant.

```
      (3 4⍴⍳6) {(↓⍺)⍳↓⍵} 10 4⍴⍳6
1 2 3 1 2 3 1 2 3 1
      (3 4⍴⍳6) ⍳ 10 4⍴⍳6
1 2 3 1 2 3 1 2 3 1
```

For further information, see *Index Of* on page 57

Mix

Mix (`↑⍵` if `⎕ml<2`; `⊃⍵` if `⎕ml≥2`) has been extended.

Previously, if any items in the argument (with the exception of scalars) had different rank, the function would signal a `RANK ERROR`. Now, it automatically extends the rank of non-scalar items in the argument to that of the largest rank by padding their shape with leading ones.

Version 13.2

```
      ↑(2 2)(2 3⍴3)
RANK ERROR
      ↑(2 2)(2 3⍴3)
     ^
```

Version 14.0

```
      ↑(2 2)(2 3⍴3)
2 2 0
0 0 0

3 3 3
3 3 3
```

In this case, the rank of the first item, the vector `(2 2)` is extended to that of the second (a matrix) by prefixing its shape with 1; so it becomes `(1 2⍴2 2)`. The result then obtains from `↑(1 2⍴2 2)(2 3⍴3)`.

Note that the extended mix is also exploited in the implementation of the new Rank and Key operators.

Mix With Axis

The APL2-compatible version of Mix (⊃⍵ when ⎕ml≥2) has been extended.

Previously, in the expression ⊃[K]Y, the axis specifier K was a scalar integer which set the position of the axes of the items of Y in the shape of the result.

Example

```
      ⊢Y←5 4⍴(⍳20)×⊂3 2⍴1
```

1 1 1 1 1 1	2 2 2 2 2 2	3 3 3 3 3 3	4 4 4 4 4 4
5 5 5 5 5 5	6 6 6 6 6 6	7 7 7 7 7 7	8 8 8 8 8 8
9 9 9 9 9 9	10 10 10 10 10 10	11 11 11 11 11 11	12 12 12 12 12 12
13 13 13 13 13 13	14 14 14 14 14 14	15 15 15 15 15 15	16 16 16 16 16 16
17 17 17 17 17 17	18 18 18 18 18 18	19 19 19 19 19 19	20 20 20 20 20 20

Notice where the (3 2) appears in the following results:

```
      ⍴⊃[1]Y
3 2 5 4
      ⍴⊃[2]Y
5 3 2 4
      ⍴⊃[3]Y
5 4 3 2
```

In Version 14.0, a vector K allows the axes of the items of Y to be distributed in the shape of the result, instead of being contiguous. Notice where the 3 and the 2 appear in the following results:

```
      ⍴⊃[1 3]Y
3 5 2 4
      ⍴⊃[1 4]Y
3 5 4 2
      ⍴⊃[4 2]Y
5 2 4 3
```

Rationalisation of Monadic Operators

Like primitive functions, monadic operators can be:

- named
- enclosed within parentheses
- displayed in the session

Examples

```
      □ ← each ← (¨)       ⍝ name and display
¨
      shape←⍴
      shape each (1 2) (3 4 5)
 2  3

      slash←/
      +slash ⍳10
55
      swap←⍨
      3 -swap 4
1
```

Note that *dyadic* operators are not promoted to first-class and so may not in general be named or displayed. A dyadic operator may however be bound with its right operand to form a monadic operator, i.e. a first class citizen. See *Right Operand Currying* on page 25.

Right Operand Currying

A dyadic operator may be bound or *curried* with its right operand to form a monadic operator:

Examples

```
      □ ← inv ← ⍣¯1  ⍝ produces monadic inverse operator
⍣ ¯1
      +\inv 1 2 3    ⍝ scan-inverse
1 1 1
      lim ← ⍣≡       ⍝ power-limit

      1 +⍥÷lim 1     ⍝ Phi
1.61803
```

Note:

The following restrictions continue to apply to Version 14.0 but may be relaxed in a later release:

- Dyadic operators are not promoted to first-class and so may not in general be named or displayed
- Left operand currying is not supported

Examples

```
      ⍥        ⍝ Dyadic operator not first-class
SYNTAX ERROR

      ⊂⍣       ⍝ No left-operand currying
SYNTAX ERROR
```

Random Link Extension

Certain applications require a non-repeatable series of pseudo-random numbers, and several programming techniques exist to meet this requirement (such as generating a seed from the system clock) which are not always satisfactory. In Version 14.0, you may have the system itself generate a random seed by assigning the value 0 to ⎕RL.

If ⎕RL is assigned the value 0, ⎕RL is initialised with a random seed generated by the operating system. This provides the means to initiate a non-repeatable series of pseudo-random numbers when using *RNG0* or *RNG1*.

Summary

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

I-Beam functionality removed from Version 14.0.

A	Derived Function
685	UNIX only: core to aplcore

I-Beam functionality extended in Version 14.0.

A	Derived Function
1111	Number of Threads/Virtual Processors

I-Beam functionality added to Version 14.0.

A	Derived Function
8	Inverted Table Index-of
181	Unsqueezed Type
219	Compress/Decompress Vector of Short Integers
220	Serialise/Deserialise Array
1159	Update Function Time and User Stamp
2002	Specify Workspace Available

A	Derived Function
2015	Data Binding
2022	Flush Session Caption
2023	Close all Windows
2400	Set Workspace Save Options
2401	Expose Root Properties

New Symbols in Classic Edition

The symbols for the 2 new operators introduced in Version 14.0, namely ⌸ (Key) and ⍤ (Rank), and the previously introduced symbol ⍠ (Variant) are not provided in Classic Edition.

If you create a function in a non-scripted namespace (including the root namespace) containing these symbols in a Unicode Edition, `)SAVE` the workspace and then `)LOAD` the workspace using Classic Edition, the symbols will be replaced by strings in the form `⎕Unnnn` as shown in the table below.

These strings may be entered and edited in the Classic Edition editor and will be refixed as the corresponding operators. In Unicode Edition, only the correct symbols may be used; the substitution strings will not be understood.

Symbol	Substitution String (Classic Edition only)
⍠	`⎕U2360`
⌸	`⎕U2338`
⍤	`⎕U2364`

Replacement of these symbols with their substitution strings and vice versa will only occur when they appear in code in the body of functions which appear in non-scripted namespaces. Replacement will not happen when they appear in:

- comments in functions
- character constants in functions
- any scripted object (this includes functions defined in scripted objects)
- variables

When attempting to `)LOAD` or `)COPY` a workspace that does not meet the above criteria into a Classic Edition, a `TRANSLATION ERROR` will be signalled. In 14.0 the error message includes the character which has caused the operation to fail; bear in mind that this is the first occurrence of the first character which will generate a `TRANSLATION ERROR`; there may be more instances of this character, and there may be other characters too.

Chapter 3:

Component File Improvements

File Read

⎕FREAD has been enhanced to allow you to read several components at one time, without the use of the Each (¨) operator. It differs from the equivalent operation using Each in that it is faster and is an atomic operation that does not permit an intervening file operation by a different user.

To effect this enhancement, the second element of the argument may now be a vector of component numbers, rather than just a single component number as before.

Example

```
      'temp'⎕FCREATE 1
      (⍳10)⎕FAPPEND¨1
      ⎕FSIZE 1
1 11 2240 1.8446744407E19


      ⎕FREAD 1 (⍳10)
1 2 3 4 5 6 7 8 9 10
```

Compressed Components

In Version 14.0, you may now create compressed components. Components are compressed using the *LZ4* compressor which delivers a medium level of compression, but is considered to be very fast compared to other algorithms.

Compression is intended to deliver a performance gain reading and writing large components on fast computers with slow (e.g. network) file access. Conversely, on a slow computer with fast file access compression may actually reduce read/write performance. For this reason it is optional at the component level.

This feature is implemented by a new file property `'Z'` which may be set using `⎕FPROPS`, or by a function derived from `⎕FCREATE` with the Variant operator `⍠`.

The default for the `'Z'` property is 0 which means no compression; 1 means compression. When written, components are compressed or not according to the current value of the `'Z'` property. Changing this property does not change any components already in the file.

A component file may therefore contain a mixture of normal and compressed components. Note that only the data in file components are compressed, the file access matrix and other header information is not compressed.

When read, compressed components are decompressed regardless of the value of the `'Z'` property.

Compression is not supported for files in which both Journalling and Checksum are disabled.

Attempting to read compressed components 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`

File Create and Variant

Previously, if you wanted to create a component file with non-default properties, it was necessary to execute two steps, namely:

1. Create the file using ⎕FCREATE
2. Set the File Properties using ⎕FPROPS

Without changing the syntax of ⎕FCREATE it is now possible to achieve this in one step because in Version 14.0 ⎕FCREATE supports the variant operator ⍠ and the following options:

- 'J' - journaling level; a numeric value
- 'C' - checksumming level; 0 or 1
- 'Z' - compression; 0 or 1

The principal option is a combination 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)

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 Check and Variant

⎕FCHK, which was implemented before the Variant operator was introduced, has been changed to support the use of Variant to set options. This has been done for consistency. However, the existing method for setting the options via its left argument will continue to be supported. There are 3 options:

- Task
- Repair
- Force

Rebuild causes the *file indices* to be discarded and rebuilt. *Repair* only takes place on files which have been checked and found to be damaged. It involves a rebuild, but that only takes place if it is needed. Note that Repair and Force only apply if Task is `'Scan'`.

Task

`Scan`	causes the file to be checked and optionally repaired (see `'Repair'` below)
`Rebuild`	causes the file to be unconditionally rebuilt

Repair (principle option)

`0`	do not repair
`1`	causes the file to be repaired if damage is found

Force

`0`	do not validate the file if it appears to have been properly closed
`1`	validate the file even if it appears to have been properly closed

Default values are highlighted thus in the above tables.

Examples

To check a file and attempt to fix it if damage is found:

```
(⎕FCHK ⍠ 1)'suspect.dcf'
```

To forcibly check a file and attempt to fix it if damage is found:

```
(⎕FCHK ⍠ ('Repair' 1)('Force'1))'suspect.dcf'
```

Note that if options are specified using both the Variant operator and a left argument, the left argument overrides Variant. For example:

```
MYFCHK ← ⎕FCHK ⍠ 'Task' 'Scan' ⍠ 'Repair' 0
```

names a variant of ⎕FCHK (identical, in this example, to the default) which can still be overridden by a left argument.

```
'rebuild' MYFCHK 'myfile'
```

Chapter 4: Miscellaneous

IDE Enhancements

New User Commands

Two new User Commands, `]boxing` and `]rows` provide alternate forms of displaying output in the Session.

`]boxing`

The `boxing` user command has the following arguments:

- `on`
- `off`

and the following modifiers:

- `-style` = `min` | `mid` | `max`
- `-trains` = `box` | `tree` | `parens`
- `-fns` = `on` | `off`
- `-chars` = `regular` | `chars`

If boxing is on, arrays resulting from expressions entered in the Session are displayed with a series of boxes bordering each sub-array. The `-style` modifier controls the amount of information provided in each border. The `-trains` modifer controls how function trains (see *Function Trains* on page 19) are displayed. The `-fns` modifier controls whether or not boxing is applied to output generated within a function. The `-chars` modifier selects the type of symbol used to draw the borders.

Examples

```
      ]boxing
Is OFF
      ⍳⍳3
 1 1 1  1 1 2  1 1 3
 1 2 1  1 2 2  1 2 3
      ]boxing on
Was OFF
      ⍳⍳3
```

1 1 1	1 1 2	1 1 3
1 2 1	1 2 2	1 2 3

]rows

The `]rows` user command controls how rows in multi-row output to the Session are displayed. In place of the standard behaviour controlled by the **auto-pw** parameter or `⎕PW`, `]rows` allows multi-row output to be truncated or wrapped to fit the Session window, or folded and cut.

Aligning Comments

There is a new Editor command, AC, which is used to align comments in a function in an edit window.

When you press the <AC> key, or select Align Comments in the Editor's context menu, the alignment of the comments in every line in the function will be changed so that the left-most comment (Lamp) symbol is in the same column as the cursor, except that:

- Comment symbols that lie between the first column and the first tab stop will remain in or be moved to the first column. For information on setting tab stops, see *Installation & Configuration Guide: Configuration Dialog (Edit/Trace Tab)*.
- Comment symbols will not move further left than the end of the statement.

When a comment is re-aligned, text to the right of the left-most comment symbol (including spaces and other comment symbols) will remain fixed in relation to that symbol.

Note that there is no keystroke associated with this command by default; the user must define one. See *Installation & Configuration Guide: Configuration Dialog (Keyboard Shortcuts Tab)*.

New Editor Toolbar Buttons

There are two new buttons on the Editor Toolbar whose functions are as follows:

⊞ Search hidden text	Specifies whether or not the search examines collapsed blocks
aA Match case	Specifies whether or not the search is case-sensitive. This setting is shared with the case-sensitivity setting in the Find/Replace Tool

New Editor Options

Remember previous Window position (ClassicModeSavePosition parameter)

This parameter specifies whether or not the current size and location of the first of the editor and tracer windows are remembered for next time. It applies only when Classic Mode is enabled.

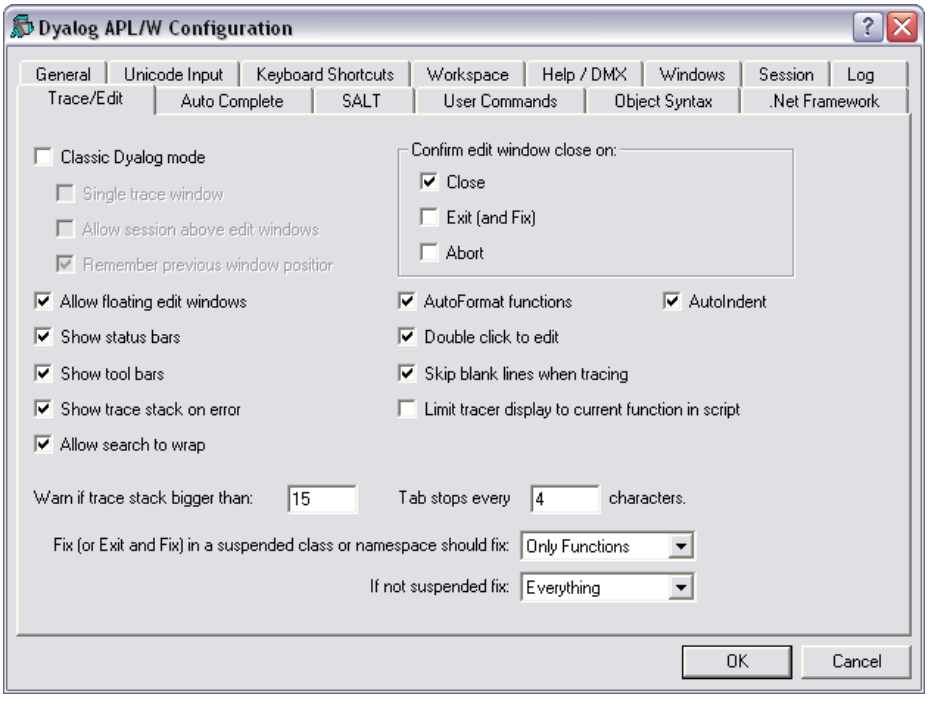
The size and location of the windows are saved in the registry in the subfolder WindowRects/EditWindow and TraceWindow.

Allow search to wrap (WrapSearch parameter)

This parameter specifies whether or not Search/Replace in the Editor stops at the bottom or top of the text (depending upon the direction of the search), or continues the search from the start or end as appropriate.

New Tracer Options

There are two new options for the Tracer namely *Skip blank lines when tracing* and *Limit tracer display to function in script*, which are enabled or disabled using the option buttons in the *Configuration Dialog: Trace/Edit Tab* as shown below.



Skipping comments and blank lines(SkipBlankLines parameter)

This parameter causes the Tracer to automatically skip lines that contain no executable statement (i.e. blank lines and comment lines), with the exception of the first line in the function, and in the case of a traditional function (not a dfn), the last line if it is a comment.

Limit tracer display to current function in script (AddClassHeaders parameter)

This parameter specifies what the Tracer displays when tracing the execution of a function in a script. If set to 1, the Tracer displays just the first line of the script and the function in question. If set to 0, the entire script is shown in the Tracer window.

Editor, Tracer and scripted objects

Editor, Tracer and objects fixed in scripted objects

A scripted object can be created either by using the Editor or by calling ⎕FIX.

The source of a scripted object can be altered only using the Editor, or by refixing using ⎕FIX. Dynamic changes to variables, fields and properties, and calling ⎕FX to generate functions do not alter the source of a scripted object. This behaviour has not changed since the introduction of scripted objects, and does not change in Version 14.0.

Prior to Version 14.0, ⎕FXing a function in a scripted object lead to apparent anomalies when tracing and editing such functions - the Tracer and Editor displayed the version of the function that was part of the source of the scripted object even though the ⎕FXed function was the version of the function that was being executed.

Now both the Tracer and the Editor display the ⎕FXed function; the statusfield will contain either "Unscripted function" or "Unscripted operator". In addition, if a function is in the source of a scripted object, the Editor and Tracer windows will display the whole script (possibly in a collapsed state) whereas a ⎕FXed function will display in a window on its own.

This behaviour is true when calling ⎕FIXed in scripted object too.

A scripted object can be created either by using the Editor or by calling ⎕FIX.

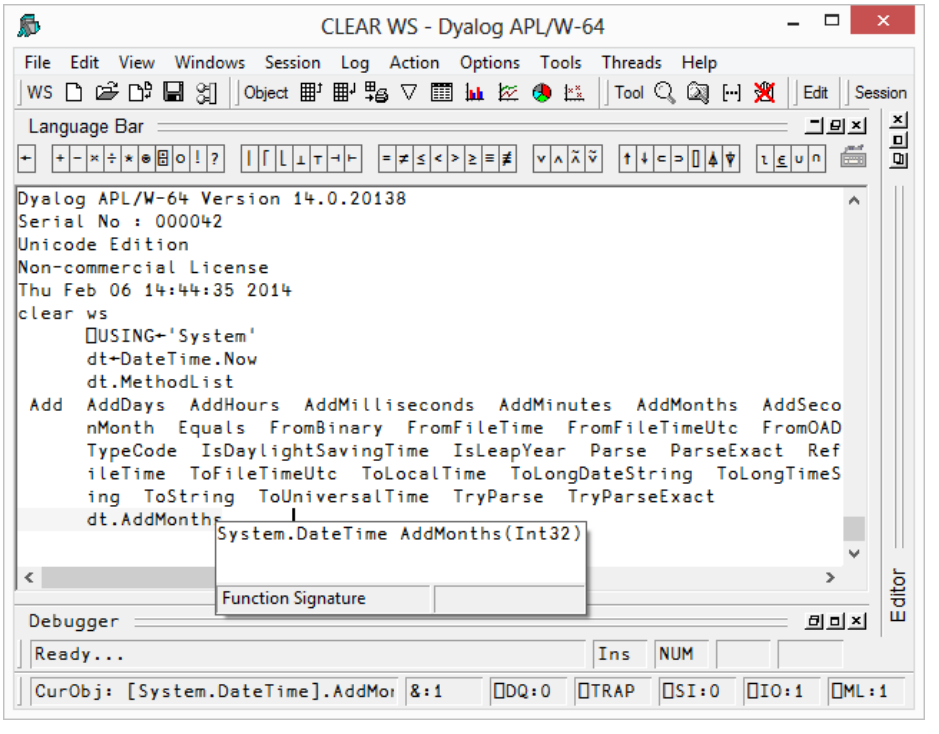
Change to fixing scripted objects in the Editor

In Versions prior to 14.0 adding or removing a Stop on a line in a function in a scripted object would have lead to the function being refixed. This is no longer true.

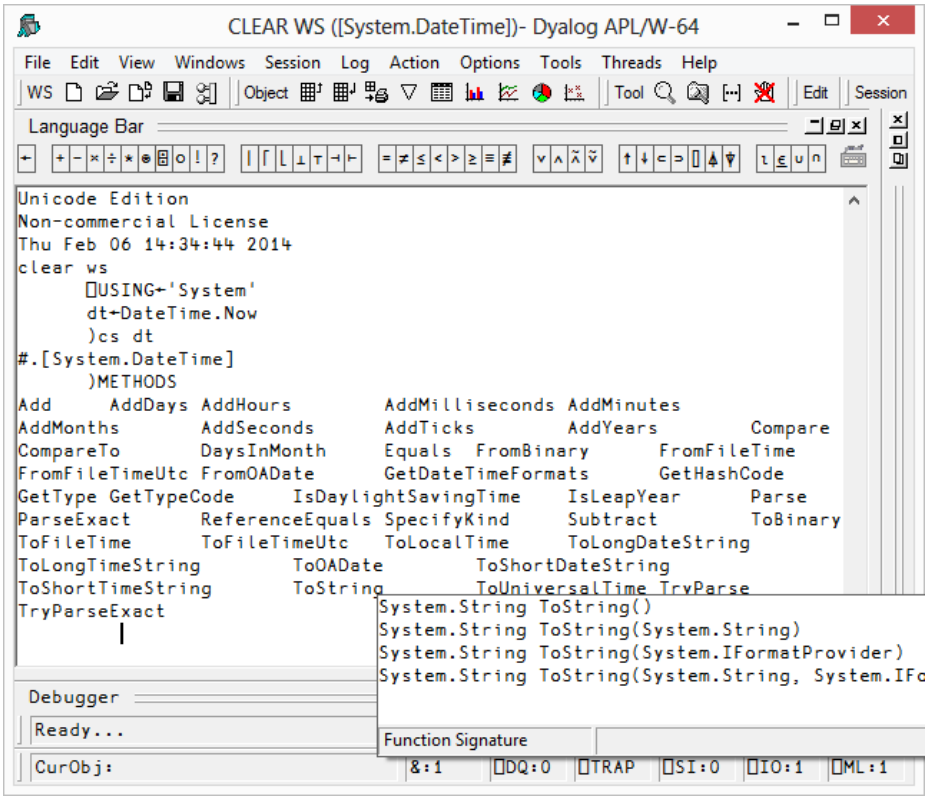
Value Tips for External Functions

Value Tips can also be used to investigate the syntax of external functions. If you hover over the name of an external function, the Value Tip displays its Function Signature.

For example, in the example below, the mouse is hovered over the external function `dt.AddMonths` and shows that it requires a single integer as its argument.



Should the external function provide more than one signature, they are all shown in the Value Tip as illustrated below. Here the function `ToString` has four different overloads.



Window Captions

The captions of the various windows that comprise the Dyalog Integrated Development Environment (IDE) are user-configurable and defined by entries in the Windows registry in the *Captions* subkey of the main Dyalog key.

Note that this only applies when the windows are floating (un-docked). When a window is docked Dyalog displays a fixed non-configurable caption.

Note also that the *Captions* subkey is not created by the interpreter; the user must create the subkey and the values.

Each entry is a string value whose name identifies the window as follows:

Window Name	**Description**
Session	The main Dyalog APL session window
Editor	The Editor window
MessageBox	The notification Message Box that is displayed in various circumstances; for example, when an object cannot be fixed by the Editor
Explorer	The Workspace Explorer tool
Rebuild Errors	The dialog box that is displayed if one or more objects cannot be re-instantiated when a workspace is loaded
Status	The Status window
Refactor	The Refactor as Method/Field/Property window that is displayed by the Editor
Event Viewer	The Event Viewer
FindReplace	The Find/Replace dialog box used by the Editor
ExitDialog	The Exit dialog box that is displayed when the user closes the Session window
WSSearch	The Find Objects tool

Each string value should contain a mixture of your own text and keywords which are enclosed in braces, e.g. {TITLE}. Keywords act like variables and are replaced at display time by corresponding values as described in the table below.

Keyword	Value
{TITLE}	The window name shown in the first column of the previous table
{WSID}	Workspace ID (⎕WSID)
{NSID}	Current Namespace
{SNSID}	Current Namespace (short version)
{PRODUCT}	The name of the Dyalog product, e.g. "Dyalog APL/W - 64"
{VER_A}	The main version number, e.g. "14"
{VER_B}	The secondary version number, e.g. "0"
{VER_C}	The tertiary version number (currently the internal revision number)
{PID}	The process ID
{CHARS}	"Classic" or "Unicode"
{BITS}	"32" or "64"
{XLOC}	The namespace currently being explored (Explorer only)

For example, if the Registry contains .*Captions**Session* whose value is:

```
My APL ({WSID}) Version {VER_A}.{VER_B}[{VER_C}]  - {PID}
```

then the caption displayed in a new Dyalog APL Session window might be:

```
My APL (CLEAR WS) Version 14.0[20105] - 4616
```

Specifying Overloads and Casts for .Net

If a .NET function is overloaded in terms of the types of arguments it accepts, Dyalog APL chooses which overload to call depending upon the data types of the arguments passed to it. For example, if a .NET function `foo()` is declared to take a single argument either of type `int` or of type `double` APL would call the first version if you called it with an integer value and the second version if you called it with a non-integer value.

In some circumstances it may be desirable to override this mechanism and explicitly specify which overload to use.

A second requirement is to be able to specify to what .NET types APL should coerce arrays before calling a .NET function. For example, if a parameter to a .NET function is declared as type `System.Object`, it might be necessary to force the APL argument to be cast to a particular *type* of `Object` before the function is called.

Both these requirements are met by calling the function via the Variant operator ⍠. There are two options, **OverloadTypes** (the Principle Option) and **CastToTypes**. Each option takes an array of refs to .NET types, the same length as the number of parameters to the function.

OverloadTypes Examples

To force APL to call the double version of function `foo()` regardless of the type of the argument `val`:

```
(foo ⍠('OverloadTypes'Double))val
```

or more simply:

```
(foo ⍠Double)val
```

Note that `Double` is a ref to the .NET type `System.Double`.

```
      ⎕USING←'System'
      Double
(System.Double)
```

Taking this a stage further, suppose that `foo()` is defined with 5 overloads as follows:

```
foo()
foo(int i)
foo(double d)
foo(double d, int i)
foo(double[] d)
```

The following statements will call the niladic, double, (double, int) and double[] overloads respectively.

```
(foo ⍠ (⊂⍬)) ⍬                                  ⍝ niladic
(foo ⍠ Double) 1                                ⍝ double
(foo ⍠(⊂Double Int32))1 1                      ⍝ double,int
(foo ⍠(Type.GetType ⊂'System.Double[]'))⊂1 1 ⍝ double[]
```

Note that in the niladic case, an enclosed empty vector is used to represent a null reference to a .NET type.

CastToTypes Example

The .NET function `Array.SetValue()` sets the value of a specified element (or elements) of an array. The first argument, the new value, is declared as `System.Object`, but the value supplied must correspond to the type of the `Array` in question. APL has no means to know what this is and will therefore pass the value *as is*, i.e. in whatever internal format it happens to be at the time. For example:

```
      ⎕USING←'System'

      ⍝ create a Boolean  array with 2 elements
      BA←Array.CreateInstance Boolean 2
      BA.GetValue 0 ⍝ get the 0th element
0

      ⍝ attempt to set the 0th element to 1 (AKA true)

      BA.SetValue 1 0
EXCEPTION: Cannot widen from source type to target type
either because the source type is a not a primitive type
or the conversion cannot be accomplished.
test[5] BA.SetValue 1 0
        ^
```

The above expression failed because APL passed the first argument 1 ,unchanged from its current internal representation, as a 1-byte integer which does not fit into a Boolean element.

To rectify the situation, APL must be told to cast the argument to a Boolean as follows:

```
      (BA.SetValue ⍠ ('CastToTypes'(Boolean Int32)))1 0
      BA.GetValue 0 ⍝ get the 0th element
1
```

Overloaded Constructors

If a class provides constructor overloads, a similar mechanism is used to specify which of the constructors is to be used when an instance of the class is created using ⎕NEW.

For example, if `MyClass` is a .NET class with an overloaded constructor, and one of its constructors is defined to take two parameters; a `double` and an `int`, the following statement would create an instance of the class by calling that specific constructor overload:

```
(⎕NEW ⎕ (⊂Double Int32)) MyClass (1 1)
```

APL Application as a Service

Introduction

Dyalog APL provides a mechanism for users to register and manage an application workspace as a Windows service. The application workspace must implement an interface to handle messages from the Windows Service Control Manager (SCM) in addition to the code required to drive the application.

Windows Services run as background tasks controlled by the SCM. When the computer is started, Windows Services are run before a user logs on to the system and do not normally interact with the desktop. A Dyalog service is run under the auspices of *Local System*.

Installing and Uninstalling a Dyalog Service

To install a Dyalog service it is necessary to run `dyalog.exe` from the command line with administrator privileges, specifying the application workspace and the following parameters, where *service_name* is a name of your choice.

- **APL_ServiceInstall=service_name**

The command must specify the full pathname to `dyalog.exe` and to the application workspace. A slightly modified version of this command line will be stored by the SCM and re-executed whenever the service is started.

Dyalog installs the service with a *Startup Type* of *Automatic*. This means that it will be started automatically whenever the computer is restarted. However, it is necessary to start it manually (using the SCM) the first time after it is installed.

The same command must be used to uninstall the service, but with:

- **APL_ServiceUninstall=service_name**

The following table summarises the parameters that can be specified by the user. Other parameters will appear on the command line in the SCM, but should not be specified by the user.

Parameter	Description
APL_ServiceInstall	Causes Dyalog to register the named service, using the current command line, but with **APL_ServiceRun** replacing **APL_ServiceInstall** in the SCM.
APL_ServiceUninstall	Causes Dyalog to uninstall the named service.

The Application Workspace

The application workspace must be designed to handle and respond (in a timely manner) to notification messages from the SCM as well as to provide the application logic. SCM notifications include instructions to start, stop, pause and resume.

SCM notification messages generate a ServiceNotification event on the Root object. To handle these messages, it is necessary to attach a callback function to this event, and to invoke the Wait method or `⎕DQ'.'` to process them. This must be executed in thread 0.

If the application is designed to be driven from events such as Timer or TCPSocket or user-defined events, it too may be implemented via callbacks in thread 0 under the control of the same Wait method or `⎕DQ'.'`. If the application uses Conga it is recommended that it runs in a separate thread.

The workspace `samples\aplservice\aplservice.dws` is included in the APL release. Its start-up function is as follows:

```
      ⎕lX←'Start'


      ∇ Start;ServiceState;ServiceControl
[1]     :If 'W'≠3⊃#.⎕WG'APLVersion'
[2]         ⎕←'This workspace only works using Dyalog APL for
                Windows version 14.0 or later'
[3]         :Return
[4]     :EndIf
[5]     :If 0∊⍴2 ⎕NQ'.' 'GetEnvironment' 'RunAsService'
[6]         Describe
[7]         :Return
[8]     :EndIf
[9]     ⍝ Define SCM constants
[10]    HashDefine
[11]    ⍝ Set up callback to handle SCM notifications
[12]    '.'⎕WS'Event' 'ServiceNotification' 'ServiceHandler'
[13]    ⍝ Global variable defines current state of the service
[14]    ServiceState←SERVICE_RUNNING
[15]    ⍝ Global variable defines last SCM notification to the
          service
[16]    ServiceControl←0
[17]    ⍝ Application code runs in a separate thread
[18]    Main&0
[19]    ⎕DQ'.'
[20]    ⎕OFF
      ∇
```

Handling ServiceNotification Events

To give the workspace (which may be busy) time to respond to SCM notifications, Dyalog responds immediately to confirm that the service has entered the appropriate pending state. For example, if the notification is SERVICE_CONTROL_STOP, Dyalog informs the SCM that the service state is SERVICE_STOP_PENDING. It is then up to the callback function to confirm that the state has reached SERVICE_STOPPED.

The following sample function is provided in `APLService.dws`.

ServiceHandler Callback Function

```
     ∇ r←ServiceHandler(obj event action state);sink
[1]   ⍝ Callback to handle notifications from the SCM
[2]
[3]   ⍝ Note that the interpreter has already responded
[4]   ⍝ automatically to the SCM with the corresponding
[5]   ⍝ "_PENDING" message prior to this callback being reached
[6]
[7]   ⍝ This callback uses the SetServiceState Method to confirm
[8]   ⍝ to the SCM that the requested state has been reached
[9]
[10]   r←0  ⍝  so returns a 0 result (the event has been handled,
[11]        ⍝ no further action required)
[12]
[13]  ⍝ It stores the desired state in global ServiceState to
[14]  ⍝ notify the application code which must take appropriate
[15]  ⍝ action. In particular, it must respond to a "STOP or
[16]  ⍝ "SHUTDOWN" by terminating the APL session
[17]
[18]   :Select ServiceControl←action
[19]   :CaseList SERVICE_CONTROL_STOP SERVICE_CONTROL_SHUTDOWN
[20]       ServiceState←SERVICE_STOPPED
[21]       state[4 5 6 7]←0
[22]
[23]   :Case SERVICE_CONTROL_PAUSE
[24]       ServiceState←SERVICE_PAUSED
[25]
[26]   :Case SERVICE_CONTROL_CONTINUE
[27]       ServiceState←SERVICE_RUNNING
[28]   :Else
[29]       :If state[2]=SERVICE_START_PENDING
[30]           ServiceState←SERVICE_RUNNING
[31]       :EndIf
[32]   :EndSelect
[33]   state[2]←ServiceState
[34]   sink←2 ⎕NQ'.' 'SetServiceState'state
     ∇
```

The Application Code

The following function illustrates how the applcation code for the service might be structured. It is merely an illustration, but however it is done, it is important that the code handles the instructions to pause, continue and stop in an appropriate manner. In this example, the function `Main` creates a log file and writes to it when the state of the service changes.

```
     ∇ Main arg;nid;log;LogFile
[1]    ⎕NUNTIE ⎕NNUMS
[2]    log←{((⍕⎕TS),' ',⍵,⎕UCS 13 10)⎕NAPPEND ⍺}
[3]    LogFile←'c:\ProgramData\TEMP\APLServiceLog.txt'
[4]    :Trap 22
[5]        nid←LogFile ⎕NCREATE 0
[6]    :Else
[7]        :Trap 22
[8]            nid←LogFile ⎕NTIE 0
[9]            0 ⎕NRESIZE nid
[10]       :Else
[11]           ⎕←'Unable to tie or create logfile'
[12]       :EndTrap
[13]   :EndTrap
[14]   nid log'Starting'
[15]   :While ServiceState≠SERVICE_STOPPED
[16]       :If ServiceControl≠0 ⋄
               nid log'ServiceControl=',⍕ServiceControl ⋄ :EndIf
[17]       :If ServiceState=SERVICE_RUNNING
[18]           nid log'Running'
[19]       :ElseIf ServiceState=SERVICE_PAUSED
[20]           ⍝  Pause application
[21]       :EndIf
[22]       ServiceControl←0 ⍝ Reset (we only want to log changes)
[23]       ⎕DL 10 ⍝ Just to prevent busy loop
[24]   :EndWhile
[25]   ⎕NUNTIE nid
[26]   ⎕OFF 0
     ∇
```

Debugging Dyalog Services

Services are run in the background under the auspices of *Local System*, and not associated with an interactive user. Neither the APL Session nor any GUI components that it creates will be visible on the desktop. This prevents the normal editing and debugging tools from being available.

However, the Dyalog APL Remote Integrated Development environment (RIDE) may be connected to any APL session, including one running as a Windows Service, and provide a debugging environment. For more information, see the *Dyalog RIDE User Guide*.

Event Logging

When a service is installed or removed, Dyalog APL records events in the Dyalog APL section of the *Applications and Services Logs* which can be viewed using the Windows system *Event Viewer*.

Causeway Tools

Overview

SharpPlot has been available for about ten years as a direct replacement for RainPro. It is a conversion of the original APL code into compiled C++ and provides all of the original functionality and more.

SharpPlot is shipped in two forms :

- `sharpplot.dll` : a Microsoft .Net assembly for Windows, which may also be run on Unix platforms using *Mono*[1].
- `sharpplot.dws` : a Dyalog APL workspace that permits SharpPlot to be used on platforms other than .Net.

Dyalog recommends upgrading from RainPro to SharpPlot as soon as possible, for a number of reasons :

- It has a more consistent API, and clearer documentation
- It supports more advanced graphics features such as alpha blending and anti-aliased raster graphics.
- It provides better performance (SharpPlot is consistently several times faster than RainPro).
- It provides multi-platform support (RainPro is supported only on windows)
- It includes free support for licensed users, including requests for enhancements

RainPro is still supported on Windows platforms, and bugs will be fixed for commercial users.

RainPro-to-SharpPlot Transition : SharpRain.dws

SharpRain is a transition tool designed to help Windows users to switch from RainPro to SharpPlot. It is a compatible emulator for RainPro/APL scripts, using SharpPlot/.Net as the backend, and provides better graphics and better performance (generally twice as fast as RainPro).

In addition, it can convert a RainPro script into a SharpPlot script, which will provide further performance improvement (generally 5 to 10 times faster than RainPro)

Further details are provided in `sharprain.dws`.

[1]Mono is an open source implementation of Microsoft's .NET Framework.
For more information, see www.mono-project.com.

Multi-Platform graphics

In Version14.0, SharpPlot is also shipped as an APL workspace (`SharpPlot.dws`), which will run on all platforms supported by Dyalog.

Raster graphics and custom fonts are not supported, and the API is slightly different from the .Net version. In particular, Properties are accessed through Get*/Set* functions rather than through a variable. These functions provide greater flexibility in terms of the arguments they will accept.

Further details are provided in `SharpPlot.dws`.

The `SharpPlot.dll` .Net assembly can still be used on non-Windows platform through *Mono*.

Chart Wizard

The chart wizard is a GUI tool to produce arbitrarily complex SharpPlot charts.

It is implemented by the `]chart` user command, which takes an APL expression as its argument. The chart wizard will then produce either the chart image, or a script that will generate the ssame image, for integration into user application.

Sessions can also be saved and loaded for complex chart elaboration.

Chapter 5:

Language Reference Changes

Cells and Sub-arrays

Certain functions and operators operate on particular cells or sub-arrays of an array, which are identified and described as follows.

K-Cells

A *rank-k* cell or *k-cell* of an array are terms used to describe a sub-array on the last `k` axes of the array. Negative `k` is interpreted as `r+k` where `r` is the rank of the array, and is used to describe a sub-array on the leading `|k` axes of an array.

If `X` is a 3-dimensional array of shape 2 3 4, the 1-cells are its 6 rows each of 4 elements; and its 2-cells are its 2 matrices each of shape 3 4. Its 3-cells is the array in its entirety. Its 0-cells are its individual elements.

Major Cells

The *major cells* of an array `X` is a term used to describe the sub-arrays on the leading dimension of the array `X` with shape `1↓⍴X`. Using the k-cell terminology, the major cells are its `¯1`-cells.

The major cells of a vector are its elements (0-cells). The major cells of a matrix are its rows (1-cells), and the major cells of a 3-dimensional array are its matrices along the first dimension (2-cells).

Examples

In the following, the major cells of `A` are 1979, 1990, 1997, 2007, and 2010; those of `B` are `'Thatcher'`, `'Major '`, `'Blair '`, `'Brown '`, and `'Cameron'`; and those of `C` are the four 2-by-3 matrices.

```
      A
1979 1990 1997 2007 2010

      B
Thatcher
Major
Blair
Brown
Cameron

   ⍴B
5 8

    ⎕←C←4 2 3⍴⍳24
 0  1  2
 3  4  5

 6  7  8
 9 10 11

12 13 14
15 16 17

18 19 20
21 22 23
```

Using the k-cell terminology, if `r` is the rank of the array, its major cells are its `r-1`-cells.

Note that if the right operand `k` of the Rank Operator `⍤` is negative, it is interpreted as `0⌈r+k`. Therefore the value `¯1` selects the major cells of the array.

Tally R←≢Y

Y may be any array. R is a simple numeric scalar.

Tally returns the number of major cells of Y. This can also be expressed as the length of the leading axis or 1 if Y is a scalar. Tally is equivalent to the function {⊃⍴⍵,1}.

Examples

```
      ≢2 3 4⍴⍳10
2
      ≢2
1
      ≢⊂⍬
0
```

Note that ≢V is useful for returning the length of vector V as a scalar. (In contrast, ⍴V is a one-element vector.)

Index Of R←X⍳Y

Y may be any array. X may be any array of rank 1 or more.

Vector Left Argument

If X is a vector, the result R is a simple integer array with the same shape as Y identifying where elements of Y are first found in X. If an element of Y cannot be found in X, then the corresponding element of R will be ⎕IO+⊃⍴X.

Elements of X and Y are considered the same if X≡Y returns 1 for those elements.

⎕IO and ⎕CT/ ⎕DCTare implicit arguments of Index Of.

Examples

```
      ⎕IO←1

      2 4 3 1 4⍳1 2 3 4 5
4 1 3 2 6

      'CAT' 'DOG' 'MOUSE'⍳'DOG' 'BIRD'
2 4
```

Higher-Rank Left Argument

If X is a higher rank array, the function locates the first occurrence of sub-arrays in Y which match major cells of X, where a major cell is a sub-array on the leading dimension of X with shape 1↓⍴X. In this case, the shape of the result R is (1-⍴⍴X)↓⍴Y.

If a sub-array of Y cannot be found in X, then the corresponding element of R will be ⎕IO+⊃⍴X.

Examples

```
      X←3 4⍴⍳12

      X
1  2  3  4
5  6  7  8
9 10 11 12

      X⍳1 2 3 4
1

      Y←2 4⍴1 2 3 4 9 10 11 12
      Y
1  2  3  4
9 10 11 12

      X⍳Y
1 3
      X⍳2 3 4 1
4

      X1←10 100 1000∘.+X
      X1
  11   12   13   14
  15   16   17   18
  19   20   21   22

 101  102  103  104
 105  106  107  108
 109  110  111  112

1001 1002 1003 1004
1005 1006 1007 1008
1009 1010 1011 1012

      X1⍳100 1000∘.+X
2 3
```

More Examples

```
      x
United Kingdom
Germany
France
Italy
United States
Canada
Japan
Canada
France
      y
United Kingdom
Germany
France
Italy
USA

Canada
Japan
China
India
Deutschland
      ⍴x
9 14
      ⍴y
2 5 14
       x⍳y
1 2  3  4 10
6 7 10 10 10

       x⍳x
1 2 3 4 5 6 7 6 3
```

Note that the expression `(y⍳x)` signals a `LENGTH ERROR` because it looks for major cells in the left argument, whose shape is `5 14(1↓⍴y)`, which is not the same as the trailing shape of `x`.

```
      y⍳x
LENGTH ERROR
      y⍳x
     ^
```

Mix (⎕ML) R←↑[K]Y or R←⊃[K]Y

The symbol chosen to represent Mix depends on the current Migration Level.

If ⎕ML<2, Mix is represented by the symbol: ↑.

If ⎕ML≥2, Mix is represented by the symbol: ⊃.

Y may be any array whose items may be uniform in rank and shape, or differ in rank and shape. If the items of Y are non-uniform, they are extended prior to the application of the function as follows:

1. If the items of Y have different ranks, each item is extended in rank to that of the greatest rank by padding with leading 1s.
2. If the items of Y have different shapes, each is padded with the corresponding prototype to a shape that represents the greatest length along each axis of all items in Y.

For the purposes of the following narrative, y represents the virtual item in Y with the greatest rank and shape, with which all other items are extended to conform.

R is an array composed from the items of Y assembled into a higher-rank array with one less level of nesting. ⍴R will be some permutation of (⍴Y),⍴y.

K is an optional axis specification whose value(s) indicate where in the result the axes of y appear. There are three cases:

1. For all values of ⎕ML, K may be a scalar or 1-element vector whose value is a fractional number indicating the two axes of Y between which new axes are to be inserted for y. The shape of R is the shape of Y with the shape ⍴y inserted between the ⌊Kth and the ⌈Kth axes of Y
2. If ⎕ML≥2, K may be a scalar or 1-element vector integer whose value specifies the position of the first axis of y in the result. This case is identical to the fractional case where K (in this case) is ⌈K (in the fractional case).
3. If ⎕ML≥2, K may be a vector, with the same length as ⍴y, each element of which specifies the position in the result of the corresponding axis of the y.

If K is absent, the axes of y appear as the last axes of the result.

Simple Vector Examples

In this example, the shape of `Y` is 3, and the shape of `y` is 2. So the shape of the result will be a permutation of 2 and 3, i.e. in this simple example, either `(2 3)` or `(3 2)`.

If `K` is omitted, the shape of the result is `(⍴Y),⍴y`.

```
      ↑(1 2)(3 4)(5 6)
1 2
3 4
5 6
```

If `K` is between 0 and 1, the shape of the result is `(⍴y),⍴Y` because `(⍴y)` is inserted between the 0^{th} and the 1^{st} axis of the result, i.e. at the beginning.

```
      ↑[.5](1 2)(3 4)(5 6)
1 3 5
2 4 6
```

If `K` is between 1 and 2, the shape of the result is `(⍴Y),⍴y` because `(⍴y)` is inserted between the 1^{st} and 2^{nd} axis of the result, i.e. at the end. This is the same as the case when `K` is omitted.

```
      ↑[1.5](1 2)(3 4)(5 6)
1 2
3 4
5 6
```

If `⎕ML≥2` an integer `K` may be used instead (Note that `⊃` is used instead of `↑`).

```
      ⎕ML←3
      ⊃(1 2)(3 4)(5 6)
1 2
3 4
5 6

      ⊃[1](1 2)(3 4)(5 6)
1 3 5
2 4 6
      ⊃[2](1 2)(3 4)(5 6)
1 2
3 4
5 6
```

Shape Extension

If the items of Y are unequal in shape, the shorter ones are extended:

```
      ⎕ML←3
      ⊃(1)(3 4)(5)
1 0
3 4
5 0
       ⊃[1](1)(3 4)(5)
1 3 5
0 4 0
```

More Simple Vector Examples:

```
      ]box on
Was OFF
       ⎕ML←3
       ⊃('andy' 19)('geoff' 37)('pauline' 21)
```

andy	19
geoff	37
pauline	21

```
       ⊃[1]('andy' 19)('geoff' 37)('pauline' 21)
```

andy	geoff	pauline
19	37	21

```
       ⊃('andy' 19)('geoff' 37)(⊂'pauline')
```

andy	19
geoff	37
pauline	

Notice that in the last statement, the shape of the third item was extended by catenating it with its prototype.

Example (Matrix of Vectors)

In the following examples, `Y` is a matrix of shape `(5 4)` and each item of `Y` (`y`) is a matrix of shape `(3 2)`. The shape of the result will be some permutaion of `(5 4 3 2)`.

```
      Y←5 4⍴(⍳20)×⊂3 2⍴1
      Y
```

1 1 1 1 1 1	2 2 2 2 2 2	3 3 3 3 3 3	4 4 4 4 4 4
5 5 5 5 5 5	6 6 6 6 6 6	7 7 7 7 7 7	8 8 8 8 8 8
9 9 9 9 9 9	10 10 10 10 10 10	11 11 11 11 11 11	12 12 12 12 12 12
13 13 13 13 13 13	14 14 14 14 14 14	15 15 15 15 15 15	16 16 16 16 16 16
17 17 17 17 17 17	18 18 18 18 18 18	19 19 19 19 19 19	20 20 20 20 20 20

By default, the axes of `⍵` appear in the last position in the shape of the result, but this position is altered by specifying the axis `K`. Notice where the `(3 2)` appears in the following results:

```
      ⍴⊃Y
5 4 3 2
      ⍴⊃[1]Y
3 2 5 4
      ⍴⊃[2]Y
5 3 2 4
      ⍴⊃[3]Y
5 4 3 2
      ⍴⊃[4]Y
INDEX ERROR
      ⍴⊃[4]Y
      ^
```

Note that `⊃[4]Y` generates an `INDEX ERROR` because 4 is greater than the length of the result.

Example (Vector K)

The axes of `Y` do not have to be contiguous in the shape of the result. By specifying a vector `K`, they can be distributed. Notice where the `3` and the `2` appear in the following results:

```
      ⍴⍉[1 3]Y
3 5 2 4
      ⍴⍉[1 4]Y
3 5 4 2
      ⍴⍉[2 4]Y
5 3 4 2
      ⍴⍉[4 2]Y
5 2 4 3
```

Rank Extension

If the items of `Y` are unequal in rank, the lower rank items are extended in rank by prefixing their shapes with 1s. Each additonal 1 may then be increased to match the maximum shape of the other items along that axis.

```
      ⎕ML←3
      Y←(1)(2 3 4 5)(2 3⍴10×⍳8)
      Y
```

1	2 3 4 5	10 20 30 40 50 60

```
      ⍴⍉Y
3 2 4
      ⍉Y
 1  0  0 0
 0  0  0 0

 2  3  4 5
 0  0  0 0

10 20 30 0
40 50 60 0
```

In the above example, the first item (1) becomes `(1 1⍴1)` to conform with the 3rd item which is rank 2. It is then extended in shape to become `(2 4↑1 1⍴1)` to conform with the 2-row 3rd item, and 4-column 2nd item.. Likewise, the 2nd item becomes a 2-row matrix, and the 3rd item gains another column.

Key R←{X}f⌸Y

`f` may be any dyadic function that returns a result.

If `X` is specified, it is an array whose major cells specify keys for corresponding major cells of `Y`. The Key operator `⌸`[1] applies the function `f` to each unique key in `X` and the major cells of `Y` having that key.

If `X` is omitted, `Y` is an array whose major cells represent keys.

In this case, the Key operator applies the function `f` to each unique key in `Y` and the elements of `⍳≢Y` having that key. `f⌸Y` is the same as `Y f⌸⍳≢Y`.

Key is similar to the GROUP BY clause in SQL.

Example

```
      cards←'2' 'Queen' 'Ace' '4' 'Jack'
      suits←'Spades' 'Hearts' 'Spades' 'Clubs' 'Hearts'

      suits,[1.5]cards
Spades  2
Hearts  Queen
Spades  Ace
Clubs   4
Hearts  Jack

      suits {⍺':'⍵}⌸ cards
 Spades  : 2 Ace
 Hearts  :  Queen  Jack
 Clubs   : 4
```

In this example, both arrays are vectors so their major cells are their elements. The function `{⍺':'⍵}` is applied between the unique elements in `suits` (`'Spades' 'Hearts' 'Clubs'`) and the elements in `cards` grouped according to their corresponding elements in `suits`, i.e. (`'2' 'Ace'`), (`'Queen' 'Jack'`) and (`'4'`).

[1] The symbol `⌸` is not available in Classic Edition, and the Key operator is instead represented by `⎕U2338`

Monadic Example

```
      {⍺ ⍵} ⌸ suits ⍴ indices of unique major cells
 Spades   1 3
 Hearts   2 5
 Clubs    4

      {⍺,≢⍵} ⌸ suits ⍴ count of unique major cells
Spades  2
Hearts  2
Clubs   1
```

Further Examples

`x` is a vector of stock codes, `y` is a corresponding matrix of values.

```
      ⍴x
10
      ⍴y
10 2
     x,y
 IBM   13 75
 AAPL  45 53
 GOOG  21  4
 GOOG  67 67
 AAPL  93 38
 MSFT  51 83
 IBM    3  5
 AAPL  52 67
 AAPL   0 38
 IBM    6 41
```

If we apply the function `{⍺ ⍵}` to `x` and `y` using the ⌸ operator, we can see how the rows (its major cells) of `y` are grouped according to the corresponding elements (its major cells) of `x`.

```
      x{⍺ ⍵}⌸y
 IBM   13 75
        3  5
        6 41
 AAPL  45 53
       93 38
       52 67
        0 38
 GOOG  21  4
       67 67
 MSFT  51 83
```

More usefully, we can apply the function `{α(+⌿ω)}`, which delivers the stock codes and the corresponding totals in `y`:

```
      x{α(+⌿ω)}⌸y
  IBM     22 121
  AAPL    190 196
  GOOG    88 71
  MSFT    51 83
```

There is no need for the function to use its left argument. So to obtain just the totals in `y` grouped by the stock codes in `x`:

```
       x{+⌿ω}⌸y
 22  121
190  196
 88   71
 51   83
```

Defined Function Example

This example appends the data for a stock into a component file named by the symbol.

```
     ∇ r←stock foo data;fid;file
[1]    file←⊃stock
[2]    :Trap 0
[3]        fid←file ⎕FTIE 0
[4]        file ⎕FERASE fid
[5]    :EndTrap
[6]    fid←file ⎕FCREATE 0
[7]    r←data ⎕FAPPEND fid
[8]    ⎕FUNTIE fid
     ∇

     x foo⌸y
1 1 1 1
```

Example

```
      {α ω} ⌸ suits ⍝ indices of unique major cells
  Spades    1 3
  Hearts    2 5
  Clubs     4

      {α,≢ω} ⌸ suits ⍝ count of unique major cells
 Spades  2
 Hearts  2
 Clubs   1
```

Another Example

Given a list of names and scores., the problem is to sum the scores for each unique name. A solution is presented first without using the Key operator, and then with the Key operator.

```
      names ⍝ 12, some repeat
 Pete  Jay  Bob  Pete  Pete  Jay  Jim  Pete  Pete  Jim
 Pete  Pete

      (∪names)∘.≡names
1 0 0 1 1 0 0 1 1 0 1 1
0 1 0 0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 1 0 0

      scores
66 75 71 100 22 10 67 77 55 42 1 78

      b←↓(∪names)∘.≡names
      ]disp b/¨⊂⍳12
┌→──────────────┬→┬→┬→──┐
│1 4 5 8 9 11 12│2 6│3│7 10│
└~──────────────→┴~─→┴→┴~──→┘

      +/¨b/¨⊂scores
399 85 71 109

      ]disp {⊂⍵}⌸ names
┌→──────────────┬→┬→┬→──┐
│1 4 5 8 9 11 12│2 6│3│7 10│
└~──────────────→┴~─→┴→┴~──→┘

      names {+/⍵}⌸ scores
399 85 71 109
```

Rank R←{X}f⍤kY

If X is omitted, f may be any monadic function that returns a result. Y may be any array.

The Rank operator ⍤[1] applies function f successively to the sub-arrays in Y specified by k. If k is positive, it selects the k-cells of Y. If k is negative, it selects the (r+k)-cells of Y where r is its rank. If k is ¯1 it selects the major cells of Y.

If X is specified, f may be any dyadic function that returns a result. Y may be any array.

In this case, the Rank operator applies function f successively between the sub-arrays in X and Y specified by k. k is a 2-element integer vector, or a scalar (which is implicitly extended), whose first element selects sub-arrays in X and whose second element selects sub-arrays of Y.

For further information, see *Programmer's Guide: Cells and Subarrays**Cells and Sub-arrays* on page 55.

Notice that it is necessary to prevent the right operand k binding to the right argument. This can be done using parentheses e.g. (f⍤1)Y. The same can be achieved using ⊢ because ⍤ binds tighter to its right operand than ⊢ does to its left argument, and ⊢ therefore resolves to Identity.

Monadic Examples

Using enclose (⊂) as the left operand elucidates the workings of the rank operator.

```
      Y
36 99 20  5
63 50 26 10
64 90 68 98

66 72 27 74
44  1 46 62
48  9 81 22
      ⍴Y
2 3 4
```

[1] The symbol ⍤ is not available in Classic Edition, and the Rank operator is instead represented by ⌺U2364

```
      ⊂⍤2 ⊢Y
```

36 99 20 5	66 72 27 74
63 50 26 10	44 1 46 62
64 90 68 98	48 9 81 22

```
      ⊂⍤1 ⊢Y
```

36 99 20 5	63 50 26 10	64 90 68 98
66 72 27 74	44 1 46 62	48 9 81 22

The function `{(⊂⍋⍵)⌷⍵}` sorts a vector.

```
   {(⊂⍋⍵)⌷⍵} 3 1 4 1 5 9 2 6 5
1 1 2 3 4 5 5 6 9
```

The rank operator can be used to apply the function to sub-arrays; in this case to sort the 1-cells (rows) of a 3-dimensional array.

```
      Y
36 99 20  5
63 50 26 10
64 90 68 98

66 72 27 74
44  1 46 62
48  9 81 22

            ({(⊂⍋⍵)⌷⍵}⍤1)Y
 5 20 36 99
10 26 50 63
64 68 90 98

27 66 72 74
 1 44 46 62
 9 22 48 81
```

Dyadic Examples

```
      10 20 30 (+⍤0 1)3 4⍴⍳12
10 11 12 13
24 25 26 27
38 39 40 41
```

Using the function `{⍺ ⍵}` as the left operand demonstrates how the dyadic case of the rank operator works.

```
      10 20 30 ({⍺ ⍵}⍤0 1)3 4⍴⍳12
```

10	0 1 2 3
20	4 5 6 7
30	8 9 10 11

Note that a right operand of `¯1` applies the function between the major cells (in this case *elements*) of the left argument, and the major cells (in this case *rows*) of the right argument.

```
      10 20 30 ({⍺ ⍵}⍤¯1)3 4⍴⍳12
```

10	0 1 2 3
20	4 5 6 7
30	8 9 10 11

Function Trains

Introduction

A *Train* is a sequence of 2 or 3 items in an expression which bind together to form a function. Each item in a train may be an array or a function but the right-most item must be a function.

Forks and Atops

The following trains are supported where `f`, `g` and `h` are functions and `A` is an array:

```
f g h
A g h
  g h
```

The 3-item trains `(f g h)` and `(A g h)` are termed *forks* while the 2-item train `(g h)` is termed an *atop*. To distinguish the two styles of *fork*, we can use the terms *fgh-fork* or *Agh-fork*.

Trains as Functions

A train is syntactically equivalent to a function and so, in common with any other function, may be:

- named using assignment
- applied to or between arguments
- consumed by operators as operands
- and so forth.

In particular, trains may be applied to a single array (monadic use) or between 2 arrays (dyadic use), providing six new constructs.

```
α(f g h)ω ↔ (α f ω) g (α h ω)   ⍝ dyadic (fgh) fork
α(A g h)ω ↔    A    g (α h ω)   ⍝ dyadic (Agh) fork
α(  g h)ω ↔         g (α h ω)   ⍝ dyadic       atop

 (f g h)ω ↔ (  f ω) g (  h ω)   ⍝ monadic (fgh) fork
 (A g h)ω ↔    A    g (  h ω)   ⍝ monadic (Agh) fork
 (  g h)ω ↔         g (  h ω)   ⍝ monadic       atop
```

Identifying a Train

For a sequence to be interpreted as a train it must be separated from the argument to which it is applied. This can be done using parentheses or by naming the derived function.

Example - fork: negation of catenated with reciprocal of

```
      (-,÷)5
¯5 0.2
```

Example - named fork

```
      negrec←-,÷
      negrec 5
 ¯5 0.2
```

Whereas, without these means to identify the sequence as a train, the expression:

```
      -,÷ 5
¯0.2
```

means the negation of the ravel of the reciprocal of 5.

Idiom Recognition

Function trains lend themselves to idiom recognition, a technique used to optimise the performance of certain expressions.

Example

An expression to find the first position in a random integer vector `X` of a number greater than 999000 is:

```
      X←?1e6⍴1e6
      (X≥999000)⍳1
1704
```

A function train is not only more concise, it is faster too.

```
      X (⍳∘1 ≥) 999000
1704
```

Trains of Trains

As a train resolves to a function, a sequences of more than 3 functions represents a train of trains. Function sequences longer than 3 are bound in threes, starting from the right:

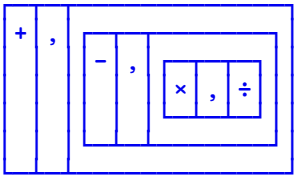
```
... fu fv fw fx fy fz → ... fu (fv fw (fx fy fz))
```

This means that, in the absence of parentheses, a sequence of an odd number of functions resolves to a 3-train (fork) and an even-numbered sequence resolves to a 2-train (atop):

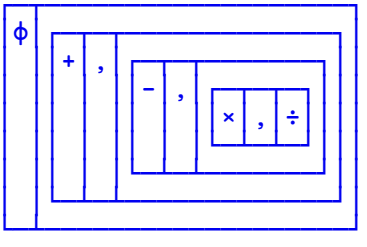
```
e f g h i j k → e f(g h(i j k))      ⍝ fork(fork(fork))
f g h i j k →   f(g h(i j k))        ⍝ atop(fork(fork))
```

Examples

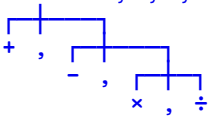
```
      6( +,-,×,÷)2      ⍝ fork:(6+2),((6-2),((6×2),(6÷2)))
8 4 12 3

      6(⍟+,-,×,÷)2      ⍝ atop: ⍟ (6+2), ...
3 12 4 8

      ]boxing on
Was OFF
      +,-,×,÷           ⍝ boxed display of fork
```




```
      ⍟+,-,×,÷          ⍝ boxed display of atop
```




```
      ]boxing -trains=tree
Was -trains=box
      +,-,×,÷           ⍝ boxed (tree) display of fork
```

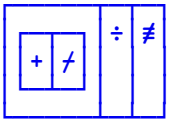


Binding Strengths

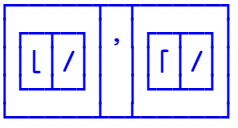
The binding strength between the items of a train is less than that of operand-operator binding. In other words, operators bind first with their function (or array) operands to form derived functions, which may then participate as items in a train.

Example:

```
      +⌿ ÷ ≢          ⍝ fork for mean value
```



```
      ⌊/,⌈/           ⍝ fork for min_max
```



This means that any of the four hybrid tokens `/ ⌿ \ ⍀` will not be interpreted as a function if there's a function to its left in the train. In order to fix one of these tokens as a replicate or expand function, it must be isolated from the function to its left:

```
      (⍳/⍳)3          ⍝ → ⍳/ atop ⍳3 → RANK ERROR
RANK ERROR

      (⍳{⍺/⍵}⍳)3      ⍝ → (⍳3){⍺/⍵}(⍳3) → (⍳3)/(⍳3)
1 2 2 3 3 3

      (⍳(/∘⊢)⍳)3      ⍝ → (⍳3)/⊢(⍳3)
1 2 2 3 3 3

      (2/⍳)3          ⍝ Agh-fork is OK
1 1 2 2 3 3
```

File Properties R←X ⎕FPROPS Y

Access Code 1 (to read) or 8192 (to change properties)

⎕FPROPS reports and sets the properties of a component file.

Y must be a simple integer scalar or 1 or 2-element vector containing the file tie number followed by an optional passnumber. If the passnumber is omitted, it is assumed to be 0.

X must be a simple character scalar or vector containing one or more valid Identifiers listed in the table below, or a 2-element nested vector which specifies an Identifier and a (new) value for that property. To set new values for more than one property, X must be is a vector of 2-element vectors, each of which contains an Identifier and a (new) value for that property.

If the left argument is a simple character array, the result R contains the current values for the properties identified by X. If the left argument is nested, the result R contains the previous values for the properties identified by X.

Identifier	Property	Description / Legal Values
S	File Size (read only)	32 = Small-span Component Files (<4GB) 64 = Large-span Component Files
E	Endian-ness (read only)	0 = Little-endian 1 = Big-endian
U	Unicode	0 = Characters will be written as type 82 arrays 1 = Characters will be written as Unicode arrays
J	Journaling	0 = Disable Journaling 1 = Enable *APL crash proof* Journaling 2 = Enable *System crash proof* Journaling; repair needed on recovery 3 = Enable full *System crash proof* Journaling
C	Checksum	0 = Disable checksum 1 = Enable checksum
Z	Compression	0 = Disable compression 1 = Enable compression

The default properties for a newly created file are as follows:

- S = 64
- U = 1 (in Unicode Edition) or 0 (in Classic Edition)
- J = 1
- C = 1
- Z = 0
- E depends upon the computer architecture.

Note that the defaults for C and J can be overridden by calling ⎕FCREATE via the Variant operator ⍠. For further information, see *File Create* on page 80.

Journaling Levels

Level 1 journaling (APL crash-proof) automatically protects a component file from damage in the event of abnormal termination of the APL process. The file state will be implicitly committed between updates and an incomplete update will automatically be rolled forward or back when the file is re-tied. In the event of an operating system crash the file may be more seriously damaged. If checksum was also enabled it may be repaired using ⎕FCHK but some components may be restored to a previous state or not restored at all.

Level 2 journaling (system crash-proof – repair needed on recovery) extends level 1 by ensuring that a component file is fully repairable using ⎕FCHK with no component loss in the event of an operating system failure. If an update was in progress when the system crashed the affected component will be rolled back to the previous state. Tying and modifying such a file without first running ⎕FCHK may however render it un-repairable.

Level 3 journaling (system crash-proof) extends level 2 by protecting a component file from damage in the event of abnormal termination of the APL process and also the operating system. Rollback of an incomplete update will be automatic and no explicit repair will be needed.

Enabling journaling on a component file will reduce performance of file updates; higher journaling levels have a greater impact.

Journaling levels 2 and 3 cannot be set unless the checksum option is also enabled.

The default level of journaling may be changed using the **APL_FCREATE_PROPS_J** parameter (see User Guide).

Checksum Option

The checksum option is enabled by default. This enables a damaged file to be repaired using ⎕FCHK. It will however reduce the performance of file updates slightly and result in larger component files. The default may be changed using the **APL_FCREATE_PROPS_C** parameter (See User Guide).

Enabling the checksum option on an existing non-empty component file will result in all previously written components without a checksum being check-summed and converted. This operation which will take place when ⎕FPROPS is changed, may not therefore be instantaneous.

Journaling and checksum settings may be changed at any time a file is exclusively tied.

Example

```
   tn←'myfile64' ⎕FCREATE 0
   'SEUJ' ⎕FPROPS tn
64 0 1 0
```

The following expression disables Unicode and switches Journaling on. The function returns the previous settings:

```
     ('U' 0)('J' 1) ⎕FPROPS tn
1 0
```

Note that to set the value of just a single property, the following two statements are equivalent:

```
     'J' 1 ⎕FPROPS tn
     (,⊂'J' 1) ⎕FPROPS tn
```

Properties may be read by a task with ⎕FREAD permission (access code 1), and set by a task with ⎕FSTAC access (8192). To set the value of the Journaling property, the file must be exclusively tied.

Recommendation

It is recommended that all component files are protected by a minimum of Level 1 Journalling and have Checksum enabled.

Unprotected files should should only be used:

- for temporary work files where speed is paramount and integrity a secondary issue
- or where compatibility with Versions of Dyalog prior to Version 12.0 is required

This recommendation is given for the following reasons:

- Unprotected files are easily damaged by abnormal termination of the interpreter
- They cannot be repaired using `⎕FCHK`
- They do not support `⎕FHIST`
- They are not well supported by the Dyalog File Server (DFS)
- They do not support compression of components
- Additional features added in future may not be supported

Compression Option

Components are compressed using the *LZ4* compressor which delivers a medium level of compression, but is considered to be very fast compared to other algorithms.

Compression is intended to deliver a performance gain reading and writing large components on fast computers with slow (e.g. network) file access. Conversely, on a slow computer with fast file access compression may actually reduce read/write performance. For this reason it is optional at the component level.

The default for the `'Z'` property is 0 which means no compression; 1 means compression. When written, components are compressed or not according to the current value of the `'Z'` property. Changing this property does not change any components already in the file.

A component file may therefore contain a mixture of normal and compressed components. Note that only the data in file components are compressed, the file access matrix and other header information is not compressed.

When read, compressed components are decompressed regardless of the value of the `'Z'` property.

An exclusive tie is not needed to change the file property.

Compression is not supported for files in which both Journalling and Checksum are disabled.

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 propeties 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.

The Principal Option is neither 'J' nor 'C' - but a combination 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 Read Components R←⎕FREAD Y

Access code 1

Y is a 2 or 3 item vector containing the file tie number, the component number(s), and an optional passnumber. If the passnumber is omitted it is assumed to be zero. All elements of Y must be integers.

The second item in Y may be scalar which specifies a single component number or a vector of component numbers. If it is a scalar, the result is the value of the array that is stored in the specified component on the tied file. If it is a vector, the result is a vector of such arrays.

Note that any invocation of ⎕FREAD is an atomic operation. Thus if compnos is a vector, the statement:

```
      ⎕FREAD tie compnos passno
```

will return the same result as:

```
      {⎕FREAD tie ⍵ passno}¨compnos
```

However, the first statement will, in the case of a share-tied file, prevent any potential intervening file access from another user (without the need for a ⎕FHOLD). It will also perform slightly faster, especially when reading from a share-tied file.

Examples

```
      ⍴SALES←⎕FREAD 1 241
3 2 12

GetFile←{⎕io←0                    ⍝ Extract contents.
    tie←⍵ ⎕fstie 0                ⍝ new tie number.
    fm to←2↑⎕fsize tie            ⍝ first and next component.
    cnos←fm+⍳to-fm                ⍝ vector of component nos.
    cvec←⎕fread tie cnos          ⍝ vector of components.
    cvec{⍺}⎕funtie tie            ⍝ ... untie and return.
}
```

File Check and Repair R←{X} ⎕FCHK Y

⎕FCHK validates and repairs component files, and validates files associated with external variables, following an abnormal termination of the APL process or operating system.

Y must be a simple character scalar or vector which specifies the name of the file to be exclusively checked or repaired. For component files, the file must be named in accordance with the operating system's conventions, and may be a relative or absolute pathname. The file must exist and must not be tied. For files associated with external variables, any filename extension must be specified even if ⎕XT would not require it. The file must exist and must not currently be associated with an external variable.

Options for ⎕FCHK are specified using the Variant operator ⍠ or by the optional left argument X. The former is recommended but the older mechanism using the left argument is still supported.

In either case, the default behaviour is as follows:

1. If the file appears to have been cleanly untied previously, return ⍬, i.e. report that the file is good.
2. Otherwise, validate the file and return the appropriate result. If the file is corrupt, no attempt is made to repair it.

The result R is a vector of the numbers of missing or damaged components. R may include non-positive numbers of "pseudo components" that indicate damage to parts of the file other than in specific components:

0	ACCESS MATRIX.
¯1	Free-block tree.
¯2	Component index tree.

Other negative numbers represent damage to the file metadata; this set may be extended in the future.

Specifying options using Variant

Using Variant, the options are as follows:

- Task
- Repair
- Force

Rebuild causes the *file indices* to be discarded and rebuilt. *Repair* only takes place on files which have been checked and found to be damaged. It involves a rebuild, but that only takes place if it is needed. Note that Repair and Force only apply if Task is `'Scan'`.

Task

`Scan`	causes the file to be checked and optionally repaired (see `'Repair'` below)
`Rebuild`	causes the file to be unconditionally rebuilt

Repair (principle option)

`0`	do not repair
`1`	causes the file to be repaired if damage is found

Force

`0`	do not validate the file if it appears to have been properly closed
`1`	validate the file even if it appears to have been properly closed

Default values are highlighted thus in the above tables.

Examples

To check a file and attempt to fix it if damage is found:

```
(⎕FCHK ⍠ 1)'suspect.dcf'
```

To forcibly check a file and attempt to fix it if damage is found:

```
(⎕FCHK ⍠ ('Repair' 1)('Force'1))'suspect.dcf'
```

Specifying options using a left argument

Using the optional left-argument, `X` must be a vector of zero or more character vectors from among `'force'`, `'repair'` and `'rebuild'`, which determine the detailed operation of the function. Note that these options are case-insensitive.

- If `X` contains `'force'`, `⎕FCHK` will validate the file even if it appears to have been cleanly untied.
- If `X` contains `'repair'`, `⎕FCHK` will repair the file, following validation, if it appears to be damaged. This option may be used in conjunction with `'force'`.
- If `X` contains `'rebuild'`, `⎕FCHK` will repair the file unconditionally.

Following a *check* of the file, a non-null result indicates that the file is damaged.

Following a *repair* of the file, the result indicates those components that could not be recovered. Un-recovered components will give a `FILE COMPONENT DAMAGED` error if read but may be replaced without error.

Repair can recover only check-summed components from the file, i.e. only those components that were written with the checksum option enabled (see *File Properties* on page 76).

Following an operating system crash, repair may result in one or more individual components being rolled back to a previous version or not recovered at all, unless Journaling levels 2 or 3 were also set when these components were written.

XML Convert R←{X} ⎕XML Y

⎕XML converts an XML string into an APL array or converts an APL array into an XML string.

Options for ⎕XML are specified using the Variant operator ⍠ or by the optional left argument X. The former is recommended but the older mechanism using the left argument is still supported.

For conversion *from* XML, Y is a character vector containing an XML string. The result R is a 5 column matrix whose columns are made up as follows:

Column	Description
1	Numeric value which indicates the level of nesting
2	Element name, other markup text, or empty character vector when empty
3	Character data or empty character vector when empty
4	Attribute name and value pairs, (`0 2⍴⊂''`) when empty
5	A numeric value which indicates what the row contains

The values in column 5 have the following meanings:

Value	Description
1	Element
2	Child element
4	Character data
8	Markup not otherwise defined
16	Comment markup
32	Processing instruction markup

Example

```
      x←'<xml><document id="001">An introduction to XML'
      x,←'</document></xml>'

     ]display v←⎕XML x
```

For conversion *to* XML, `Y` is a 3, 4 or 5 column matrix and the result `R` is a character vector. The columns of `Y` have the same meaning as those described above for the result of converting *from* XML.:

Example

```
      ⎕XML v
<xml>
  <document id="001">An introduction to XML</document>
</xml>
```

Introduction to XML and Glossary of Terms

XML is an open standard, designed to allow exchange of data between applications. The full specification[1] describes functionality, including processing directives and other directives, which can transform XML data as it is read, and which a full XML processor would be expected to handle.

The ⎕XML function is designed to handle XML to the extent required to import and export APL data. It favours speed over complexity - some markup is tolerated but largely ignored, and there are no XML query or validation features. APL applications which require processing, querying or validation will need to call external tools for this, and finally call ⎕XML on the resulting XML to perform the transformation into APL arrays.

XML grammar such as processing instructions, document type declarations etc may optionally be stored in the APL array, but will not be processed or validated. This is principally to allow regeneration of XML from XML input which contains such structures, but an APL application could process the data if it chose to do so.

The XML definition uses specific terminology to describe its component parts. The following is a summary of the terms used in this section:

Character Data

Character data consists of free-form text. The free-form text should not include the characters '>', '<' or '&', so these must be represented by their entity references ('>', '<' and '&' respectively), or numeric character references.

Entity References and Character References

Entity references are named representations of single characters which cannot normally be used in character data because they are used to delimit markup, such as > for '>'. Character references are numeric representations of any character, such as &#20; for space. Note that character references always take values in the Unicode code space, regardless of the encoding of the XML text itself.

⎕XML converts entity references and all character references which the APL character set is able to represent into their character equivalent when generating APL array data; when generating XML it converts any or all characters to entity references as needed.

There is a predefined set of entity references, and the XML specification allows others to be defined within the XML using the <!ENTITY > markup. ⎕XML does not process these additional declarations and therefore will only convert the predefined types.

[1] http://www.w3.org/TR/2008/REC-xml-20081126/

Whitespace

Whitespace sequences consist of one or more spaces, tabs or line-endings. Within character data, sequences of one or more whitespace characters are replaced with a single space when this is enabled by the whitespace option. Line endings are represented differently on different systems (0x0D 0x0A, 0x0A and 0x0D are all used) but are normalized by converting them all to 0x0A before the XML is parsed, regardless of the setting of the whitespace option.

Elements

An element consists of a balanced pair of tags or a single empty element tag. Tags are given names, and start and end tag names must match.

An example pair of tags, named TagName is

```
<TagName></TagName>
```

This pair is shown with no content between the tags; this may be abbreviated as an empty element tag as

```
<TagName/>
```

Tags may be given zero or more attributes, which are specified as name/value pairs; for example

```
<TagName AttName="AttValue">
```

Attribute values may be delimited by either double quotes as shown or single quotes (apostrophes); they may not contain certain characters (the delimiting quote, '&' or '<') and these must be represented by entity or character references.

The content of elements may be zero or more mixed occurrences of character data and nested elements. Tags and attribute names *describe* data, attribute values and the content within tags contain the data itself. Nesting of elements allows structure to be defined.

Because certain markup which describes the format of allowable data (such as element type declarations and attribute-list declarations) is not processed, no error will be reported if element contents and attributes do not conform to their restricted declarations, nor are attributes automatically added to tags if not explicitly given.

Attributes with names beginning **xml:** are reserved. Only **xml:space** is treated specially by ⎕XML. When converting both from and to XML, the value for this attribute has the following effects on space normalization for the character data within this element and child elements within it (unless subsequently overridden):

- **default** – space normalization is as determined by the **whitespace** option.
- **preserve** - space normalization is disabled – all whitespace is preserved as given.
- **any other value** – rejected.

Regardless of whether the attribute name and value have a recognised meaning, the attribute will be included in the APL array / generated XML. Note that when the names and values of attributes are examined, the comparisons are case-sensitive and take place after entity references and character references have been expanded.

Comments

Comments are fully supported markup. They are delimited by '<!--' and '-->' and all text between these delimiters is ignored. This text is included in the APL array if markup is being preserved, or discarded otherwise.

CDATA Sections

CDATA Sections are fully supported markup. They are used to delimit text within character data which has, or may have, markup text in it which is not to be processed as such. They and are delimited by '<![CDATA[' and ']]>'. CDATA sections are never recorded in the APL array as markup when XML is processed – instead, that data appears as character data. Note that this means that if you convert XML to an APL array and then convert this back to XML, CDATA sections will not be regenerated. It is, however, *possible* to generate CDATA sections in XML by presenting them as markup.

Processing Instructions

Processing Instructions are delimited by '<&' and '&>' but are otherwise treated as other markup, below.

Other markup

The remainder of XML markup, including document type declarations, XML declarations and text declarations are all delimited by '<!' and '>', and may contain nested markup. If markup is being preserved the text, including nested markup, will appear as a single row in the APL array. ⎕XML does not process the contents of such markup. This has varying effects, including but not limited to the following:

- No validation is performed.
- Constraints specified in markup such element type declarations will be ignored and therefore syntactically correct elements which fall outside their constraint will not be rejected.
- Default attributes in attribute-list declarations will not be automatically added to elements.
- Conditional sections will always be ignored.
- Only standard, predefined, entity references will be recognized; entity declarations which define others entity references will have no effect.
- External entities are not processed.

Conversion from XML

- The level number in the first column of the result R is 0 for the outermost level and subsequent levels are represented by an increase of 1 for each level. Thus, for
- <xml><document id="001">An introduction to XML </document></xml>
- The *xml* element is at level 0 and the *document id* element is at level 1. The text within the *document id* element is at level 2.
- Each tag in the XML contains an element name and zero or more attribute name and value pairs, delimited by '<' and '>' characters. The delimiters are not included in the result matrix. The element name of a tag is stored in column 2 and the attribute(s) in column 4.
- All XML markup other than tags are delimited by either '<!' and '>', or '<?' and '>' characters. By default these are not stored in the result matrix but the **markup** option may be used to specify that they are. The elements are stored in their entirety, except for the leading and trailing '<' and '>' characters, in column 2. Nested constructs are treated as a single block. Because the leading and trailing '<' and '>' characters are stripped, such entries will always have either '!' or '&' as the first character.
- Character data itself has no tag name or attributes. As an optimisation, when character data is the sole content of an element, it is included with its parent rather than as a separate row in the result. Note that when this happens, the level number stored is that of the parent; the data itself implicitly has a level number one greater.

- Attribute name and value pairs associated with the element name are stored in the fourth column, in an (*n x 2*) matrix of character values, for the *n* (including zero) pairs.
- Each row is further described in the fifth column as a convenience to simplify processing of the array (although this information could be deduced). Any given row may contain an entry for an element, character data, markup not otherwise defined, a comment or a processing instruction. Furthermore, an element will have zero or more of these as children. For all types except elements, the value in the fifth column is as shown above. For elements, the value is computed by adding together the value of the row itself (1) and those of its children. For example, the value for a row for an element which contains one or more sub-elements and character data is 7 – that is 1 (element) + 2 (child element) + 4 (character data). It should be noted that:
- Odd values always represent elements. Odd values other than 1 indicate that there are children.
- Elements which contain just character data (5) are combined into a single row as noted previously.
- Only immediate children are considered when computing the value. For example, an element which contains a sub-element which in turn contains character data does not itself contain the character data.
- The computed value is derived from what is actually preserved in the array. For example, if the source XML contains an element which contains a comment, but comments are being discarded, there will be no entry for the comment in the array and the fifth column for the element will not indicate that it has a child comment.

Conversion to XML

Conversion to XML takes an array with the format described above and generates XML text from it. There are some simplifications to the array which are accepted:

- The fifth column is not needed for XML generation and is effectively ignored. Any numeric values are accepted, or the column may be omitted altogether.
- If there are no attributes in a particular row then the `(0 2⍴⊂'')` may be abbreviated as `⍬` (zilde). If the fifth column is omitted then the fourth column may also be omitted altogether.
- Data in the third column and attribute values in the fourth column (if present) may be provided as either character vectors or numeric values. Numeric values are implicitly formatted as if `⎕PP` was set to 17.

The following validations are performed on the data in the array:

- All elements within the array are checked for type.
- Values in column 1 must be non-negative and start from level 0, and the increment from one row to the next must be ≤ +1.
- Tag names in column 2 and attribute names in column 4 (if present) must conform to the XML name definition.

Then, character references and entity references are emitted in place of characters where necessary, to ensure that valid XML is generated. However, markup, if present, is *not* validated and it is possible to generate invalid XML if care in not taken with markup constructs.

Options

There are 3 options which may be specified using the Variant operator ⍠ (recommended) or by the optional left argument `X` (retained for backwards compatibility). The names are different and are case-sensitive; they must be spelled exactly as shown below.

Option names for Variant	Option names for left argument
Whitespace	whitespace
Markup	markup
UnknownEntity	unknown-entity

The values of each option are tabulated below. In each case the value of the option for Variant is given first, followed by its equivalent for the optional left argument in brackets; e.g. **UnknownEntity (unknown-entity)**.

Note that the default value is shown first, and that the option names and values are case-sensitive.

If options are specified using the optional left argument, `X` specifies a set of option/-value pairs, each of which is a character vector. `X` may be a 2-element vector, or a vector of 2-element character vectors. In the examples below, this method is illustrated by the equivalent expression written as a comment, following the recommended approach using the Variant operator ⍠. i.e.

```
]display (⎕XML⍠'Whitespace' 'Strip')eg
⍝      'whitespace' 'strip' ⎕XML eg
```

Errors detected in the input arrays or options will all cause `DOMAIN ERROR`.

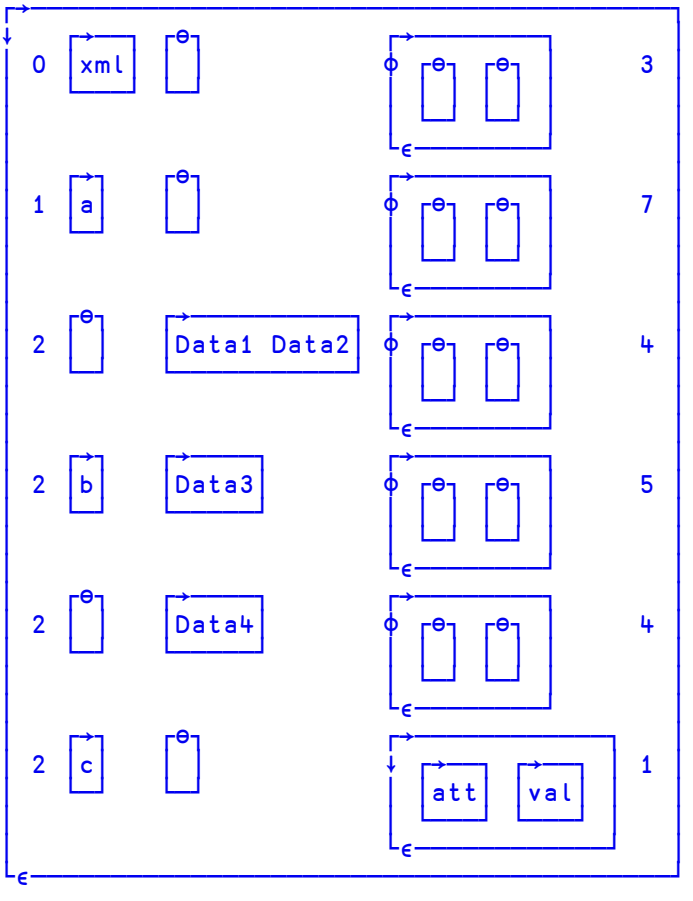
Whitespace (whitespace)

When converting from XML `Whitespace` specifies the default handling of white space surrounding and within character data. When converting to XML `Whitespace` specifies the default formatting of the XML. Note that attribute values are not comprised of character data so white space in attribute values is always preserved.

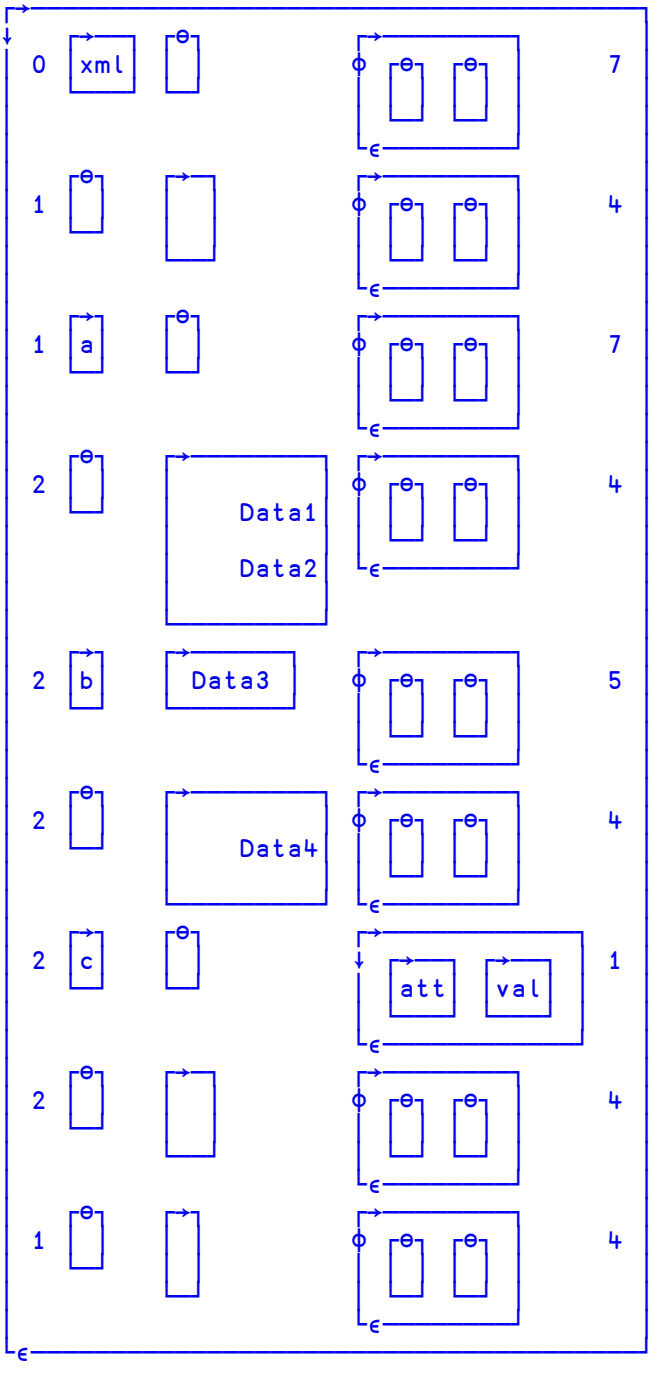
Converting from XML	
`Strip (strip)`	All leading and trailing whitespace sequences are removed; remaining whitespace sequences are replaced by a single space character
`Trim (trim)`	All leading and trailing whitespace sequences are removed; all remaining white space sequences are handled as preserve
`Preserve (preserve)`	Whitespace is preserved as given except that line endings are represented by Linefeed (`UCS 10`)
Converting to XML	
`Strip (strip)`	All leading and trailing whitespace sequences are removed; remaining whitespace sequences within the data are replaced by a single space character. XML is generated with formatting and indentation to show the data structure
`Trim (trim)`	Synonymous with `strip`
`Preserve (preserve)`	White space in the data is preserved as given, except that line endings are represented by Linefeed (`UCS 10`). XML is generated with no formatting and indentation other than that which is contained within the data

```
      ]display eg
┌→──────────────────┐
│<xml>              │
│  <a>              │
│    Data1          │
│    <!-- Comment --> │
│    Data2          │
│    <b> Data3 </b> │
│    Data4          │
│    <c att="val"/> │
│  </a>             │
│</xml>             │
└───────────────────┘
```

```
      ]display (⎕XML⍠'Whitespace' 'Strip')eg
      ⍝      'whitespace' 'strip' ⎕XML eg
```



```
]display (⎕XML⍠'Whitespace' 'Preserve')eg
A          'whitespace' 'preserve' ⎕XML eg
```



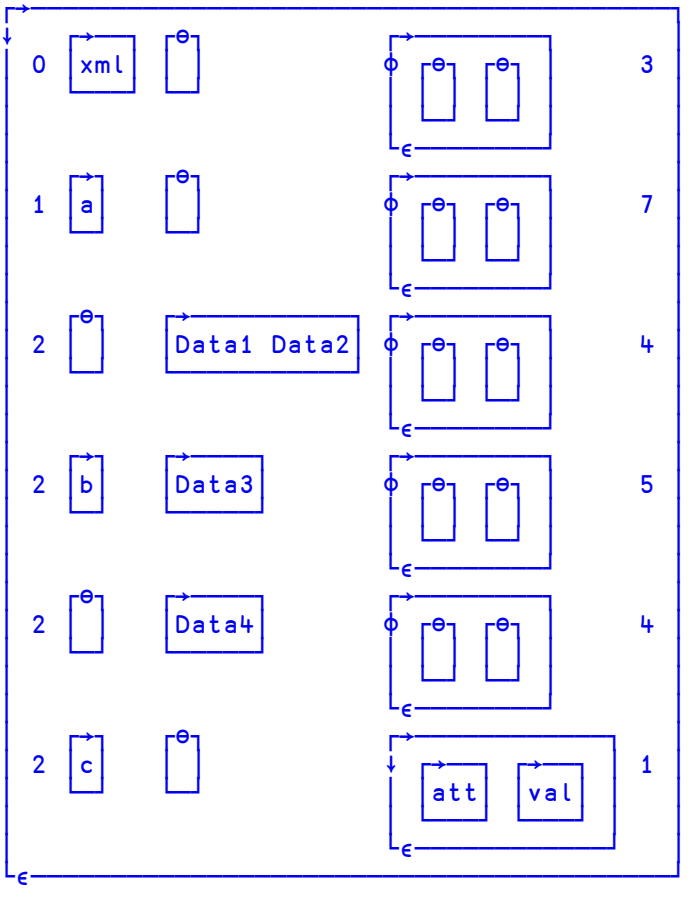
Markup (markup)

When converting from XML, `Markup` determines whether markup (other than entity tags) appears in the output array or not. When converting to XML `Markup` has no effect.

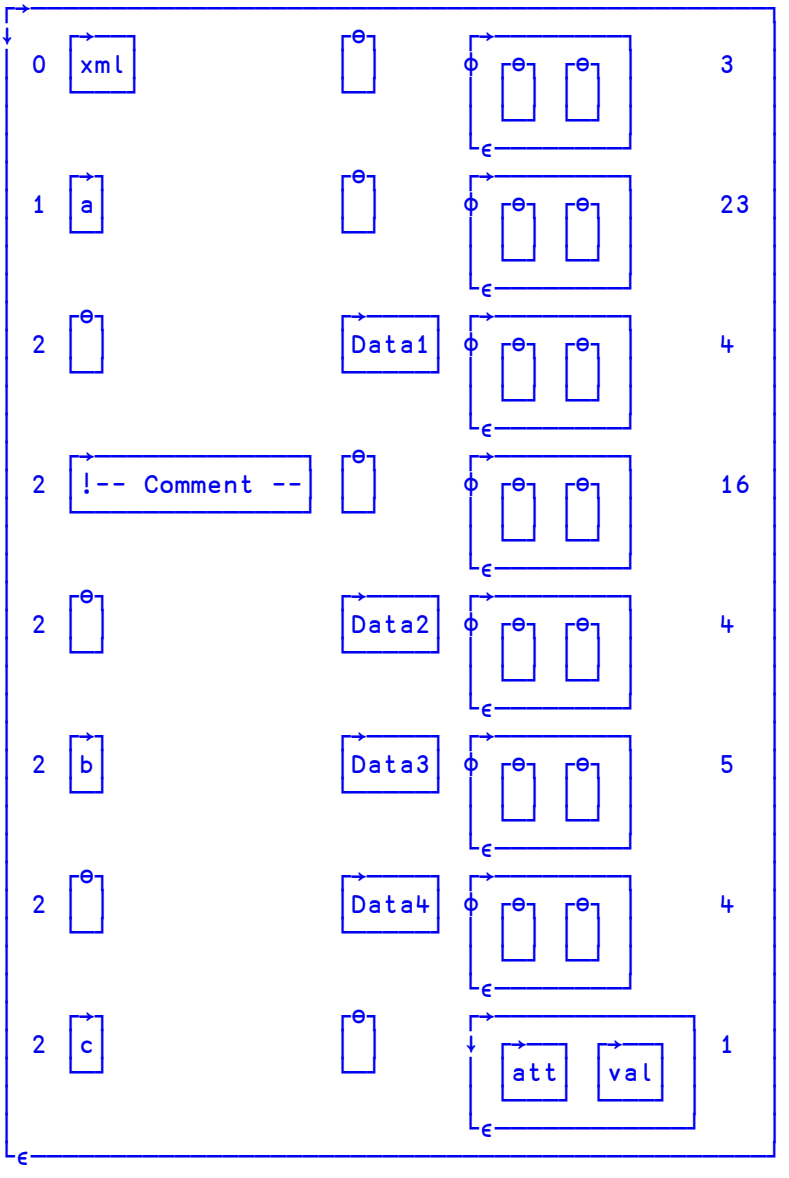
Converting from XML	
`Strip (strip)`	Markup data is not included in the output array
`Preserve (preserve)`	Markup text appears in the output array, without the leading '<' and trailing '>' in the tag (2nd) column

```
      ]display eg
┌→──────────────────────┐
<xml>
  <a>
    Data1
    <!-- Comment -->
    Data2
    <b> Data3 </b>
    Data4
    <c att="val"/>
  </a>
</xml>
└──────────────────────┘
```

```
]display (⎕XML⍠'Markup' 'Strip')eg
⍴      'markup' 'strip' ⎕XML eg
```



```
]display (⎕XML⍠'Markup' 'Preserve')eg
⍴          'markup' 'preserve' ⎕XML eg
```



UnknownEntity (unknown-entity)

When converting from XML, this option determines what happens when an unknown entity reference, or a character reference for a Unicode character which cannot be represented as an APL character, is encountered. In Classic versions of Dyalog APL that is any Unicode character which does not appear in ⎕AVU. When converting to XML, this option determines what happens to Esc characters (⎕UCS 27) in data.

Converting from XML	
Replace (replace)	The reference is replaced by a single '?' character
Preserve (preserve)	The reference is included in the output data as given, but with the leading '&' replaced by Esc (⎕UCS 27)
Converting to XML	
Replace (replace)	Esc (⎕UCS 27) is preserved
Preserve (preserve)	Esc (⎕UCS 27) is replaced by '&'

Roll R←?Y

Y may be any non-negative integer array. R has the same shape as Y at each depth.

For each positive element of Y the corresponding element of R is an integer, pseudo-randomly selected from the integers ⍳Y with each integer in this population having an equal chance of being selected.

For each zero element of Y, the corresponding element of R is a pseudo-random floating-point value in the range 0 - 1, but excluding 0 and 1, i.e. (0<R[I]<1).

⎕IO and ⎕RL are implicit arguments of Roll. A side effect of Roll is to change the value of ⎕RL.

Note that different random number generators are available; see 16807⌶ for more information.

Examples

```
      ?9 9 9
2 7 5
      ?3⍴0
0.3205466592 0.3772891947 0.5456603511
```

Chapter 6:

I-Beam Reference Changes

Summary

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

I-Beam functionality removed from Version 14.0.

A	Derived Function
685	UNIX only: core to aplcore

I-Beam functionality extended in Version 14.0.

A	Derived Function
1111	Number of Threads/Virtual Processors

I-Beam functionality added to Version 14.0.

A	Derived Function
8	Inverted Table Index-of
181	Unsqueezed Type
219	Compress/Decompress Vector of Short Integers
220	Serialise/Deserialise Array
1159	Update Function Time and User Stamp
2002	Specify Workspace Available

A	Derived Function
2015	Data Binding
2022	Flush Session Caption
2023	Close all Windows
2400	Set Workspace Save Options
2401	Expose Root Properties

Inverted Table Index Of R←X(8⌶)Y

This function computes X index-of Y (viz. X⍳Y) where X and Y are compatible inverted tables. R is the indices of Y in X.

An inverted table is a (nested) vector all of whose items have the same number of major cells. That is, 1=⍴⍴⍵ and (≢⊃⍵)=≢¨⍵. An inverted table representation of relational data is more efficient in time and space than other representations.

The following is an example of an inverted table:

```
      X←(10 3⍴⎕a) (⍳10) 'metalepsis'
      X
```

ABC DEF GHI JKL MNO PQR STU VWX YZA BCD	0 1 2 3 4 5 6 7 8 9	metalepsis

Using inverted tables, it is often necessary to perform a table look-up to find the "row" indices of one in another. Suppose there is a second table Y:

```
      Y←(⊂⊂3 1 4 1 5 9)⌷¨X
      Y
 GHI  3 1 4 1 5 9  tmamli
 ABC
 JKL
 ABC
 MNO
 YZA
```

To compute the indices of Y in X using dyadic ⍳, it is necessary to first un-invert each of the tables in order to create nested matrices that ⍳ can handle.

```
      unvert ← {⍵↑⊂⍤¯1⍤1⍵}
      unvert X
```

ABC	0	m
DEF	1	e
GHI	2	t
JKL	3	a
MNO	4	l
PQR	5	e
STU	6	p
VWX	7	s
YZA	8	i
BCD	9	s

```
      (unvert X) ⍳ (unvert Y)
3 1 4 1 5 9
```

Each un-inverted table requires considerably more workspace than its inverted form, so if the inverted tables are large, this operation is potentially expensive in terms of both time and workspace.

`8⌶` is an optimised version of the above expression.

```
      X (8⌶) Y
3 1 4 1 5 9
```

Unsqueezed Type R←181⌶Y

Y is any array.

The result R is an integer scalar containing an integer value which indicates the type of the array.

181⌶ is functionally identical to monadic ⎕DR, except that no attempt is made to squeeze the data into smaller data types. ⎕DR always attempts to squeeze the data; 181⌶ does not, but if a workspace compaction occurs during execution of 181⌶, the data may still be squeezed before the type is identified.

Example

```
      ⎕dr 1↑1 1000
11
      (181⌶) 1↑1 1000
163
```

Compress Vector of Short Integers R←X(219⌶)Y

In this section, the term *sint_vector* is used to refer to a simple integer vector whose items are all in the range ¯128 to 127 i.e. they are type 83.

In most cases this I-Beam functionality will be used in conjunction with 220⌶ (Serialise/Deserialise Array). However, it may be possible to pass the raw compressed data to and from other applications.

X specifies the operation to be performed, either compression or decompression, the compression library to be used, and any optional parameters. Y contains the data to be operated on.

Compression

Y must be a *sint_vector*.

R is a two item vector, each of which is a *sint_vector*. R[1] describes the compression, and R[2] contains the raw data which is the result of applying the compression library to the input data Y.

X is specified as follows:

X[1]	X[2]	Compression Library
1	n/a	LZ4
2	0 .. 9	zlib
3	0 .. 9	gzip

If LZ4 compression is required, then X must either be a scalar or a one element vector. Otherwise, X[2], if present, specifies the compression level; higher numbers produce better compression, but take longer.

Decompression

R is a *sint_vector*, containing the output of applying the decompression library to the input data, Y.

If X is a scalar or a one item vector, and has the value 0, then Y must be a vector of two items which is the result of previously calling 219⌶ to compress a *sint_vector*.

Otherwise, X is a scalar or one or two element vector. The first element of X must be one of the following values.

X[1]	Compression Library
¯1	LZ4
¯2	zlib
¯3	gzip

The second, optional, element of X specifies the length of the uncompressed data. Its presence results in a more efficient use of the compression library.

X may not be a two item vector whose first item has the value 0.

Examples

```
      sint←{⍵-256×⍵>127}
      utf8←'UTF-8'⍪⎕ucs
      str←'empty←⍬'
      ⍝v←sint utf8  str
101 109 112 116 121 ¯30 ¯122 ¯112 ¯30 ¯115 ¯84
      ⍝comp←1 (219⌶) v
8 ¯55 1 0 0 0 0 11  ¯80 101 109 112 116 121 ¯30 ¯122 ¯112
¯30 ¯115 ¯84

      utf8 256| 0(219⌶)comp
empty←⍬
      utf8 256| ¯1(219⌶)2⊃comp
empty←⍬
```

Serialise/Deserialise Array R←X(220⌶)Y

In this section, the term *sint_vector* is used to refer to a simple integer vector whose items are all in the range ¯128 to 127 i.e. they are type 83.

It is expected that in many cases this I-Beam functionality will be used in conjunction with 219⌶ - Compress/Decompress vector of short integers. It would also be possible to encrypt the serialised form and write to a file (either component or native), and reverse the process at a later date.

X is a scalar which can take the value 0 or 1.

When X is 1, Y can be any array. The result R is the serialised form of the array, presented as a *sint_vector*.

When X is 0, Y must be a *sint_vector*. The result R is an array whose serialised form is represented by Y.

Typically it is not possible to construct a vector which can be deserialised; it is expected that the only source of a vector which can be deserialised is the result of using 1(220⌶) to serialise an array.

The result of 1(220⌶) will differ between interpreters of differing widths and editions, but the resulting vector can be deserialised in other interpreters, with the exception that, like arrays in component files, it may not be possible to deserialise an array which was serialised in a later interpreter

The following identity holds true:

```
      A≡ 0(220⌶) 0(219⌶) 1(219⌶) 1(220⌶) A
```

Example

```
      a←'ab'
      b←1(220⌶)a
      b
¯33 ¯108 5 0 0 0 31 39 0 0 2 0 0 0 97 98 0 0
      c←0(220⌶)b
      c≡a
1
```

Number of Threads R←1111⌶Y

Specifies how many threads are to be used for parallel execution.

If Y has the value ⍬, R is the number of virtual processors in the machine.

Otherwise, Y is an integer that specifies the number of threads that are to be used henceforth for parallel execution. Prior to this call, the default number of threads is specified by the environment variable APL_MAX_THREADS. If this variable is not set, the default is the number of virtual processors that the machine is configured to have.

R is the previous value.

To reset the number of threads to be the same as the number of virtual processors run:

```
{}1111⌶ 1111⌶⍬
```

Update Function Time Stamp {R}←X(1159⌶)Y

Y is an array of function names in the same format as the right argument of ⎕AT. For further information, see .

X is an array of function attributes in the same format as the output of ⎕AT.

The shy result R is a vector of numeric items, one per each specified function containing the following values:

0	No change was made; the name is not that of a function, or the function was locked
1	The time and user stamp were updated

Note that the last item of the function time stamp must be set to 0 otherwise 1159 ⌶ will generate a DOMAIN ERROR. Additionally, the time stamp must be greater than 1970 1 1 0 0 0 0.

Example

```
      ]disp ⎕AT'Christmas'
┌→──────────────────────────┬─┬───────┐
│0 0 0│2013 3 1 11 14 58 0│0│Richard│
└~────→└~──────────────────→└─┴───────┘

      x←⎕AT 'Christmas'
x[2 4]←(2012 12 25 11 59 0 0)('Santa')
x (1159⌶) 'Christmas'

      ]disp ⎕AT'Christmas'
┌→──────────────────────────┬─┬─────┐
│0 0 0│2012 12 25 11 59 0 0│0│Santa│
└~────→└~──────────────────→└─┴────→┘3
```

Specify Workspace Available R←2002⌶Y

This function is identical to the system function ⎕WA except that it provides the means to specify the amount of memory [1] that is *committed* for the workspace rather than have it assigned by the internal algorithm. Committed memory is memory that is allocated to a specific process and thereby reduces the amount of memory available for other applications.

Like ⎕WA, 2002⌶ compacts the workspace so that it occupies the minimum number of bytes possible, adds an *extra amount*, and then de-commits all the remaining memory that it is currently using, allowing it to be allocated by the operating system for use by other applications.

The argument Y is an integer which specifies the size, in bytes, of this *extra amount*.

The purpose of the *extra amount* is to reduce the likelihood that APL will immediately have to ask the operating system to re-commit memory that it has just de-committed, something that would have a deleterious effect on performance. At the same time, if the *extra amount* were to be excessively large, APL could starve other applications of memory which itself could reduce the effective performance of the system. Whereas ⎕WA calculates the size of the *extra amount* using a simple internal algorithm, 2002⌶ uses a value specified by the programmer.

R is an integer which reports the size in bytes of the memory committed for the workspace, and is the sum of the minimum amount required by the workspace itself and the argument Y.

If the size of the committed workspace would be smaller than the minimum value (specified by 2000⌶) or larger than the maximum value (which defaults to MAXWS), a DOMAIN ERROR is signalled.

Note that this function does not change the size of the *extra amount* that will be applied subsequently by ⎕WA or by an automatic compaction.

[1] The term *memory* here means virtual memory which includes memory mapped to disk.

Data Binding R←{X}2015⌶Y

Creates an object that may be used as a data source for WPF data binding.[1]

Data binding connects a *Binding Target* to a *Binding Source*. In WPF a Binding Target is a particular property of a user interface object; for example, the `Text` property of a `TextBox` object. A Binding Source is a *Path* to a value in a data object (which may contain other values). The value of the Binding Source determines the value of the Binding Target. If two-way binding is in place, a change in a user-interface component causes the bound data value to change accordingly. In the example of the `TextBox`, the value in the Binding Source changes as the user types into the `TextBox`.

`Y` is a character vector containing one of the following:

- the name of a variable
- the name of a namespace containing one or more variables
- the name of a variable containing a vector of refs to namespaces, each of which contains one or more variables.

If the name specified by `Y` doesn't exist or represents neither a variable nor a namespace, the function reports `DOMAIN ERROR`. Currently, no further validation of the structure and contents of `Y` is performed, but nothing other than the examples described herein is supported.

If the optional left argument `X` is given and `Y` is an variable other than a ref, `X` specifies the binding type for that variable. If `Y` specifies one or more namespaces, `X` specifies the names and binding types of each of the variables which are to be bound, contained in the namespaces specified by `Y`.

The structure of `X` depends upon the structure of `Y` and is discussed later in this topic.

If `X` is omitted, all of the variables specified by `Y` are bound with default binding types.

Here the term *bind variable* refers to any variable specified by `X` and `Y` to be bound, and the term *binding type* means the .NET data type to which the value of the bind variable is converted before it is passed to the .NET interface.

[1] It is beyond the scope of this document to fully explain the concepts of WPF data binding. See Microsoft Developer Network, Data Binding Overview.

`2015⌶` creates a Binding Source object `R`. This is a .NET object which contains *Path* (s) to one or more bind variables. This object may then be assigned to a property of a WPF object or passed a s as a parameter to a WPF method that requires a Binding Source.

Bind Variables and Bind Types

A bind variable should be of rank 2 or less. Higher rank arrays are not supported.

If not specified by `X`, the binding type of a bind variable is derived from its content at the time `2015⌶` is executed. The binding type is then stored with the variable in the workspace. There is no mechanism to change a variable's binding type without erasing the variable and re-executing `2015⌶`. If you change the type or rank of a bind variable while it is bound (for example from a variable to a namespace), the behaviour of the system is unpredictable.

The default binding type is derived as follow:

If the bind variable is a simple scalar number the default binding type is `System.Object`. At the point when the value of the variable is passed to the .NET interface this will be cast to a numeric type such as `System.Int16`, `System.Int32`, `System.Int64`, or `System.Double`, depending upon the internal representation of the data. The .NET property to which it is bound will typically only accept a single Type (for example `System.Int32`), so to avoid unpredictable behaviour, it is recommended that the left argument `X` be used to specify the binding type for numeric data.

If the bind variable is a character scalar or vector, the default binding type is also `System.Object`, but at the point when the value of the variable is passed to the .NET interface it will always be passed as `System.String`,which is suitable for binding to any property that accepts a `System.String`, such as the `Text` property of a `TextBox`.

If the bind variable is a vector other than a simple character vector, such as a vector of character vectors, a simple numeric vector, or a vector of .NET objects, the bind type will be a collection. This is suitable for binding to any property that represents a collection (list) of items, for example the `ItemsSource` property of a `ListBox`.

If the bind variable is a matrix, the default binding type is `System.Object`. It is likely that in a future release a rank-2 array will be bound as a `DataTable`.

All the examples that follow assume `⎕USING←'System'`.

Binding Single Variables

In this case, `Y` specifies the name of a variable which is one of the following:

- character vector (or scalar)
- numeric scalar
- scalar .NET object (not currently supported)
- vector of character vectors
- numeric vector
- vector of .NET objects
- matrix (not currently supported)

`X` (if specified) defines the binding type for the bind variable named by `Y` and is a single .NET Type.

Note that in the following examples, the reason for expunging the name first is discussed in the section headed *Rebinding a Variable*.

Binding a Character Vector

This example illustrates how to bind a variable which contains a character vector.

```
⎕EX'txtSource'
txtSource←HELLO WORLD'
bindsource←2015⌶'txtSource'
```

In this example, the binding type of the variable **`txtSource`** will be `System.String`, suitable for binding to any property that accepts a String, such as the `Text` property of a `TextBox`.

Binding a Numeric Scalar

This example illustrates how to bind a variable which contains a numeric scalar value.

```
⎕EX'sizeSource'
sizeSource←36
bindSource←Int32(2015⌶)'sizeSource'
```

In this example, the left argument **`Int32`** specifies that the binding type for the variable sizeSource is to be `System.Int32`. This means that whenever APL passes the value of **`sizeSource`** to the control, it will first be cast to an `Int32`. This makes it suitable, for example, for binding to the `FontSize` property of a `TextBox`.

A number of controls have a `Value` property which must be expressed as a `System.Double`. The next example shows how to create a Binding Source for such a variable.

```
]EX'valSource'
valSource←42
bindSource←Double(2015⌶)'valSource'
```

Binding a Scalar .NET Object

This is not supported in the first release of Version 14.0. It is intended that it will be added in due course.

Binding a Vector of Character Vectors

WPF data binding provides the means to bind controls that display lists of items, such as the `ListBox`, `ListView`, and `TreeView` controls, to collections of data. These controls are all based upon the `ItemsControl` class. To bind an `ItemsControl` to a collection object, you use its `ItemsSource` property.

This example illustrates how to bind a variable which contains a vector of character vectors.

```
]EX'itemsSource'
itemsSource←'beer' 'wine' 'water'
bindsource←2015⌶'itemsSource'
```

In this example, the binding type of the variable `itemsSource` will be `System.Collection`, suitable for binding to the `ItemSource` property of an `ItemsControl`.

Binding a Numeric Vector

By default, a numeric vector is bound in the same way as a vector of character vectors, i.e. as a `System.Collection`, suitable for binding to the `ItemSource` property of an `ItemsControl`.

```
]EX'yearsSource'
yearsSource←2000+⍳20
bindSource←2015⌶'yearsSource'
```

In principle, a numeric vector may alternatively be bound to a WPF property that requires a 1-dimensional numeric array, by specifying the appropriate data type (e.g. `Int32`, `Double`) for the array as the left argument. For example:

```
]EX'arraySource'
arraySource←42 24
bindSource←Int32 (2015⌶)'arraySource'
```

Binding a Vector of .NET Objects

A vector of .NET objects is bound in the same way as a vector of character vectors, i.e. as a `System.Collection`, suitable for binding to the `ItemSource` property of an `ItemsControl`.

```
      ↑Easter
2015 4 12
2016 5  1
2017 4 16
2018 4  8
2019 4 28
2020 4 19
2021 5  2
2022 4 24
2023 4 16
2024 5  5
      dt←{⎕NEW DateTime ⍵}¨Easter
      bindSource←2015I'dt'
```

Binding a Matrix

Currently, the system allows a bind variable to contain a matrix (simple or nested) but the default binding type is `System.Object`. This is unlikely to be of any use. It is intended that in a future release of Dyalog APL a matrix will be bound as a `DataTable` or similar.

Rebinding a Variable

As mentioned earlier, when a variable is bound its binding type is stored with it in the workspace. If you subsequently attempt to rebind the variable there is no mechanism in place to alter the binding type. If the current binding type (whether specified by the left argument `X`, or by being the default) differs from the saved one, the function will generate a `DOMAIN ERROR`.

```
      num←42
      bs←2015I'num'

      bs←'Int32'(2015I)'num'
DOMAIN ERROR: You cannot redefine the binding types
      bs←'Int32'(2015I)'num'
     ^
```

In this example, perhaps the programmer realised after binding `num` (with a default binding type of `System.Object`) that the binding type should really be `System.Int32`, and simply was trying to correct the error. To avoid this problem, it is recommended that you expunge the name before using it.

```
⎕EX 'num'
num←42
bs←2015⌶'num'⍝ (default) binding type System.Object

⎕EX 'num'
num←42
bs←Int32(2015⌶)'num'
```

Binding A Namespace

In this case, `Y` specifies the name of a namespace that contains one or more variables. By default, each variable is bound using its default binding type as described above. Objects other than variables are ignored.

If it is required to specify the binding type of any of the variables, or if certain variables are to be excluded, the left argument is a 2-column matrix. The first column contains the names of the variables to be bound, and the second column their binding types.

Example

The following code snippet binds a namespace containing two variables named **`txtSource`** and **`sizeSource`**. In this case, the name of each variable may be specified as the Path for a WPF property that requires a `String` or an `Int32`. For example, if `bindSource` were assigned to the `DataContext` property of a `TextBox`, its `Text` property could be bound to **`txtSource`** and its `FontSize` property to **`sizeSource`**.

```
src←⎕NS''
src.txtSource←'Hello World'
src.sizeSource←36
options←2 2⍴'txtSource'String'sizeSource'Int32
bindSource←options(2015⌶)'src'
```

Binding a Vector of Namespaces

In this case, `Y` specifies the name of a variable that contains a vector of refs to namespaces. In this case, the result `R` is of type `Dyalog.Data.DataBoundCollectionHandler` which is suitable for binding to a WPF property that requires an `IEnumerable` implementation, such as the `ItemsSource` property of the `DataGrid`.

Each namespace in `Y` represents one of a collection of instances of an object, which exports a particular set of properties for binding purposes. For example, `Y` could specify a wine database where each namespace represents a different wine, and each namespace contains the same set of variables that contain the name, price (and so forth) of each wine.

Example

```
winelist←⎕NS¨(⍴Wines)⍴⊂''
winelist.Name←Wines
winelist.Price←0.01×10000+?(⍴Wines)⍴10000

bindSource←2015⌶'winelist'
```

Flush Session Caption R←2022⌶Y

Under Windows, the Session Caption displays information such as the name of the current workspace. The contents of the Caption can be modified: see *Window Captions* in the *Installation and Configuration Guide* for more details.

However, the Caption is updated only at the six-space prompt; calling ⎕LOAD for example from within a function will not result in the Caption being updated at the end of the ⎕LOAD.

This I-Beam causes the Session Caption to be updated (flushed) when called. Note that this I-Beam does not alter the contents of the Caption.

Example

```
2022⌶0
```

Close All Windows R←2023⌶Y

Under Windows the option, *Windows -> Close All Windows* allows the user to close all open Editor and Tracer Windows, but does not reset the *State Indicator*.

This I-Beam mimics this behaviour, thus allowing the user to write code which can close all windows before attempting to save the workspace; it is not possible to save a workspace if any editor or tracer windows are open.

Under UNIX, this is the only mechanism for closing all such windows.

Example

```
2023⌶0
```

Set Workspace Save Options R←2400⌶Y

This function sets a flag in the workspace that determines what happens when it is saved. The flag itself is part of the workspace and is saved with it.

If the flag is set, all Trace, Stop and Monitor settings will be cleared whenever the workspace is saved, whether by `)SAVE`, `⎕SAVE` or by *File/Save* from the Session menubar.

`Y` must be 1 (set the flag) or 0 (clear the flag).

The result `R` is the previous value of the flag.

This function may be extended in the future and a left-argument may be added.

Example

```
      (2400⌶)1
0
      )SAVE
0 Trace bits cleared.
3 Stop bits cleared.
0 Monitor bits cleared.
temp saved Sat Apr 05 17:01:30 2014
```

Expose Root Properties R←2401⌶Y

This function is used to expose or hide Root Properties, Event and Methods.

If Y is 1, Root Properties, Events and Methods are exposed.

If Y is 0, no further Root Properties, Events or Methods are exposed; however any that have already been exposed will remain so.

This functionality is available in Windows versions by selecting or unselecting the *Expose Root Properties* MenuItem in the *Options* Menu in the Session. Note that deselecting this MenuItem only affects future references to Root Properties, Events or Methods.

This function is the only mechanism available under non-Windows versions of Dyalog APL; the state of this setting is saved in the workspace, and therefore cannot be controlled by an environment variable.

Example

```
      #.GetEnvironment'MAXWS'
VALUE ERROR
      #.GetEnvironment'MAXWS'
     ^

      2401⌶1
0
      #.GetEnvironment'MAXWS'
64M

      2401⌶0
1
      #.GetEnvironment'MAXWS'
64M
      #.GetCommandLine
VALUE ERROR
      #.GetCommandLine
     ^
```

Close All Windows R←2023⌶Y

Under Windows the option, *Windows -> Close All Windows* allows the user to close all open Editor and Tracer Windows, but does not reset the *State Indicator*.

This I-Beam mimics this behaviour, thus allowing the user to write code which can close all windows before attempting to save the workspace; it is not possible to save a workspace if any editor or tracer windows are open.

Under UNIX, this is the only mechanism for closing all such windows.

Example

```
2023⌶0
```

SessionPrint Event 526

Applies To: Session

Description

If enabled, this event is reported when a value is about to be displayed in the Session. It is generated by the display of a variable or the result of a function including system variables and functions. Error messages and output from system commands do not generate this event.

The event message reported as the result of ⎕DQ, or supplied as the right argument to your callback function, is a 2-element vector as follows :

[1]	Object	ref or character vector
[2]	Event	'SessionPrint' or

The attachment of a callback function intercepts and annuls the normal display of any value.

Note that this event may be extended in future; in particular the number of elements in the event message may be increased, and the event may be generated by some system commands. You should therefore allow for such extensions in any code which refers to SessionPrint.

When the event is generated, the left argument of the callback function contains the value which was about to be displayed. The callback function may display this or any other value, using default output or by assignment to ⎕. If so, this output will be processed normally, without generating a subsequent SessionPrint event. If the callback fails to explicitly display anything, nothing will appear in the Session.

Example

```
      ⎕VR'⎕SE.TimeStamp'
     ∇ VAL TimeStamp EV
[1]    ⎕TS VAL
      ∇

      '⎕SE'⎕WS'Event' 'SessionPrint' '⎕SE.TimeStamp'

      2
 2014 9 18 16 20 38 318  2

      ⎕A
 2014 9 18 16 20 44 668  ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The result (if any) of the callback function is ignored.

You may not disable the event (by setting its action to ¯1), nor generate the event using □NQ, nor call it as a method.

Chapter 7:

Object Reference Changes

Error Messages when setting GUI Properties

The message displayed to indicate an error in the right argument to ⎕WC, ⎕WS and ⎕NEW has been improved.

The following statement may be deemed incorrect for two reasons.

```
      'F'⎕WC'Form' ('Posn' 10 10)('TheCaption' 'andys')
```

- the second property specification `('TheCaption' 'andys')` is malformed because there is no such property as `'TheCaption'`.
- In the absence of a valid property name/value pair, the system expects a value for the Size property because Size comes after Posn in the list of Properties that apply to a Form, and a vector of character vectors is an invalid value for Size.

Previously, APL assumed the second case and displayed the error message:

```
RANK ERROR: There was an error processing the "Size" prop
erty
      'F'⎕WC'Form'('Posn' 10 10)('TheCaption' 'andys')
     ^
```

This was confusing because the statement contains no mention of the Size property.

The system will now generate the following error:

```
RANK ERROR: There was an error processing the property at
position 2 of the right argument
      'F'⎕WC'Form'('Posn' 10 10)('TheCaption' 'andys')
```

CursorObj new style

The CursorObj Property has been extended; setting the Property to a value of 14 specifies the (Pointing) *Hand* cursor.

Redraw Property

The Redraw Property has been extended; setting the Property to a value of 3 causes the object and all of its children to be redrawn immediately. As in previous versions of Dyalog APL, a value of 2 causes the object to be redrawn immediately, but not its children.

DragDrop Event

The DragDrop Event used to always report the *name* of the dragged object in the event message regardless of the syntax used to specify the callback function.

Henceforth, if the callback function is specified using the `onDragDrop` syntax, the 3rd element of the event message is a *ref* to the dragged object rather than its name. If you use the Event property to establish the callback, the 3rd element of the event message is the name of the object as before.

This change makes the DragDrop event consistent with other events in the way that objects are reported in the event message.

Chapter 8:

Windows Presentation Foundation

Introduction

Windows Presentation Foundation is a graphical system that includes a programmable Graphical User Interface. It is supplied as a set of Microsoft .NET assemblies and is supported on all current Windows platforms.

The WPF GUI is in many ways more sophisticated and powerful than either Dyalog APL's own built-in GUI or the GUI provided by Windows Forms.

Like any other set of .NET classes, WFP can be integrated into Dyalog APL applications via the .NET interface. Dyalog APL users may therefore develop GUI applications that are based upon WPF as an alternative to the built-in Dyalog GUI or Windows Forms.

Quite apart from its advanced GUI capabilities, WPF supports *data binding*. This is a complex subject, but putting it very simply, data binding allows a property of a user-interface object (such as the `Text` property of a `TextBox` object) to be bound to some data. When the data changes, the bound property of the object changes and vice versa.

Dyalog APL Version 14 includes a data binding function (`2015⌶`[1]) which supports data binding to APL arrays and namespaces.

A WPF GUI can be built dynamically by creating a set of component objects (using `⎕NEW`) in a similar way to the Dyalog APL GUI and Windows Forms. However, the same user-interface can instead be specified statically using XAML, a text markup system that describes the GUI using XML. Along with data binding, this feature allows the application logic and the user-interface to be developed and maintained separately.

The examples described in this section are provided in the workspace `WPFINtro.dws`

[1] This function may remain as an i-beam or be replaced by one or more system functions in a future Version of Dyalog APL.

Temperature Converter Tutorial

This tutorial illustrates how to go about developing a simple WPF application in Dyalog APL. It is functionally identical to the GUI tutorial example that illustrates how to develop a GUI application using the built-in Dyaog APL Grahical user Interface. See *Interface Guide: GUI Tutorial*.

Like the GUI Tutorial, this is necessarily an elementary example, but illustrates the principles that are involved. The example is a simple Temperature Converter.

The user may enter a temperature value in either Fahrenheit or Centigrade and have it converted to the other scale.

No attempt has been made to update the WPF example, in terms of its user-interface, from the original version which was developed for Windows 3. This allows a direct comparison to be made between using the WPF and using the built-in Dyalog GUI.

There are two versions provided. The first uses XAML to describe the user-interface with code to drive it. The second version is written entirely in APL code. The two versions of this example may be found in `WPFINtro.dws` in the namespaces `UsingXAML` and `UsingCode` respectively.

Using XAML

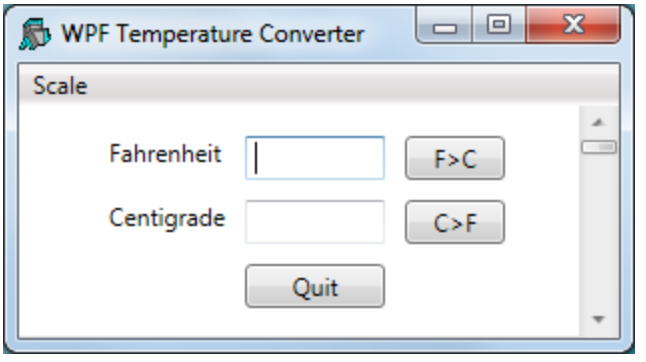
The functions and data for this example are provided in the workspace `WPFINtro.dws` in the namespace `WPF.UsingXAML`. To run the example:

```
)LOAD WPFINtro
WPF.UsingXAML.TempConverter
```

Arguably the easiest way to create a WPF GUI is to define it using XAML. The XAML defines the structure, layout and appearance of the user-interface in a very concise manner. It is still necessary to write code to display the XAML and to respond to user actions, but the amount of code involved is minimal.

The XAML for the Temperature Converter is shown below.

```
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Name="Temp"
 Title="WPF Temperature Converter"
 SizeToContent="WidthandHeight">
  <DockPanel LastChildFill="False">
    <Menu DockPanel.Dock="Top">
        <MenuItem Header="_Scale">
            <MenuItem Name="mnuFahrenheit" Header="_Fahrenheit"
             IsCheckable="True" IsChecked="True"/>
            <MenuItem Name="mnuCentigrade" Header="_Centigrade"
             IsCheckable="True"/>
        </MenuItem>
    </Menu>
    <Grid Width="230" Margin="40,10,10,10">
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="80"/>
        <ColumnDefinition Width="60"/>
      </Grid.ColumnDefinitions>
    <Label Grid.Row="0" Grid.Column="0" Content="Fahrenheit"/>
    <Label Grid.Row="1" Grid.Column="0" Content="Centigrade"/>
    <TextBox Name="txtFahrenheit" Grid.Row="0" Grid.Column="1"
     Margin="5"/>
    <TextBox Name="txtCentigrade" Grid.Row="1" Grid.Column="1"
     Margin="5"/>
    <Button Name="btnF2C" Grid.Row="0" Grid.Column="2"
     Content="F>C" Margin="5"/>
    <Button Name="btnC2F" Grid.Row="1" Grid.Column="2"
     Content="C>F" Margin="5"/>
    <Button Name="btnQuit" Grid.Row="2" Grid.Column="1"
     Content="Quit" Margin="5"/>
    </Grid>
    <ScrollBar Name="scrTemp" DockPanel.Dock="Right"  Width="20"
     Orientation="Vertical" Minimum="1" Maximum="213">
    </ScrollBar>
  </DockPanel>
</Window>
```



The window defined by this XAML is illustrated in the screen image shown above. Let us examine the XAML, component by component.

Parent and Child Controls

First, notice how the structure of the GUI is defined by enclosing the child components inside the opening and closing tags of its parent. So:

```
<Window
...
  <DockPanel>
...
  </DockPanel>
</Window>
```

specifies a Window control that *contains* a DockPanel control.

Similarly,

```
    <Menu>
        <MenuItem ... >
            <MenuItem ... />
            <MenuItem ... />
        </MenuItem>
    </Menu>
```

defines a Menu that contains a `MenuItem`, that itself contains two other `MenuItem` objects.

Named and Un-named Controls

Secondly, notice that certain objects are *named* whereas others are not. For example: `TextBox Name="mnuFahrenheit` defines a `TextBox` named*txtFahenheit*; whereas `<Dockpanel ...>` defines an unnamed `DockPanel` object.

Objects are given names so that they can be referenced from the code that displays content in the user-interface or handles the user actions. In this case, the code will read the content of the *txtFahrenheit* `TextBox` but has no need to reference the `DockPanel`.

The Main Window

```
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Name="Temp"
 Title="WPF Temperature Converter"
 SizeToContent="WidthandHeight">
...
</Window>
```

This extract of XAML defines a Window control; a top-level window that is equivalent to a Dyalog APL GUI Form.

The *xmlns* attributes define the XML namespaces (effectively the vocabulary of the xml scheme) and are mandatory in an XAML document.

The name of the `TextBox` is *Temp*, and its caption is *WFP Temperature Converter*. The `SizeToContent` property is set to "WidthandHeight" , which causes the `TextBox` to automatically size itself to fit its content in both horizontal and vertical directions.

The DockPanel

```
 <DockPanel LastChildFill="False">
..
 </DockPanel>
```

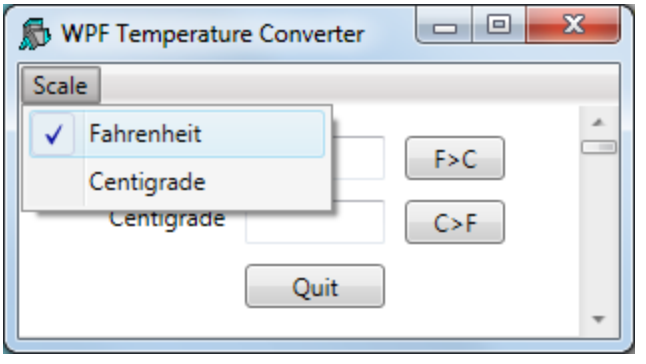
WPF provides a number of *layout controls*. These are containers whose only purpose is to arrange child controls in a particular way, and to dictate how they are re-arranged when the parent window is resized. The `DockPanel` is one of the simplest of the WPF layout controls.

In this case, the `DockPanel` is controlling 3 child windows a Menu, a `Grid` and a `ScrollBar`.

The attachment of a particular child control is specified by setting its `DockPanel.Dock` property. By default, the last control added to a `DockPanel` is stretched to fill the remaining space when the window is expanded. In this case, the requirement is for a fixed-width scrollbar attached to the right edge, so the default is overriden by setting the `LastChildFill` property to "False".

The Menu

```
<Menu DockPanel.Dock="Top">
    <MenuItem Header="_Scale">
        <MenuItem Name="mnuFahrenheit" Header="_Fahrenheit"
         IsCheckable="True" IsChecked="True"/>
        <MenuItem Name="mnuCentigrade" Header="_Centigrade"
         IsCheckable="True"/>
    </MenuItem>
</Menu>
```



The above extract from the XAML defines a `Menu`. Setting `Dock` to "Top" causes the `Menu` as a whole to be docked, so that it appears like a menubar, along the top of the `DockPanel`. The `Menu` contains a single `MenuItem` labelled *Scale* which itself contains two sub-items labelled *Fahrenheit* and *Centigrade* respectively. The `IsCheckable` property specifies whether or not the user can check the `MenuItem`, and the `IsChecked` property sets and reports its checked state. The underscore characters (e.g. as in "_Scale") identify the following character as a keyboard shortcut.

The Grid

```
<Grid Width="230" Margin="40,10,10,10">
...
</Grid>
```

The `Grid` object is another WPF layout control that organises other controls in rows and columns. Here, the XAML defines a `Grid` with a width of 230; a left margin if 40, and a top, right and bottom margin of 10. As there is no explicit unit specified, the system uses the default device-independent unit (*px*) of $1/96^{th}$ inch.

The rows and columns of a `Grid` are defined by collections of `RowDefinition` and `ColumnDefinition` objects.

Here the XAML specifies that the `Grid` contains 3 rows, each of which has a `Height` set to "Auto" which means that its height depends upon the height of its content.

```
<Grid.RowDefinitions>
  <RowDefinition Height="Auto"/>
  <RowDefinition Height="Auto"/>
  <RowDefinition Height="Auto"/>
</Grid.RowDefinitions>
```

Similarly, there are 3 columns. The first column (which will contain labels) takes its width from its content, i.e. it will be just wide enough to display the longest label. The other columns for the edit boxes and buttons are specified to be 80px and 60px wide respectively. In this case, the content (`TextBox` and `Button` objects) will take their widths from that of the column.

```
<Grid.ColumnDefinitions>
   <ColumnDefinition Width="Auto"/>
   <ColumnDefinition Width="80"/>
   <ColumnDefinition Width="60"/>
 </Grid.ColumnDefinitions>
```

The Label Objects(Column 1)

```
<Label Grid.Row="0" Grid.Column="0" Content="Fahrenheit"/>
<Label Grid.Row="1" Grid.Column="0" Content="Centigrade"/>
```

Here the XAML specifies `Label` objects Fahrenheit and Centigrade. Because they are defined within the `<Grid> ...</Grid>` tags, they are child objects of the Grid. In addition it is necessary to specify in which cells they are displayed using their `Grid.Row` and `Grid.Column` properties. Note that the cell coordinates have zero origin.

The TextBox Objects(Column 2)

```
<TextBox Name="txtFahrenheit" Grid.Row="0" Grid.Column="1"
 Margin="5"/>
<TextBox Name="txtCentigrade" Grid.Row="1" Grid.Column="1"
 Margin="5"/>
```

The XAML specifies two `TextBox` objects named *txtFahrenheit* and *txtCentigrade* respectively. Setting `Margin` to "5" means that a margin of 5px is applied around each edge; otherwise the text boxes would occupy the entire width of the column (80px). The effective width of each `TextBox` will therefore be 70px `(80-2×5)`.

The `Button` Objects (Column 3)

```
<Button Name="btnF2C" Grid.Row="0" Grid.Column="2"
 Content="F>C" Margin="5"/>
<Button Name="btnC2F" Grid.Row="1" Grid.Column="2"
 Content="C>F" Margin="5"/>
<Button Name="btnQuit" Grid.Row="2" Grid.Column="1"
 Content="Quit" Margin="5"/>
```

The XAML specifies three named `Button` controls. Note that the caption on a `Button` is specified by its `Content` property.

The ScrollBar Object

This example uses a `ScrollBar` which the user may scroll to input a value, either in Fahrenheit or Centigrade depending upon which of the two menu items (*Fahrenheit* or *Centigrade*) is checked.[1]

```
<ScrollBar Name="scrTemp" DockPanel.Dock="Right"  Width="20"
 Orientation="Vertical" Minimum="1" Maximum="213">
</ScrollBar>
```

This XAML snippet defines a `ScrollBar` named *scrTemp*.

Setting `DockPanel.Dock` to "Right" means that it will be docked (aligned) on the right edge of the `DockPanel`. It will be a vertical scrollbar, have a fixed width of 20px and a default height. The range of the `ScrollBar` is defined by its `Minimum` and `Maximum` properties which are set so that the `ScrollBar` will specify a value in Fahrenheit.

Note that in order to cause the `ScrollBar` to be docked (aligned) along the right edge of the `DockPanel` it is necessary to set `LastChildFill` to "False" (for the `DockPanel`) and `Dock` to "Right" (for the `ScrollBar`), because the value of `LastChildFill` (default "True") overrides the `Dock` value of the last defined child of the `DockPanel`.

Note

The XAML that defines this user-interface is at the same time both simple and complex. It is simple because (in this case) it is readily understood., It is complex because in order to write it, the user-interface designer must understand precisely how the various controls and their properties behave and work together. For these details, you should refer to the appropriate documentation and check out the large number of examples published on the internet.

---

[1] A `ScrollBar` is not the ideal choice of control for this type of user-interation, but this example is designed to look and behave like the original Dyalog GUI example, which was writtenfor the original version of Dyalog APL for Windows .

The Code to display the XAML

The function `TempConverter` shown below contains the code needed to display and operate the user interface whose layout is defined by the XAML described above.

```
     ∇ TempConverter;str;xml;win;txtFahrenheit;txtCentigrade;
                      mnuFahrenheit;mnuCentigrade;btnF2C;
                      btnC2F;btnQuit;scrTemp;sink
[1]    ⎕USING←'System'
[2]    ⎕USING,←⊂'System.IO'
[3]    ⎕USING,←⊂'System.Windows.Markup'
[4]    ⎕USING,←⊂'System.Xml,system.xml.dll'
[5]    ⎕USING,←⊂'System.Windows.Controls.Primitives,
                 WPF/PresentationFramework.dll'
[6]
[7]    str←⎕NEW StringReader(⊂XAML)
[8]    xml←⎕NEW XmlTextReader str
[9]    win←XamlReader.Load xml
[10]
[11]   txtFahrenheit←win.FindName⊂'txtFahrenheit'
[12]   txtCentigrade←win.FindName⊂'txtCentigrade'
[13]   mnuFahrenheit←win.FindName⊂'mnuFahrenheit'
[14]   mnuFahrenheit.onClick←'SET_F'
[15]   mnuCentigrade←win.FindName⊂'mnuCentigrade'
[16]   mnuCentigrade.onClick←'SET_C'
[17]   (btnF2C←win.FindName⊂'btnF2C').onClick←'f2c'
[18]   (btnC2F←win.FindName⊂'btnC2F').onClick←'c2f'
[19]   (btnQuit←win.FindName⊂'btnQuit').onClick←'Quit'
[20]   (scrTemp←win.FindName⊂'scrTemp').onScroll←'F2C'
[21]   sink←win.ShowDialog
     ∇
```

The variable `XAML` (a character vector) contains the XAML described previously.

Note that apart from the names given to the objects by the XAML and used by the function, the XAML and the code are independent.

`TempConverter[7-8]` create an `XamlReader` object from the character vector via `StringReader` and `XmlTextReader` objects.

```
[7]    str←⎕NEW StringReader(⊂XAML)
[8]    xml←⎕NEW XmlTextReader str
```

`TempConverter[9]` instantiates the XAML content by calling its `Load` method, which returns a reference `win` to the top-level control (in this case a Window) defined therein. The Window is not yet visible.

```
[9]    win←XamlReader.Load xml
```

Earlier, it was explained that objects defined bt the XAML must be *named* in order that they can be referenced (used) by the code. The mechanism to achieve this is to call the `FindName` method of the Window, which returns a reference to the specified (named) object. So these statements:

```
[11]   txtFahrenheit←win.FindName⊂'txtFahrenheit'
[12]   txtCentigrade←win.FindName⊂'txtCentigrade'
```

obtain refs (in this case named **`txtFahrenheit`** and **`txtCentigrade`**) to objects named *txtFahrenheit* and *txtCentigrade*. It is convenient (but not essential) to use the same name for the ref as is used for the control.

Most of the remaining statements obtain refs to the `MenuItem`, `Button` and `ScrollBar` objects and attach callback functions to their `Click` and `Scroll` events respectively.

```
[13]   mnuFahrenheit←win.FindName⊂'mnuFahrenheit'
[14]   mnuFahrenheit.onClick←'SET_F'
[15]   mnuCentigrade←win.FindName⊂'mnuCentigrade'
[16]   mnuCentigrade.onClick←'SET_C'
[17]   (btnF2C←win.FindName⊂'btnF2C').onClick←'f2c'
[18]   (btnC2F←win.FindName⊂'btnC2F').onClick←'c2f'
[19]   (btnQuit←win.FindName⊂'btnQuit').onClick←'Quit'
[20]   (scrTemp←win.FindName⊂'scrTemp').onScroll←'F2C'
```

Finally the code displays the Window and hands it over to the user by calling the `ShowDialog` method of the top-level Window.

```
[21]   sink←win.ShowDialog
```

`ShowDialog` displays the Window *modally*; i.e. until it is closed, the user may interact only with that Window. It is equivalent to **`]DQ win`** or **`win.Wait`** in the Dyalog built-in GUI.

The CallBack Functions

The callback functions are named as they are in the basic Dyalog GUI example and are remarkably similar. See *Interface Guide: GUI Tutorial*.

Callback function **`f2c`** which is attached to the `Click` event of the **`btnF2C`** button (labelled *F>C*) reads the character string in the **`txtFahrenheit`** `TextBox`, converts it to a number using **`Text2Num`**, calculates the equivalent in centigrade and then displays the result in the **`txtCentigrade`** `TextBox`.

```
     ∇ f2c;value
[1]    ⍝ Callback to convert Fahrenheit to Centigrade
[2]    :If 1=⍴,value←Text2Num txtFahrenheit.Text
[3]        txtCentigrade.Text←2⍕(value-32)×5÷9
[4]    :Else
[5]        txtCentigrade.Text←'invalid'
[6]    :EndIf
     ∇
```

For completeness, the `Text2Num` function is shown below. Note that if the user enters an invalid number, `Text2Num` returns an an empty vector, and the callback displays the text *invalid* instead.

```
     ∇ num←Text2Num txt;val
[1]    val num←⍎⍞VFI txt
[2]    num←val/num
     ∇
```

The `c2f` function converts from Centigrade to Fahrenheit when the user presses the button labelled *C>F*.

```
     ∇ c2f;value
[1]   ⍝ Callback to convert Centigrade to Fahrenheit
[2]    :If 1=⍴,value←Text2Num txtCentigrade.Text
[3]        txtFahrenheit.Text←2⍕32+value÷5÷9
[4]    :Else
[5]        txtFahrenheit.Text←'invalid'
[6]    :EndIf
     ∇
```

The callbacks `F2C` and `C2F`, one of which at a time is attached to the `Scroll` event of the `ScrollBar` object are shown below. The argument `Msg` contains two items, namely:

`[1]`	Object	a ref to the ScrollBar object
`[2]`	Object	a ref to an object of type System.Windows.Controls.Primitives.ScrollEventArgs

In this case the code uses the `NewValue` property of the ScrollEventArgs object. An alternative would be to refer to the `Value` property of the `ScrollBar` object

```
     ∇ F2C Msg;C;F;val
[1]   ⍝ Callback for Fahrenheit input via scrollbar
[2]    txtFahrenheit.Text←2⍕val←213-(2⊃Msg).NewValue
[3]    txtCentigrade.Text←2⍕(val-32)×5÷9
     ∇
```

```
     ∇ C2F Msg;C;F;val
[1]    ⍝ Callback for Centigrade input via scrollbar
[2]    txtCentigrade.Text←2⍕val←101-(2⊃Msg).NewValue
[3]    txtFahrenheit.Text←2⍕32+val÷5÷9
     ∇
```

The callbacks `SET_F` and `SET_C` which are attached to the `Click` events of the two `MenuItem` objects are shown below.

```
     ∇ SET_F
[1]    ⍝ Sets the scrollbar to work in Fahrenheit
[2]    scrTemp.(Minimum Maximum)←1 213
[3]    scrTemp.onScroll←'F2C'
[4]    mnuFahrenheit.IsChecked←1
[5]    mnuCentigrade.IsChecked←0
     ∇


     ∇ SET_C
[1]    ⍝ Sets the scrollbar to work in Centigrade
[2]    scrTemp.(Minimum Maximum)←1 101
[3]    scrTemp.onScroll←'C2F'
[4]    mnuCentigrade.IsChecked←1
[5]    mnuFahrenheit.IsChecked←0
     ∇
```

Finally, the callback function `Quit` which is attached to the `Click` event on the *Quit* button, simply calls the `Close` method of the Window:

```
     ∇ Quit arg
[1]    win.Close
     ∇
```

Notice that unlike its equivalent in the Dyalog GUI, it is not appropriate to close the Window using the expression `⎕EX 'win'`. This would expunge the ref to the Window but have no effect on the Window itself.

Using Code

The functions for this example are provided in the workspace `WPFINtro.dws` in the namespace `WPF.UsingCode`. To run the example:

```
)LOAD WPFINtro
WPF.UsingCode.TempConverter
```

The following function `TempConverter` performs *exactly* the same task of defining and manipulating the user-interface for the Temperature Converter example using XAML which was discussed previously.

The callback functions it uses are identical.

```
     ∇ TempConverter;⎕USING;win;dp;mnu;mnuFahrenheit;
                     mnuCentigrade;gr;tn;rd1;rd2;rd3;
                     rc1;rc2;rc3;l1;l2;txtFahrenheit;
                     txtCentigrade;btnF2C;btnC2F;
                     btnQuit;sink;mnuScale;scrTemp
[1]
[2]    ⎕USING←,⊂'System.Windows.Controls,
                 WPF/PresentationFramework.dll'
[3]    ⎕USING,←⊂'System.Windows.Controls.Primitives,
                 WPF/PresentationFramework.dll'
[4]    ⎕USING,←⊂'System.Windows,
                 WPF/PresentationFramework.dll'
[5]    ⎕USING,←⊂'System.Windows,
                 WPF/PresentationCore.dll'
[6]
[7]    win←⎕NEW Window
[8]    win.SizeToContent←SizeToContent.WidthAndHeight
[9]    win.Title←'WPF Temperature Converter'
[10]
[11]   dp←⎕NEW DockPanel
[12]   dp.LastChildFill←0
[13]
[14]   mnu←⎕NEW Menu
[15]
[16]   mnuScale←⎕NEW MenuItem
[17]   mnuScale.Header←'_Scale'
[18]   sink←mnu.Items.Add mnuScale
[19]
[20]   mnuFahrenheit←⎕NEW MenuItem
[21]   mnuFahrenheit.Header←'Fahrenheit'
[22]   mnuFahrenheit.IsCheckable←1
[23]   mnuFahrenheit.IsChecked←1
[24]   mnuFahrenheit.onClick←'SET_F'
[25]   sink←mnuScale.Items.Add mnuFahrenheit
[26]
[27]   mnuCentigrade←⎕NEW MenuItem
[28]   mnuCentigrade.Header←'_Centigrade'
[29]   mnuCentigrade.IsCheckable←1
[30]   mnuCentigrade.IsChecked←0
[31]   mnuCentigrade.onClick←'SET_C'
[32]   sink←mnuScale.Items.Add mnuCentigrade
[33]
[34]   sink←dp.Children.Add mnu
[35]   dp.SetDock mnu Dock.Top
[36]
[37]   gr←⎕NEW Grid
[38]   gr.Width←230
[39]   gr.Margin←⎕NEW Thickness(40 10 10 10)
[40]
[41]   rd1←⎕NEW RowDefinition
[42]   rd1.Height←GridLength.Auto
```

```
[43]    rd2←⍞NEW RowDefinition
[44]    rd2.Height←GridLength.Auto
[45]    rd3←⍞NEW RowDefinition
[46]    rd3.Height←GridLength.Auto
[47]    gr.RowDefinitions.Add¨rd1 rd2 rd3
[48]
[49]    rc1←⍞NEW ColumnDefinition
[50]    rc1.Width←GridLength.Auto
[51]    rc2←⍞NEW ColumnDefinition
[52]    rc2.Width←⍞NEW GridLength 80
[53]    rc3←⍞NEW ColumnDefinition
[54]    rc3.Width←⍞NEW GridLength 60
[55]    gr.ColumnDefinitions.Add¨rc1 rc2 rc3
[56]
[57]    l1←⍞NEW Label
[58]    l1.Content←'Fahrenheit'
[59]    sink←gr.Children.Add l1
[60]    gr.SetRow l1 0
[61]    gr.SetColumn l1 0
[62]
[63]    l2←⍞NEW Label
[64]    l2.Content←'Centigrade'
[65]    sink←gr.Children.Add l2
[66]    gr.SetRow l2 1
[67]    gr.SetColumn l2 0
[68]
[69]    txtFahrenheit←⍞NEW TextBox
[70]    txtFahrenheit.Margin←⍞NEW Thickness 5
[71]    sink←gr.Children.Add txtFahrenheit
[72]    gr.SetRow txtFahrenheit 0
[73]    gr.SetColumn txtFahrenheit 1
[74]
[75]    txtCentigrade←⍞NEW TextBox
[76]    txtCentigrade.Margin←⍞NEW Thickness 5
[77]    sink←gr.Children.Add txtCentigrade
[78]    gr.SetRow txtCentigrade 1
[79]    gr.SetColumn txtCentigrade 1
[80]
[81]    btnF2C←⍞NEW Button
[82]    btnF2C.Content←'F>C'
[83]    btnF2C.Margin←⍞NEW Thickness 5
[84]    btnF2C.onClick←'f2c'
[85]    sink←gr.Children.Add btnF2C
[86]    gr.SetRow btnF2C 0
[87]    gr.SetColumn btnF2C 2
[88]
[89]    btnC2F←⍞NEW Button
[90]    btnC2F.Content←'C>F'
[91]    btnC2F.Margin←⍞NEW Thickness 5
[92]    btnC2F.onClick←'c2f'
[93]    sink←gr.Children.Add btnC2F
```

```
[94]   gr.SetRow btnC2F 1
[95]   gr.SetColumn btnC2F 2
[96]
[97]   btnQuit←⎕NEW Button
[98]   btnQuit.Content←'Quit'
[99]   btnQuit.Margin←⎕NEW Thickness 5
[100]  btnQuit.onClick←'Quit'
[101]  sink←gr.Children.Add btnQuit
[102]  gr.SetRow btnQuit 2
[103]  gr.SetColumn btnQuit 1
[104]
[105]  sink←dp.Children.Add gr
[106]
[107]  scrTemp←⎕NEW ScrollBar
[108]  scrTemp.Width←20
[109]  scrTemp.Orientation←Orientation.Vertical
[110]  scrTemp.Minimum←1
[111]  scrTemp.Maximum←213
[112]  scrTemp.onScroll←'F2C'
[113]
[114]  sink←dp.Children.Add scrTemp
[115]  dp.SetDock scrTemp Dock.Right
[116]
[117]  win.Content←dp
[118]
[119]  sink←win.ShowDialog
     ∇
```

Although this approach appears at first sight to be considerably more verbose than using XAML (a 120-line function compared with a 21-line function and a 44-line block of XAML) each line of code performs only one very simple task, and no attempt has been made to write utility functions to perform the same task for similar controls, as might be done in a real application.

As before, let us examine the code line-by-line.

`TempConverter[2-5]` define `⎕USING` so that the appropriate .NET assemplies are on the search-path. Note that the `ScrollBar` control is in `System.Windows.Controls.Primitives` and not `System.Windows.Controls` like the others.

```
[2]     ⎕USING←,⊂'System.Windows.Controls,
                  WPF/PresentationFramework.dll'
[3]     ⎕USING,←⊂'System.Windows.Controls.Primitives,
                  WPF/PresentationFramework.dll'
[4]     ⎕USING,←⊂'System.Windows,
                  WPF/PresentationFramework.dll'
[5]     ⎕USING,←⊂'System.Windows,
                  WPF/PresentationCore.dll
```

`TempConverter[8-9]` creates a `Window` and sets its `SizeToContent` and `Title` properties as in the XAML example. Notice however that whereas using XAML the string `SizeToContent="WidthandHeight"` is sufficient, when using code it is necessary to *get theType right*. In this case, the `SizeToContent` property must be set to a specific member (in this case `WidthAndHeight`) of the `System.Windows.SizeToContent` enumeration. Other members of this Type are `Width`, `Height` and `Manual` (the default).

```
[7]     win←⎕NEW Window
[8]     win.SizeToContent←SizeToContent.WidthAndHeight
[9]     win.Title←'WPF Temperature Converter'
```

`TempConverter[11-12]` create a `DockPanel` control ansd set its `LastChildFill` property to 0. In this case the APL value 0 is used instead of the string "False" in XAML.

```
[11]    dp←⎕NEW DockPanel
[12]    dp.LastChildFill←0
```

`TempConverter[14]` creates a `Menu` control.

```
[14]    mnu←⎕NEW Menu
```

`TempConverter[16-18]` create a `MenuItem` control with the caption *Scale*, and then add the control to the `Items` collection of the main `Menu` using its `Add` method. This illustrates one significant difference between using XAML and code. In XAML, the parent/child relationships between controls are defined by the structure and order of the XML. Using code, child controls must be explicitly added to the appropriate list of child controls managed by the parent.

```
[16]    mnuScale←⎕NEW MenuItem
[17]    mnuScale.Header←'_Scale'
[18]    sink←mnu.Items.Add mnuScale
```

`TempConverter[20-25]` create a `MenuItem` control labelled *Fahrenheit*.The `IsCheckable` and `IsChecked` properties are set to 1, which is equivalent to "True" in XAML. The callback function `SET_F` is assigned to the `Click` event exactly as in the XAML version of this example. The last line in this section makes the *Fahrenheit* `MenuItem` a child of the *Scale* `MenuItem`.

```
[20]    mnuFahrenheit←⎕NEW MenuItem
[21]    mnuFahrenheit.Header←'Fahrenheit'
[22]    mnuFahrenheit.IsCheckable←1
[23]    mnuFahrenheit.IsChecked←1
[24]    mnuFahrenheit.onClick←'SET_F'
[25]    sink←mnuScale.Items.Add mnuFahrenheit
```

The code used to create the *Centigrade* `MenuItem` is more or less the same.

`TempConverter[34-35]` adds the top-level `Menu` to the `DockPanel`. Note that in the case of a `DockPanel`, the list of its child controls is represented by its `Children` property. Furthermore, to define how it is docked this is done, using code, by the `SetDock` method of the `DockPanel`. This contrasts with the way this is achieved using XAML (`DockPanel.Dock="Top"`). Note too that the argument to `SetDock` is not just a simple string as in XAML, but a member of the `System.Windows.Controls.Dock` enumeration.

```
[34]    sink←dp.Children.Add mnu
[35]    dp.SetDock mnu Dock.Top
```

`TempConverter[37-39]` create the `Grid` control. Its `Width` property will accept a simple numeric value, but its `Margin` property must be given an instance of a `System.Windows.Thickness` structure. In this case, the `ThickNess` constructor is given a 4-element numeric vector that specifies its `Left`, `Top`, `Right` and `Bottom` members respectively.

```
[37]    gr←⎕NEW Grid
[38]    gr.Width←230
[39]    gr.Margin←⎕NEW Thickness(40 10 10 10)
```

`TempConverter[41-47]` create instances of 3 `RowDefinition` classes and add them to the `RowDefinitions` collection of the `Grid`. Note that whereas in XAML the `Height` can be specified as a string, using code it is necessary once again to use the correct Type. In this case, `Height` must be specified by a member of the `System.Windows.GridLength` structure.

```
[41]    rd1←⎕NEW RowDefinition
[42]    rd1.Height←GridLength.Auto
[43]    rd2←⎕NEW RowDefinition
[44]    rd2.Height←GridLength.Auto
[45]    rd3←⎕NEW RowDefinition
[46]    rd3.Height←GridLength.Auto
[47]    gr.RowDefinitions.Add¨rd1 rd2 rd3
```

Similarly, `TempConverter[49-55]` create instances of 3 `ColumnDefinition` classes and add them to the `ColumnDefinitions` collection of the `Grid`. Note that The `Width` property will not accept a simple numeric value, it must be a member of the `GridLength` structure. To set the `Width` to 80, it is necessary first to create an instance of a `GridLength` structure giving this value as the argument to its constructor.

```
[49]    rc1←⎕NEW ColumnDefinition
[50]    rc1.Width←GridLength.Auto
[51]    rc2←⎕NEW ColumnDefinition
[52]    rc2.Width←⎕NEW GridLength 80
[53]    rc3←⎕NEW ColumnDefinition
[54]    rc3.Width←⎕NEW GridLength 60
[55]    gr.ColumnDefinitions.Add¨rc1 rc2 rc3
```

`TempConverter[57-61]` create a `Label` control with the caption *Fahrenheit*. To display the `Label` in a `Grid` it is necessary to first add it to the `Children` collection of the `Grid`, and then set its position in the `Grid` using its `SetRow` and `SetColumn` methods. Similar code is used to create and position the second `Label`.

```
[57]    l1←⎕NEW Label
[58]    l1.Content←'Fahrenheit'
[59]    sink←gr.Children.Add l1
[60]    gr.SetRow l1 0
[61]    gr.SetColumn l1 0
```

`TempConverter[69-73]` create and position a `TextBox` control, in the same way as the `Label` controls. Notice that in this case, the constructor for the Thickness structure is given a single value that specifies all four of its `Left`, `Top`, `Right` and `Bottom` members.

```
[69]    txtFahrenheit←⎕NEW TextBox
[70]    txtFahrenheit.Margin←⎕NEW Thickness 5
[71]    sink←gr.Children.Add txtFahrenheit
[72]    gr.SetRow txtFahrenheit 0
[73]    gr.SetColumn txtFahrenheit 1
```

`TempConverter[81-87]` create and position a `Button` control. The callback function `f2c` is attached to the `Click` event in the same way as in the XAML version of this example.

```
[81]    btnF2C←⎕NEW Button
[82]    btnF2C.Content←'F>C'
[83]    btnF2C.Margin←⎕NEW Thickness 5
[84]    btnF2C.onClick←'f2c'
[85]    sink←gr.Children.Add btnF2C
[86]    gr.SetRow btnF2C 0
[87]    gr.SetColumn btnF2C 2
```

`TempConverter[105]` adds the `Grid` to the list of `Children` to be managed by the `DockControl`.

```
[105]   sink←dp.Children.Add gr
```

`TempConverter[107-112]` create a `ScrollBar` control. Its `Width`, `Minimum` and `Maximum` properties all accept simple numeric values. However, its `Orientation` property must be set to a member of the `System.Windows.Controls.Orientation` enumeration.

```
[107]   scrTemp←⎕NEW ScrollBar
[108]   scrTemp.Width←20
[109]   scrTemp.Orientation←Orientation.Vertical
[110]   scrTemp.Minimum←1
[111]   scrTemp.Maximum←213
[112]   scrTemp.onScroll←'F2C'
```

`TempConverter[114-115]` add the `ScrollBar` to the list of `Children` managed by the `DockPanel`, and use its `SetDock` method to cause it to be right-aligned.

```
[114]   sink←dp.Children.Add scrTemp
[115]   dp.SetDock scrTemp Dock.Right
```

Finally, the `DockPanel` is assigned to the `Content` property of the `Window`, and the `Window` displayed as in the XAML version of this example. Note that a `Window` may contain just one control.

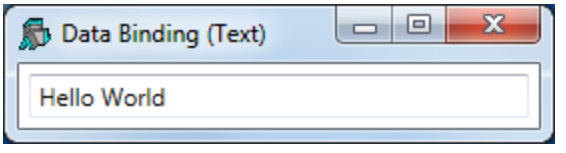
```
[117]  win.Content←dp
[118]
[119]  sink←win.ShowDialog
```

Data Binding

This section provides some simple examples of WPF data binding using Dyalog APL. Each example builds upon the one before, so it is advisable to read them in order.

Example 1

This example illustrates data binding using XAML to specify the user-interface coupled with an APL function to drive it and handle the data binding.



The XAML

The XAML shown below, describes a Window containing a TextBox.

```
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Name="Temp"
 Title="Data Binding (Text)"
 SizeToContent="WidthandHeight">
     <TextBox Name="txt" Width="300" Margin="5"
      Text="{Binding txtSource,Mode=TwoWay,
             UpdateSourceTrigger=PropertyChanged}"/>
</Window>
```

It contains a data binding expressions, namely:

```
      Text="{Binding txtSource,Mode=TwoWay,
             UpdateSourceTrigger=PropertyChanged}"
```

This specifies that the `Text` property of the `TextBox` is bound to a value in the Binding Source (which has yet to be defined) whose path is `txtSource`. The binding mode is set to `TwoWay` which means that any change in the `TextBox` will be reflected in a new value in the Binding Source, and vice-versa. The value in the Binding Source will be updated when the property (in this case the `Text` Property) changes.

The APL Code

Th function `Text` which generates this example is shown below.

The argument `txt` is the text to be displayed initially in the `TextBox`. Note that the variable `XAML_Text` contains the XAML that describes the user-interface listed above.

```
     ∇ Text txt;⎕USING;str;xml;win
[1]    ⎕USING,←,⊂'System.Windows.Controls,
                  WPF/PresentationFramework.dll'
[2]    win←LoadXAML XAML_Text
[3]    win.txtBox←win.FindName⊂'txt'
[4]
[5]    ⎕EX'txtSource'
[6]    txtSource←txt
[7]    win.txtBox.DataContext←2015⌶'txtSource'
[8]
[9]    win.Show
     ∇
```

The utility function `LoadXAML` incorporates the 3 lines of code, used to create a WPF window from XAML, that were coded in-line in previous examples in this chapter.

```
     ∇ win←LoadXAML xaml;⎕USING;str;xml
[1]    ⎕USING←'System.IO'
[2]    ⎕USING,←⊂'System.Windows.Markup'
[3]    ⎕USING,←⊂'System.Xml,system.xml.dll'
[4]    ⎕USING,←⊂'System.Windows.Controls,
                 WPF/PresentationFramework.dll'
[5]    str←⎕NEW StringReader(⊂xaml)
[6]    xml←⎕NEW XmlTextReader str
[7]    win←XamlReader.Load xml
     ∇
```

`Text[1]` defines the .NET search path needed to access the WPF controls.

```
[1]    ⎕USING,←,⊂'System.Windows.Controls,
                  WPF/PresentationFramework.dll'
```

`Text[2-3]` uses the utility function `LoadXAML` to load a WPF user-interface from the XAML and then uses the `FindName` method to obtain a reference to the object named *txt*.

```
[2]    win←LoadXAML XAML
[3]    win.txtBox←win.FindName⊂'txt'
```

`Text[5-6]` initialise a new global variable named `txtSource` to the value of the argument. When using a *global* variable as a data binding source, it is generally advisable to establish a new variable by first expunging it.[1]

```
[5]   ⎕EX'txtSource'
[6]   txtSource←txt
```

`Text[7]`creates a Binding Source object using `2015⌶` and assigns it to the `DataContext` property of the `TextBox` object. Because it is a character vector, the exported Type for the bound variable `txtSource` is `System.String` which is appropiate for the `Text` property of a `TextBox`.

```
[7]    win.txtBox.DataContext←2015⌶'txtSource'
```

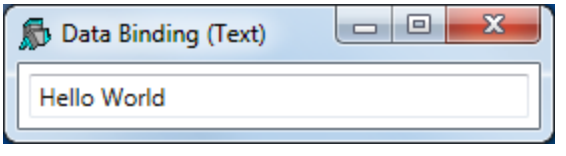
`Text[9]` displays the Window. Note that although the APL local variable `win` goes out of scope when the function terminates, the Window remains visible until the user has closed it.

```
[9]   win.Show
```

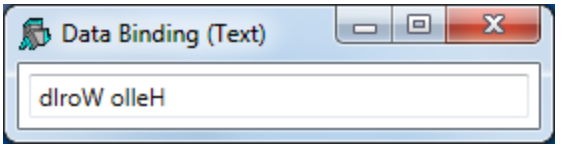
Testing the Data Binding

The following expressions may be used to explore the effect of data binding.

```
)LOAD WPFIntro
)CS DataBinding.Text

Text 'Hello World'
```

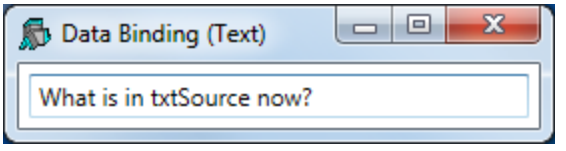



```
txtSource←⌽txtSource
```



[1]This is because its binding type (the exported type of the data bound variable) is stored in the workspace along with its value, and the binding type (were it to be incorrect) may not be changed once it has been established.

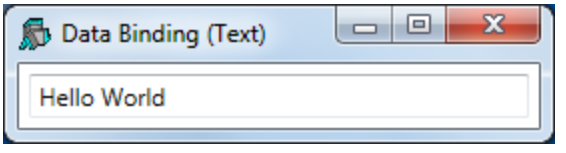
Typing into the TextBox changes the value of the bound variable.




```
      txtSource
What is in txtSource now?
```

Example 2

This example illustrates the use of the optional left argument to `2015⌶` to specify the data type used to export the value of the bound variable.



The XAML

The XAML shown below, describes the same Window containing a TextBox as before.

```
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Name="Temp"
 Title="Data Binding (FontSize)"
 SizeToContent="WidthandHeight">
     <TextBox Name="txt" Text="Hello World" Width="300"
      Margin="5"
      FontSize="{Binding sizeSource,Mode=OneWay}"/>
</Window>
```

This time, the data binding expression is:

```
      FontSize="{Binding sizeSource,Mode=OneWay}"/>
```

This specifies that the `FontSize` property of the `TextBox` is bound to a value in the Binding Source (which has yet to be defined) whose path is `sizeSource`. The binding mode is set to `OneWay` which means that the `FontSize` property depends on the data value but not vice versa. Were the `FontSize` to change for any external reason (which is admittedly unlikely in the case of `FontSize`), it would not alter the value in `sizeSource` to which it is bound.

The APL Code

The function `FontSize` is almost identical to the function `Text` which is described in Example 1.

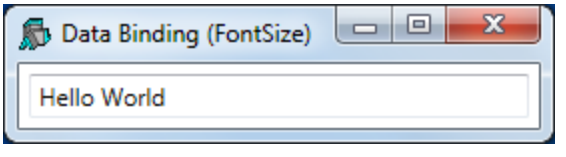
```
     ∇ FontSize size;⎕USING;win
[1]    ⎕USING←'System'
[2]    ⎕USING,←⊂'System.Windows.Controls,
                 WPF/PresentationFramework.dll'
[3]    win←LoadXAML XAML
[4]    win.txtBox←win.FindName⊂'txt'
[5]
[6]    ⎕EX'sizeSource'
[7]    sizeSource←size
[8]    win.txtBox.DataContext←Int32(2015⌶)'sizeSource'
[9]
[10]   win.Show
     ∇
```

The key difference is in `FontSize[8]`. Here the left argument of `(2015⌶)` is `Int32`. This means that the exported Type of the variable `sizeSource` will be `Int32`. This Type (a 32-bit integer) is required by the `FontSize` property of a `TextBox`; no other Type will do. If this were omitted, APL would export the value of the variable using a Type dependent on its internal format (most likely `Int16`) and the binding would fail.

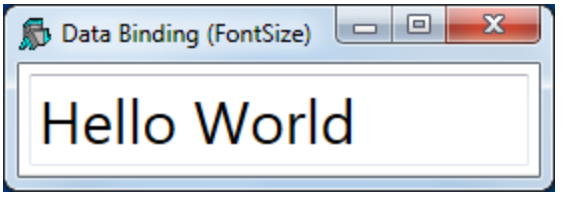
```
[8]    win.txtBox.DataContext←Int32(2015⌶)'sizeSource'
```

Testing the Data Binding

```
      )LOAD WPFIntro
      )CS DataBinding.FontSize

      FontSize 12
```




```
      sizeSource
12
      sizeSource←30
```



Example 3

This example, uses APL code to both build the user-interface (instead of using XAML) and handle the data binding. In this case both the Text and the FontSize properties are bound to APL variables. The function is shown below:

```
     ∇ TextFontSize(txt size);⎕USING;win;sink
[1]    ⎕USING←'System'
[2]    ⎕USING,←,⊂'System.Windows.Controls,
                  WPF/PresentationFramework.dll'
[3]    ⎕USING,←⊂'System.Windows.Controls.Primitives,
                 WPF/PresentationFramework.dll'
[4]    ⎕USING,←⊂'System.Windows,
                 WPF/PresentationFramework.dll'
[5]    ⎕USING,←⊂'System.Windows,
                 WPF/PresentationCore.dll'
[6]
[7]    ⍝ Create a Window, DockPanel and TextBox
[8]    win←⎕NEW Window
[9]    win.SizeToContent←SizeToContent.WidthAndHeight
[10]   win.Title←'Data Binding (Text and FontSize)'
[11]   win.txtBox←⎕NEW TextBox
[12]   win.txtBox.Width←350
[13]   win.Content←win.txtBox
[14]
[15]   ⍝ Define data binding from variable "txtSource"
[16]   ⍝ to the Text property of TextBox win.txtBox
[17]   ⎕EX'txtSource'
[18]   txtSource←txt
[19]   win.txtbinding←⎕NEW Data.Binding(⊂'txtSource')
[20]   win.txtbinding.Source←2015⌶'txtSource'
[21]   win.txtbinding.Mode←Data.BindingMode.TwoWay
[22]   win.txtbinding.UpdateSourceTrigger←
           Data.UpdateSourceTrigger.PropertyChanged
[23]   sink←win.txtBox.SetBinding
                 TextBox.TextProperty win.txtbinding
[24]
[25]   ⍝ Define data binding from variable "sizeSource"
[26]   ⍝ to the FontSize property of TextBox win.txtBox
[27]   ⎕EX'sizeSource'
[28]   sizeSource←size
[29]   win.fntbinding←⎕NEW Data.Binding(⊂'sizeSource')
[30]   win.fntbinding.Source←Int32(2015⌶)'sizeSource'
[31]   win.fntbinding.Mode←Data.BindingMode.OneWay
[32]   sink←win.txtBox.SetBinding
                 TextBox.FontSizeProperty win.fntbinding
[33]
[34]   win.Show
     ∇
```

Apart from the code that creates the controls, the only material difference between this and the previous examples is the way that the bindings are handled.

In code (as opposed to using XAML) this is done using explicit `Binding` objects[1]. The code for binding the `Text` property to the `txtSource` variable is as follows:

```
[19]   win.txtbinding←⍞NEW Data.Binding(⊂'txtSource')
[20]   win.txtbinding.Source←2015⌶'txtSource'
[21]   win.txtbinding.Mode←Data.BindingMode.TwoWay
[22]   win.txtbinding.UpdateSourceTrigger←
           Data.UpdateSourceTrigger.PropertyChanged
[23]   sink←win.txtBox.SetBinding
               TextBox.TextProperty win.txtbinding
```

Line [19] creates a `Binding` object, passing the constructor the the name of the APL variable `txtSource` as the `Path` to the binding value.

```
[19]   win.txtbinding←⍞NEW Data.Binding(⊂'txtSource')
```

Line [20] creates a Binding Source object using `2015⌶` as before, but this time assigns it to the `Source` property of the `Binding` object.

```
[20]   win.txtbinding.Source←2015⌶'txtSource'
```

Line [21] sets the `Mode` property of the `Binding` object to `TwoWay` (a field of the `BindingMode` Type). As in Example 1, this specifies two-way binding.

```
[21]   win.txtbinding.Mode←Data.BindingMode.TwoWay
```

Line [22] sets the `UpdateSourceTrigger` property of the `Binding` object to `PropertyChanged` (a field of the `UpdateSourceTrigger` Type). This causes the value in the Binding Source (in this case `txtSource`) to be changed whenever the property (in this case the `Text` property) of the `TextBox` changes. This will occur on every keystroke.

```
[22]   win.txtbinding.UpdateSourceTrigger←
           Data.UpdateSourceTrigger.PropertyChanged
```

(Note that the three types `Binding`, `BindingMode` and `UpdateSourceTrigger` are located in `System.Windows.Data`)

The code that establishes the binding between the `sizeSource` variable and the `FontSize` property is very similar.

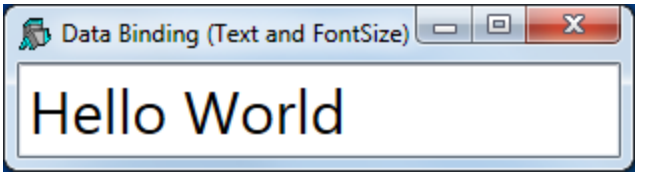
[1]Binding objects are implicit in all binding operations, but are created declaratively when using XAML.

```
[29]    win.fntbinding←⎕NEW Data.Binding(⊂'sizeSource')
[30]    win.fntbinding.Source←Int32(2015⌶)'sizeSource'
[31]    win.fntbinding.Mode←Data.BindingMode.OneWay
[32]    sink←win.txtBox.SetBinding
                TextBox.FontSizeProperty win.fntbinding
```

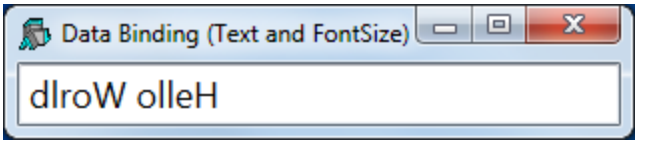
Note however that (as in Example 2) the left-argument to `(2015⌶)` specifies that the exported data type of the `sizeSource` variable is to be `Int32`.

Testing the Data Binding

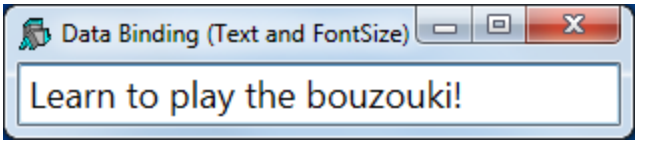
```
      )LOAD WPFIntro
      )CS DataBinding.TextFontSizeCode

      TextFontSize 'Hello World' 30
```




```
      txtSource sizeSource←(⍎txtSource) 18
```



As in previous examples, when the user changes the text, the new text appears in `txtSource`.




```
      txtSource
Learn to play the bouzouki!
```

Note

It is perhaps worth mentioning that if you want to bind two properties *of the same object* to two APL variables, it has to be done by writing code as shown in this example, using two separate Binding Source objects. This is because using XAML you may only associate a single Binding Source to an object.

However, this minor restriction is easily surmounted by using an APL namespace as a Binding Source as illustrated in the next Example.

Example 4

This example uses XAML to specify the user-interface and the main components of the data binding.

The XAML

The XAML is much the same as in Example 1 and 2 except that it connects two properties `Text` and `FontSize` of the same `TextBox` to two Paths *txtSource* and *sizeSource*.

```
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Name="Temp"
 Title="Data Binding (Text and FontSize)"
 SizeToContent="WidthandHeight">
     <TextBox Name="txt" Width="350" Margin="5"
      Text="{Binding txtSource,Mode=TwoWay,
             UpdateSourceTrigger=PropertyChanged}"
      FontSize="{Binding sizeSource,Mode=OneWay}"/>
</Window>
```

The APL Code

The function `TextFontSize` is shown below.

```
     ∇ TextFontSize(txt size);⎕USING;win;options
[1]    ⎕USING←'System'
[2]    ⎕USING,←⊂'System.Windows,
                 WPF/PresentationFramework.dll'
[3]
[4]    win←LoadXAML XAML
[5]
[6]    src←⎕NS''
[7]    src.(txtSource sizeSource)←txt size
[8]    options←2 2⍴'txtSource'String'sizeSource'Int32
[9]
[10]   win.DataContext←options(2015⌶)'src'
[11]
[12]   win.Show
     ∇
```

Lines [6-7] create a new namespace *src* containg two variables `txtSource` and `sizeSource` which are initialised to the arguments of the function.

```
[6]    src←⎕NS''
[7]    src.(txtSource sizeSource)←txt size
```

Line [8] creates a local variable named options which will be used as the left argument of `2015⌶)`. It is a 2-column matrix. The first column is a list of the names of the variables which are to be exported by the namespace when used as a Binding Source. The second column specifies their data types.

```
[8]    options←2 2⍴'txtSource'String'sizeSource'Int32
```

Line [10] creates a Binding Source object from the namespace `src` and a left argument `options` and assigns it to the `DataContext` property of the `Window win`.

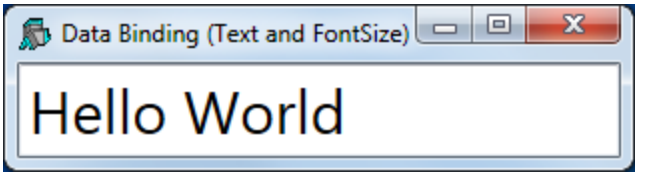
```
[10]   win.DataContext←options(2015⌶)'src'
```

An alternative would be to assign it to the `DataContext` property of the `TextBox` object, but this would require one further line of code to identify it. The reason this works is that the DataContext property of a `TextBox` (and many other controls) is inherited from its parent Window. This feature allows a single Binding Source namespace to be used to specify data bindings between its component variables and any number of properties of any number of controls in the same Window.

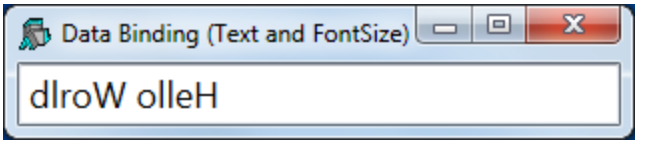
As shown before, the left argument of `2015⌶)` is optional. Without it, the namespace would export all its variables using default binding types. In this case, because the binding type of `sizeSource` must be specifed as `Int32`, it is necessary to use a left argument, which means specifying all the variables involved.

Testing the Data Binding

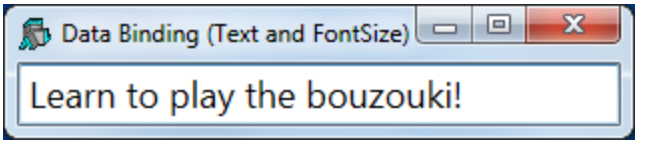
```
      )LOAD WPFIntro
      )CS DataBinding.TextFontSizeXAML

      DB_Text_FontSize_XAML'Hello World' 30
```




```
      src.(txtSource sizeSource←(⍎txtSource) 18)
```



As in previous examples, when the user changes the text, the new text appears in `txtSource`.




```
      src.txtSource
Learn to play the bouzouki!
```

Example 5

WPF data binding provides the means to bind controls that display lists of items, such as the `ListBox`, `ListView`, and `TreeView` controls, to collections of data. These controls are all based upon the `ItemsControl` class. To bind an `ItemsControl` to a collection object, you use its `ItemsSource` property.

If the right argument of `2015⌶` names a variable, or a namespace containing a variable, that is a vector other than a simple character vector, it returns a Binding Source object that provides the necessary interfaces to bind the variable as a collection to the `ItemSource` property of an `ItemsControl`.

The APL variable will normally contain a vector of character vectors, because most `ItemsControl` objects deal with collections of strings. However, any APL vector other than a simple character vector will be treated in this way.

This example illustrates binding between a variable containing a vector of character vectors, to the items of a `ListBox`.

Incidentally, the `ItemsSource` property overrides the `Items` collection as a means to specify the content of the `ItemsControl`. When the `ItemsSource` property is set, the `Items` collection becomes read-only and of fixed-size. Note that the `ItemsSource` property supports OneWay binding by default.

The XAML

The variable `XAML_FilteredList`, shown below, contains XAML to specify a Window containing a `StackPanel`. The `StackPanel` control is a WPF layout control that organises child controls in a single line, by default vertically. In this example, the `StackPanel` contains a `TextBox` and, below it, a `WrapPanel`, and below that a `TextBlock`. The `WrapPanel` is also a layout control that organises its child controls sequentially from left to right. The `WrapPanel` contains two `ListBox` controls.

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentat
ion"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Filtered List Example"
    SizeToContent="WidthAndHeight"
    Topmost="true">
    <StackPanel>
        <TextBox Name="filter" Margin="5"
         Text="{Binding Filter,Mode=TwoWay,
                UpdateSourceTrigger=PropertyChanged}"/>
        <WrapPanel>
            <ListBox Name="all" Width="135" Height="440"
             Margin="5" ItemsSource="{Binding DyalogNames}"/>
            <ListBox Name="filtered" Width="135" Height="440"
             Margin="5" ItemsSource="{Binding FilteredList}"/>
        </WrapPanel>
        <TextBlock Text="Dyalog WPF Demo" Margin="5"/>
    </StackPanel>
</Window>
```

The Code

```
     ∇ FilteredList;MySource;win;sink
[1]
[2]    MySource←⎕NS''
[3]    MySource.Filter←''
[4]    MySource.FilteredList←0⍴⊂''
[5]    MySource.DyalogNames←DyalogNames
[6]
[7]    win←LoadXAML XAML_FilteredList
[8]    win.DataContext←2015⌶'MySource'
[9]    (win.FindName⊂'filter').onTextChanged←
                              'FilteredList_TextChanged'
[10]   sink←win.ShowDialog
     ∇
```

Like the previous example, this example uses a namespace `MySource` containing the bound variables `Filter`, `FilteredList` and `DyalogNames`.

`FilteredList[8]` creates a Binding Source object and assigns it to the `DataContext` property of the `Window` `win`.

```
[8]    win.DataContext←2015⌶'MySource'
```

The `DataContext` property is inherited by all child controls , so they all share the same Binding Source. Their different `Paths` to different values in the Binding Source are specified in the XAML as follows.

The `Text` property of the `TextBox` named *filter* is bound to the variable `Filter` by the expression `Text="{Binding Filter,...`

```
<TextBox Name="filter" Margin="5"
 Text="{Binding Filter,Mode=TwoWay,
```

The `ItemsSource` property of the `ListBox` named *all* is bound to the variable `DyalogNames` by the expression `ItemsSource="{Binding DyalogNames}"`

```
<ListBox Name="all" Width="135" Height="440"
 Margin="5" ItemsSource="{Binding DyalogNames}"/>
```

Thirdly, the `ItemsSource` property of the `ListBox` named *filtered* is bound to the variable `FilteredList` by the expression `ItemsSource="{Binding FilteredList}"`

```
<ListBox Name="filtered" Width="135" Height="440"
 Margin="5" ItemsSource="{Binding FilteredList}"/>
```

Testing the Data Binding

```
      FilteredList
```

If the user types a single character, in this case "e", into the `TextBox`, this fires a `TextChanged` event which in turn fires the callback function shown below:

```
     ∇ FilteredList_TextChanged a;hits
[1]    hits←(⊂MySource.Filter){⍵/⍨⍺∈⍵}¨DyalogNames
[2]    MySource.FilteredList←hits/DyalogNames
     ∇
```

When the callback runs, the variable `MySource.Filter`, which is bound to the `Text` property of the `TextBox`, will contain "e". The function calculates a mask `hits` which identifies which members of the variable `DyalogNames` contain this string. It then assigns that subset to the variable `MySource.FilteredList`. This is bound to the `ItemsSource` property of the right-hand `ListBox`, so the result is as follows:

Similarly, typing "er" into the `TexBox` reduces the number of hits as shown below:

Example 6

This example illustrates data binding using a vector of .NET objects, in this case `DateTime` objects.

The XAML

The XAML shown below, describes a Window containing a `StackPanel`, inside which is a `ListBox`.

```
<Window

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentat
ion"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="NetObjects (DateTime) Example"
    SizeToContent="WidthAndHeight" >
    <StackPanel>
         <TextBlock Text="Dates of forthcoming Orthodox Easters"
          FontSize="18" Margin="5"/>
         <ListBox Name="EasterDates" Height="100"
          Margin="5" />
    </StackPanel>
</Window>
```

The APL Code

The function `NetObjects` is shown below.

```
     ∇ NetObjects;⎕USING;win;dt
[1]    ⎕USING←'System'
[2]    win←LoadXAML XAML
[3]    win.dates←win.FindName⊂'EasterDates'
[4]    dt←{⎕NEW DateTime ⍵}¨Easter
[5]    win.dates.ItemsSource←2015↓'dt'
[6]    sink←win.ShowDialog
     ∇
```

`NetObjects[3]` uses `FindName` to obtain a ref to the `ListBox` (defined in the XAML) named *EasterDates:*

```
[3]    win.dates←win.FindName⊂'EasterDates'
```

The global variable `Easter` contains a vector of 3-element numeric vectors representing the dates of forthcoming Orthodox Easter Sundays.

```
      ↑Easter
2015 4 12
2016 5  1
2017 4 16
2018 4  8
2019 4 28
2020 4 19
2021 5  2
2022 4 24
2023 4 16
2024 5  5
```

`NetObjects[4]` creates a vector of `DateTime` objects from the global variable `Easter`.

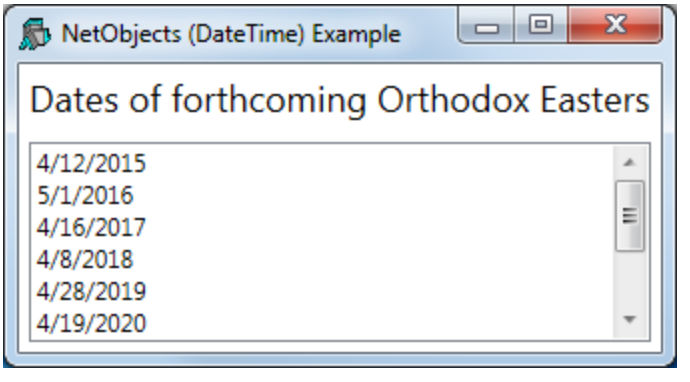
```
[4]    dt←{⍎'NEW DateTime ⍵'}¨Easter
```

Then, NetObjects[5] creates a binding source object from this array and assigns it to the ItemsSource property of the ListBox.

```
[5]    win.dates.ItemsSource←2015⌶'dt'
```

Testing the Data Binding

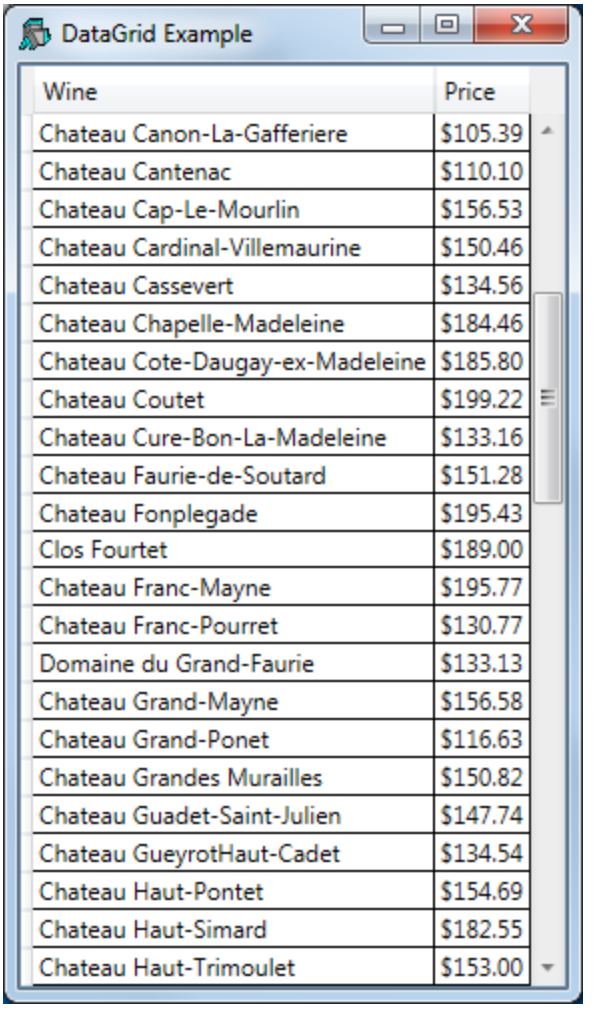
```
      )LOAD WPFIntro
      DataBinding.NETObjects.NETObjects
```



Example 7

This example illustrates data binding using 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.



Wine	Price
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

The XAML

The XAML shown below, describes a Window containing a `DockPanel`, inside which is a `DataGrid`.

```
 <Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentat
ion"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DataGrid 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

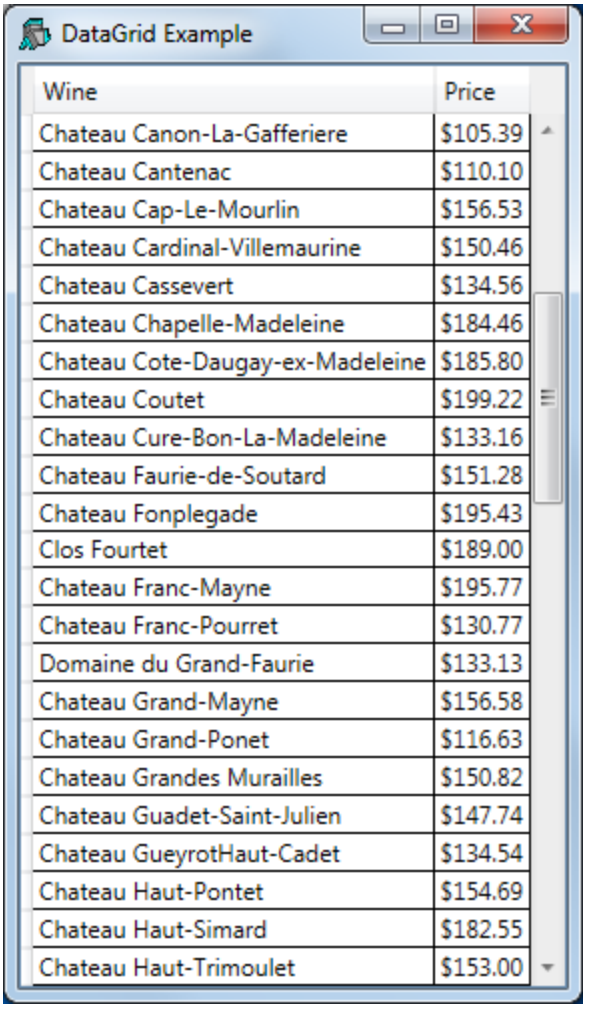
The function `Grid` is shown below.

```
     ∇ Grid;⎕USING;MySource;win
[1]    ⎕USING←'System'
[2]    winelist←⎕NS¨(⍴Wines)⍴⊂''
[3]    winelist.Name←Wines
[4]    winelist.Price←0.01×10000+?(⍴Wines)⍴10000
[5]
[6]    win←LoadXAML XAML
[7]    win.DataContext←2015⌶'winelist'
[8]    win.Show
     ∇
```

The global variable `Wines` contains a vector of character vectors, each of which is the name of a wine. `Grid[2-4]` creates `winelist`, a vector of namespaces, of the same length, each of which contains two variables c `Name` and `Price`.

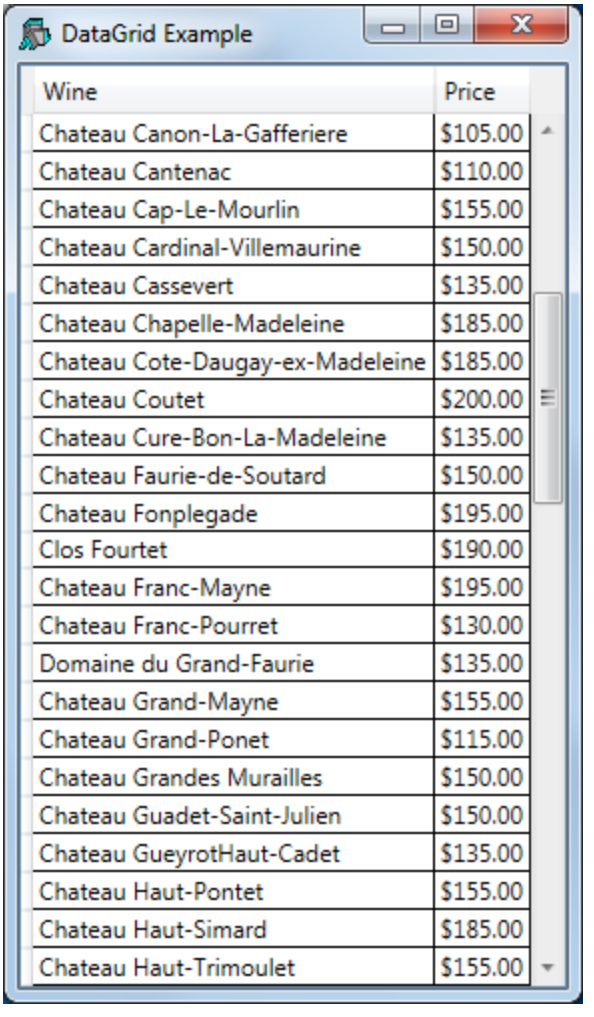
Testing the Data Binding

```
)LOAD WPFIntro
)CS DataBinding.DataGrid
Grid
```



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

```
winelist.Price←5×⌊0.5+winelist.Price÷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

Syncfusion Libraries

Under a licensing agreement with Syncfusion, Dyalog includes the Syncfusion library of WPF controls. These may be used by Dyalog APL users to develop applications, and may be distributed with Dyalog APL run-time applications.

The Syncfusion libraries comprise a set of .NET assemblies which are supplied in the *Syncfusion/4.5* sub-directory of the main Dyalog APL installation directory (for example: *c:\Program Files\Dyalog\Dyalog APL-64 14.0 Unicode\Syncfusion\4.5*.

Requirements

To use the Syncfusion libraries you must be using Microsoft .NET Version 4.5. See *User's Guide: Configuration Dialog:.NET Framework Tab*.

In addition, to use the controls contained in these assemblies it is necessary to perform one or both of the following steps.

Using XAML

If using XAML, the XAML must include the appropriate `xmlns` statements that specify where the Syncfusion controls are to be found. For example:

```
xmlns:syncfusion="clr-namespace:Syncfusion.Windows.Gauge;
                  assembly=Syncfusion.Gauge.WPF"
```

The above statement defines the prefix `syncfusion` to mean the specified Syncfusion namespace and assembly that contains the various Gauge controls. When the prefix `syncfusion` is subsequently used in front of a control in the XAML, the system knows where to find it. For example:

```
<syncfusion:CircularGauge Name="fahrenheit" Margin="10">
```

⎕USING

In common with all .NET types, when a Syncfusion control is loaded using XAML or using ⎕NEW it is essential that the current value of ⎕USING identifies the .NET namespace and assembly in which the control will be found. For example:

```
      ⎕USING,←⊂'Syncfusion.Windows.Gauge,
                Syncfusion/4.5/Syncfusion.Gauge.WPF.dll'
```

This statement tells APL to search the .NET namespace named *Syncfusion.Windows.Gauge*, which is located in the assembly file whose path (relative to the Dyalog installation directory) is `Syncfusion/4.5/Syncfusion.Gauge.WPF.dll`.

Syncfusion Circular Gauge Example

The XAML

Like most Syncfusion controls, the `CircularGauge` is made up of a complex structure of objects, and the XAML (see variable `XAML_SF`) is too extensive to describe in detail herein. It was created from the sample XAML from the Syncfusion documentation for this control entitled *Essential Gauge for WPF*, which may be downloaded from http://help.syncfusion.com/wpf/gauge.

The key statements in the XAML are as follows:

```
xmlns:syncfusion="clr-namespace:Syncfusion.Windows.Gauge;
                  assembly=Syncfusion.Gauge.WPF"
```

The above statement defines the prefix `syncfusion` to mean the specified Syncfusion namespace and assembly. When the prefix `syncfusion` is subsequently used in front of a control in the XAML, the system knows where to find it.

The next two statements define `CircularPointer` controls (the needles on the gauges); one for the Fahrenheit gauge (named *f_pointer*) and one for the Centigrade gauge (named *c_pointer*).

```
<syncfusion:CircularPointer Name="f_pointer" BorderWidth="0.3"
 PointerLength="100" PointerPlacement="Inside" PointerWidth="20"
 Value="32"/>

<syncfusion:CircularPointer Name="c_pointer" BorderWidth="0.3"
 PointerLength="100" PointerPlacement="Inside" PointerWidth="20"
 Value="0"/
```

The APL Code

The following functions were used to produce the example illustrated above. The main function is `SF_TC_XAML`.

```
     ∇ SF_TC_XAML;⎕USING;win;f_pointer;c_pointer;sink
[1]
[2]    win←LoadXAML XAML_SF
[3]
[4]    f_pointer←win.FindName⊂'f_pointer'
[5]    c_pointer←win.FindName⊂'c_pointer'
[6]
[7]    f_pointer.onMouseEnter←'MouseEnter'
[8]    c_pointer.onMouseEnter←'MouseEnter'
[9]
[10]   sink←win.ShowDialog
     ∇
```

After creating the Window from the text in `XAML_SF`, the function `SF_TC_XAML` obtains refs to the two `CircularPointer` controls named *f_pointer* (in the Fahrenheit gauge) and *c_pointer* (in the Centrigrade gauge). It then attaches the `MouseEnter` callback to each of these objects.

```
     ∇ MouseEnter(this ev);ptrs
[1]    ptrs←f_pointer c_pointer
[2]    ptrs.onValueChanged←(ptrs⍳this)⌽0 'TempChanged'
     ∇
```

In this example, the user grabs one of the gauge needles and moves it around the face of the gauge. When the user moves the mouse into one of these needles, the `MouseEnter` callback fires. The function `MouseEnter` receives the `CircularPointer` object that generated the event `this` as the first item in its argument.

The code simply attaches the callback function `TempChanged` to `this`, and disables any callback on the other `CircularPointer` object.

Note that if both `CircularPointer` objects had callbacks on `TempChanged` at the same time, the system would enter a callback loop.

```
     ∇ TempChanged(obj ev)
[1]    :Select obj
[2]    :Case f_pointer
[3]        c_pointer.Value←(obj.Value-32)×5÷9
[4]    :Case c_pointer
[5]        f_pointer.Value←32+obj.Value÷5÷9
[6]    :EndSelect
     ∇
```

The `LoadXAML` function used in this example is subtly different from previous examples.

```
     ∇ win←LoadXAML xaml;⎕USING;str;xml
[1]    ⎕USING←'System.IO'
[2]    ⎕USING,←⊂'System.Windows.Markup'
[3]    ⎕USING,←⊂'System.Xml,system.xml.dll'
[4]    ⎕USING,←⊂'System.Windows.Controls,
                 WPF/PresentationFramework.dll'
[5]    ⎕USING,←⊂'Syncfusion.Windows.Gauge,
                 Syncfusion/4.5/Syncfusion.Gauge.WPF.dll'
[6]    str←⎕NEW StringReader(⊂xaml)
[7]    xml←⎕NEW XmlTextReader str
[8]    win←XamlReader.Load xml
     ∇
```

In particular, it contains the all-important statement:

```
[5]     ⎕USING,←⊂'Syncfusion.Windows.Gauge,
                  Syncfusion/4.5/Syncfusion.Gauge.WPF.dll'
```

This statement tells APL to search the .NET namespace named *Syncfusion.Windows.Gauge*, which is located in the assembly file whose path (relative to the Dyalog installation directory) is `Syncfusion/4.5/Syncfusion.Gauge.WPF.dll`.

Chapter 9:

UNIX Specific Features

Summary

This section summarises the UNIX specific changes in Dyalog APL Version 14.0.

- Previous versions of Dyalog APL for UNIX included the *Dyalog APL for UNIX Installation and User Guide*; in version 14.0 this has been split into the *Dyalog APL for UNIX Installation and Configuration Guide* and the *Dyalog APL for UNIX User Guide*
- The environment variable SKIPBLANKLINES is used to control whether blank lines and comment-only lines are skipped when tracing. If SKIPBLANKLINES=0 then the behaviour in previous versions is retained
- The keystroke CMD-a forces comment alignment
- All shared libraries are now located in $DYALOG/lib
- In preparation for RIDE, ride*.so and lbar.xml are included in the release, and the mapl script has support for enabling RIDE
- To cater for various terminal windows, translate tables xterm-256, screen and screen-256 have been added
- `685⌶` has been removed; should APL terminate, both an aplcore and a core file will be generated, removing the need for this functionality
- The function geterrno has been added to dyalog.so; be aware that the error number returned may have been generated subsequently to the original error condition
- The buildse workspace is now included with UNIX releases; this allows the user to recreate their default session file
- The Key character cannot be entered directly at present (May 2014). See the *Dyalog APL for UNIX Installation and User Guide* for more information.
- From 14.0.22176 onwards, it is possible to close all Editor and Tracer windows using `2023⌶`. This mimics the *Close all Windows* menuitem in versions of Dyalog for Windows.

Index

V

W

X