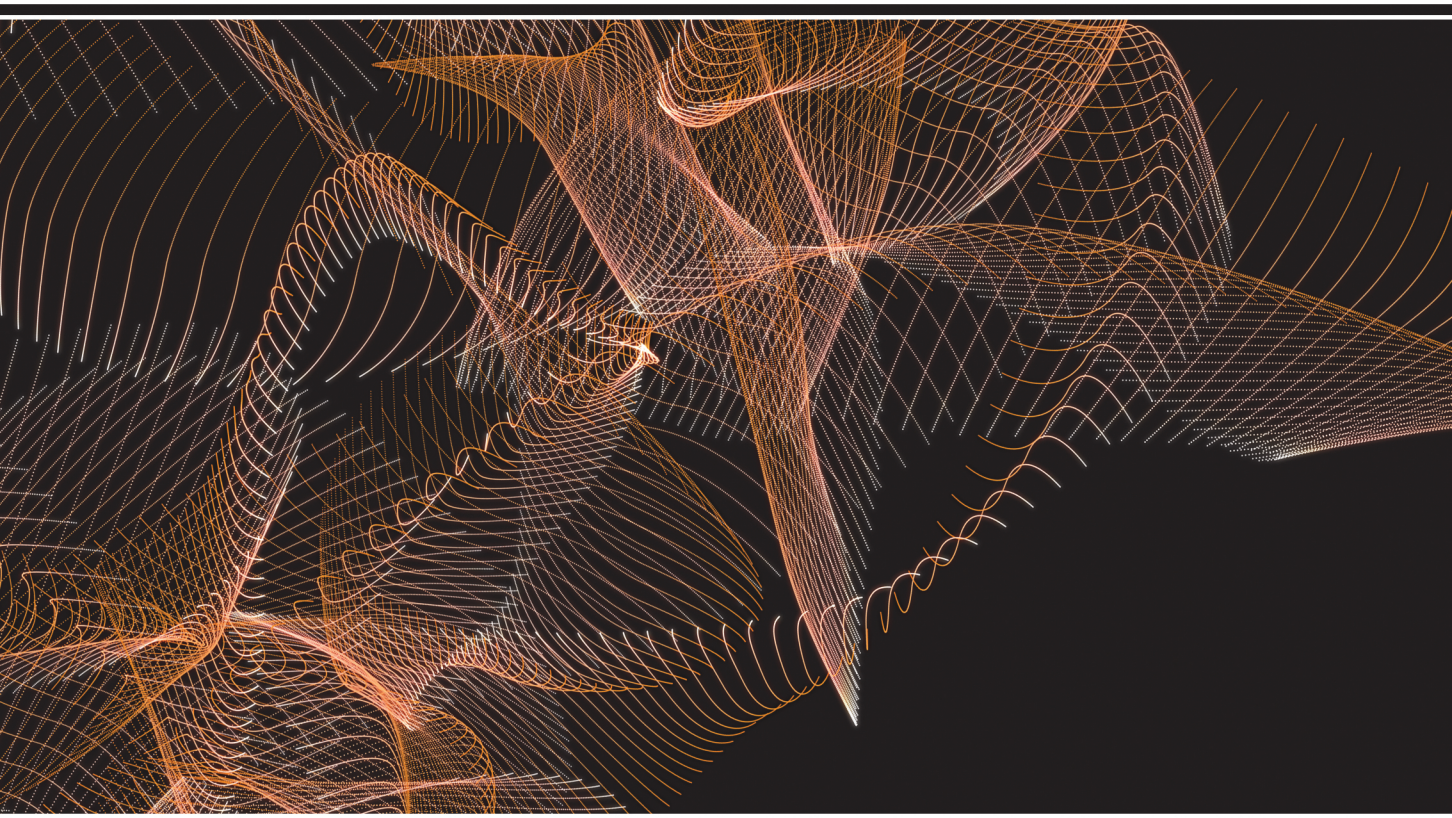
Dyalog
Release Notes

Dyalog version 15.0



DYALOG

The tool of thought for software solutions

Version: 16.0

Revision: 20170303

Please note that unless otherwise stated, all the examples in this document assume that ⎕IO is 1, and ⎕ML is 1.

email: support@dyalog.com
http://www.dyalog.com

Contents

Chapter 1:

Introduction

Key Features

Dyalog APL Version 16.0 provides the following new features, enhancements and changes:

Installation

I t is now possible to install Dyalog without Administrator privileges, although there are a number of consequential limitations. See *Non Admin Installation* on page 6.

Performance Improvements

Version 16.0 includes a considerable amount of research and development work designed to substantially improve speed of execution. See *Performance Improvements* on page 17.

Language Enhancements

New Language Features

- The Random Link system variable `⎕RL` has been extended to specify the random number generator. See *Random Link* on page 51. This enhancement supersedes the use of and replaces the use of (`16807⌶`) which is deprecated and will be removed in the next release.
- Catenate reduction has been extended to operate on empty arrays instead of generating a `DOMAIN ERROR`. See *Reduce* on page 48.

- New system functions are provided to plug some gaps in the native file sub-system. Like the existing native file functions, the new ones are portable across all the operating systems supported by Dyalog, and obviate the need for non-portable solutions (such as the use of the .NET classes for handling directories). See *File Name Parts* on page 65, *Make Directory* on page 56, *Native File Delete* on page 57, *Native File Exists* on page 58, *Native File Information* on page 62, *Read Text File* on page 59 and *Write Text File* on page 67.
- The native file functions `⎕NREAD` and `⎕NREPLACE` have been enhanced for compatibility with APLX. The current file position may now be specified by the value `¯1`.

New I-Beam Features

- A new I-beam function transforms character arrays into lower or upper case and is provided to optimise performance for these operations. See *Case Convert* on page 76.
- A new I-beam function is provided to disassociate data binding from a data-bound variable. See *Remove Data Binding* on page 79.
- Two new I-beams have been added to control when APL threads allocated to incoming .NET calls are re-used or discarded. See *Discard Thread on Exit* on page 81 and *Discard Parked Threads* on page 81.
- A new I-beam function is provided to control the error handling behaviour of `:Trap` and `⎕TRAP`. See *Trap Control* on page 75.
- A new I-beam function creates a hashed array on which dyadic `⍳` and other set functions can be expected to perform faster. See *Hash Array* on page 77.
- The Dyalog compiler remains *experimental* but the I-beam to control it is now documented. See *Compiler Control* on page 72.
- A previously undocumented I-beam is provided to obtain the APL code for methods in classes. See *Canonical Representation* on page 71.
- A new I-beam is provided to identify the precise type of a delegate required by a .NET method or property. See *Create .NET Delegate* on page 80.
- A new I-beam function is provided to assist with inverting matrices. See *Singular Value Decomposition* on page 84.
- Two new I-beams are provided to obtain information concerning files loaded for editing or scripts fixed from file. See *List Loaded Files* on page 1 and *Examples:* on page 1.
- A new I-beam is provided to control connections to the RIDE. See *Manage RIDE Connections* on page 82.

IDE Enhancements

- Native Look and Feeel is now enabled by default for the Dyalog Session and for GUI applications. See *Native Look and Feel* on page 27.
- The Editor has been extended to allow you to edit Dyalog script files and arbitrary text files. See *Editing Scripts and Text Files* on page 30.
- The Editor has a new *Syntax* menu. See *Editor: Syntax Menu* on page 39. The Edit menu has two additional options. See *Editor: Edit Menu* on page 40.
- The Editor now has an option to identify lines of a function which failed to compile. See *Editor: Compiler Errors* on page 41.
- The *Options* menu in the Session provides a new option to Disable suspended `:Trap` blocks or localised `⎕TRAP`s. See *Disable traps in session* on page 38.

GUI Enhancements

- There is a new GetFocusObj method that identifies the GUI object which currently has the input focus. GetFocusObj returns a ref to the object, whereas GetFocus returns its name. See *GetFocusObj* on page 87.
- Native Look and Feel is now enabled by default; whereas previously it was disabled.
- It is now possible to allocate values for the Dispatch ID (DISPID) of exported methods, properties and events of OLEServer and ActiveXControl objects. See *DISPID (Dispatch ID)* on page 86.

Rationalisation of filename extensions

The process of rationalising filename extensions, which was begun in Version 14.1 with the introduction of the **WSEXT** parameter for workspaces, has been extended to component files. See *Component Files without File Extensions* on page 34. This is intended to simplify file naming conventions for cross-platform development.

.NET Interface Enhancements

- The I-beam functions Create Data Binding Source (`2015⌶`) and Update Data Table (`2010⌶`) have been enhanced to handle `DateTime` data more efficiently. In cases where the functions expect DateTime data, it is no longer necessary to create instances of DateTime objects in the workspace. Instead, the data may be represented by 7-element integer vectors (`⎕TS` format) or character strings that can be parsed by the `DateTime.Parse (String)` method. Using either form is typically faster than creating DateTime objects in the workspace, especially for large arrays. This applies only to Create Data Binding Source and Update Date Table. See *Example 6a (Casting to DateTime)* on page 1.
- When a .NET method/property/field etc. returns a null result, it is now represented by `⎕NULL`. In previous versions of Dyalog a null result from a .NET method generates a `VALUE ERROR`. Be aware that this may result in your code returning a `SYNTAX ERROR` rather than a `VALUE ERROR` (as in 14.1 and earlier) in code such as `(⎕NEW Window).Owner.Left`
- The `web.config` file now includes a key `DyalogBinDirectory` to specify the location of the Dyalog .NET interface, the Dyalog DLLs and the Dyalog script compiler. The `web.config` file installed in `samples\asp.net` sets this to the directory that is identified by the **dyalog** parameter. This new key allows different Dyalog ASP.NET applications to use different versions of Dyalog.

Including Script Files in Scripts

- There is a new `:Require` directive that if present **must** precede any code (either a workspace script or a script file). A `:Require` statement instructs Dyalog to load and fix a script file prior to processing the current script. See *Including Script Files in Scripts* on page 42.

Unicode Encodings for □S and □R

The introduction of □NGET and □NPUT has caused some changes to □S and □R so that they match. In particular, the InEnc, OutEnc and Enc options take the same encoding values as □NGET and □NPUT use:

- UTF8 and other UTF encodings are now spelled UTF-8 etc (old spellings are retained for backward compatibility)
- UTF-32 is a new addition
- UTF-16 and UTF-32 may optionally be qualified with BE or LE and assume host endianness when omitted; previously the endianness had to be specified
- All UTF variants can be qualified with -BOM or -NOBOM when neither is specified
- UTF-8 assumes -NOBOM and UTF-16/UTF-32 assume -BOM. Before only -BOM was an option and not specifying was equivalent to -NOBOM. Note that this means that the old UTF16LE and UTF16BE have changed their behaviour: before there would have been no BOM, now there is.
- Windows-1252 "replaces" ANSI (ANSI is a synonym, but Windows-1252 is the preferred name).

File Explorer Integration

Using Unicode Edition, it is now possible to browse workspaces and script (.dyalog) files from the Windows File Explorer. Dyalog presents the contents of workspaces and scripts in the File Explorer preview pane. In addition script files can be opened from File Explorer using the standard Dyalog Editor, changed and re-saved; all without starting Dyalog itself. See *File Explorer Integration* on page 35.

The facility to browse Dyalog script files in a preview pane extends to Microsoft Outlook ,which will use Dyalog to preview such files when received as attachments in emails.

This feature is not supported in Classic Edition.

Non Admin Installation

If you run the `setup.exe` program without Administrator rights, you are offered the choice to continue or to restart the `setup-exe` program with Administrator privileges.

If you install Dyalog without Administrator privileges, the following limitations apply:

- The APL 385 font is not installed. Dyalog will use the correct font, but no other application will have access to it.
- The IME is not installed. Instead the keyboard is defined as a series of Registry entries. This means that the Dyalog keyboard layout can be used only with Dyalog and not with other software.
- No attempt is made to install the VS2015 redistributables globally; they are therefore not available outside the interpreter unless they have been installed beforehand.
- A non-administrator installation is available only to the user who performed the installation; on a shared machine all users would have to perform a non-administrator installation.

Dyalog APL is installed under your AppData directory, for example

c:\users\andys\AppData\Local\Programs\Dyalog\Dyalog APL 15.0 Unicode. Note that the VS2015 redistributables will also be placed in this directory. The registry entries for your version of Dyalog will appear under HKEY_CURRENT_USER\Software\Dyalog.

System Requirements

Microsoft Windows

Dyalog APL Version 16.0 supports versions of Windows from Microsoft Windows Vista up to and including Microsoft Windows 10 and Microsoft Windows Server 2015. Dyalog APL Version 16.0 will not run on earlier versions.

Microsoft .NET Interface

Dyalog APL Version 16.0 .NET Interface requires Version 4.0 or greater of the Microsoft .NET Framework. It does *not* operate with earlier versions of .NET.

For full Data Binding support (including support for the `INotifyCollectionChanged` interface[1]), and Syncfusion, Version 16.0 requires .NET Version 4.5.

The examples provided in the sub-directory `Samples/asp.net` require that IIS is installed. If IIS and ASP.NET are not present, the `asp.net` sub-directory will not be installed during the Dyalog installation.

AIX

For AIX, Version 16.0 requires AIX 6.1 or higher, and a POWER6 chip or higher.

Raspberry Pi

On the Raspberry Pi, Dyalog (32-bit Unicode) supports Raspbian Wheezy and Jessie.

Non-Pi Linux

For non-Pi Linux, Version 16.0 only exists as 64-bit interpreters - there are no 32-bit versions. It is built on RedHat 6, and runs on all recent distributions, including Ubuntu 14.01 and openSUSE 13.2. Contact Dyalog for information about other platforms.

Mac OS X

16.0 requires Mac OS X Yosemite onwards. The target Mac must have been introduced in 2010 or later.

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

Inter-operability

Introduction

Workspaces and component files are stored on disk in a binary format (illegible to text editors). This format differs between machine architectures and among versions of Dyalog. For example, a file component written by a PC may well have an internal format that is different from one written by a UNIX machine. Similarly, a workspace saved from Dyalog Version 16.0 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, but the copy will be fully inter-operable.

The following sections describe other limitations in inter-operability:

Code and ⎕ORs

Code that is saved in workspaces, or embedded within `⎕OR`s stored in component files, can only be read by the Dyalog 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`s 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 that 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.

Null Items (⎕NULL)

`⎕NULL`s created in Version 16.0 can be brought into Versions 13.2, 14.0 and 14.1 provided that the interpreters have been patched to revision 27114 or higher. Attempts to bring `⎕NULL` into earlier versions will fail with a `DOMAIN ERROR`.

Object Representations (⎕OR)

From Version 13.2 onwards, an attempt to `⎕FREAD` a component containing a `⎕OR` that was created by a later version of Dyalog APL will generate `DOMAIN ERROR: Array is from a later version of APL`. This also applies to APL objects passed via Conga or TCPSockets, or objects that have been serialised using `220⌶`.

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; however it is still possible to *read* and *write* small span files. Setting the second item of the right argument of `⎕FCREATE` to anything other than 64 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 are subject to the same restrictions as these files. External variables are unlikely to be developed further; Dyalog recommends that applications which use them should switch to using mapped files or traditional component files. Please 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.

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

Unicode vs. Classic Editions

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

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

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

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

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

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

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

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

AVU changes

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

DECFs and Complex numbers

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

Very large array components

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

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

File Journaling

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

File Component Compression

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

Dyalog 14.0.27143 and 14.1.27143 and later can read compressed components written using 15.0. Attempts to read such components in lower levels of 14.0 and 14.1 will generate `DOMAIN ERROR: Array is from a later version of APL`

Attempting to read a compressed component using versions of Dyalog APL earlier than 14.0 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 and Conga

TCPSockets and Conga can be used to communicate between differing versions of Dyalog APL and are subject to similar limitations to those described above for component files.

Auxiliary Processors

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

Session Files

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

Announcements

Withdrawal of Support for Version 13.2

The supported Versions of Dyalog APL are now Version 15.0, 14.1, and Version 14.0. Version 13.2 and earlier are no longer supported.

Withdrawal of Support for Old Workspaces

Dyalog APL workspaces saved by Version 10.1 (released 2004) and earlier may not be loaded using Version 15.0. Only workspaces saved by Version 11.0 and later may be loaded. Contact Dyalog support if this affects you.

Withdrawal of Support for Windows XP

Version 15.0 is not supported under Windows XP.

Withdrawal of Support for .NET earlier than Version 4.

Version 15.0 requires .NET 4. As no other versions of .NET are supported, the *.NET Framework* tab has been removed from the *Options/Configure* dialog.

dfns workspace now frozen in Classic Editions

Dyalog does not intend to enhance the dfns workspace in Classic Editions any more; only bug fixes will be made. The workspace will continue to be developed for Unicode Editions; the latest version of the dfns workspace is as ever available from *http://dfns.dyalog.com*.

Planned Operating System Requirements for the next version

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

Operating System	**Version**
Microsoft Windows	Vista or Server 2008
AIX	7.1 on POWER 7
Linux	RedHat/Centos 6 or equivalent
OS X	OS X Yosemite 10.10.x

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

Planned Hardware Requirements for next version

The same as Dyalog 15.0.

C32 and C64 Calling Conventions

⎕NA used to support syntax such as `dllname.C32`, which specified that 32-bit calling conventions were to be used.

This has not been necessary for several versions; in Version 15.0 the interpreter checks that if this syntax is present and that it is one of `.C32` or `.C64`, but otherwise ignores it.

Dyalog intends to remove all traces of this syntax from supplied APL code (either in workspaces or in scripts) and it is recommended that users do the same. Dyalog may in future generate an error when this syntax is found, rather than simply ignoring it.

Copies of the `files` namespace and code that manipulates the registry are most likely to contain this syntax.

Default Value for Random Link ⎕RL

The default value for ⎕RL will be changed to `0` in the next major release.

Withdrawal of Random Number Generator I-Beam `16807⌶`

The Random Number Generator I-Beam `16807⌶` is deprecated and will be removed in the next release.

SharpPlot workspace (Classic Edition)

The `sharpplot.dws` workspace is no longer supported in Classic Edition and has been removed from the Classic Edition build.

Array Editor

Version 1.1.0.91 of the Array Editor is included in the Unicode version 15.0 for Windows installation images.

Syncfusion libraries

Version 14.1.0.41 of the Syncfusion libraries are included in the version 15.0 for Windows installation images.

dyalogdata4.5.dll

`dyalogdata4.5.dll` is no longer needed. The bridge dll has been compiled with .NET 4.0 which includes the required functionality.

The use of GR to mean German

For reasons that are over 20 years old, the BuildSE workspace and the input and output translate tables in Classic installations use the abbreviation "GR" to represent "German". In the next version Dyalog intends to replace all such occurrences with "DE".

WSEXT parameter

The default value for **WSEXT** on non-Windows platforms has been changed so that it favours extensions. On AIX and Linux the default **WSEXT** is now `.dws:.DWS:`; on OS X it is `.dws:`.

Trailing directory delimiters

In version 15.0 all environment variables or registry entries which contains a directory or directories have been made consistant in that there are no trailing directory delimeter. This affects the *dyalog* registry entry.

Namespaces and External Variables

From version 15.0 onwards it is no longer possible to assign namespaces and other `⎕NL 9` objects, nor the `⎕OR` of such objects to external variables. Earlier versions could make such assignments, but were apt to lead to APL terminating abnormally, or the external variable on disk not being correctly updated.

Performance Improvements

As part of the ongoing Performance Quality Assurance project [1] , Version 16.0 includes a number of performance improvements as summarised in the following table.

Expression	Factor	Comments
`↑x`	5-12	for elements with the same rank and same shape with possible exception of leading axis
`↓x`	1-1.75	AKA `⊂⍤1⊢x`
`⊂⍤r⊢x`	1-1.75	AKA `⊂[(-r)↑⍳⍴⍴x]x`
`↓[a]x`	1.5-5	
`⍋b`	1-40	
`⍋int8`	1-10	
`x⍋b`	1-300	for `x` having 1 or 2 elements
`x⍋int8`	1-180	for `x` having 2 elements, the first of which is non-zero
`b∘.f x`	1-8	
`b⊥b`	3-∞	equivalent to `+/∧\⌽b`, the number of trailing 1s in `b`
`b(≠⍤⊂)x`	TBA	partition pseudo operator
`b(f/⍤⊂)x`	TBA	partition pseudo operator, for `f` one of `+ × ⌈ ⌊ ∧ ∨ ≠ =`

`>/b`
`≥/b`
`∨/b`
`∧/b`
`</b`
`≤/b`
	1800-87e3	

`∧/b` `∨/b`	1-∞	

[1] http://www.dyalog.com/blog/2016/03/pqa/

Expression	Factor	Comments
`+/b`	1-8	
`,/y`	1-∞	for arguments with arrays with rank > 1
`2 f/ x`	1.2-4	for vector `x` and `f` one of `+ - × ⌈ ⌊ = ≠ < ≤ ≥ = ∨ ∧ ⍲ ⍱`
`n,/[a]x`	1-50	for simple array `x`

`n+/[a]x`
`n≠/[a]b`
`n=/[a]b`
	TBA	

`n⌈/[a]x` `n⌊/[a]x`	TBA	
`x∘.,y`	2-4	
`∧.∧` `∨.∨`	5-∞	for Boolean vector arguments
`x⍳⍤1 0⊢y`	13	
`x⌷⍤r⊢y`	1-370	where the effective left rank is 0
`b[; ;i]` `b[i]`	1-4	Boolean column indexing and Boolean vector indexing
`x[i; ;]`	2-18	row (major cell) indexing AKA `(⊂i)⌷x`AKA `i⌷⍤0 15⊢x`

`+z,`
`-z`
`y+z`
`y-z`
`∘z`
	1-6	complex arguments; greatest speed-ups are only on Linux

`x≡y` `x≢y`	1-8.7	64-bit floats or complex arguments
`x≡y` `x≢y`	1-1.4	non float simple arguments
`x⊥y`	1-26	special case for scalar or vector x
`x⊤y`	1-3.3	special case for integer vector `x` and integer `y`
`f\[a]x`	2-20	`f` is `+ ×` (deferred from 14.0)

Expression	Factor	Comments
`-\[a]x` `÷\[a]x`	1-∞	
`α⍒ω` `α ⎕FMT ω`	varies	E F and I formats, non-DECF data
`⍋p`	10-50	for permutations
`⍒p`	10-50	for permutations
`⍋ω` `⍒ω`	1-1.7	result in smallest datatype
`⍋ω` `⍒ω`	2-20	for small-range arguments
`÷ω`	1-1.1	for non .NET systems only
`≡ω`	2-∞	
`{α,f⌿ω}⌸` `{α(f⌿ω)}⌸`	TBA	for `f` one of `+ ⌈ ⌊ = ≠ ∨ ∧`
`x*b`	1-90	scalar `x` and Boolean `b`
`f⍤r`	1-1.5	general case of rank operator on small arguments
`f⌸`	1-1.8	general case of key operator on small arguments and/or in monadic case
`f.g`	1-1.6	general case of inner product
`f¨`		`f` is any monadic or dyadic scalar function
`≠¨ω`	25	
`f/¨ω`	1-16	`f` one of `+ ⌈ ⌊ ∧ ∨ = ≠`
`α+.×ω`	1-11	Boolean vector `α` and Boolean array `ω`; also +.⋀
`α+.f ω`	4	Boolean array α and Boolean vec `ω`, for `f` one of `× ∧ > < ≠ ∨ ⍱ = ≥ ≤ ⍲`
`⌈/ω` `⌊/ω`	1-29	for non-vector `ω`
`+/x`	6-35	for integer `x`
`0∈b` `1∈b`	1-400	

Expression	Factor	Comments
`⌽[a]x`	1-3.7	
`⍋b` `{ω[⍋ω;]}b`	1-7	also `⍒` instead of `⍋`; for boolean `b` with a multiple of 8 columns
`⍋b` `{ω[⍋ω;]}b`	1-5	also `⍒` instead of `⍋`; for boolean `b` with 1-element major cells
`⍋b` `{ω[⍋ω;]}b`	5-27	also `⍒` instead of `⍋`; for boolean `b` with NOT a multiple of 8 columns
`?m⍴n`	1.1-1.4	result is smallest datatype based on `n` >
`x∧.=1`	1-100	also `1∧.=x, x∨.≠1, 1∨.≠x`; also 0 instead of 1
`⍉bm`	1.2	POWER8 only, transposing a matrix with multiple of 8 rows and columns
`y∈x`	3-17	where `x` and/or `y` have 1- or 2-byte items
`y~x`	3-12	where `x` and/or `y` have 1- or 2-byte items
`y∩x`	3-12	where `x` and/or `y` have 1- or 2-byte items
`x∪y`	3-13	where `x` and/or `y` have 1- or 2-byte items
`∪x`	3-16	where `x` has 1- or 2-byte items

New Idiom

The following new idiom is recognised:

Expression	Description
`XA↓⍨←NS`	This idiom applies only when `NS` is negative, when it removes the last `-NS` items from `XA` along its leading axis. For example, if `NS` is `¯3` then the idiom removes the last `-¯3` (i.e. 3) items.

Bug Fixes

A number of bug fixes implemented in Version 16.0 may change the way that existing code operates and are therefore documented in this section.

011659: Modified, selective assignment returns wrongly enclosed result

In 14.1 and earlier the interpreter incorrectly enclosed the result of a modified, selective assignment. This has been fixed in 15.0.

15.0 behaviour:

```
      var←⊂'Dyalog'
      'APL'≡(⊃var),←'APL'
1
      var
 DyalogAPL
```

14.1 and earlier behaviour:

```
      var←⊂'Dyalog'
      'APL'≡(⊃var),←'APL'
0
      var
 DyalogAPL
```

Change to Component File Names

There were exceptions to the following rules which have been corrected:

- The file name returned by ⎕FNAMES should be precisely the same as that which was specified when the file was tied.
- The left argument to ⎕FERASE should be identical to the name used to tie the file.

Examples of errors that have been fixed in Version 15.0

```
      t←'a.DCF' ⎕TIE 0 ⋄ ⎕FNAMES
a     A  WAS INCORRECT, SHOULD BE a.DCF

      t←'a' ⎕FTIE 0 ⋄ 'a.DCF' ⎕FERASE t
A INCORRECTLY DELETED THE FILE

      t←'a.DCF' ⎕FTIE 0 ⋄ 'a' ⎕FERASE t
A INCORRECTLY DELETED THE FILE
```

Length of left argument of ⎕WC limited to 2*15

The maximum length of the left argument of □WC is now limited to 2*15 in order to eliminate a number of system crashes.

Change to User-Defined Event Message

Previously, if you created a GUI object using □NEW to which you attached a user-defined event, the first element of the event message (that identifies the object) reported for that event was a generic character vector. It is now a ref.

```
      f←□NEW ⊂'Form'
      f.on999←'foo'

     ∇ foo m
[1]    m
     ∇

      □NQ f 999
 #.[Form]  999
```

Correction to ⎕FIX

⎕FIX now handles the inheritance of system variables by sub-namespaces correctly.

Example:

```
      script←':Namespace ML2' '⎕ML←2'
      script,←':Namespace NOML' ':EndNamespace'
      script,←⊂ ':EndNamespace'
      ↑script
:Namespace ML2
⎕ML←2
:Namespace NOML
:EndNamespace
:EndNamespace

      ⎕ML←1
      ⎕FIX script
      ML2.NOML.⎕ML
2
```

In Version 14.1, `ML2.NOML.⎕ML` is 1.

dfns no longer accept 3 or more colons in statements

In version 14.1 and earlier expressions such as

```
      {1::::3}
```

were treated as valid. In version 15.0 such expressions will generate a `SYNTAX ERROR`.

Change to behaviour of selective modified assignment

In version 14.1 and earlier the result of some selective modified assignments was incorrectly nested. This has been fixed in 15.0. For example in 14.1:

```
      text←⊂'Dyalog '
      'APL'≡(⊃text),←'APL'
0
```

In version 15.0 the result is correctly not enclosed:

```
      text←⊂'Dyalog '
      'APL'≡(⊃text),←'APL'
1
```

Comparisons involving the prototype of Null Item

In version 14.1 and earlier expressions such as

```
(0⍴⊂NULL)≡0⍴⊂NULL
```

or

```
⊂null≡⍬0⍴⊂null
```

incorrectly generated `NONCE ERROR`s. In version 15.0 onwards the above expressions both return 1.

Note that comparisons involving the prototype of `#` or `⎕SE` will still generate a `NONCE ERROR`.

Classic Edition: `'UTF-8'⎕UCS` errors

In version 14.1 and earlier in Classic interpreters an attempt to convert an invalid UTF-8 sequence into a Unicode code point resulted in a `TRANSLATION ERROR`. In 15.0 onwards a `TRANSLATION ERROR` is only generated if the resulting (valid) Unicode code point is not in `⎕AVU`. Attempts to convert invalid UTF-8 sequences generate `DOMAIN ERROR`s; the same behaviour as in Unicode interpreters.

Debug modifier for user commands

Having run `]udebug on` to enable debugging user commands, the user command **must** be called with "-" as the last modifier. That is, the "-" must be preceded by at least one space. If there is no space the "-" is assumed to be part of the preceding modifier or flag. In previous versions of SALT the "-" was treated as an instruction to debug the user command whether or not it is preceded by a space.

⎕SI and ⎕XSI can return the wrong name when an inner dfn is called via an operator

In earlier versions of Dyalog ⎕SI and ⎕XSI could return the wrong name when an inner dfn is called via an operator; ⎕LC would however report the correct line numbers. This has been fixed in Version 15.0.

Example

```
      ∇r←hoo;foo;goo;dop
       dop←{
           r←1
           +⍺⍺ ⍵
       }
       foo←{
       ⍝ comment
       ⍝ comment1
       ⍝ comment2
       ⍝ comment3
       ⍝ comment4
           goo←{
               +{
                       ⎕XSI,¨⎕LC
               }⍵
           }
           +goo dop ⍵
       }
       r←foo 1
      ∇
       hoo ⍝ Version 15.0
#.foo 8  #.foo 7  #.dop 2  #.foo 11  #.hoo8 18
       hoo ⍝ Version 14.1 and prior. Wrong.
#.goo 8  #.goo 7  #.dop 2  #.foo 11  #.hoo 18
```

Chapter 2: Miscellaneous

Native Look and Feel

Native Look and Feel is a Dyalog option that affects the appearance of the controls provided by the Dyalog GUI Interface and those used by the Dyalog Session. It is implemented by the **XPLookAndFeel** parameter.

Most of the Dyalog controls (with the notable exception of the Dyalog Grid) are standard Windows user-interface components provided by the Windows Common Controls library `comctl32.dll`. Successive versions of Windows have introduced new versions of the Windows Common Control Library which typically provide additional features as well as certain differences in appearance. However, each version of Windows continues to support older versions of the Common Control Library as well as the latest one. The decision as to which is loaded is made at run-time.

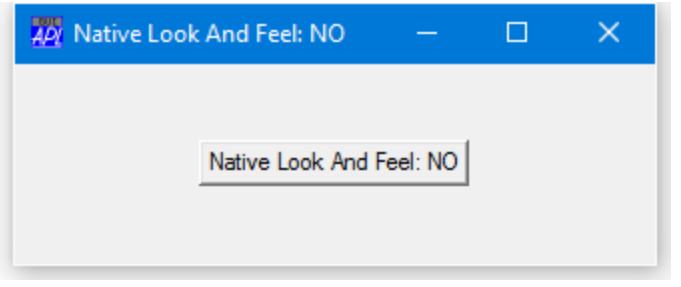
A second factor that affects the appearance of user-interface controls is the application of *Visual Styles*[1] and Themes. These features enable users to tailor the UI to accommodate their individual needs and preferences. From Windows 8 onwards, the default appearance of certain Common Controls is overridden by the Visual Styles in use. However, this applies only if *Native Look and Feel* is enabled.

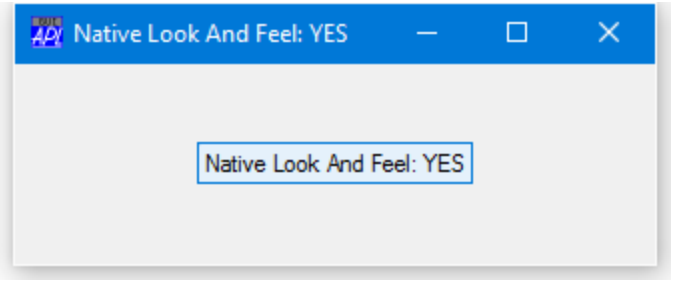
If *Native Look and Feel* is enabled, Windows loads the latest version of `comctl32.dll` (and potentially other Windows dlls) that is appropriate for the version of Windows in use. If *Native Look and Feel* is disabled, an earlier version may be loaded. The specific version that is loaded is not determined by Dyalog, but by Windows.

The Dyalog for Microsoft Windows Object Reference Guide identifies which features require *Native Look and Feel* to be enabled. It documents the typical appearance of controls with *Native Look and Feel* disabled but does not specify how the appearance of controls is affected by enabling *Native Look and Feel*, which is in any case affected by the Visual Styles selected by the user.

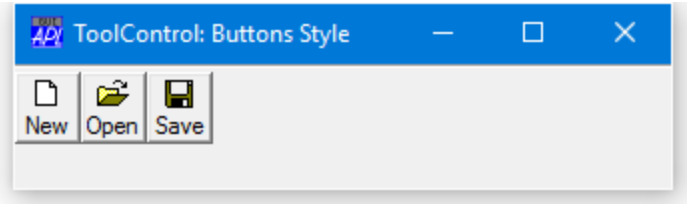
[1] See msdn.microsoft.com for details.

The following pictures illustrate the appearance of a simple Button created with and without *Native Look and Feel* under Windows 10.

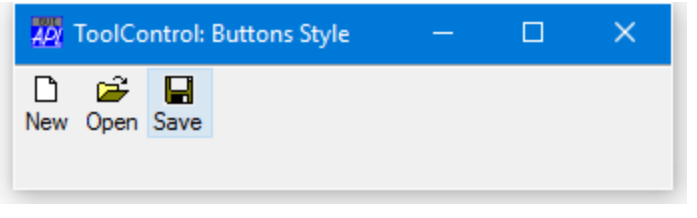




The next shows the appearance of ToolButttons with Style `Buttons` under Windows 10 with *Native Look and Feel* disabled. The appearance is the same in earlier versions of Windows.



When *Native Look and Feel* is enabled, the buttons become transparent and Windows applies the Visual Styles associated with the current theme. This in turn implies that with *Native Look and Feel* enabled the Styles `Button` and `FlatButton` have the same appearance.

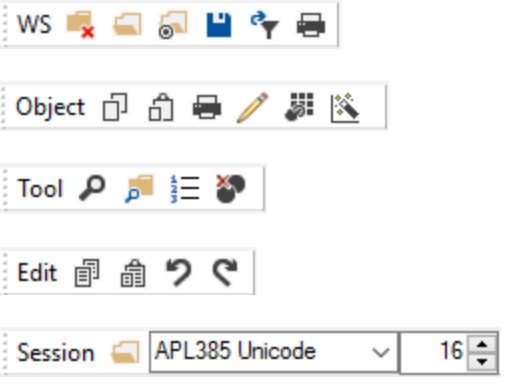


Native Look and Feel and the Dyalog Session

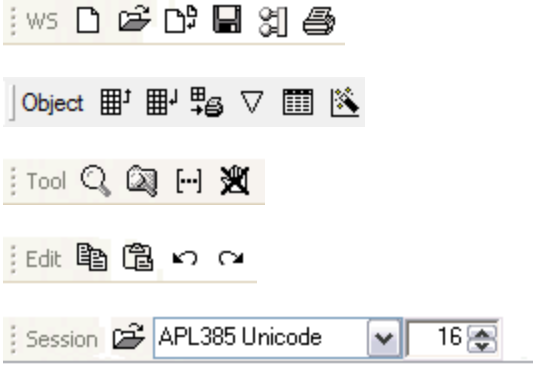
During development, both the Dyalog Session and the Dyalog APL GUI will display native style buttons, combo boxes, and other GUI components if *Native Look and Feel* is enabled. The option is provided in the *General* tab of the *Configuration* dialog and in Version 15.0 onwards is enabled by default.

Furthermore, from Version 14.1 onwards, the appearance of the Session tool buttons on the Session, Editor, Tracer and other windows is different according to whether or not *Native Look and Feel* is enabled.

Native Look and Feel Enabled



Native Look and Feel Disabled



Editing Scripts and Text Files

The Editor may also be used to edit Dyalog script files (`.dyalog` files) and general text files.

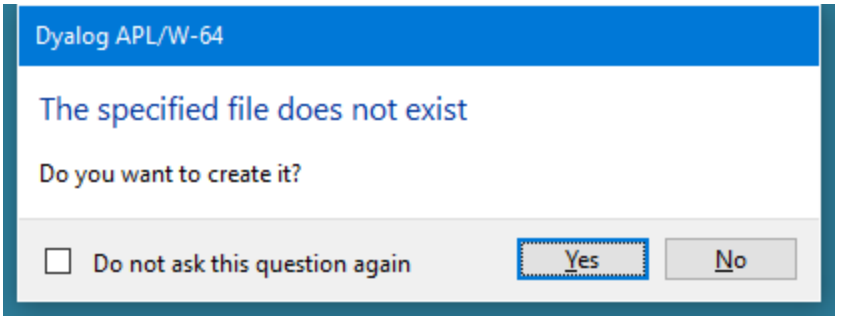
There are two ways to choose the file to be edited. If the file exists, you can select it from the *Open source file* dialog by clicking *File/Edit Text File* from the Session menu bar.

Alternatively, type `)ED` followed by the pathname to the file. To identify the name given as a file, it must either contain a slash character ("\" or "/") or be preceded by one.

Examples

```
)ED c:\Dyalog15.0\myscript.dyalog
)ED c:\Dyalog15.0\pete.txt
)ED /x.txt  ⍝ x.txt in current directory
)ed / x.txt ⍝ ditto
)ed / y     ⍝ y in current directory
```

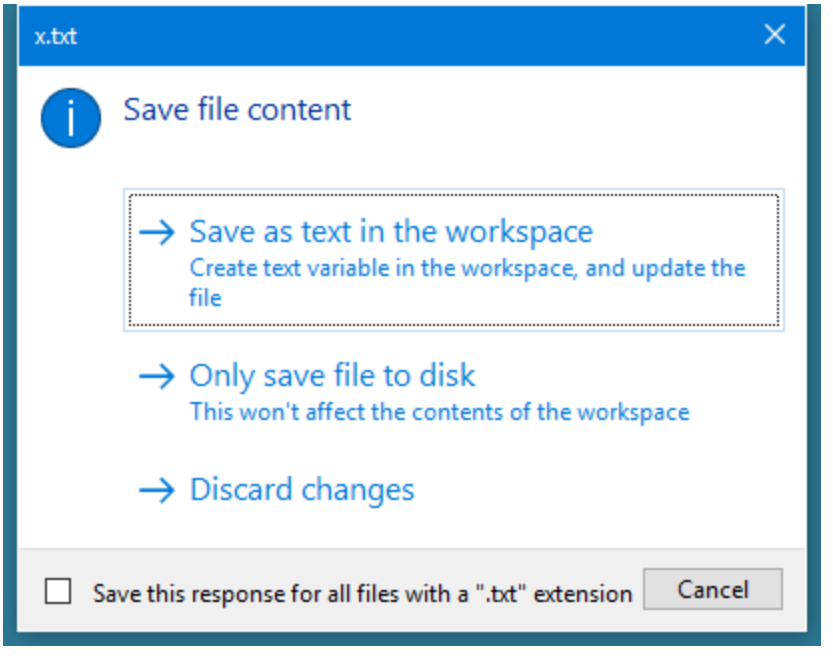
If the named file does not exist, you will be asked whether or not you want to create it:



If you edit a Dyalog script file, the editor will treat it as such and provide the same formatting and syntax colouring as if it were a script in the workspace.

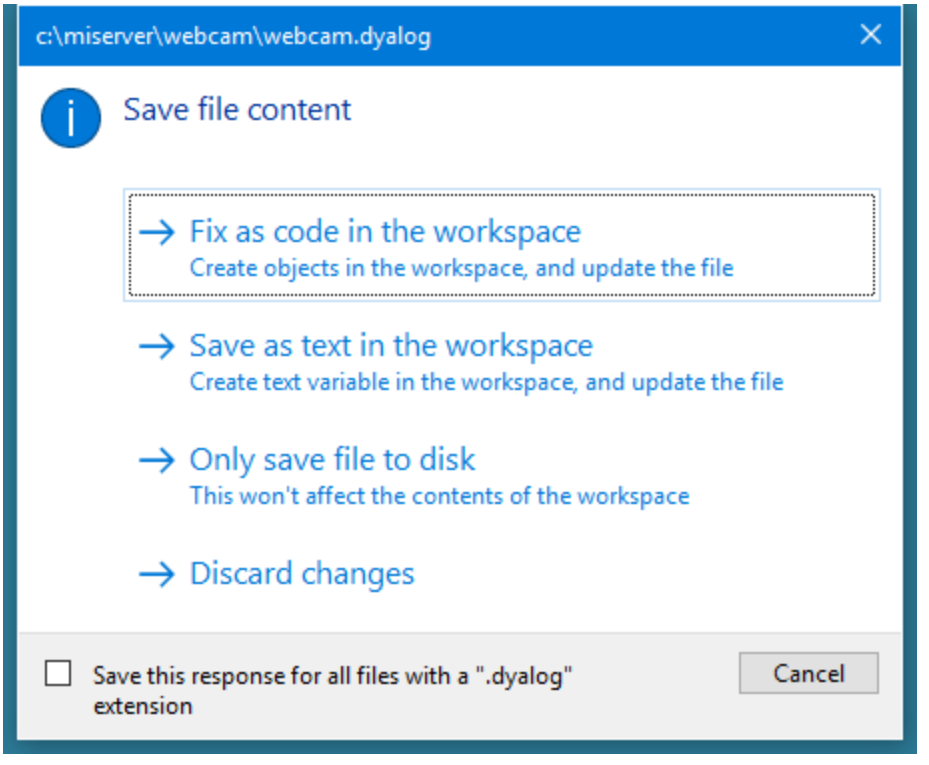
Otherwise, the file will be edited as if it were a character vector with embedded new-lines.

When you exit the editor with *Exit and fix*, you will be offered a number of alternatives depending upon the type of file, as shown below.



Saving a Text file.

Note that if you choose *Save as text in the workspace*, information about the file and the text variable associated with it is retained in the workspace. This information may be obtained using `5176I` and `5177I`. See *Language Reference Guide: List Loaded Files* and *List Loaded File Objects*.



Saving a Script file.

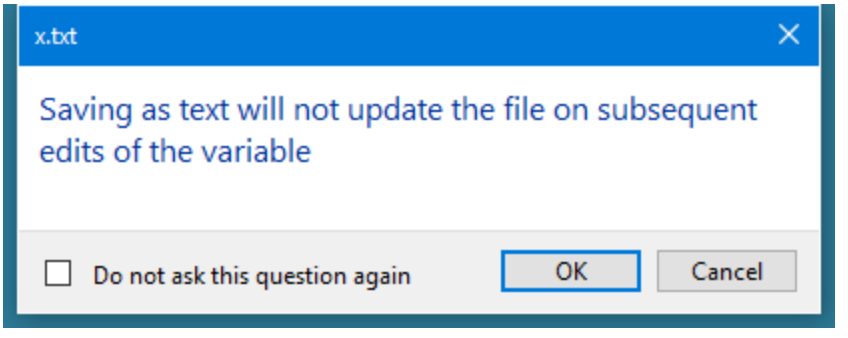
Note that if you choose *Fix as code in the workspace* or *Save as text in the workspace*, information about the file and the text variable associated with it is retained in the workspace. This information may be obtained using `5176⌶` and `5177⌶`. See *Language Reference Guide: List Loaded Files* and *List Loaded File Objects*.

Fix as code in the workspace

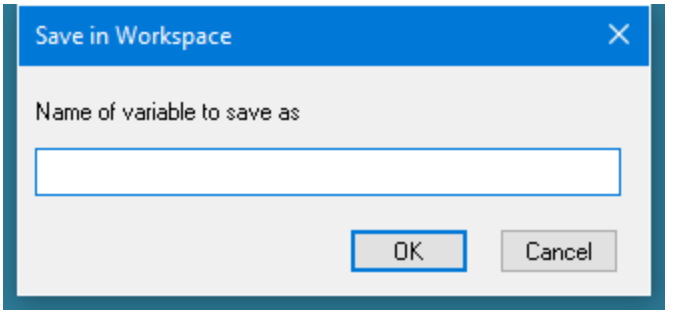
If you choose this option, the file will be updated and the script will also be fixed in the workspace. Note that if the script refers to a base class or other external elements, it cannot be fixed unless these elements are also present in the workspace.

Save as text in the workspace

If you choose this option, the file will be updated and the contents of the file will also be saved to a variable in the workspace. First you will see the following warning dialog, which may be disabled subsequently by checking *Do not ask this question again*.



Then you will be prompted to supply its name, which may be a new name or the name of an existing variable:



Only save file to disk

If you choose this option, the file will be updated but nothing will be changed in the workspace.

Discard changes

If you choose this option, all changes will be discarded and nothing saved.

Component Files without File Extensions

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

This parameter specifies component file filename extensions.

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

If undefined, CFEXT defaults to `.dcf:` on Windows and OS X, and `.dcf:.DCF:` on all other platforms.

Creating a Component File

If the new file name specified for ?FCREATE, ?FCOPY or ?FRENAME contains no extension, the first extension in **CFEXT** will be added.

Tieing a Component File

If the component file name specified for ?FTIE, ?FSTIE or ?FCHK does not end with a file extension, the corresponding file is located by appending the specified file name with each extension in **CFEXT** in turn until the file is found.

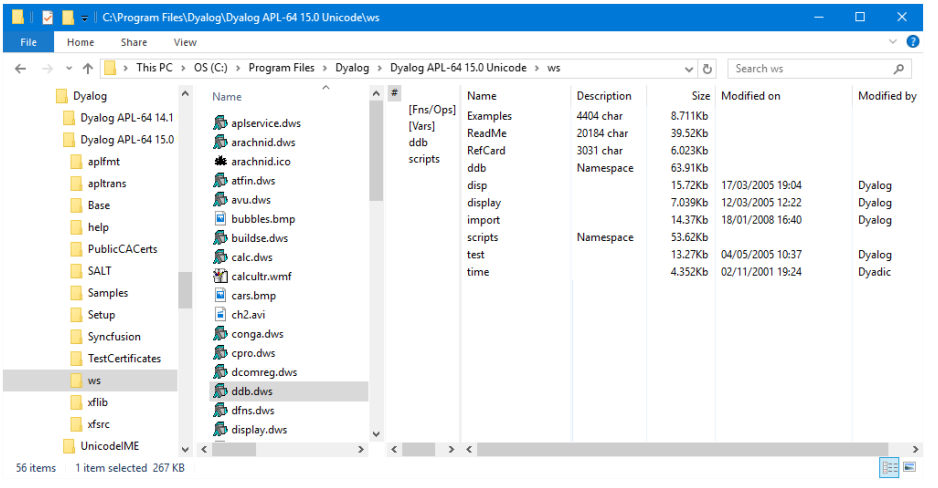
Consequential Change to ?FLIB

As a result of the implementation of **CFEXT**, ?FLIB now reports all file extensions. Previously, ?FLIB removed "standard" file extensions .dcf and .DCF.

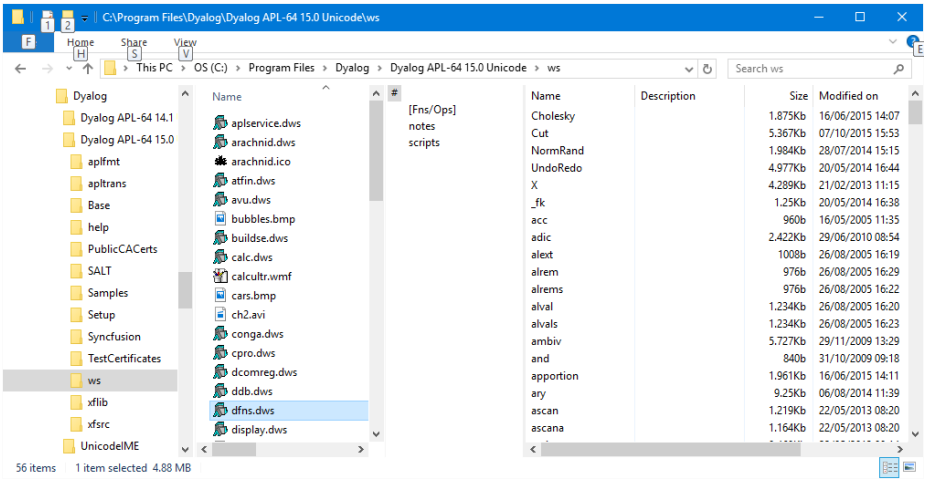
File Explorer Integration

Unicode Edition Only

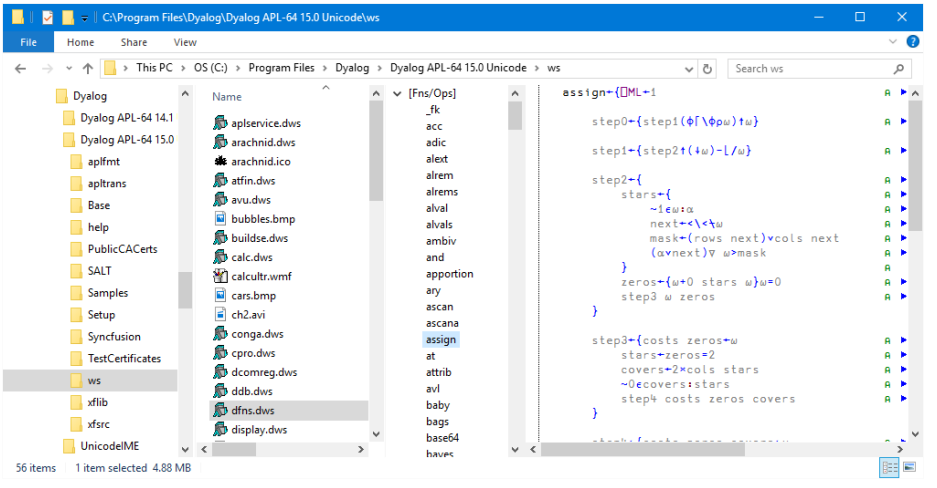
You can browse the contents of workspaces and Dyalog script files using the preview pane of Windows File Explorer. The following example show what you see in the preview pane when you select the supplied workspace `ddb.dws`.



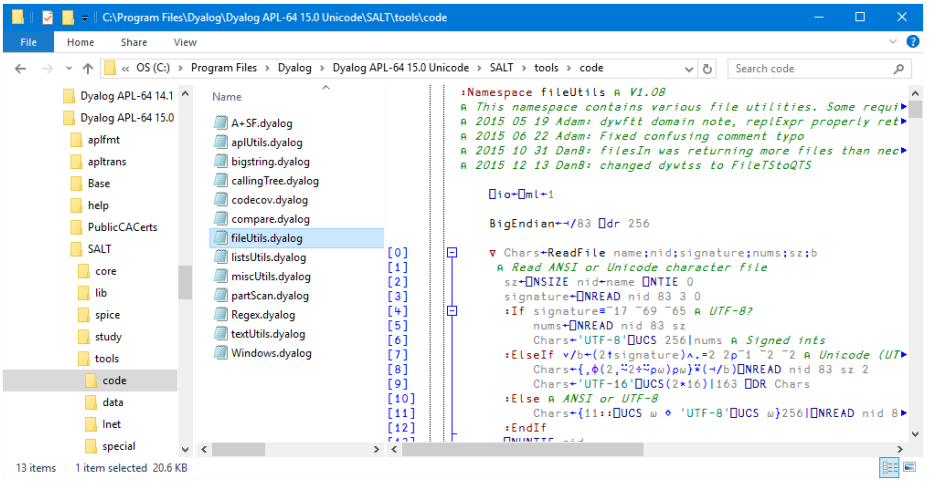
When you move the cursor to the next workspace in the list, `dfns.dws`, the preview pane is immediately updated to show its contents.



If you open the Fns/Ops node and click on a function name, the function is displayed. The next picture shows the function `assign`.



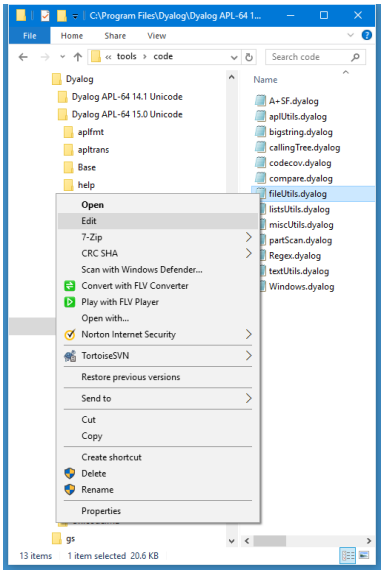
You can also browse Dyalog script files. The following picture shows what you see when you select the `fileUtils.dyalog` file.



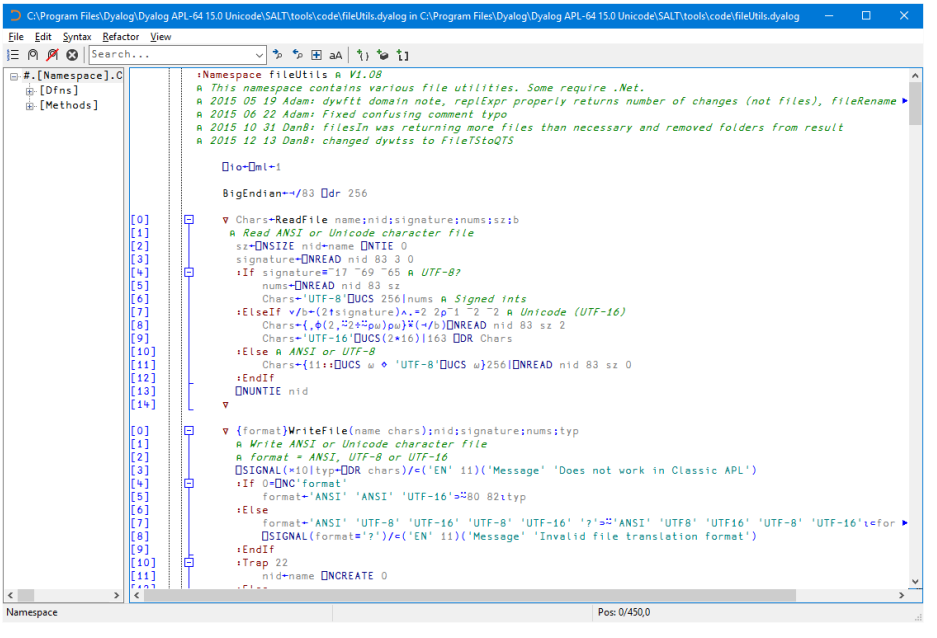
Note that you may only *view* workspace objects and scripts in the preview pane, it is not possible to *edit* them in the preview pane.

Editing Dyalog Scripts

You may edit a script file from File Explorer by first selecting the script file and then choosing *Edit* from the File Explorer context menu.



This brings up the standard Dyalog Editor, in a stand-alone window, just as it would appear if undocked from the Session, as shown in the next picture.

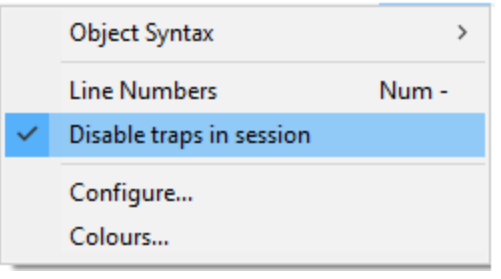


IDE Enhancements

Disable traps in session

There is a new menu item on the *Options* menu labelled *Disable traps in session*. This is designed to prevent undesirable behaviour when an expression executed in the session inadvertently triggers a suspended trap expression.

This option has no effect on the behaviour of error-guards in dfns.



If there is a suspension on the stack and the suspended code is within a `:Trap` block or contains a localised `⎕TRAP`, this option disables the trap.

Example

```
     ∇ r←Recip x
[1]    :Trap 11
[2]        r←÷x
[3]    :Else
[4]        r←1
[5]    :EndTrap
     ∇

      Recip 3
0.3333333333
      Recip 0
1
      2 ⎕STOP 'Recip'
      Recip 0

Recip[2]
      )SI
#.Recip[2]*
```

Now there is a suspended `:Trap` on the stack set to fire when a `DOMAIN ERROR` (error number 11) is generated.

This can lead to undesirable behaviour if an expression executed directly in the Session, or within a function executed from the Session, causes a `DOMAIN ERROR`.

In Versions prior to Version 15.0, or if the new option to *Disable traps in the session* is OFF, any expression that generates a `DOMAIN ERROR` will invoke the suspended `:TRAP`. In this case, that will cause the suspended function to execute the `:Else` clause and return the value 0.

```
      'a'+2
1
```

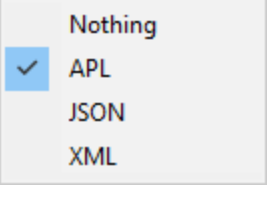
However, if the option to *Disable traps in the session* is ON, the system will ignore the `:Trap` and the `DOMAIN ERROR` will be handled as if it were not there:

```
      'a'+0
DOMAIN ERROR
      'a'+2
     ^
```

Note that this behaviour can also be controlled by calling `600⌶`. See *Trap Control* on page 75.

Editor: Syntax Menu

There is a new *Syntax Menu* for the Editor.

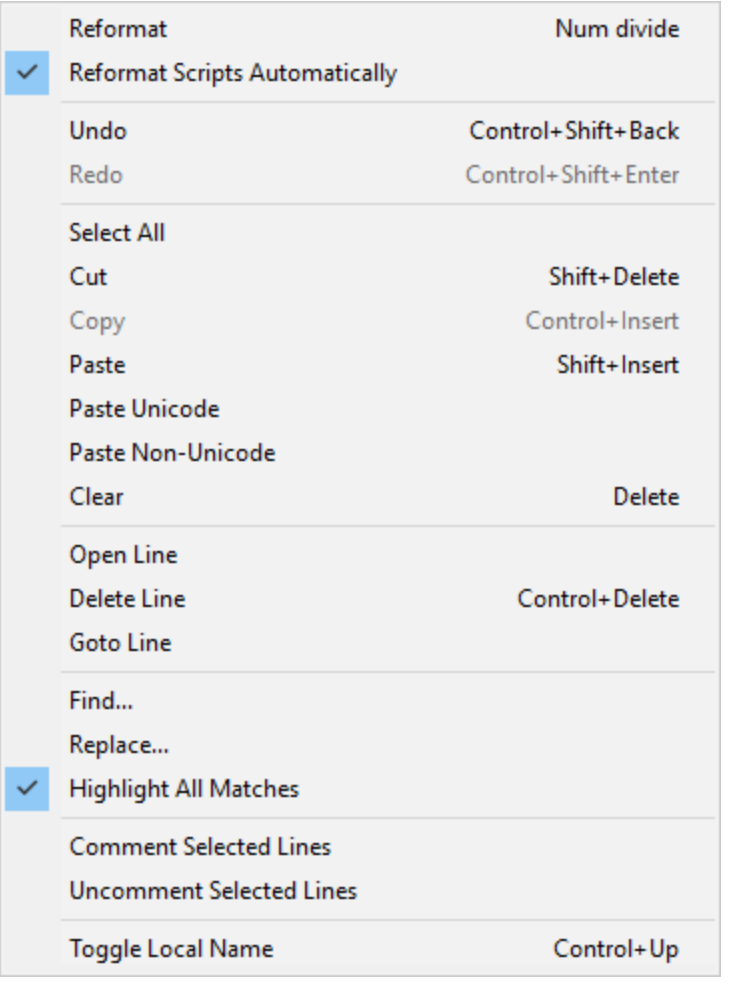


The *Syntax* menu illustrated above provides options to specify how the data displayed in the Editor window is to be syntax coloured. For workspace objects, the default is *APL* for functions and operators, and *Nothing* for variables.

Item	Syntax Colour as
Nothing	Variable
APL	Function
JSON	JSON array
XML	XML array

Editor: Edit Menu

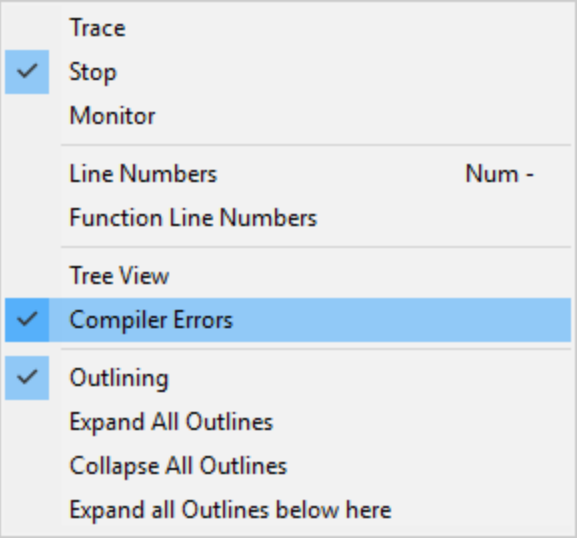
There are two new options on the Edit menu of the Editor.



Item	Description
Reformat Scripts Automatically	If checked, the Editor will automatically reformat a Dyalog script when it loads it.
Highlight All Matches	If checked, all strings in the object being edited that match the search string are highlighted. The highted items change dynamically as the seacrh string is entered or changed.

Editor: Compiler Errors

The *View Menu* of the Editor now has an option *Compiler Errors* which, if enabled, identifies which lines of a function failed to compile.



Aligning Comments

Comments which are preceded only by white space (i.e. in lines which contain no executable code) are ignored (remain unaltered) when comments are aligned, unless only such lines are selected.

Hitting <F1> immediately to the right of ⎕ or :

In earlier versions on Windows, hitting <F1> on or immediately to the right of a ⎕ or : result in the help system being opened for ⎕ or : respectively. In version 15.0 the help system will be opened for the system object or control structure respectively.

Altering the hint on SysTray icons

"SysTray" has been added to the list of Window Names for which the caption can be set. This defines the hint that appears when hovering over Dyalog icons in the System Tray, and can be used to identify more easily which icon is associated with which Dyalog process. Dyalog suggests including "{PID}" to the caption of both the SysTray and the Session. See *Installation & Configuration Guide: Window Captions.*

Including Script Files in Scripts

A Class or Namespace script in the workspace or in a script file may specify that other script files are to be loaded prior to the fixing of the script itself. To do so, it must begin with one or more `:Require` statements, with the following syntax:

```
:Require file://[path]/file
```

If no `path` is specified, the path is taken to be relative to the current script file or, if in a workspace script, the current working directory. Note that a leading `'./'` or `'.\'` in `path` is not allowed, to avoid any potential confusion with "current directory".

`:Require` is a directive to the Editor (more specifically, to the internal mechanism that fixes a script as an object in the workspace) and can appear in any script containing APL code, but **must** precede all code in the script. `:Require` is thus not valid within a function, class, namespace or any other definition.

The prefix `file://` allows for the possibility of a future extension of `http://` and `ftp://`.

In version 16.0 `⎕:require` is a synonym for `:Require`. This allows the user to create scripts which can be used in multiple versions of Dyalog; in 14.1 and earlier SALT parses `⎕:require` statements and loads the appropriate files, in 16.0 it is the interpreter loads the file named in `⎕:require` statements. Dyalog intends to remove support for the `⎕:require` statement from the interpreter in a future version. Note that unlike `:Require`, `⎕:require` can appear within code.

DateTime Enhancement

To illustrate the extension for `DateTime` objects, the following example has been added to the section on *Data Binding* in the chapter on *Windows Presentation Foundation* in the *.NET Interface Guide:* .

Example 6a (Casting to DateTime)

This example is similar to Example 6 but illustrates how numeric data in `⎕TS` format can be converted to `DateTime` type.

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/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DateTimes using ⎕TS data"
    SizeToContent="WidthAndHeight" >
    <StackPanel>
         <TextBlock Text="Some High Tides at Portsmouth, England"
          FontSize="18" Margin="5"/>
         <ListBox Name="TideTimes" Height="200"
          Margin="5" />
    </StackPanel>
</Window>
```

The APL Code

The function `Tides` is shown below.

```
     ∇ Tides;⎕USING;win;dt;Highs
[1]    ⎕USING←'System'
[2]    win←LoadXAML XAML_Tides
[3]    win.times←win.FindName⊂'TideTimes'
[4]    Highs←(⊂2016 2 18),¨(7 9)(8 44)(19 47)(21 47)
[5]    Highs,←(⊂2016 2 19),¨(8 17)(10 12)(20 51)(22 51)
[6]    dt←7↑¨Highs
[7]    win.times.ItemsSource←DateTime(2015⌶)'dt'
[8]    sink←win.ShowDialog
     ∇
```

`Tides[3]` uses `FindName` to obtain a ref to the `ListBox` (defined in the XAML) named *TideTimes*:

```
[3]     win.times←win.FindName⊂'TideTimes'
```

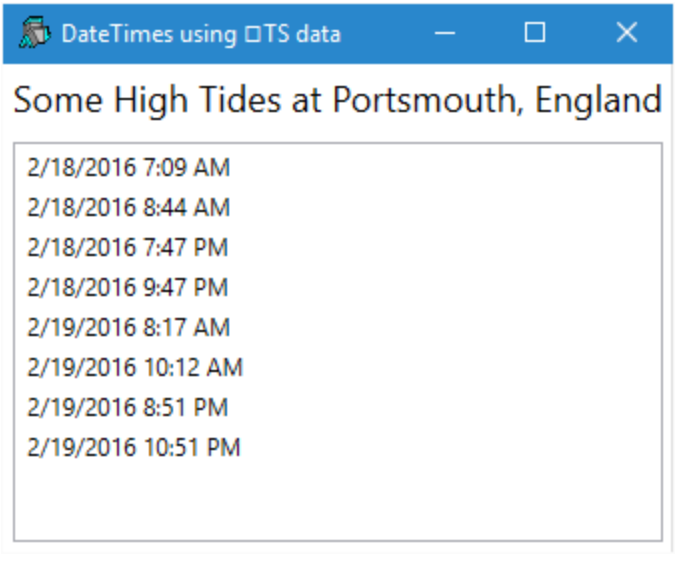
`Tides[4-5]` creates a vector of integer vectors each of which species the time and date of a high tide at Portsmouth. `Tides[6]` extends each to 7-elements, which is required to represent a `DateTime` object.

Then, `Tides[7]` creates a binding source object from this array and assigns it to the `ItemsSource` property of the `ListBox`. Note that the left argument `DateTime` specifies that the data be cast to that type.

```
[7]     win.times.ItemsSource←DateTime(2015⌶)'dt'
```

Testing the Data Binding

```
      )LOAD wpfintro
      DataBinding.NetObjects.Tides
```



Other Changes

Changes to ⎕R and ⎕S

The PCRE engine used to support ⎕R and ⎕S has been upgraded from version 8.12 to 8.37 and includes a number of bug fixes in the engine. The full list of changes to the engine is at: http://pcre.org/original/changelog.txt.

The options **InEnc** and **OutEnc** have been changed to provide compatibility with ⎕NGET and ⎕NPUT.

⎕R and ⎕S now support the specification of characters in transformation strings using the syntax \x{nnnn}, i.e.

\x{*nnnn*}	represents a Unicode code point; *nnnn* is a hexadecimal sequence of characters yielding a value between 0x1 and 0x0FFFFFFF.

Change to)SI

As a consequence of performance improvements, certain primitive operators including Commute, Axis and some cases of Outer Product are no longer reported by)SI and ⎕STACK.

Version 15.0

```
      2 foo¨ 3
SYNTAX ERROR
foo[2] ∘
       ^
      )SI
#.foo[2]*
      ⎕STACK
      *
 ∇foo
      *
```

Version 14.1 and Prior

```
      2 foo¨ 3
SYNTAX ERROR
foo[2] ∘
       ^
      )SI
#.foo[2]*
¨
      ⎕STACK
      *
 ∇foo
      ¨
      *
```

Chapter 3:

Language Reference Changes

Language Changes

The following table summarises the main changes to language features in Version 15.0.

Function	Descritpion	Change
R←f/[K]Y	Reduce	Enhanced
□RL	Random Link	Enhanced
□MKDIR	Make Directory	NEW
□NDELETE	Native File Delete	NEW
□NEXISTS	Native File Exists	NEW
□NGET	Read Text File	NEW
□NINFO	Native File Information	NEW
□NPARTS	File Name Parts	NEW
□NPUT	Write Text File	NEW

Reduce R←f/[K]Y

`f` must be a dyadic function. `Y` may be any array whose items in the sub-arrays along the `K`th axis are appropriate to function `f`.

The axis specification is optional. If present, `K` must identify an axis of `Y`. If absent, the last axis of `Y` is implied. The form `R←f⌿Y` implies the first axis of `Y`.

`R` is an array formed by applying function `f` between items of the vectors along the `K`th (or implied) axis of `Y`. For a typical vector `Y`, the result `R` is:

```
R  ↔  ⊂(1⊃Y)f(2⊃Y)f.....f(n⊃Y)
```

The shape `S` of `R` is the shape of `Y` excluding the `K`th axis, i.e.

```
S  ↔  ⍴R  ↔  (K≠⍳⍴⍴Y)/⍴Y
```

If `Y` is a scalar then for any scalar `f`, `R` is `Y`.

If the length of the `K`th axis of `Y` is 1, or if the length of any other axis of `Y` is 0, then `f` is not applied and `R` is `S⍴Y`.

Otherwise, if the length of the `K`th axis is 0 then the result depends on `f` and on `⊃Y` (the prototypical item of `Y`) as follows:

> If `f` is one of the functions listed in the table below then `R` is `S⍴⊂I`, where `I` is formed from `⊃Y` by replacing each depth-zero item of `⊃Y` with the identity item from the table.
>
> Otherwise, if `f` is Catenate along the `J`th (specified or implied) axis, `R` is `S⍴⊂0/[J]⊃Y`.
>
> Otherwise, `DOMAIN ERROR` is reported.

Table 1: Identity Elements

Function		Identity
Add	+	0
Subtract	-	0
Multiply	×	1
Divide	÷	1
Residue	|	0
Minimum	⌊	M[1]
Maximum	⌈	-M
Power	*	1
Binomial	!	1
And	∧	1
Or	∨	0
Less	<	0
Less or Equal	≤	1
Equal	=	1
Greater	>	0
Greater or Equal	≥	1
Not Equal	≠	0
Encode	⊤	0
Union	∪	⍬
Replicate	/⌿	1
Expand	\⍀	1
Rotate	⌽⊖	0

[1] M represents the largest representable value: typically this is 1.7E308, unless ⎕FR is 1287, when the value is 1E6145.

Examples

```
      ∨/0 0 1 0 0 1 0
1
      MAT
1 2 3
4 5 6

      +/MAT
6 15

      +⌿MAT
5 7 9

      +/[1]MAT
5 7 9

      +/(1 2 3)(4 5 6)(7 8 9)
 12 15 18

      ,/'ONE' 'NESS'
 ONENESS

      +/⍳0
0

      (⊂⍬)≡,/⍬
1
      (⊂'')≡,/0⍴'Hello' 'World'
1
      (⊂0 3 4⍴0)≡⍪/0⍴⊂2 3 4⍴0
1
```

Random Link ⎕RL

⎕RL is a 2-element vector. Its first element contains the base or *random number seed* and its second element is an integer that identifies the random number generator that is currently in use. Together these items define how the system generates random numbers using Roll and Deal.

In a `clear ws` ⎕RL is `(0 1)`.

Random Number Seed

The facility to set the seed to a specific value provides the means to generate a repeatable sequence of random numbers, such as might be required by certain types of simulation modelling. This capability is not provided by RNG2.

If the seed is set to 0, the seed is set randomly but may be retrieved and subsequently re-assigned to create a repeatable sequence.

If the seed is set to 0, Dyalog is able to take advantage of certain optimisations which deliver maximum performance. In this case, the actual seed in use is intentionally hidden and ⎕RL[1] always reports 0, regardless of the Random Number Generator in use.

Random Number Generators

The 3 random number generators are listed in the table below. The 4th column of the table contains the values of seeds that may be assigned to them.

Id	Name	Algorithm	Valid Seed Values
0	RNG0	Lehmer linear congruential generator.	0, 0, or an integer in the range 1 to ¯2+2*31
1	RNG1	Mersenne Twister.	0, 0, an integer in the range 1 to ¯2+2*31 or a 625-element integer vector
2	RNG2	Operating System random number generator.	0

The default random number generator in a `CLEAR WS` is 1 (Mersenne Twister). This algortithm *RNG1* produces 64-bit values with good distribution.

The Lehmer linear congruential generator *RNG0* was the only random number generator provided in versions of Dyalog APL prior to Version 13.1. The implementation of this algorithm has several limitations including limited value range `(2*31)`, short period and non-uniform distribution (some values may appear more frequently than others). It is retained for backwards compatibility.

Under Windows, the Operating System random number generator algorithm *RNG2* uses the `CryptGenRandom()` function. Under UNIX/Linux it uses `/dev/urandom`.

Random Number Sequences

Random number sequences may be predictable or not and repeatable or not. A predictable and repeatable sequence is obtained by starting with the same specific value for the seed. A non-predictable sequence is obtained by starting with a seed which is itself chosen at random, but such a sequence is repeatable if the value of the seed (chosen at random) is visible. A non-predictable and non-repeatable sequence of random numbers is obtained where the initial seed is chosen completely at random and is unknown.

Using *RNG0* or *RNG1*:

- To obtain an entirely predictable random sequence, set the seed to a non-zero value
- To obtain a non-predictable, but repeatable sequence, set the seed to `0`
- To obtain a non-predictable, non-repeatable series of random numbers, set the seed to `⍬`

RNG2 does not support a user modifiable random number seed, so when using this scheme, it is not possible to obtain a repeatable random number series and the seed must always be `⍬`.

Implementation Note:

⎕RL does not behave quite like a regular 2-element variable; it has its own rules relating to assignment and reference.

Reference

⎕RL *returns* a 2-element vector whose second element identifies the scheme in use (0, 1 or 2).

If ⎕RL[1] is set to ⍬, ⎕RL[1] always reports ⍬.

Otherwise if the seed ⎕RL[1] is set to a value other than ⍬:

- using *RNG0*, ⎕RL[1] is an integer which represents the *seed* for the next random number in the sequence.
- using *RNG1*, the system internally retains a block of 312 64-bit numbers which are used one by one to generate the results of roll and deal. When the first block of 312 have been used up, the system generates a second block. In this case, ⎕RL[1] is an integer vector of 32-bit numbers of length 625 (the first is an index into the block of 312) which represents the internal state of the random number generator. This means that, as with *RNG0*, you may save the value of ⎕RL in a variable and reassign it later.
- Using *RNG2*, the seed is purely internal and ⎕RL[1] is always zilde.

Assignment

⎕RL may only be assigned in its entirety. Indexed and selective assignment may not be used to assign values to individual elements.

To preserve compatibility with Versions of Dyalog prior to Version 15.0 (in which ⎕RL specifies just the seed) if the value assigned to ⎕RL represents a valid seed for the random number generator in use, it is taken to be the new seed. Otherwise, the value assigned to ⎕RL must be a 2-element vector, whose first item is the seed and whose second item is 0, 1 or 2 and specifies the random number generator to be used subsequently.

Examples (specific seeds for repeatable sequences)

```
      )CLEAR
clear ws
      ⎕RL←16807
      10?10
4 1 6 5 2 9 7 10 3 8
      5↑⍕⎕RL
10 0 16807 1819658750 ¯355441828
      X←?1000⍴1000
      5↑⍕⎕RL
100 ¯465541037 ¯1790786136 ¯205462449 996695303

      ⎕RL←16807
      10?10
4 1 6 5 2 9 7 10 3 8
      Y←?1000⍴1000
      X≡Y
1
      5↑⍕⎕RL
100 ¯465541037 ¯1790786136 ¯205462449 996695303

      ⎕RL←16807 0 ⍝ Select RNG0
      ⎕RL
16807 0
      ?9 9 9
2 7 5
      ?9
7
      ⎕RL
984943658 0

      ⎕RL←16807
      ?9 9 9
2 7 5
      ?9
7
      ⎕RL
984943658 0

      ⎕RL←16807 1 ⍝ Select RNG1
      5↑⍕⎕RL
100 ¯465541037 ¯1790786136 ¯205462449 996695303
```

Examples (0 seed)

When you set the seed to 0, a random seed is created for you:

```
      ⎕RL←0 0
      ⎕RL
865618822 0
      ⎕RL←0
      ⎕RL
1100783275 0
```

Setting the seed to 0 gives you a new, unpredictable random sequence yet it is repeatable because you can retrieve (and subsequently re-use) the actual seed after you set it:

```
      ?10⍴100
14 22 18 30 42 22 71 32 32 12
      ⎕RL←1100783275
      ?10⍴100
14 22 18 30 42 22 71 32 32 12
```

Example (zilde)

When you set the seed to zilde, you get the same random initialisation as setting it to 0 but you can't retrieve the actual value of the seed. When it is set to ⍬ it is subsequently reported as ⍬ and the internal value of the seed is hidden.

```
      ⎕RL←⍬
      ⎕RL
```

Make Directory {R}←{X}⎕MKDIR Y

This function creates a new directory.

`Y` is a character vector or scalar containing a file name that conforms to the naming rules of the host Operating System.

By default, the path specified by `Y` must exist and the base name specified by `Y` must not exist (see *File Name Parts* on page 65), otherwise an error is signalled. The optional left argument `X` is the numeric scalar 0, 1, 2 or 3 which amends this behaviour as shown in the following table. If omitted, it is assumed to be 0.

`0`	Default behaviour.
`1`	No action is taken if the directory specified by `Y` already exists. The return value may be used to determine whether a new directory was created or not.
`2`	Any part of the *path* specified in `Y` which does not already exist will be created in preparation of creating `Y` itself.
`3`	Combination of 1 and 2.

The shy result `R` is 1 if a directory was created otherwise it is 0.

Examples

```
      ⎕NEXISTS '\Users\Pete\Documents\temp'
0
      ⎕←⎕MKDIR '\Users\Pete\Documents\temp'
1
      ⎕←⎕MKDIR '\Users\Pete\Documents\temp'
FILE NAME ERROR: Directory exists
      ⎕←⎕MKDIR'\Users\Pete\Documents\temp'
     ^

      ⎕←⎕MKDIR'\Users\Pete\Documents\temp\t1\t2'
FILE NAME ERROR: Unable to create directory ("The system
cannot find the path specified.")
      ⎕←⎕MKDIR'\Users\Pete\Documents\temp\t1\t2'
     ^


      ⎕←2 ⎕MKDIR'\Users\Pete\Documents\temp\t1\t2'
1
```

Native File Delete {R}←{X}⎕NDELETE Y

This function deletes a file or a directory.

Y is a character vector or scalar containing a file name that conforms to the naming rules of the host Operating System.

The optional left argument X is a numeric scalar; valid values are shown in the following table. If omitted, its default value is 0.

0	The file or directory with the given name must exist.
1	If the file or directory with the given name does not exist then no action is taken. The result R may be used to determine whether the file or directory was deleted or not.

The shy result R is 1 if the file or directory was deleted otherwise it is 0.

If Y is the name of a symbolic link, ⎕NDELETE deletes that symbolic link; the target of the symbolic link is unaffected.

Examples

```
      ⎕NEXISTS'\Users\Pete\Documents\temp\t1\t2'
1
      ⎕←⎕NDELETE'\Users\Pete\Documents\temp\t1\t2'
1
      ⎕←⎕NDELETE'\Users\Pete\Documents\temp\t1\t2'
FILE NAME ERROR: Invalid file or directory name ("The
system cannot find the file specified.")
      ⎕←⎕NDELETE'\Users\Pete\Documents\temp\t1\t2'
     ^

      ⎕←1 ⎕NDELETE'\Users\Pete\Documents\temp\t1\t2'
0
```

If the file is in use or the current user is not authorised to delete it, ⎕NDELETE will not succeed but will instead generate a FILE ACCESS ERROR.

Native File Exists R←⎕NEXISTS Y

This function reports whether or not a file or directory exists.

Y is a character vector or scalar containing a file name that conforms to the naming rules of the host Operating System.

The result R is 1 if the file or directory specified by Y exists (can be accessed), otherwise R is 0.

Example

```
      ⎕←⎕MKDIR'\Users\Pete\Documents\temp\t1\t2'
1
      ⎕NEXISTS'\Users\Pete\Documents\temp\t1\t2'
1
      ⎕NEXISTS'\Users\Pete\Documents\temp\t1\t2\pd'
0
```

Note: If Y is a symbolic link, ⎕NEXISTS will return 1 whether or not the target of the symbolic link exists.

Read Text File R←{X} ⎕NGET Y

This function reads the contents of the specified text file. See also *Write Text File* on page 67.

`Y` is either a character vector/scalar containing the name of the file to be read, or a 2-item vector whose first item is the file name and whose second is an integer scalar specifying `flags` for the operation.

If `flags` is 0 (the default value if omitted) the content in the result `R` is a character vector. If `flags` is 1 the result is a nested array of character vectors corresponding to the lines in the file.

The optional left-argument `X` is a character vector that specifies the file-encoding:

Table 2: File Encodings

Encoding	Description
`UTF-8`	The data is encoded as UTF-8 format.
`UTF-16LE`	The data is encoded as UTF-16 little-endian format.
`UTF-16BE`	The data is encoded as UTF-16 big-endian format.
`UTF-16`	The data is encoded as UTF-16 with the endianness of the host system (currently BE on AIX platforms, LE on all others).
`UTF-32LE`	The data is encoded as UTF-32 little-endian format.
`UTF-32BE`	The data is encoded as UTF-32 big-endian format.
`UTF-32`	The data is encoded as UTF-32 with the endianness of the host system (currently BE on AIX platforms, LE on all others).
`ASCII`	The data is encoded as 7-bit ASCII format.
`Windows-1252`	The data is encoded as 8-bit Windows-1252 format.
`ANSI`	ANSI is a synonym of Windows-1252.

The above UTF formats may be qualified with -BOM or -NOBOM (e.g. UTF-8-BOM). See *Write Text File* on page 67.

Whether or not `X` is specified, if the start of the file contains a recognised Byte Order Mark (BOM) , the file is decoded according to the BOM. Otherwise, if `X` is specified the file is decoded according to the value of `X`. Otherwise, the file is examined to try to decide its encoding and is decoded accordingly.

The result `R` is a 3-element vector comprising `(content) (encoding) (newline)` where:

`content`	A simple character vector, or a vector of character vectors, according to the value of `flags`.
`encoding`	The encoding that was actually used to read the file. If this is a UTF format, it will always include the appropriate endianness (except for UTF-8 to which endianness doesn't apply) and a -BOM or -NOBOM suffix to indicate whether or not a BOM is actually present in the file. For example, UTF-16LE-BOM.
`newline`	Determined by the first occurrence in the file of one of the newline characters identified in the line separator table, or `0` if no such line separator is found.

If `content` is simple then all its line separators (listed in the table below) are replaced by (normalised to) `⎕UCS 10`, which in the Classic Edition must be in `⎕AVU` (else `TRANSLATION ERROR`).

If `content` is nested, it is formed by splitting the contents of the file on the occurrence of *any* of the line separators shown in the table below. These line separators are removed.

The 3rd element of the result `newline` is a numeric vector from the *Value* column of the table below corresponding to the first occurrence of any of the **newline characters** in the file. If none of these characters are present, the value is `0`.

Table 3: Line separators:

Value	Code	Description
newline characters		
13	`CR`	Carriage Return (U+000D)
10	`LF`	Line Feed (U+000A)
13 10	`CRLF`	Carriage Return followed by Line Feed
133	`NEL`	New Line (U+0085)
other line separator characters		
11	`VT`	Vertical Tab (U+000B)
12	`FF`	Form Feed (U+000C)
8232	`LS`	Line Separator (U+2028)
8233	`PS`	Paragraph Separator (U+2029)

Note:

⎕NGET currently queries the size of the file, and then attempts to read that many bytes from the file. Certain types of file, including pipes, sockets, FIFOs and the files under /proc always return a size of 0, so ⎕NGET will return 0-length content. In this case you should use ⎕NREAD to read the contents of such files. This limitation may be removed in a future release.

Example (Linux)

```
      z←⎕sh'echo $PPID'
      ≠⍤⎕NGET '/proc/',z,'/stat' 1
0 11 0
```

No data although the files does exist, and can be read using ⎕NTIE/⎕NREAD/⎕NUNTIE.

Native File Information R←{X}⎕NINFO Y

This function returns information about one or more files or directories.

Y may either be a numeric scalar containing the tie number of a native file, or a character vector or scalar containing a file name that conforms to the naming rules of the host Operating System.

This function may be applied using the Variant operator with two options; Wildcard (the Principal option) and Follow. The default value for Wildcard is 0, and for Follow is 1.

If Wildcard is 1, the part of Y that specifies the *base name* and *extension* (see *File Name Parts* on page 65), may also contain the wildcard characters "?" and "*" and potentially identifies more than one file. An asterisk is a substitute for any 0 or more characters in a file name or extension; a question-mark is a substitute for any single character.

The Follow option affects the properties of a symbolic link If Follow is 1, the properties reported for a symbolic link are those of the target of the symbolic link; if Follow is 0, they are of the symbolic link itself.

The optional left argument X is a simple numeric array containing values shown in the following table.

X	Property	Default
0	Name of the file or directory, as a character vector. If Y is a tie number then this is the name which the file was tied.	
1	Type, as a numeric scalar: 0=Not known 1=Directory 2=Regular file 3=Character device 4=Symbolic link (only when Follow is 0) 5=Block device 6=FIFO (not Windows) 7=Socket (not Windows)	0
2	Size in bytes, as a numeric scalar	0
3	Last modification time, as a timestamp in ⎕TS format	7⍴0
4	Owner user id, as a character vector – on Windows a SID, on other platforms a numeric userid converted to character format	''

X	Property	Default
5	Owner name, as a character vector	''
6	Whether the file or directory is hidden (1) or not (0), as a numeric scalar. On Windows, file properties include a "hidden" attribute; on non-Windows platforms a file or directory is implicitly considered to be hidden if its name begins with a ".".	¯1
7	Target of symbolic link (when Type is 4).	''

Each value in X identifies a property of the file(s) or directory(ies) identified by Y whose value is to be returned in the result R. If omitted, the default value of X is 0. Values in X may be specified in any order and duplicates are allowed. A value in X which is not defined in the table above will not generate an error but results in a ⍬ (Zilde) in the corresponding element of R.

R is the same shape as X and each element contains value(s) determined by the property specified in the corresponding element in X. The depth of R depends upon whether or not the Wildcard option is enabled. If, for any reason, the function is unable to obtain a property value, (for example, if the file is in use exclusively by another process) the default value shown in the last column is returned instead.

If the Wildcard option is not enabled (the default) then Y specifies exactly one file or directory and must exist. In this case each element in R is a single property value for that file. If the name in Y does not exist, the function signals an error. On non-Windows platforms "*" and "?" are treated as normal characters. On Windows an error will be signalled since neither "*" nor "?" are valid characters for file or directory names.

If the Wildcard option is enabled, zero or more files and/or directories may match the pattern in Y. In this case each element in R is a vector of property values for each of the files; Note that no error will be signalled if no files match the pattern.

When using the Wildcard option, matching of names is done case insensitively on Windows and OS X, and case sensitively on other platforms. The names '.' and '..' are excluded from any matches. The order in which the names match is not defined.

Examples

```
      (0 1 2) ⎕NINFO 'c:\Users\Pete\Documents'
┌→──────────────────────────────────┐
│ ┌→──────────────────────┐ 1 163840 │
│ │c:/Users/Pete/Documents│          │
│ └~──────────────────────┘          │
└∊──────────────────────────────────┘
      ⍕1⎕NPARTS ''
c:/Users/Pete/
      (⎕NINFO⍠1)'D*'
┌─────────────────────────────────────┐
│┌───────┬─────────┬─────────┬───────┐│
│|Desktop|Documents|Downloads|Dropbox||
│└───────┴─────────┴─────────┴───────┘│
└─────────────────────────────────────┘
      (⎕NINFO⍠1)'Documents\*.zip'
┌──────────────────────┐
│┌────────────────────┐│
│|Documents/dyalog.zip||
│└────────────────────┘│
└──────────────────────┘
      ⍪ (0,⍳6) ⎕NINFO 'Documents\dyalog.zip'
┌──────────────────────────────────────────────┐
│Documents/dyalog.zip                          │
├──────────────────────────────────────────────┤
│2                                             │
├──────────────────────────────────────────────┤
│3429284                                       │
├──────────────────────────────────────────────┤
│2016 1 22 16 43 58 0                          │
├──────────────────────────────────────────────┤
│S-1-5-21-2756282986-1198856910-2233986399-1001│
├──────────────────────────────────────────────┤
│HP\Pete                                       │
├──────────────────────────────────────────────┤
│0                                             │
└──────────────────────────────────────────────┘
```

File Name Parts R←{X} ⎕NPARTS Y

Splits a file name into its constituent parts.

Y is a character vector or scalar containing a file name that conforms to the naming rules of the host Operating System. The file need not exist; indeed this system function makes no attempt to identify or locate it.

The optional left-argument X specifies whether or not the file name specified by Y is *normalised* before being processed. The default value 0 means no normalisation; 1 means normalise as follows:

- Pathnames are made absolute.
- On Windows, all "\" directory separators are changed to "/".
- The resultant name is simplified by removing extraneous directory separators etc. On Windows, this includes resolving occurrences of "." and ".." within the name. On non-Windows platforms single "." are removed. Note that ".." and symbolic links interact differently on Windows to other platforms; on other platforms they cannot be removed without reference to the file system itself and are left in place.

The result R is a 3-element vector of character vectors as follows:

[1]	*path*
[2]	*base name*
[3]	*extension*

The *path* identifies the directory in which the file exists.

The *base name* is the name of the file stripped of its path and extension, if any.

The *extension* is the file extension including the leading ".".

Examples

```
      ⎕CMD 'CD'⍵ Current working directory
c:\Users\Pete

      1 ⎕NPARTS 'a'
```


```
      1 ⎕NPARTS '\Users\Pete\Documents\dyalog.zip'
```


```
      ⍕'.'⎕wg'APLVersion'
AIX-64
      1 ⎕nparts'/home/andys/./..'
```

Note that `⍕1 ⎕NPARTS ''` returns the current working directory.

```
      ⍕1 ⎕NPARTS ''
```

Write Text File {R}←X ⎕NPUT Y

This function writes character data to a text file. See also *Read Text File* on page 59.

`Y` is either a simple character vector or scalar containing the name of the file to be written, or a 2-item vector whose first item is the file name and whose second is an integer scalar specifying `flags` for the operation.

If `flags` is 0 (the default value if omitted) the file will not be overwritten if it already exists and `⎕NPUT` will signal an error. If `flags` is 1 the file will be written regardless.

The left-argument `X` is comprised of 1, 2 or 3 items which identify `(content) (encoding) (newline)` respectively.

`content` is either a vector of character vectors, each of which represents a line in the file to be written, or a simple character vector.

If specified, `encoding` is a character vector from the first column in the table *File Encodings* on page 59. If `encoding` specifies a UTF format, it may be qualified with -BOM (e.g. UTF-8-BOM), which causes a Byte Order Mark (BOM) to be written at the beginning of the file or -NOBOM which does not. If the -BOM or -NOBOM suffix is omitted, UTF-8 defaults to UTF-8-NOBOM, while the other UTF formats default to -BOM.

If omitted, `encoding` defaults to UTF-8-NOBOM.

If specified, `newline` is numeric and is either `⍬` or a scalar or vector from the column labelled *Value* in the **newline characters** section of the table *Line separators:* on page 60. Any other value causes `DOMAIN ERROR`. If `newline` is omitted it defaults to `(13 10)` on Windows and `10` on other platforms.

If `content` is nested, each element is considered to be to a logical line in the file, and when the file is written, a line separator character corresponding to `newline` is appended to each and every element, i.e. the data written to the file (excluding the BOM) is:

```
∈content,¨⊂⎕UCS newline
```

If `content` is simple each and every LF (`⎕UCS 10`) character that it contains is first replaced by the character corresponding to `newline`. If not present, one LF character is added to the end of the array prior to these replacements.

In both cases, any other line separator characters are written *as is* to the file. This allows the APL programmer to insert other line endings if so desired.

If content contains anything other than a character vector or scalar (or these, nested) then a `DOMAIN ERROR` is signalled.

The shy result `R` is the number of bytes written to the file.

Note that when `content` is a vector of character vectors and `encoding` is omitted; it is necessary to enclose the left argument.

Example:

```
      txt←'mene' 'mene' 'tekel' 'upharsin'
      ⎕←(⊂txt) ⎕NPUT 'writing.txt'
25
```

Whereas:

```
      txt ⎕NPUT 'writing.txt'
LENGTH ERROR: Left argument should be content, optional
encoding and optional line ending
      txt ⎕NPUT'writing.txt'
     ^
```

Chapter 4:

I-Beam Reference Changes

I-Beam Changes

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

The column labelled *O/S* indicates if a function applies only on Windows (W) or only on non-Windows (X) platforms.

I-beam functionality added to Version 16.0.

A	Derived Function	O/S
180	Canonical Representation	
400	Compiler Control	
600	Trap Control	
819	Case Convert	
1500	Hash Array	
2014	Remove Data Binding	W
2016	Create .NET Delegate	W
2041	Override COM Defrault Value	W
2501	Discard thread on exit	W
2502	Discard parked threads	W
3502	Manage RIDE Connections	
5176	List Loaded Files	
5177	List Loaded File Objects	
8415	Singular Value Decomposition	

Error Messages

When attempting to use I-Beam with an unsupported operation value, Version 15.0 now reports three different error messages:

- Invalid I-Beam function selection
- I-Beam function xxx has been withdrawn
- I-Beam function xxx is not supported by this interpreter

This allows the user to distinguish between operation values that have never been used, those that have been used in earlier versions but are no longer included in the current version, and those that are valid in other editions or on other platforms other than the current interpreter.

Canonical Representation R←180⌶Y

This function is the same as the system function ⎕CR except that it can be used to obtain the canonical representation of methods in classes. 180⌶ is used by]PROFILE.

Example

```
      )load ComponentFile
C:\Program Files\Dyalog\Dyalog APL-64 15.0 Unicode\...

      180⌶'ComponentFile.Close'
 Close
 :Implements Destructor
 :If tie∈⎕FNUMS
     :If temp ⋄ Name ⎕FERASE tie
     :Else ⋄ ⎕FUNTIE tie
     :EndIf
 :EndIf
```

Compiler Control R←{X}(400⌶)Y

Controls the actions of the Compiler. For further information, see *Compiler User Guide*.

The optional left-argument X must be one of the following:

X	Description
0	Set automatic compilation options (default)
1	Determine whether the function/operator Y has been successfully compiled
2	Compile the function/operator Y
3	Discard compiled form of the function/operator Y
4	Show bytecode for the compiled function/operator Y
nsref	Compile the function/operator Y using user-defined callbacks in this namespace to provide information about global names

The nature of Y and R depend on the value of X as follows:

X=0 : Control Automatic Compilation (default)

Y must be an integer 0, 1, 2, or 3.

Y	Description
0	disable automatic compilation (initial setting)
1	compile functions when they are fixed (with ⎕FX or from the function editor)
2	compile operators the first time they are executed
3	compile functions when they are fixed (with ⎕FX or from the function editor) and compile operators the first time they are executed

The result R is the previous value of Y.

The automatic compilation setting is maintained within the workspace, and is saved and loaded with the workspace.

X=1: Query Compilation State

Y must be a character vector, matrix or vector of vectors specifying the name of a function or operator or a list of such names.

The result R is a Boolean scalar or vector, with the value 1 if the corresponding function/operator has been successfully compiled or 0 if it has not.

X=2: Compile

Y must be a character vector, matrix or vector of vectors specifying the name of a function or operator or a list of such names that should be compiled.

The result R is a matrix of diagnostic information or, if Y was either a matrix or a vector of vectors, a vector of such matrices. Each row of the matrix describes a problem that caused the compilation to fail, with four columns corresponding to:

1. the APL error number
2. the line number in the function/operator
3. the column number (currently always 0)
4. the error message

If the matrix R has zero rows then the compilation was successful.

If this mechanism is used to compile operators, then the compiled bytecode will assume that the operator's operands are functions rather than arrays. At run time, the operands will be checked – if they are functions then the compiled bytecode will be used, otherwise the operator will be interpreted.

X=3: Discard Compiled Form

If Y is empty, discard any compiled bytecode for all functions and operators in the workspace. If Y is a character vector, matrix or vector of vectors specifying the name of a function or operator or a list of such names, discard any compiled bytecode for the name(s) specified by Y. R is always 0

X=4: Show Bytecode

Y must be a character vector, matrix or vector of vectors specifying the name of a function or operator or a list of such names.

The result R is a multi-line string (that is, a character vector with embedded newlines) or, if Y was either a matrix or a vector of vectors, a vector of such strings. Each string is a human-readable representation of the bytecode of a compiled function or operator.

This functionality is provided for information and diagnostic purposes only. The human-readable form of the bytecode is subject to change at any time.

X is a namespace reference: Compile With Callbacks

Y must be a character vector, matrix or vector of vectors specifying the name of a function or operator or a list of such names. The specified functions or operators are compiled in the same way as when X = 2 except that the compiler uses the user-defined callback functions in the namespace X to obtain information about global names. The namespace X can contain any or all of following callback functions:

Callback	Description
quadNC	analogous to the system function ⎕NC. When applied monadically to an enclosed character vector it should return the detailed name class of that name. For example, given the name of a global dfn it should return the value 3.2.
quadAT	analogous to the system function ⎕AT. When applied monadically to an enclosed character vector it should return a 1 by 4 matrix whose first item is a vector of 3 integers describing (respectively) the result, function valence and operator valence of the name.
getValue	used to obtain the value of global constants. When applied monadically to a character vector that is a global constant it should return the enclose of the constant value, otherwise it returns ⍬.

Each of these callback functions returns information about names that should be guaranteed to exist when the compiled functions are executed. The compiler assumes that the information returned by the callbacks is correct, and generates bytecode accordingly. In the case of quadNC and quadAT, if the information returned by the callbacks turns out not to be correct when the compiled function is executed, then a runtime error is generated.

The result R is a matrix of diagnostic information or, if Y was either a matrix or a vector of vectors, a vector of such matrices. Each row of the matrix describes a problem that caused the compilation to fail, with four columns corresponding to:

1. the APL error number
2. the line number in the function/operator
3. the column number (currently always 0)
4. the error message

If the matrix R has zero rows then the compilation was successful.

Trap Control R←600⌶Y

This function is used to temporarily disable the error trapping mechanism used by `:Trap` and `⎕TRAP`. This can be useful in debugging applications.

`Y` is an integer 0, 1 or 2 as shown in the following table.

`R` is the previous value (0, 1, or 2) of the trap state.

Y	Effect
0	Enable all traps.
1	Disable all traps.
2	Disable traps in suspended functions from triggering when an error is generated in the Session.

Note that the *Disable traps in session* option of the Session *Options* menu performs the same tasks as `(600⌶0)` and `(600⌶2)`.

For error-guards in dfns `600⌶0` and `600⌶2` are equivalent; in neither case is an error generated in the session caught by an error guard in a suspended dfn.

Case Convert R←{X}(819⌶)Y

Converts character data in Y to upper or lower-case. This function is considerably faster than any comparable function coded in APL, especially on nested arrays.

This function is Unicode only and is not available in the Classic variants of Dyalog.

Y may be any array of arbitrary depth so long as all the elements are characters.

The optional left-argument X is 0 (convert to lower-case) or 1 (convert to upper-case). If omitted, the default is 0.

The result R has the same structure as Y but each character element is case folded to upper or lower case.

Characters are converted per the default case mappings specified by The Unicode Consortium, described at:

```
ftp://ftp.unicode.org/Public/3.0-Update/UnicodeData-3.0.0.html
```

and using the table at:

```
http://unicode.org/Public/UNIDATA/UnicodeData.txt
```

If conversion is being used to do case-insensitive character comparisons then converting everything to lower case is generally preferable to converting everything to upper. This is because converting to lower case can be faster.

Examples

```
      (819⌶) 'How many Roads must a man walk down'
how many roads must a man walk down
      1 (819⌶) 'How many Roads must a man walk down'
HOW MANY ROADS MUST A MAN WALK DOWN

      data←1000⍴⊂'Hello there.'
      lc_data←819⌶ data
      4↑lc_data
 hello there.  hello there.  hello there.  hello there.
```

Hash Array R←{X}1500⌶Y

This function creates a hashed array, returns an unhashed copy of an array or reports the state of hashing of an array.

Y may be any array.

If X is omitted, the result R is a copy of Y that has been invisibly marked as hashed. R behaves the same as Y in all respects. The only difference is that dyadic ⍳ and related functions are expected to run faster when applied to a hashed array. The *hash* will be created the first time the array is used as an argument to ⍳ or other set functions. The *hashed* property is preserved across assignments and argument passing, but in general is not preserved by any primitive functions.

If X is 1, the result R returns an indication of whether Y has been marked for hashing or whether the hash has been created:

R	State of Y
0	Y has not been marked for hashing
1	Y has been marked for hashing, but the hash tables has not yet been created
2	Y has a hash table

If X is 2, the result R is the unhashed form of Y.

Examples:

```
      R←1500⌶1 2 3    ⍝ R is marked for hashing

      1 (1500⌶)R
1
      S←R             ⍝ S is marked for hashing
      {⍵⍳2 3 5}R      ⍝ R is now hashed
      1 (1500⌶)R
2
      U←(⍴R)⍴R        ⍝ U is not hashed
      U←⊃⊂R           ⍝   ditto
      1 (1500⌶)U
0
```

If R is a hashed array then certain forms of modified assignment will preserve and efficiently update the hash table:

```
      R,←Y    ⍝ only for scalar or vector R
      R⍪←Y
      R↓⍨←Y   ⍝ only for negative singleton Y
```

Examples:

```
      R←1500⌶1 2 3 ⍝ R is hashed

      R,←5     ⍝ ,← preserves and updates
               ⍝ the hash table
      R
1 2 3 5
      R⍳2 4 6
2 5 5

      R⍫⍤←¯2   ⍝ ⍫⍤← preserves and updates
               ⍝ the hash table
      R
1 2
      R⍳2 4 6
2 3 3
```

The *hashed* property survives `)SAVE`/`)LOAD` and `)SAVE`/`)COPY`. It does not currently survive writing to a component file and reading back again.

Remove Data Binding R←2014⌶Y

Windows only.

This function disassociates a data-bound variable from its data binding source.

`Y` is any array.

If `Y` or an element of `Y` is a character vector that contains the name of a data-bound variable, that variable is dissociated from its data binding source.

The result `R` is always 1.

Example

```
      2014⌶'txtSource'
1
```

Create .NET Delegate R←2016⌶Y

Windows only.

.NET methods (and properties) may specify a parameter to be a *delegate*. A delegate is a place holder for a function, normally with a particular signature and result type, that should be supplied when the method is called. Sometimes the signature of a .NET method that takes a delegate as a parameter does not provide enough information for Dyalog to determine automatically what type of delegate is required. `2016⌶` allows you to specify the type so that Dyalog can perform the necessary conversion(s) at run-time.

`Y` is a 2-element array. The first element is a .NET type that inherits from the abstract .NET Class `System.Delegate`. The second item is either the name of or the `⎕OR` of an APL function which is to be invoked via a .NET method or property.

The result `R` is a ref to an instance of a .NET type specified by the first element of `Y`, which internally is associated with the function identified by the second element of `Y`.

Example

```
      ∇foo∇
     ∇ foo(ev arg)
[1]    ⍝ Callback for .NET method
     ∇
      ⎕USING←'System'
      del←2016⌶ EventHandler'foo'
      del
System.EventHandler
```

Then, when calling a .NET method that requires a `Delegate` of type `System.Eventhandler`, but whose signature is imprecise in this respect, the object `del` should be used instead.

Discard Thread on Exit R←2501⌶Y

APL threads that Dyalog creates to serve incoming .NET requests are not terminated when their work is done. They persist so that if another call comes in on the same .NET thread the same APL thread can handle it. In effect the thread is *parked* until it is needed again. If the thread is not required, there is a small performance cost in maintaining it in this state.

`(2501⌶0)` must be called from WITHIN one of these threads and tells the interpreter NOT to park the thread on termination, but to discard the thread completely.

Discard Parked Threads R←2502⌶Y

APL threads that Dyalog creates to serve incoming .NET requests are not terminated when their work is done. They persist so that if another call comes in on the same .NET thread the same APL thread can handle it. In effect the thread is *parked* until it is needed again. If the thread is not required, there is a small performance cost in maintaining it in this state.

`(2502⌶0)` removes all parked threads from the workspace.

Manage RIDE Connections R←3502⌶Y

3502⌶ gives control over RIDE connections to the interpreter. More details about RIDE can be found in the *RIDE User Guide*. This I-Beam has been significantly changed in version 15.0.

Y may be either 0 or 1 or a simple character vector.

R has the value 0 if the call to 3502⌶ was successful; if unsuccessful the value may be either a positive or negative integer.

If Y is 0, then any active RIDE connections are disconnected, and no future connections may be made.

If Y is 1, then the interpreter attempts to enable RIDE, using the value of the initialisation string to determine the connection details. If the current initialisation string is ill-defined, or the RIDE DLL/shared library is not available, then R will be non-zero.

If Y is a character vector and RIDE is currently disabled, then the current initialisation string is unconditionally replaced by the contents of Y. If RIDE is currently enabled, the initialisation string is not replaced, and R will have the value ¯2.

The initialisation string has the same syntax as the value of the **RIDE_INIT** configuration parameter which is described in the *RIDE User Guide*

If RIDE is currently disabled, and 3502⌶0 is called or if RIDE is currently enabled and 3502⌶1 is called, no action is taken and R will have the value ¯1.

The configuration parameter **RIDE_INIT** can still be used to establish the initial value of the RIDE initialisation string.

The runtime interpreter has RIDE disabled by default, whether or not **RIDE_INIT** is set; the only method of enabling RIDE in a runtime interpreter is to call 3502⌶1.

If **RIDE_INIT** is set when a development interpreter is called, RIDE will be enabled provided that the RIDE DLL/shared library is available and the **RIDE_INIT** variable is properly formed. If the connection is of type SERVE the port must not be in use. If any of these conditions are not met, then the interpreter fails with a non-zero exit code. If **RIDE_INIT** is not set then the development interpreter will start, but with RIDE disabled. It is therefore possible to override the **RIDE_INIT** variable in the development interpreter with code similar to:

```
r←3502⌶0              ⍝ Stop RIDE
r←3502⌶'SERVE::4511'  ⍝ Update init string
r←3502⌶1              ⍝ Start RIDE
```

And similarly for altering the RIDE settings in an active APL session.

Notes:

In 14.1 and earlier `3502⌶0` was used to enable RIDE; this value is still valid, albeit deprecated: code should call `3502⌶1` instead.

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

Singular Value Decomposition R←(8415⌶)Y

Y is a simple numeric matrix.

The result R is a 4 element vector whose elements are as follows.

[1]	U	a unitary matrix
[2]	S	a diagonal matrix
[3]	V	a unitary matrix
[4]	f	a Boolean flag indicating whether the algorithm converged or not

This function computes a factorisation of the matrix Y such that:

```
M ≡ U +.× S +.× ⍉+V
```

This can be useful for analysing matrices for which ⌹ cannot compute an inverse, because they are singular or nearly singular.

For further information, see https://en.wikipedia.org/wiki/Singular_value_decomposition.

Chapter 5:

Object Reference Changes

GUI Enhancements

The following table summarises the main changes to GUI features in Version 15.0.

Name	**Change**
DISPID	
Native look and feel	Default changed to enabled
GetFocusObj	New method
SetEventInfo	Enhanced to set DISPID
SetFnInfo	Enhanced to set DISPID
SetPropertyInfo	Enhanced to set DISPID

DISPID (Dispatch ID)

COM objects created by Dyalog (OLEServer and ActiveXControl objects) export their members (methods, properties and events) using the standard *IDispatch* interface.

Using this interface, a client application may discover the names and parameters of the members supported by an object at run-time, and then access them by name. Alternatively, a client application may compile references to the object's members in advance using their *Dispatch IDs* or *DISPIDs*.

Prior to Version 14.1, Dyalog assigned all DISPIDs automatically[1], making it impractical for them to be compiled into client applications.

From Version 14.1 onwards, the SetFnInfo, SetPropertyInfo and SetEventInfo methods allow the Dyalog programmer to assign DISPIDs so that they may be used directly by client applications. The specified DISPID must be a non-zero integer. The special value `¯1` causes Dyalog to assign the DISPID automatically as before.

Note

Each of the DISPIDs exported by a COM object must be unique. Furthermore, the behaviour of a COM object with non-unique DISPIDs is undefined. Non-unique DISPIDs may prevent the COM object from being registered (with or without generating an error) or may cause a run-time failure. If Dyalog assigns all the DISPIDs of an object, they will be unique. If you choose to allocate your own DISPIDs to **any** of the members of a Dyalog COM object, the responsibility to ensure that they are **all** unique is yours. In this case, Dyalog does not guarantee nor check for uniqueness.

[1] An automatically assigned DISPID is its index into the list of the names of the object's members in alphabetic order, and may therefore change when this list is altered in any way.

GetFocusObj Method 509

Applies To: ActiveXControl, Animation, Button, ButtonEdit, Calendar, ColorButton, Combo, ComboEx, CoolBar, DateTimePicker, Edit, Form, Grid, Group, Label, List, ListView, MDIClient, ProgressBar, PropertyPage, PropertySheet, RichEdit, Root, Scroll, SM, Spinner, Static, StatusBar, SubForm, TabBar, TabControl, ToolBar, ToolControl, TrackBar, TreeView, UpDown

Description

This method is used to obtain a ref to the object that currently has the input focus.

The GetFocusObj method is niladic.

The result is a ref. If there is no Dyalog APL GUI object with the input focus, the result is `⎕NULL`.

See also: *GetFocus* on page 1.

SetEventInfo Method 547

Applies To: ActiveXControl, OLEServer

Description

This method is used to register an event that may be generated by an ActiveXControl or OLEServer object.

A host application that wishes to attach a callback function to an event in a Dyalog APL ActiveXControl or OLEServer, needs to know the name of the event and the number and data types of any parameters that the event may supply. It also needs to know the data type (if any) of the result that the callback function may be expected to pass back to the control.

An ActiveXControl or OLEServer generates an event in the host application using `4 ⎕NQ`. The right argument is a vector whose first 2 elements are character vectors containing the names of the ActiveXControl or OLEServer and the event respectively. The parameters for the event are passed as additional elements in the argument.

Another way to think about it is that when you generate an event using `4 ⎕NQ`, you are effectively calling a function, of your specification, in the host application. To enable the host application to accept the function call, it needs to know the number of parameters that you will supply and their data types.

A further consideration is that if you specify that the data type of a parameter is a *pointer* (e.g. `'VT_PTR TO I4'`) this will allow a callback function to modify the parameter in-situ. If so, the result returned by `4 ⎕NQ` will be the modified values of any such parameters; this is a similar mechanism to `⎕NA`.

The argument to SetEventInfo is a 1, 2, 3 or 4-element array as follows:

`[1]`	Event name	character vector
`[2]`	Event info	nested array (see below)
`[3]`	Help ID	integer
`[4]`	DISPID	integer. See *DISPID (Dispatch ID)* on page 86

Event info

Event info, specifies an optional help string which describes what the event does, the data type of the result (if any) and the names and data types of its arguments.

If the event is fully described, each element of Event Info is a 2-element vector of character vectors. The first element contains the help string and the COM data type of the result that the callback function in the host application is expected to supply. Subsequent elements contain the name and COM data type of each of the parameters supplied by the event.

However, both the help string and the names of the parameters are optional and may be omitted. If so, one or more elements of *Event Info* may be a simple character vector.

Help ID

This is an integer value that identifies the help context id for the event within the help file associated with the HelpFile property of the ActiveXControl object. The value `¯1` means that no help is provided. APL stores this information in the registry from where it may be retrieved by the host application.

Example

The example *Dual* ActiveXControl, that is fully described elsewhere, generates a ChangeValue1 event. This event occurs when the user moves the thumb in a TrackBar that is internal to an instance of the ActiveXControl.

The external ChangeValue1 event is fired by an internal APL callback function (called `ChangeValue`) that is attached to ThumbDrag and Scroll events on the TrackBar object. The internal callback function is :

```
[0]   ChangeValue MSG
[1]   ⍝ Callback for ThumbDrag and Scroll
[2]   Value1←4 ⎕NQ'' 'ChangeValue1'(>¯1↑MSG)
[3]   CalcValue2
[4]   'V1'⎕WS'Text'(⍕Value1)
[5]   'V2'⎕WS'Text'(⍕Value2)
```

Note that `ChangeValue[2]` generates the external ChangeValue1 event by invoking `4 ⎕NQ`, passing it the new value provided by the TrackBar. However, the host application is permitted to modify that value, returning it in the result of `4 ⎕NQ`. This result, rather than the TrackBar value itself, is then used to update other (Label) controls in the object.

The following statements were used to declare the *ChangeValue1* event The event provides a single parameter named *Value1* that may be modified in-situ by a callback function in the host application. The callback is not, otherwise, expected to return a result.

```
      INFO←⊂'Occurs when the value of the control is
changed' 'VT_VOID'
      INFO,←⊂'Value1' 'VT_PTR TO VT_I4'
      F.Dual.SetEventInfo 'ChangeValue1' INFO
```

If the host application was Visual Basic, a suitable callback function might be:

```
Private Sub Dual1_ChangeValue1(Value1 As Long)
Value1=2*(Value1\2)
End Sub
```

This callback function receives the proposed new value of the control as the parameter Value1, and modifies it, forcing it to be an even number.

SetFnInfo Method 545

Applies To: ActiveXControl, OLEServer

Description

This method is used to describe an APL function that is to be exported as a method, a Property Get Function, or a Property Put Function of an ActiveXControl or OLEServer object.

An exported function must be a niladic or monadic defined function (dfns and derived functions are not allowed) and may optionally return a result. Ambivalent functions (functions with optional left argument) are allowed, but will be called monadically by the host application.

COM syntax differs from APL syntax in many ways and the SetFnInfo method is required to declare an APL function to COM in terms that COM understands. In particular, although monadic APL functions take just one argument, COM functions may take several parameters, and some may be optional.

A function exported by SetFnInfo will be called by a host application with the number of parameters that SetFnInfo has described. The argument received when the function is called by a host application, will be a nested vector of this length.

The argument to SetFnInfo is a 2, 3, 4, 5 or 6-element array as follows:

[1]	Function name	character vector
[2]	Function info	nested array (see below)
[3]	Help ID	integer
[4]	Function type	integer
[5]	Property name	character vector
[6]	DISPID	integer. See *DISPID (Dispatch ID)* on page 86

Function info

This specifies an optional help string which describes what the function does, the data type of the result (if any) and the names and data types of its arguments.

If the function syntax is fully described, each element of *Function Info* is a 2-element vector of character vectors. The first element contains the help string and the COM data type of the function's result. Subsequent elements contain the name and COM data type of each parameter.

However, both the help string and the names of the parameters are optional and may be omitted. If so, one or more elements of *Function Info* may be a simple character vector.

Consider a very basic function `ADD` in an ActiveXControl called `F.dbase`, that is designed to add a record to a personnel database. The database consists only of a list of names, ages and addresses.

Function `ADD` expects to be called with a name (character string), age (number) and address (character string), and returns a result 0 or 1 (Boolean) according to whether the record was successfully added. This function could be declared as follows:

```
      HELP←'Adds a new record to the personnel database'
      SPEC←⊂(HELP 'VT_BOOL')      ⍝ Result is Boolean
      SPEC,←⊂('Name' 'VT_BSTR')   ⍝ 1st param called
'Name' is a string
      SPEC,←⊂('Age' 'VT_I4')      ⍝ 2nd param called
'Age' is an integer
      SPEC,←⊂('Address' 'VT_BSTR')⍝ 3rd param called
'Address' is a string

      F.dbase.SetFnInfo 'ADD' SPEC
```

Alternatively, but much less helpfully, the function could be declared to take a single unnamed nested argument, leaving it to the host application programmer to guess at its structure :

```
      SPEC←⊂('' 'VT_BOOL')              ⍝ No help string,
result is Boolean
      SPEC,←⊂('' 'VT_ARRAY OF VT_VARIANT') ⍝ Param is a
nested array
      F.dbase.SetFnInfo 'ADD'SPEC
```

Help ID

This is an integer value that identifies the help context id within the help file associated with the HelpFile property of the ActiveXControl object. The value `¯1` means that no help is provided. APL stores this information in the registry from where it may be retrieved by the host application.

Function type

This specifies the type of function being exported. This is an integer with one of the following values:

1	Function is a *method*
2	Function is a *property get* function
4	Function is a *property put* function

In both these last two cases, the name of the property, which is totally independent of the name of the APL function, is given as *Property name*.

If omitted, the function type is *method*.

SetPropertyInfo Method 554

Applies To: OCXClass, OLEClient

Description

This method is used to redefine a property that is exported by a COM object. SetPropertyInfo is used to override the information provided by the object's Type Library.

The argument to SetPropertyInfo is a 2 or 3-element array as follows:

[1]	Property name	character vector
[2]	Property info	nested vector
[3]	Property function	integer
[4]	DISPID	integer. See *DISPID (Dispatch ID)* on page 86

For example, the Visible property exported by *Excel.Application* has the data type VT_BOOL and may be declared as follows:

```
'EX' □WC 'OLEClient' 'Excel.Application'
EX.SetPropertyInfo 'Visible' 'VT_BOOL'
```

Property function may be required if the property value is retrieved or set via a function. This typically applies if the property takes parameters and will result in the property being fixed as a function rather than as a variable. Such properties may have a PropertyGet function, a PropertyPut function and/or a PropertyPutByReference function. If so, it is necessary to say to which of these three functions the details apply. The value of *Property function* is an integer 2 (PropertyGet), 4 (PropertyPut), or 8 (PropertyPutByReference).

For example, the following statement declares the PropertyGet function for the Item property of the Fields collection of the OLE object DAO.DBEngine. This property takes an index (into the collection) and returns an object.

```
      Fields.SetPropertyInfo 'Item'('VT_DISPATCH' 'VT_
I4')2
```

Chapter 6:

UNIX Specific Features

Summary

This section summarises the changes specific to Dyalog APL Version 16.0 on UNIX-based platforms. This list currently consists of:

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

Hardware Requirements

AIX

For AIX, Version 16.0 requires AIX 6.1 or higher, and a POWER6 chip or higher.

Raspberry Pi

On the Raspberry Pi, Dyalog (32-bit Unicode) supports Raspbian Wheezy and Jessie.

Non-Pi Linux

For non-Pi Linux, Version 16.0 only exists as 64-bit interpreters - there are no 32-bit versions. It is built on RedHat 6, and runs on all recent distributions, including Ubuntu 14.01 and openSUSE 13.2. Contact Dyalog for information about other platforms.

Mac OS X

16.0 requires Mac OS X Yosemite onwards. The target Mac must have been introduced in 2010 or later.

Obtaining the exit code from ⎕SH

In Version 16.0, ⎕DMX.Message contains the exit code from all calls to ⎕SH which result in an error:

```
      z←⎕SH'exit 17'
DOMAIN ERROR
      ⎕DMX.Message
Command interpreter returned failure code 17
```

In previous versions of Dyalog APL it was necessary to use an expression such as

```
      ⎕SH'mycmd; echo $? ; exit 0'
```

to obtain the exit code.

Dyalog intends to return the exit code in a more usable form in a future version.

Suppressing error traps in the session

600⌶ allows the user to control what happens when an error is generated in the session when there are functions suspended which have traps assigned.

Under Windows this can be toggled using a MenuItem in the session; on non-Windows platforms you must use 600⌶ to achieve the same result. Note that if SALT is active, setting 600⌶0 (disable all traps everywhere) will result in the SALT-related editor backend code suspending, which is undesirable. See the *Dyalog Language Reference* or the I-Beam section of the online help for more information.

RIDE 3.0 and Dyalog APL 16.0

Note that Dyalog Version16.0 supports RIDE 3 only; RIDE 2 is not supported.

The *Dyalog RIDE Reference Guide* details how to configure the APL session to support the underscored alphabet; contact support@dyalog.com if you wish to be able to generate key-chords which result in the underscored alphabet being entered into APL.

Linux Window Managers and APL characters

If your Linux window manager does not include support for APL characters (Gnome is an example), then the first time that you run Dyalog having started the window manager afresh, you must run

```
$ dyalog -kbd
```

Subsequent invocations of dyalog should not require this flag.

Change to event handling

The handling of events from "GUI" objects (`TCPSockets` and `Timers`) has been altered in version 15.0 on non-Windows platforms to eliminate the possibility of Dyalog hanging. In previous versions all GUI events were added to a pipe, and processed only when `⎕DQ` was called. This could lead to a situation where the pipe filled, at which point Dyalog would hang. In version 15.0 the code has been changed so that at the end of each line of APL code, any valid events are moved to an internal APL message queue. This brings the non-Windows versions more in line with the Windows version of Dyalog.

Index