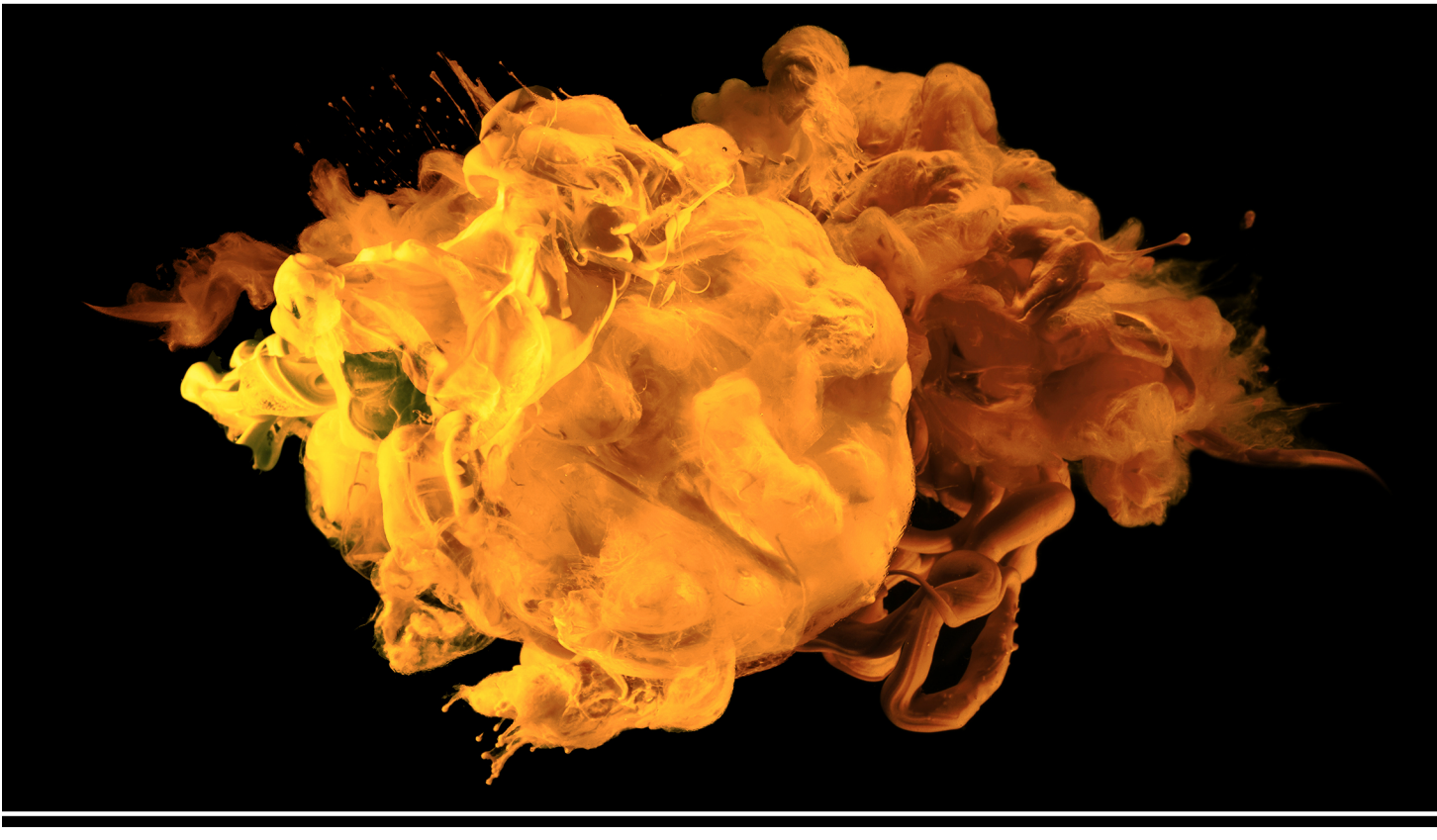
Dyalog Release Notes

Dyalog version 18.0



DYALOG

The tool of thought for software solutions

Dyalog is a trademark of Dyalog Limited
Copyright © 1982-2020 by Dyalog Limited
All rights reserved.

Dyalog Release Notes

Dyalog version 18.0
Document Revision: 20240212_180

Unless stated otherwise, all examples in this document assume that `⎕IO ⎕ML ← 1`

No part of this publication may be reproduced in any form by any means without the prior written permission of Dyalog Limited.

Dyalog Limited makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Dyalog Limited reserves the right to revise this publication without notification.

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

TRADEMARKS:

SQAPL is copyright of Insight Systems ApS.
Array Editor is copyright of davidliebtag.com
Raspberry Pi is a trademark of the Raspberry Pi Foundation.
Oracle®, Javascript™ and Java™ are registered trademarks of Oracle and/or its affiliates.
UNIX® is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.
Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.
Windows® is a registered trademark of Microsoft Corporation in the United States and other countries.
macOS® and OS X® (operating system software) are trademarks of Apple Inc., registered in the U.S. and other countries.

All other trademarks and copyrights are acknowledged.

Contents

Chapter 1: Highlights **1**
Key Features 1
Introducing Configuration Files 4
Multi-line Session Input 7
Extension to Where 8
Extensions to Mix 8
Regex Variant Option 9
Serialising Namespaces 10
Load 12
LX 12
Bug Fixes 13
Announcements 14
System Requirements 15
Interoperability 16

Chapter 2: Configuration **21**
Session Initialisation 21
File Associations 23
Configuration Parameters 23
New Configuration Parameters 25
Dyalog_NETCore 25
APL_TextInAplCore 25
Configuration Files 25
Run-Time Applications and Components 29

Chapter 3: Language Reference Changes **35**
Language Changes 35
Atop 36
Beside 36
Bind 37
Constant 38
Over 38
Unique Mask 39
Partitioned Enclose 40
Case Convert 42
Date-time 43
Format Date-time 50

Chapter 4: Object Reference Changes **59**

ExecuteJavaScript 59
InterceptedURLs 59
WebSocketUpgrade 61

Index **63**

Chapter 1:

Highlights

Key Features

Installation and Configuration

- Version 18.0 introduces platform-independent configuration files. See *Introducing Configuration Files* on page 4.
- During installation, `setup.exe` associates a number of new file extensions with the Dyalog APL Editor `dyaedit.exe`. See *File Associations* on page 23.
- A new feature is provided to instantiate code in the Session (`⎕SE`) when the Dyalog program starts. See *Session Initialisation* on page 21.

Executing Source Code Files

- Version 18.0 provides the capability of launching the Dyalog program on text files containing APL source code. This is achieved by two new parameters which may be specified on the command-line. See *Load* on page 12 and *LX* on page 12.

.NET Core Bridge

- Under Microsoft Windows, Linux (including the Raspberry Pi) and macOS, Version 18.0 includes a bridge to the .NET Core, providing APL developers with access to a large and rapidly growing collection of APIs and frameworks.
- The .NET Core bridge will be almost identical in functionality to the existing bridge to the .NET Framework, which is only available under Windows,. However, in Version 18.0 it still lacks some features, most importantly the ability to encapsulate APL code as .NET Core assemblies.
- The .NET Core bridge is documented in the dotNET Core Interface Guide. See *Dyalog NETCore* on page 25.

New Language Features

- New monadic ≠ function. See *Unique Mask* on page 39.
- Version 18.0 introduces 3 new operators. See *Atop* on page 36, *Constant* on page 38 and *Over* on page 38.
- New System Function ⎕C. This new function is intended to replace 819⌶ which is deprecated. See *Case Convert* on page 42.
- New system function ⎕DT. See *Date-time* on page 43.
- New I-beam function. See *Format Date-time* on page 50.

Improved Language Features

- ⎕JSON provides a new **HighRank** option to handle arrays with rank >1 automatically, and a new **Dialect** option that permits JSON5 extensions.
- ⎕NPUT provides a new **NEOL** option that specifies how embedded line separators are treated.
- The Where function (monadic ⍸), which previously accepted only a Boolean argument, has been extended to allow non-negative integers. See *Extension to Where* on page 8.
- Partitioned enclose has been extended to permit empty result elements. See *Partitioned Enclose* on page 40.
- Mix has been improved so that certain APL2 extensions which were provided only when ⎕ML≥2 are now provided at all levels of ⎕ML. See *Extensions to Mix* on page 8.
- ⎕S and ⎕R now have a **Regex** Variant option that can be used to disable regular expression matching. See *Regex Variant Option* on page 9.
- The rules concerning the serialisation of arrays that contain external references have been re-defined. See *Serialising Namespaces* on page 10.

Documentation Changes

- The Composition operator, which was previously described in terms of 4 forms has been renamed Beside (forms I and IV) and Bind (forms II and III). Only the descriptions have been updated, the features and functionality have not changed. See *Beside* on page 36 and *Bind* on page 37.
- The documentation for Run Time applications has been updated to take advantage of the new configuration files. See *Run-Time Applications and Components* on page 29.

UI Improvements

- The Session now permits multi-line input. See *Multi-line Session Input* on page 7.
- The Editor now identifies:
  - matching occurrences of the word under the caret (except the actual instance under the caret)
  - matching parentheses, brackets and braces
  - all control words associated with the one under the caret (e.g. if the caret is on `:If`, then `:Else` and `:EndIf` are highlighted)
- The Find/Replace Dialog box now has an option to move to the next occurrence of the target string following a replace operation.
- The `MO` (Goto matching outline) keystroke, now includes nested control words.

HTMLRenderer Changes

- The protocol for initiating a WebSocket has been improved to permit manual control. See *InterceptedURLs* on page 59 and *WebSocketUpgrade* on page 61.
- The HTMLRenderer provides a new method. See *ExecuteJavaScript* on page 59.
- A number of other minor changes have been made to WebSocket events.

New Configuration Parameters

- There is a new parameter to select the .NET Core interface in place of the .NET Framework. See *Dyalog_NETCore* on page 25.

Introducing Configuration Files

Background

The operating systems on which Dyalog APL has been available through the years have used a variety of mechanisms for system configuration. Environment variables have been available since the beginning, first under UNIX and later under Microsoft DOS and IBM OS/2. For a while, INI files became the norm under Microsoft Windows, and were briefly used by Dyalog APL before the Registry replaced them.

Today, Dyalog APL for Windows relies on the Windows Registry for configuration (but allows environment variables to override settings). Other platforms mostly use environment variables, which in practice require a script surrounding the invocation of the Dyalog program. Settings can also be provided on the command line, but this cannot be done in the same way on all platforms, and some platforms limit the length of a command line.

In order to simplify cross-platform deployment of applications, Version 18.0 introduces a new configuration mechanism based on JSON5 text files. This is a superset of JSON which is easier to read and write, and allows the inclusion of comments.

Benefits of Configuration Files

The key benefits of configuration files include:

- Configuration files are text-based, which means they can easily be managed along with the source code for an application, using industry standard tools for source code management and continuous integration.
- *Application configuration files* can be placed in application folders and define all configuration settings that each application relies on.
- *User configuration files* can provide settings that are the same for all applications. (Typically, these files will be used to configure the development environment.)
- Configuration of the interpreter can be done in the same way on all supported platforms.
- Version 18.0 also adds an option to allow Dyalog APL to be launched not only on a binary workspace file, but also on a text file defining a function, namespace or class. If a configuration file exists with the same name as the loaded file (but with a `.dcfg` extension), it will configure the interpreter for that use.
- Configuration files are easy to read, and to write by hand or using `⎕JSON`, which has been extended to support JSON5.

- Both application and user configuration files can *cascade*, overriding settings defined in a more generic configuration file, simplifying the configuration of components which share some configuration.

Current and Future Policy

The new configuration files work in exactly the same way on all platforms.

On non-Windows platforms, Version 18.0 will create and initialise a User Configuration file `~/.dyalog/dyalog.180U64.dcfg` (for a Unicode 64-bit installation), and the default launch script will set the **USERCONFIGFILE** parameter to enable it.

Under Windows, because the Windows Registry is a convenient and familiar mechanism for maintaining user preferences, especially those relating to the IDE, Version 18.0 continues to record user settings in the Windows Registry. As before, the Configuration dialog reflects the Windows Registry settings and ignores parameters defined in other ways.

The current expectation is that future releases will split the contents of the existing Windows Registry between settings that configure the development environment (which will remain in the registry) and settings that configure the interpreter (which will be relocated to a user configuration file, similar to the one created for Version 18.0 on non-Windows platforms).

Dyalog recommends that you immediately start using configuration files for all run-time applications, and eliminate the use of the Windows Registry or Environment Variables for this purpose.

Implementation

A configuration file is an optional text file containing configuration parameters and values. It may cascade, i.e. it can extend (inherit) configuration values from other configuration files, and supplement and/or override them.

Configuration files use JSON5 (a superset of standard JSON) syntax. These files are portable across all systems supported by Dyalog.

Precedence

All existing mechanisms for configuration continue to be supported. The following precedence table shows how configuration files have been inserted into the existing precedence rules when a setting is defined in multiple places:

- Command-line settings override
- *Application configuration file* settings, which override
- Environment variable settings, which override
- *User configuration file* settings, which override
- Settings in the registry (Windows only), which override
- Built-in defaults

For further information, see *Configuration Files* on page 25.

Multi-line Session Input

Extended multi-line input is a new approach to handling multi-line statements in the session. The details here are intentionally imprecise because the behaviour is likely to change, and it is intended to be intuitive.

This new feature is optional and is controlled by the value of the **Dyalog_ LineEditor_Mode** parameter which by default is 0 (off). To enable the new behaviour, you must set the parameter to 1.

Changes to the line editor

- Functions entered into the session with the Line (`∇`) Editor are syntax coloured as a whole.
- the `[]` syntax to manipulate lines is deprecated. Instead, the `IL` and `EL` keystrokes (see below) should be used.

Entry of control structures and multi-line dfns

Broadly speaking, if the interpreter detects an un-terminated control structure or dfn on a single line of input it:

- Enters a new multi-line mode which accumulates lines until the control structure or dfn is terminated.
- The completed block of lines is then executed as if it were a niladic defined function.

In all cases:

- The session considers all related lines to be a *group*.
- Grouped lines are syntax coloured as a whole.
- If a change is made to one or more lines in a group then the whole group is marked to be re-executed when `ER` is pressed.
- Lines can be inserted into a group with the `IL` keystroke.
- The current line can be cleared with the `EL` keystroke. (Note, this exists because it is not possible to UNDO a delete line in the session).

Extension to Where

The Where function (monadic ⍸), which previously accepted only a Boolean argument, has been extended to allow non-negative integers.

The model for Where can be expressed as `{(,⍵)/,⍳⍴⍵}`, and the same model may be used to explain the extension. Note that a neater way to write this using the new Over operator is `{⍵/⍤⊖,⍳⍴⍵}`.

Examples:

```
      ⍸ 0 1 0 2
2 4 4

      {⍵/⍤⊖,⍳⍴⍵} 0 1 0 2
2 4 4
      ⍸2 2⍴0 1 2 3
```

1 2	2 1	2 1	2 2	2 2	2 2

Extensions to Mix

Certain APL2 extensions which were only previously implemented when `⎕ML≥2` have been implemented at all levels of `⎕ML`.

```
      ⎕ML←2
      ⍴⊃[1]2 3⍴⊂4⍴0      ⍝ APL2 style, inserts new axis
4 2 3
      ⎕ML←0
      ⍴↑[1]2 3⍴⊂4⍴0      ⍝ Now works when ⎕ML is <2
4 2 3

⎕ML←2
      ⍴⊃[1 3]2 3⍴⊂4 5⍴0 ⍝ insert new axes
4 2 5 3
      ⎕ML←0
      ⍴↑[1 3]2 3⍴⊂4 5⍴0 ⍝  Now works when ⎕ML is <2
4 2 5 3
```

Regex Variant Option

⎕S and ⎕R now have a **Regex** Variant option that can be used to disable regular expression matching.

Regex Option

This option may be used to disable regular expression matching which is enabled by default. It is a singleton Boolean value that applies to both search and transformation patterns, or a 2-element vector of Boolean values that applies to them separately.

1	Regular expression matching is applied.
0	regular expression matching is disabled.

Examples

```
      STR
The cat sat on the mat

      ('.at' ⎕S '\u0') STR
 CAT  SAT  MAT
      ('.at' ⎕S '\u0' ⍠('Regex' 0)) STR

      ('.at'⎕R'\u0')STR
The CAT SAT on the MAT
      ('.at'⎕R'\u0'⍠('Regex' (1 0)))STR
The \u0 \u0 on the \u0
```

Serialising Namespaces

The Serialisation of an array is its conversion from its internal representation, which may contain pointers to other structures in the workspace, into a self-contained series of bytes. This allows the array to be written to a file, transmitted over a socket or used in a variety of other ways. The de-serialisation of an array is the conversion back to an internal format whose content and structure is identical to the original array.

If an array contains a reference to a namespace or object that is within the same array, it can be serialised and de-serialised normally.

If an array contains a reference to a namespace or object that is not internal to the array itself, this presents a problem, which is resolved as follows:

1. If the reference is a direct reference to Root (#) or to ⎕SE, it is serialised as a reference to that symbol, but the contents of # or ⎕SE are not included. When the array is de-serialised, this results in a reference to the Root (#) or ⎕SE in the current workspace. The newly reconstituted array is not strictly identical to the original because the contents of # or ⎕SE may be different.
2. If the reference is to an arbitrary external namespace or object, a copy of that object is included but its path is discarded. When the array is de-serialised, the copy is reconstituted as a sibling (i.e. as a child of the same parent as the de-serialised array). In this case the contents of the external namespace or object are preserved, but not its path. The newly reconstituted array is not strictly identical to the original because the path to the external reference has changed.
3. If however, the external namespace or object itself contains an external reference, the operation fails with `DOMAIN ERROR`.

The following example uses 220⌶ but applies equally to an array serialised by, for example ⎕FAPPEND.

Examples:

```
        'A' ⎕NS ''
        'B' ⎕NS ''
        'C' ⎕NS ''
         A.b←B
         B.c←C
         s←1 (220⌶)A
        )erase A B C
        )obs

       New←0(220⌶)s
       New
#.A
       New.b
#.B
       New.b.c
#.C

       )clear
clear ws
       'A' ⎕NS ''
       'B' ⎕NS ''
       'X'⎕NS ''
       'X.C'⎕NS ''
       A.b←B
       B.c←X.C
       s←1(220⌶)A
DOMAIN ERROR: Namespace is not self contained
       s←1(220⌶)A
          ^
```

Note that a successful 0(220⌶) does not mean that a 1(220⌶) on the result will succeed. If the original reference was to, say, the MenuBar of ⎕SE you cannot reconstitute that in #.

Load

This parameter is a character string that specifies the name of a workspace, or a text file containing APL source code, to be loaded when Dyalog starts. It will normally be specified on the command line or in a Configuration file.

Dyalog determines whether or not the file is a workspace by its internal signature. If it is a workspace, the expression specified by its Latent Expression ⎕LX will be executed by default. This expression may be overridden by the **LX** parameter.

Otherwise, if the file extension is `.aplf` `.aplc` or `.apln` Dyalog will attempt to fix the contents of the file as APL source code. If successful, it will by default run the expression shown in the table below, where `filename` is the file name specified by the Load parameter without its extension. This expression may be overridden by the **LX** parameter.

File Extension	Type	Expression
`.aplf`	Function source code	`filename ⍬⊂''`
`.aplc`	Class source code	`filename.Run ⍬⊂''`
`.apln`	Namespace source code	`filename.Run ⍬⊂''`

Notes:

- The **Load** parameter overrides a workspace name specified as the last item on the command line.
- The option to load APL source code from a text file applies only to the Unicode version and is not supported by the Classic version.

LX

This parameter specifies an expression to be executed after Dyalog has started and loaded a workspace or a text file containing APL source code. Also see *Load* on page 12. This expression is run only on Dyalog start-up and overrides the workspace latent expression ⎕LX.

The LX parameter applies only to the development version of Dyalog and is ignored in run-time applications.

The LX parameter is ignored when a workspace is loaded other than at start-up of the Dyalog program.

Bug Fixes

A number of bug fixes implemented in Version 18.0 may change the way that existing code operates and are therefore documented in this section.

Change to ⍓S

⍓S *with a transformation pattern* returns a vector of character vectors, one per match. When there are no matches it should therefore return a zero-length array of character vectors, but it used to erroneously return an empty character vector.

```
      ]box on -style=max

      ('X'⍓S'&')'Y' ⍝ Old behaviour
┌⊖┐
└─┘

      ('X'⍓S'&')'Y' ⍝ Corrected behaviour:
┌⊖┐
│┌⊖┐│
│└─┘│
└∈──┘
```

Note: there is no change to the result if used with a transformation code or transformation function, and ⍓R is not affected.

Minor Correction to ⍞

Version 17.1:

```
      ⍞←20↑'hello' ⋄ x←⍞
hello                     ⍝ user presses ER
      x
hello
      ⍴x
5
```

Version 18.0

```
      ⍞←20↑'hello' ⋄ x←⍞
hello                     ⍝ user presses ER
      x
hello
      ⍴x
20
```

APLTextInAplCore parameter

The default for this parameter is now 1 on all platforms.

Announcements

Withdrawal of Support for Version 16.0

The supported Versions of Dyalog APL are now Version 18.0, 17.1, and 17.0. Version 16.0 and earlier versions are no longer supported.

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	Windows 8/Server 2012
AIX	POWER 8 (if you have a POWER 7 requirement, please contact sales@dyalog.com).
Linux	Versions of distributions which are in standard support for at least 3 months from when the next version of Dyalog is released
macOS	macOS Catalina
Raspberry Pi	Raspbian Buster

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

Planned Hardware Requirements for next version

The same as Dyalog Version 18.0.

Case Convert

`(819⌶)` has been replaced by `⎕C` and is deprecated. It will be removed in a future release. See *Case Convert* on page 42.

⎕SE.⎕WX

The default value of `⎕SE.⎕WX` is now 3, matching the recommended default value.

System Requirements

Microsoft Windows

Dyalog APL Version 18.0 is supported on versions of Microsoft Windows from Windows 8 or Windows Server 2012, up to and including Windows 10 and Windows Server 2016.

Microsoft .NET Interface

Dyalog APL Version 18.0 .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]) Version 18.0 requires .NET Version 4.5. The Syncfusion libraries supplied with Version 18.0 require .NET 4.6.

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 18.0 requires AIX 7.2 or higher, and a POWER8 chip or higher.

Raspberry Pi

On the Raspberry Pi, Dyalog 32-bit Unicode supports Raspbian Buster or later (Bookworm requires Dyalog version 18.0.48479).

Non-Pi Linux

For non-Pi Linux, Version 18.0 only exists as 64-bit interpreters - there are no 32-bit versions. It is built on Ubuntu 18.04; it should run on all recent distributions. See https://forums.dyalog.com/viewtopic.php?f=20&t=1652 for a list of tested platforms.

macOS/Mac OS X

Version 18.0 requires macOS High Sierra or later. 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.

Interoperability

Introduction

Workspaces and component files are stored on disk in a binary format (illegible to text editors). This format differs between machine architectures and among versions of Dyalog. For example, a file component written by a PC may well have an internal format that is different from one written by a UNIX machine. Similarly, a workspace saved from Dyalog Version 18.0 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 and the following sections describe limitations in interoperability:

Code and ⎕ORs

Code that is saved in workspaces, or embedded within ⎕ORs 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 ⎕ORs 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 interoperability 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.

Full cross-platform interoperability of component files is only available for large-span component files.

Null Items (⎕NULL) and Compressed Components

⎕NULLs and components from compressed component files that were created in Version 18.0 and later can be brought into Versions 16.0, 17.0 and 17.1 provided that the interpreters have been patched to revision 38151 or higher. Attempts to bring ⎕NULL or compressed component into earlier versions of Dyalog APL or lower revisions of the aforementioned versions will fail with:

```
DOMAIN ERROR: Array is from a later version of APL.
```

Object Representations (⎕OR)

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

It is no longer possible to *create* or write to small-span (32-bit) files; however it is still currently possible to *read* from 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 subject to the same restrictions as small-span component files regarding Unicode data. 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

There is complete interoperability 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

Two editions are available on some platforms. Unicode editions work with the entire Unicode character set. Classic editions (which are only available to commercial and enterprise users for legacy applications) 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 will issue a TRANSLATION ERROR if it attempts to read a component containing Unicode data that is not in ⎕AVU from a component file.

A TRANSLATION ERROR will also be issued when a Classic edition attempts to)LOAD or)COPY 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 ERRORs can occur on transmission or reception in the same way as when updating or reading a file component.

The symbols ⊆, ⍳, ⍤, ⌸, ⌸, ⌺ and ⍥ used for the Nest (Interval Index) and Where (Partition) functions, the Rank, Variant, Key, Stencil and Over operators respectively are available only in the Unicode edition. In the Classic edition, these symbols are replaced by ⎕U2286, ⎕U2378, ⎕U2364, ⎕U2360, ⎕U2338, ⎕U233a and ⎕U2365 respectively. In both Unicode and Classic editions Variant may be represented by ⎕OPT.

Very large array components

An attempt to read a component greater than 2GB in 32-bit interpreters will result in a WS FULL.

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. Under Microsoft Windows, Session files may only be used by the architecture (32-bit-or 64-bit) of the Version of Dyalog that saved them.

Chapter 2:

Configuration

Session Initialisation

Background

This enhancement aims to provide a standard, documented mechanism that will allow Dyalog and users to add content to the session (⎕SE) upon start-up of a Dyalog session, simply by placing the code to be loaded in a directory. The directory may be in a default location, or be specified using configuration parameters.

Each time Dyalog starts it loads and executes an initialisation file whose name is defined by the **DyalogStartup** parameter. If this is undefined, the default file is named `startup.dyalog` in the Dyalog directory. This process was introduced in Version 17.1 to load Link. From Version 18.0, the code defined in `startup.dyalog` also performs Session initialisation.

Implementation

Code to be installed in ⎕SE is specified in APL source code files contained in *Session initialisation* directories identified by the **DyalogStartupSE** parameter. If this parameter is not specified, the default is a directory named `StartupSession` located in three standard locations as described below.

Only content stored in files matching the wildcard patterns `*.dyalog` and `*.apl?` will be loaded. All such files must be appropriate for ⎕FIX.

For each sub-directory in a Session initialisation directory, a corresponding namespace is created in ⎕SE, and any source code files in these sub-directories will be fixed in their respective corresponding namespaces. There is currently no support for additional subdirectories inside these subdirectories, although this feature is planned.

The Session initialisation directories are processed in order and code defined in each directory will replace code with the same name defined previously. In effect, this means that user-supplied content can replace content supplied by Dyalog Ltd. and version-specific content can replace version-agnostic content.

Default Session Initialisation Directories

If the **DyalogStartupSE** parameter is undefined, APL looks for Session initialisation directories named `StartupSession` in the following three locations, and processes them in that order:

1. The Dyalog installation directory (which contains the dyalog executable)
2. A version-agnostic sub-directory in the user directory (the standard directory for user-related Dyalog APL files)
3. A version-specific sub-directory in the user directory, whose name is derived as described below.

Under Windows these might be:

1. `C:\Program Files\Dyalog\Dyalog APL-64 18.0 Unicode`
2. `C:\Users\Pete\Documents\Dyalog APL Files`
3. `C:\Users\Pete\Documents\Dyalog APL-64 18.0 Unicode Files`

The version-specific name is :

```
Dyalog APL{bit} {version} {edition}
```

where:

- {bit} is "`-64`" if 64-bit version, otherwise nothing
- {version} is the main and secondary version numbers of dyalog.exe separated by ".".
- {edition} is "`Unicode`" for the Unicode Edition, otherwise nothing

On non-Windows platforms these might be:

1. `/opt/mdyalog/18.0/64/unicode`
2. `/home/Pete/dyalog.files`
3. `/home/Pete/dyalog.180U64.files`

The version-specific name is:

```
dyalog.{version}{edition}{bit}.files
```

where:

- {version} is main and secondary version numbers without separator
- {edition} is an uppercase "`U`" or "`C`" for Unicode/Classic
- {bit} is "`64`" or "`32`" depending on bit-width

File Associations

During installation, `setup.exe` associates a number of file extensions with Dyalog applications.

Workspace files with extension `.dws` and files with extension `.dyapp`, which are used to bootstrap SALT-based applications[1], are associated with `dyalog.exe`.

The following file types are associated with the Dyalog APL Editor `dyaedit.exe`. They are used by various source code management tools, including Link[2] and SALT[3] and 3rd party tools like Acre Desktop[4].

`.aplf`	`Functions`
`.aplo`	`Operators`
`.apln`	`Namespaces`
`.aplc`	`Classes`
`.apli`	`Interfaces`
`.dyalog`	`Generic`

Additionally, Link uses `.apla` files to store serialised arrays. These are likely to become associated with `dyaedit.exe` in a future release.

Configuration Parameters

Introduction

Dyalog APL is customised using a set of configuration parameters. These may be defined in a number of ways, which take precedence as follows:

- Command-line settings
- Application configuration file settings
- Environment variable settings
- User configuration file settings
- Settings in the registry section defined by the **IniFile** parameter (Windows only)
- Built-in defaults

[1] http://docs.dyalog.com/latest/SALT User Guide.pdf#page=12

[2] https://github.com/Dyalog/link/blob/master/help/Link.md

[3] http://docs.dyalog.com/latest/SALT User Guide.pdf

[4] https://github.com/the-carlisle-group/Acre-Desktop/wiki

This scheme provides a great deal of flexibility, and a system whereby you can override one setting with another. For example, you can define your normal workspace size (*maxws*) in the Registry, but override it with a new value specified on the APL command line. The way this is done is described in the following section.

Furthermore, you are not limited to the set of parameters employed by APL itself as you may add parameters of your own choosing.

Although for clarity parameter names are given here in mixed case, they are case-independent under Windows. Under UNIX and Linux, if Dyalog parameters are specified as environment variables they must be named entirely in upper-case.

Note that the value of a parameter obtained by the GetEnvironment method (see *Object Reference Guide: GetEnvironment Method*) uses exactly the same set of rules.

The following section details those parameters that are implemented by Registry Values in the top-level folder identified by **IniFile**. Values that are implemented in sub-folders are *mainly* internal and are not described in detail here. However, any Value that is maintained via a configuration dialog box will be named and described in the documentation for that dialog box in The APL Environment.

Specifying Size-related Parameters

Several of the configuration parameters define sizes.

The value of the parameter must consist of an integer value, optionally followed immediately by a single character which denotes the units to be used. If the value contains no character the units are assumed to be KiB.

Valid values for units are:

K(KiB), M(MiB), G(GiB), T(TiB), P(PiB) and E(EiB).

Specifying an invalid value will prevent Dyalog APL from starting.

Changing parameter values in the Registry

You can change parameters in the Registry in one of two ways:

- Using the Configuration dialog box that is obtained by selecting *Configure* from the *Options* menu on the Dyalog APL/W session.
- By directly editing the Windows Registry using `REGEDIT.EXE` or `REGEDIT32.EXE`. This is necessary for parameters that are not editable via the Configuration dialog box.

New Configuration Parameters

Dyalog_NETCore

This Boolean parameter specifies whether the .NET Core interface is enabled. On Windows the default is 0 which disables the .NET Core interface in favour of the .NET Framework interface. If it is set to 1, Dyalog uses .NET Core instead of the .NET Framework.

On other platforms which support the .NET Core, the default is 1.

APL_TextInAplCore

This Boolean parameter specifies whether or not certain information is written to an *aplcore* file when a system error occurs. The default is 1.

Configuration Files

Introduction

A configuration file is an optional text file containing configuration parameters and their values. It may cascade, i.e. it can extend (inherit) configuration values from other configuration files, and supplement and/or override them.

Configuration files use JSON5 (a superset of standard JSON) syntax, as described below. These files are portable across all systems supported by Dyalog.

Names of configuration parameters defined in Configuration files may be specified in any combination of alphabetic case.

Dyalog processes up to two kinds of configuration file (each of which may cascade):

1. An application configuration file which contains configuration values associated with a specific application
2. A user configuration file which defines configuration values for the current, and possibly only, user of the system.

Application Configuration File

When Dyalog starts, it derives the name of the application configuration file as follows:

- The name in the configuration parameter **ConfigFile** if it is set, otherwise
- The name of the workspace or script loaded at start-up using the **Load** parameter, with the extension replaced by `.dcfg`, if that file exists, otherwise
- Nothing.

User Configuration File

The name of the user configuration file is specified by the **UserConfigFile** parameter. Under Windows, this parameter is not set by default but may be defined by the user.

Precedence

Configuration files supplement existing methods of defining parameters. The following precedence table shows the order of precedence when a setting is defined in multiple places:

- Command-line settings override
- Application configuration file settings, which override
- Environment variable settings, which override
- User configuration file settings, which override
- Settings in the registry (Windows only), which override
- Built-in defaults

Configuration Files and The Configuration Dialog

The Configuration Dialog reflects the values of parameters stored in the Windows Registry and ignores overriding values defined on the command-line, in configuration files or in environment variables. If the user changes parameters using the Configuration Dialog, the new values are recorded in the Registry, but remain overridden by those that take precedence.

Configuration File Structure

Configuration files define configuration parameters using JSON5. A JSON object contains data in the form of key/value pairs and other JSON objects. The keys are strings and the values are the JSON types. Keys and values are separated by colon. Each entry (key/value pair) is separated by comma.

The top-level object defines an optional key named **Extend** and an optional object named **Settings**.

Extend is a string value containing the name of a configuration file to import. The extended (imported) file may in turn extend another configuration file. Configuration values from the imported file(s) may be overridden by redefining them. The file name is implicitly relative to the name of the file which imports it. Any file name extension must be explicitly specified.

Settings is an object containing the names of configuration parameters and their values. The values may be:

- A string
- A number
- An array of strings

The names and values correspond to configuration parameters, but names are not case sensitive. Any named values may be defined; an APL application may query the values using `+2⎕NQ'.' 'GetEnvironment' name`, or using the `]config` user command. Note that `GetEnvironment` returns the value in use as defined by the precedence rules (see *Precedence* above).

Example

```
      +2 ⎕NQ '.' 'GetEnvironment' ('MaxWS' 'Captions\Session')
```

256M	My Dyalog V18.0 Session

```
      ]config MaxWS Captions\Session
```

MaxWS	256M
Captions\Session	My Dyalog V18.0 Session

A warning will be given if names are redefined in the same configuration file; the second and subsequent definitions will be discarded.

File Names

Pathnames specified in configuration files should be specified using portable forward slashes "/" rather than back-slashes "\" as the latter are used as escape characters by JSON.

`WSPATH: ["c:/Dyalog18.0"]` or `WSPATH: ["c:\\Dyalog18.0"]` specifies the file `c:\Dyalog18.0`.

whereas,

`WSPATH: ["c:\Dyalog18.0"]` means `c:Dyalog18.0`.

Example

```
{
  Extend: "my_default_configuration.dcfg",

  Settings: {
      // maximum workspace
      MAXWS: "2GB",
      WSPATH: ["/dir1", "/dir2", ""],
      UserOption: 123,
      ROOTDIR: "/my/root/directory",
      // references to other configuration parameters
      FNAME: "[rootdir]/filename",
  }
}
```

Arrays

An array may be used to define file paths etc. For example,

```
WSPATH: ["/dir1", "/dir2"]
```

The only parameters which may be defined as arrays are **WSPATH**, **WSEXT** and **CFEXT**.

References to other Configuration Parameters

Configuration parameters which are string values may include references to other configuration parameters (regardless of where they are defined) using square bracket delimiters. For example:

```
MySetting: "[DYALOG]/MyFile"
```

will replace `[DYALOG]` with the value of the **DYALOG** configuration value.

Note that:

- If the referenced configuration parameter is not defined then no substitution will take place; the reference, including square bracket delimiters, will remain in place.
- To include square brackets in a string, prefix the '[' with a '\' character.

Nested Structures

Some parameters are stored in sub-folders in the Windows Registry. Currently, all such parameters used by Dyalog APL itself relate to the Windows IDE, but you can create your own application-specific structures..

The Configuration file supports this structure by defining an object that corresponds to a Registry sub-folder. For example:

```
Captions: {
    Session: "My Dyalog Session",
    Status: "My Status window",
}
```

```
      +2 ⎕NQ '.' 'GetEnvironment' 'Captions\Session'
My Dyalog Session
```

Run-Time Applications and Components

Using Dyalog APL you may create different types of run-time applications and components. Note that the distribution of run-time applications and components requires a Dyalog APL Run-Time Agreement. Please contact Dyalog or your distributor, or see the Dyalog web page for more information.

The various types of run-time applications and components are as follows:

1. Workspace or source code run-time
2. Stand-alone run-time
3. Bound run-time
4. Out-of-Process COM Server
5. In-Process COM Server
6. ActiveX Control
7. Microsoft .NET Assembly

All but the first of these are made using the *Export* dialog box accessed from the *File/Export* menu item of the Session window.

Configuration Parameters

Configuration parameters for these run-time applications, both for the Dyalog engine and for your own application settings, may be specified in a number of ways. See *Configuration Parameters* on page 23.

Nevertheless, it is strongly recommended that you use Configuration files. In this section we will discuss only Application Configuration files, although User Configuration files may be used as well.

Workspace or source code based run-time

A workspace or source code based run-time application consists of the Dyalog APL Run-Time Program (Run-Time EXE), a separate workspace or text file containing APL source code, and an optional configuration file. To distribute your application, you need to supply and install:

1. your workspace or source code
2. the Run-Time EXE
3. a configuration file (optional)
4. whatever additional files that may be required by your application
5. a command-line to start the application

The command-line for your application invokes the Run-Time EXE and directly or indirectly specifies the name of the workspace or source code file and the optional configuration file. You will need to associate your own icon with your application during its installation.

The name of the workspace or source code file may be specified by the **Load** parameter on the command line. If the application uses a workspace, the name of the workspace may instead be supplied as the last item on the command-line.

The name of the configuration file may be specified on the application command-line, using the **ConfigFile** parameter. Alternatively, the name of the configuration file is derived from the name of the workspace or source code file.

The action to start the application when a workspace or source code file is loaded is specified by the **LX** parameter or, for a workspace, by its latent expression (`⎕LX`).

In the command-line examples that follow, the name of the Run-Time EXE has been shortened to `dyalogrt.exe` for brevity.

Using a workspace

```
dyalogrt.exe myapp.dws
```

The application starts by running `⎕LX` in `myapp.dws`. If a configuration file named `myapp.dcfg` in the same directory, it is loaded and applied.

Using a source code file

```
dyalogrt.exe Load=myfn.aplf
```

The application loads the file named `myfn.aplf` which contains the source code for a function, and executes the expression `(myfn 0⍴⊂'')` (see *Load* on page 12). If a configuration file named `myfn.dcfg` in the same directory, it is loaded and applied.

If your application uses any component of the Microsoft .NET Framework, you must distribute the Bridge DLL and DyalogNet DLLs. These DLLs must be placed in the same directory as your EXE.

Stand-alone and Bound run-times

A stand-alone run-time is a single .EXE that contains a workspace and a copy of the Run-Time version of the Dyalog APL interpreter. It is the simplest type of run-time to install because it has the fewest number of dependencies.

A bound run-time is a workspace packaged as a .EXE that relies upon and requires the separate installation of the Run-Time DLL. Compared with the stand-alone executable option, bound run-times may save disk space and memory if your customer installs and runs several different Dyalog applications.

Both these run-times are created using the *File/Export* menu item on the Session window.

To distribute your application, you need to supply and install:

1. your stand-alone or bound .EXE
2. the Run-Time DLL (bound .EXE only)
3. a configuration file (optional)
4. whatever additional files that may be required by your application
5. a command-line to start the application

When you build your .EXE using the Export dialog, you may specify the name(s) of the configuration file(s) using the **ConfigFile** and/or **UserConfigFile** parameters in the field labelled *Command Line*.

An alternative is to specify these parameters in the command-line that you use to run your .EXE (note that this is not the same as the *Command Line* in the *Export* dialog box). If so, the Dyalog parameter(s) must be preceded by the **-apl** option.

If your application uses any component of the Microsoft .NET Framework, you must distribute the Bridge DLL and DyalogNet DLLs. These DLLs must be placed in the same directory as your EXE.

Out-of-process COM Server

To make an out-of-process COM Server, you must:

1. establish one or more OLEServer namespaces in your workspace, populated with functions and variables that you wish to export as methods, properties and events.
2. use the *File/Export ...* menu item on the Session window to register the COM Server on your computer so that it is ready for use.

The command-line for your COM Server must be specified in the field labelled *Command Line* in the *Export* dialog box. The field is initialised to invoke the Run-Time EXE with the name of your workspace in the same fashion as the workspace-based run-time discussed above. This command-line is recorded in the Windows Registry to be invoked when a client application requests it.

You may change the contents of the *Command Line* field to use a configuration file, in the same way as for a workspace-based runtime. The following example uses the Loan COM Server. See *Interface Guide: The LOAN Workspace*.

Example:

```
dyalog.exe C:\Dyalog18.0\myloan.dws
```

The command-line above will, on invocation, cause Dyalog to load the `myloan.dws` workspace together with the configuration file `myloan.dcfg` if it exists in that directory.

To distribute an out-of-process COM Server, you need to supply and install the following files:

1. your workspace
2. the associated Type Library (.tlb) file (created by *File/Export*)
3. the Run-Time EXE
4. a configuration file (optional)
5. whatever additional files that may be required by your application

To install an out-of-process COM Server you must set up the appropriate Windows registry entries. See Interface Guide for details.

In-process COM Server

To make an in-process COM Server, you must:

1. establish one or more OLEServer namespaces in your workspace, populated with functions and variables that you wish to export as methods, properties and events.
2. use the *File/Export ...* menu item on the Session window to create an in-process COM Server (DLL) which contains your workspace bound to the Run-Time DLL. This operation also registers the COM Server on your computer so that it is ready for use.

As there is no command-line available, to specify a configuration file for an in-process COM server, it is necessary to define the **ConfigFile** parameter and/or the **UserConfigFile** parameter as an environment variable or in the registry.

To distribute your component, you need to supply and install

1. Your COM Server file (DLL)
2. the Run-Time DLL
3. a configuration file (optional) and the means to define **ConfigFile** and/or **UserConfigFile**
4. whatever additional files that may be required by your COM Server.

Note that you must register your COM Server on the target computer using the `regsvr32.exe` utility.

ActiveX Control

To make an ActiveX Control, you must:

1. establish an ActiveXControl namespace in your workspace, populated with functions and variables that you wish to export as methods, properties and events.
2. use the *File/Export ...* menu item on the Session window to create an ActiveX Control file (OCX) which contains your workspace bound to the Run-Time DLL. This operation also registers the ActiveX Control on your computer so that it is ready for use.

As there is no command-line available, to specify a configuration file for an in-process COM server, it is necessary to define the **ConfigFile** parameter and/or the **UserConfigFile** parameter as an environment variable or in the registry.

To distribute your component, you need to supply and install

1. Your ActiveX Control file (OCX)
2. the Run-Time DLL
3. a configuration file (optional) and the means to define **ConfigFile** and/or **UserConfigFile**
4. whatever additional files that may be required by your ActiveX Control.

Note that you must register your ActiveX Control on the target computer using the `regsvr32.exe` utility.

Microsoft .NET Assembly

A Microsoft .NET Assembly contains one or more .NET Classes. To make a Microsoft .NET Assembly, you must:

1. establish one or more NetType namespaces in your workspace, populated with functions and variables that you wish to export as methods, properties and events.
2. use the *File/Export ...* menu item on the Session window to create a Microsoft .NET Assembly (DLL) which contains your workspace bound to the Run-Time DLL.

If the option selected in the *Isolation Mode* field of the *Export* dialog is either:

- Each assembly has its own workspace, or
- Each assembly attempts to use local bridge and interpreter libraries

you may enter configuration parameters or specify a Configuration file for your Dyalog assembly in the field labelled *Command Line*.

For the other isolation modes, this is not appropriate because only the command line from the first assembly loaded into the interpreter could be honoured, and the order in which assemblies are loaded is unpredictable. However, configuration files may be specified using the **ConfigFile** parameter and/or the **UserConfigFile** parameter specified as an environment variable or in the registry.

To distribute your .NET Classes, you need to supply and install

1. your Assembly file (DLL)
2. the Run-Time DLL
3. the Bridge DLL
4. the DyalogNet DLL
5. a configuration file (optional) and, depending upon the isolation mode, the means to define **ConfigFile** and/or **UserConfigFile**
6. whatever additional files that may be required by your .NET Assembly.

All the DLLs and subsidiary files must be installed in the same directory as the .NET Assembly.

Chapter 3:

Language Reference Changes

Language Changes

The following table summarises the main changes to language features in Version 18.0.

Function/Operator	Description	Change
{X}f⍤gY	Atop	New operator
{X}f∘gY	Beside	Renamed operator
A∘gY (f∘B)Y	Bind	Renamed operator
{X}(A⍨)Y	Constant	New operator
{X}f⍤gY	Over	New operator
≠	Unique Mask	New monadic function
⎕C	Case Convert	New system function
⍳	Where	Function extended
X⊂Y	Partitioned Enclose	Function extended
⎕DT	Date-time	New system function
1200⌶	Format Date-time	New I-beam function

Atop {R}←{X}f⍤gY

Classic Edition: the symbol ⍤ is not available in Classic Edition, and the Atop operator is instead represented by ⎕U2364.

f can be any monadic function. Y can be any array that is suitable as the right argument to function g with the result of g being appropriate to function f.

If X is omitted, g must be a monadic function. If X is specified, g must be a dyadic function and X can be any array that is suitable as the left argument to function g.

The derived function is equivalent to fgY or fXgY and need not return a result.

The Atop operator allows functions to be *glued* together to build up more complex functions. For further information, see *Function Composition* on page 1.

Examples:

```
      -⍤÷ 4       ⍴ (  f⍤g y) ≡  f   g y
¯0.25
      12 -⍤÷ 4    ⍴ (x f⍤g y) ≡ (f x g y)
¯3
      3 1 4 1 5 ~⍤∈ 1 2 3
0 0 1 0 1
```

Beside {R}←{X}f∘gY

g can be any monadic function which returns a result. Y can be any array appropriate to function g with gY being suitable as the right argument to function f.

If X is omitted, f must be a monadic function. If X is specified, f must be a dyadic function and X can be any array that is suitable as the left argument to function f.

The derived function is equivalent to fgY or XfgY and need not return a result.

The Beside operator allows functions to be *glued* together to build up more complex functions. For further information, see *Function Composition* on page 1.

Examples

```
      RANK ← ⍴∘⍴
      RANK ¨ 'JOANNE' (2 3⍴⍳6)
 1  2
```

```
      +/∘⍳¨2 4 6
3 10 21


      ⎕VR'SUM'
     ∇ R←SUM X
[1]    R←+/X
     ∇

      SUM∘⍳¨2 4 6
3 10 21

      +∘÷/40⍴1        ⍝ Golden Ratio! (Bob Smith)
1.618033989

      0,∘⍳¨⍳5
0 1  0 1 2  0 1 2 3  0 1 2 3 4  0 1 2 3 4 5
```

Bind

{R}←A∘fY
{R}←(f∘B)Y

The Bind operator binds an array A or B to a dyadic function f either as its left or its right argument respectively. The former may be described as *left argument currying* and the latter as *right argument currying*.

A, B and Y may be any arrays whose items are appropriate to function f. In the case where B is bound as the right argument of function f, the parentheses are required in order to distinguish between the operand B and the argument Y.

The derived function is equivalent to AfY or YfB and need not return a result.

Examples

```
      2 2∘⍴ ¨ 'AB'
 AA  BB
 AA  BB

      SINE ← 1∘○

      SINE 10 20 30
¯0.5440211109 0.9129452507 ¯0.9880316241

      (*∘0.5)4 16 25
2 4 5

      SQRT ← *∘.5

      SQRT 4 16 25
2 4 5
```

The following example uses both forms of Bind to list functions in the workspace:

```
      ⎕NL 3
ADD
PLUS

      ⎕∘←∘⎕VR¨↓⎕NL 3
     ∇ ADD X
[1]    →LABρ⍨0≠⎕NC'SUM' ⋄ SUM←0
[2]   LAB:SUM←SUM++/X
     ∇
     ∇ R←A PLUS B
[1]    R←A+B
     ∇
```

Constant R←{X}(A⍨)Y

A, X and Y are arrays. The Constant operator returns array A.

Examples:

```
      'mu'⍨ 'any' ⎕NULL   ⍝ Always returns its operand
mu
      1E100 ('mu'⍨) 1j1
mu
      ¯1⍨¨ ⍳2 3
¯1 ¯1 ¯1
¯1 ¯1 ¯1
```

Over {R}←{X}fögY

Classic Edition: the symbol ö is not available in Classic Edition, and the Over operator is instead represented by ⎕U2365.

g can be any monadic function which returns a result. Y can be any array that is suitable as the argument to function g with gY being suitable as the right argument to function f.

If X is omitted, f must be a monadic function. If X is specified, f must be a dyadic function and X can be any array that is suitable as argument to function g with gX being suitable as the left argument to function f.

The derived function is equivalent to fgY or (gX)f(gY) and need not return a result.

The Over operator allows functions to be *glued* together to build up more complex functions. For further information, see *Function Composition* on page 1.

Examples

```
      2 3 ,⍤¯1 'text'   ⍝ ,⍤¯1  ↔  {⍺⍵}
```

2 3	text

```
     scores←82 90 76
     weights←20 35 45
     (weights×scores)÷⌿(+/)weights ⍝ Weighted average
82.1
```

Unique Mask R←≠Y

Y may be any array.

R is a Boolean vector whose length is the number of major cells in Y. For each major cell of Y, the corresponding element of R is 1 if it is the first occurrence of that value, and 0 if it is a duplicate of an earlier major cell.

⎕CT and ⎕DCT are implicit arguments of Unique.

Examples

```
      ≠22 10 22 22 21 10 5 10
1 1 0 0 1 0 1 0

      ≠ v←'CAT' 'DOG' 'CAT' 'DUCK' 'DOG' 'DUCK'
1 1 0 1 0 0

      ⊢mat←↑v
CAT
DOG
CAT
DUCK
DOG
DUCK
      ≠mat
1 1 0 1 0 0
```

Partitioned Enclose (⎕ML<3) R←X⊂[K]Y

Y may be any array. X must be a simple integer scalar or vector. If X is a scalar it is extended to (≠Y)⍴X.

The axis specification is optional. If present, it must be a simple integer scalar or one-element vector. The value of K must be an axis of Y. If absent, the last axis of Y is implied.

R is a vector of items selected from Y by inserting 0 or more dividers, specified by X, between its major cells.

Each element of X species the number of dividers to insert before the corresponding major cell of Y. The maximum length of X is 1+≠Y, when the last element of X specifies the number of trailing dividers. Note that major cells of Y that precede the first divider (identified by the first non-zero element of X) are excluded from the result.

The length of R is +/X (after possible extension).

Examples

```
      0 0 1 0 0 1 0⊂'abcdefg'
```

cde	fg

```
      2 0 1 3 0 2 0 1⊂'abcdefg'
```

	ab	c			de		fg	

```
      0 2 0 1⊂'abcdefg'
```

	bc	defg

The above examples may be explained pictorially by the diagram below.

Further Examples

```
      1 0 1⊂[1]3 4⍴⍳12
```

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

```
      1 0 0 1⊂[2]3 4⍴⍳12
```

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

Case Convert R←{X}⎕C Y

Y is any array. R is an identical array except that character arrays within it are either folded for case-less comparison, or mapped to upper or lower case.

For a discussion of case folding and case conversion (mapping), see https://unicode.org/faq/casemap_charprop.html.

If the optional left-argument X is omitted, R is a copy of Y with character arrays folded, for case-less comparison.

If X is specified, the following cases are supported:

X	Description
1	R is a copy of Y with character arrays mapped to upper case.
¯1	R is a copy of Y with character arrays mapped to lower case.
¯3	R is a copy of Y with character arrays folded, for case-less comparison (this is equivalent to monadic use).

Examples

```
      ⎕C 42 'Pete' 'Πέτρος'
42  pete  πέτροσ
      1 ⎕C 42 'Pete' 'Πέτρος'
42  PETE  ΠΈΤΡΟΣ
      ¯1 ⎕C 42 'Pete' 'Πέτρος'
42  pete  πέτρος

      (⊂'pete'){α≡⎕C ω}¨'PETE' 'Pete' 'pEte'
1 1 1
```

Example

Greek has two forms of lower-case Sigma, namely "σ" and "ς" but a single upper-case Sigma "Σ". Each lower-case form remains unchanged when mapped to lower-case, but both fold to "σ", while "Σ" is mapped to lower-case "σ" .

```
      ⎕C 'ίσως'
ίσωσ
      1 ⎕C 'ίσως'
ΊΣΩΣ
      ¯1⎕C 1 ⎕C 'ίσως'
ίσωσ
```

Note:

Refs in Y are not followed but just returned unchanged.

Date-time R←X ⎕DT Y

This function validates date-times or converts date-times between one format and another.

A *date-time* is a date and time of day represented by a *timestamp*, a *time number* or a *military time-zone character*.

- A *timestamp* is a date-time expressed as a multiple element numeric vector, of which there are several different sorts (principally `⎕TS` format).
- A *time number* is a date-time expressed as a scalar numeric value, of which there are several different sorts.
- A *military time zone character* is a scalar character that represents the current date-time ("now") in a particular time zone. For example, `'A'` represents the current date-time (UTC) + 1 hour.

`Y` is an array of any shape whose elements contain a timestamp, time number or military time zone character, in any combination.

`X` may be a single integer value or a 2-element integer vector.

When `X` is a single integer value, it must be either 0 or a date-time code listed in the tables below. 0 specifies that the elements of `Y` are to be validated. A non-zero value specifies the date-time representation to which the elements of `Y` are to be converted. In this case, the numeric elements of `Y` are interpreted as follows:

- scalars are assumed to be time numbers of type Dyalog Date Number (code 1)
- vectors are assumed to be `⎕TS` timestamps (code ¯1)

When `X` is a 2-element integer vector, `X[1]` is a date-time code that explicitly specifies the date-time representation of the numeric elements in `Y`. `X[2]` is either 0 or a date-time code listed in the tables below. 0 specifies that the elements of `Y` are to be validated. A non-zero value specifies the date-time representation to which the elements of `Y` are to be converted.

Character scalars in `Y` are always interpreted as meaning "now".

`R` is an array of the same shape as `Y`, where each element is either a timestamp, time number or Boolean value as determined by the second or only element in `X`.

Note: time numbers in `R` may be of type DECF even if `⎕FR` is 645 if their magnitude can be too great to store precisely in a double. See the table below for the type numbers where this is so.

Time Numbers

If a value in `X` is positive it indicates that a time number type is expected in `Y` or generated in `R`, as follows. Note that the last column indicated whether (Yes) or not (No) negative numbers are allowed.

Code	Description	Category	Epoch[1]	Neg[8]
1	Dyalog Date Number	Day count with fractional part	1899-12-31 00:00	Yes
2	Dyalog component time	Tick count 1/60thsec ticks[2]	1970-01-01 00:00	Yes
10	J (J nanosecond time)	Tick count 1ns ticks[2]	2000-01-01 00:00	Yes
11	Shakti K	Tick count 1ms ticks[2]	2024-01-01 00:00	Yes
12	Javascript / D / Q	Tick count 1ms ticks[2]	1970-01-01 00:00	Yes
13	R (R chron format)	Day count with fractional part	1970-01-01 00:00	Yes
20	Unix time	Tick count 1s ticks[2]	1970-01-01 00:00	Yes
30	Microsoft DOS date/time	Encoded broken-down time 2s resolution	N/A	No
31	Microsoft Win32 FILETIME	Tick count[3] 100ns ticks	1601-01-01 00:00	No
32	Microsoft CLR DateTime (.NET)(Ticks property thereof)	Tick count[3] 100ns ticks	0001-01-01 00:00	No
33	Microsoft OLE Automation Date(also known as Variant Time)	Day count with fractional part	1899-12-30 00:00	Yes[7]

Code	Description	Category	Epoch[1]	Neg[8]
40	Excel (1900 Date System)[4] / Lotus 1-2-3	Day count with fractional part[5]	1899-12-31 00:00[6]	No
41	Excel (1904 Date System)[4]	Day count with fractional part	1904-01-01 00:00	No
42	Stata statistics package	Tick count 1ms ticks[2]	1960-01-01 00:00	Yes
43	SPSS statistics package	Tick count 1s ticks[2]	1582-10-14 00:00	No
44	SAS	Tick count 1s ticks[2]	1960-01-01 00:00	Yes
50	Julian Date	Day count with fractional part	¯4714-11-24 00:00	No
51	J (J dayno)	Day count with fractional part	1800-01-01 00:00	No
52	Reduced Julian Date	Day count with fractional part	1858-11-16 12:00	Yes
53	Modified Julian Date	Day count with fractional part	1858-11-17 00:00	Yes
54	Dublin Julian Date	Day count with fractional part	1899-12-31 12:00	Yes
55	CNES Julian Date	Day count with fractional part	1950-01-01 00:00	Yes

Code	Description	Category	Epoch[1]	Neg[8]
56	CCSDS Julian Date	Day count with fractional part	1958-01-01 00:00	Yes
60	Floating-point decimal encoded format[9] Digits take the form yyyymmdd.hhmmss	Encoded broken-down time 1s resolution	N/A	No
61	Integer decimal encoded format[9] Digits take the form yyyymmddhhmmss (J digit time)	Encoded broken-down time 1s resolution	N/A	No

Footnotes

1. In the Proleptic Gregorian Calendar.
2. There are the same number of ticks per day regardless of leap seconds.
3. Generated as DECF values regardless of the setting of □FR due to their magnitude.
4. Excel supports two time number conventions. On Windows the 1900 Date System is the default and on macOS the 1904 Date System is the default. Both systems can use either convention and the convention in use is stored in the worksheet so that the platforms interoperate.
5. Count includes the invalid date 1900-02-29.
6. Microsoft Excel converts day 0 to the invalid date 1900-01-00.
7. For negative numbers, the integral part counts backward from 1899-12-30 and the fractional part counts forward from the date so reached.
8. No date-time may represent a date earlier than ¯4713-01-01 00:00.
9. Decimal encoded formats encode human-readable dates and times into a single number with the most significant part in the most significant decimal digit, for example 2020/01/23 (year/month/day) is encoded as 20200123, and 13:17:56 (hour:minute:second) is encoded as 131756. The date must be between 1 January 0001 and 28 February 4000 in the Proleptic Gregorian Calendar.

Timestamps

If a value in X is negative it indicates that a timestamp type is expected in Y or generated in R, as follows:

Code	Description	Max elements	Element contents[1]	Elided element implicit values (in Y)[3]
-1	Millisecond precision (TS)	7	Year, month, day-of-month, hour, minute, second, millisecond	1 1 1 0 0 0 0
-2	Microsecond precision	7	Year, month, day-of-month, hour, minute, second, microsecond	1 1 1 0 0 0 0
-3	Nanosecond precision (J expanded digit time)	7	Year, month, day-of-month, hour, minute, second, nanosecond	1 1 1 0 0 0 0
-10	ISO day-of-year components	6	Year, day-of-year, hour, minute, second, microsecond	1 1 0 0 0 0
-11	ISO day-of-week components	7	Year, week, day-of-week, hour, minute, second, microsecond	1 1 1 0 0 0 0
-20	Decimal encoded date and time[2]	2	Decimal encoded date, decimal encoded time	10101 0

Footnotes

1. All dates must be between 1 January 0001 and 28 February 4000 in the Proleptic Gregorian Calendar.
2. Decimal encoded formats encode human-readable dates and times into a single number with the most significant part in the most significant decimal digit, for example 2020/01/23 (year/month/day) is encoded as 20200123, and 13:17:56 (hour:minute:second) is encoded as 131756.
3. If a timestamp has fewer than the maximum number of elements, the remaining elements take the default values shown.

Military time zone characters

Any element in `Y` may be specified as a military time zone character and is implicitly replaced by the current time in the time zone they represent. The time zones are as follows:

Character	Time zone name	Time zone
A	Alpha	UTC +1
B	Bravo	UTC +2
C	Charlie	UTC +3
D	Delta	UTC +4
E	Echo	UTC +5
F	Foxtrot	UTC +6
G	Golf	UTC +7
H	Hotel	UTC +8
I	India	UTC +9
J	Juliet	Local time
K	Kilo	UTC +10
L	Lima	UTC +11
M	Mike	UTC +12
N	November	UTC -1
O	Oscar	UTC -2
P	Papa	UTC -3
Q	Quebec	UTC -4
R	Romeo	UTC -5
S	Sierra	UTC -6
T	Tango	UTC -7
U	Uniform	UTC -8
V	Victor	UTC -9
W	Whisky	UTC -10
X	X-ray	UTC -11
Y	Yankee	UTC -12

Character	Time zone name	Time zone
Z	Zulu	UTC +0

Note that the resolutions of system clocks vary by platform.

Examples: Timestamp to time number conversion

```
      ¯1 1 ⎕DT ⊂⎕TS
43886.48039
      1 ⎕DT ⊂⎕TS
43886.48039
      1 ⎕DT ⎕TS 'J'
43886.48039 43886.48039

      1 ⎕DT ⊂⍬ ⍝ cf Elided element implicit values
¯693594
      1 ⎕DT ⊂1 1 1 0 0 0 0
¯693594
```

Examples: Time number to timestamp conversion

```
      1 ¯1 ⎕DT 0 43508.42843
┌──────────────────┬──────────────────┐
│1899 12 31 0 0 0 0│2019 2 13 10 16 56 352│
└──────────────────┴──────────────────┘
      ¯1 ⎕DT 0 43508.42843
┌──────────────────┬──────────────────┐
│1899 12 31 0 0 0 0│2019 2 13 10 16 56 352│
└──────────────────┴──────────────────┘
      2 ¯1 ⎕DT 3⊃⎕FRDCI 1 1
┌──────────────────┐
│2020 2 26 11 33 54 466│
└──────────────────┘
```

Examples: Time number to time number conversion

```
      2 1 ⎕DT 3⊃⎕FRDCI 1 1
43886.48188
      1 ⎕DT 'J'
43886.48371
      ⍝ Local time is UTC-05:00
      3600÷⍨-/20 ⎕DT 'JZ'
¯5
```

Example: Validation

```
      0 ⎕DT ⎕TS (2020 13 1) 'J' 'DT' #
1 0 1 0 0
```

Format Date-time R←X(1200⌶)Y

Y is a numeric array of any shape, where every element contains a Dyalog Date Number that represents a date between 1 January 0001 and 28 February 4000 in the Proleptic Gregorian Calendar.

X is a character scalar or vector specifying a pattern with which the elements in Y should be formatted.

R is an array of the same shape as Y, whose elements are enclosed character vectors.

Formatting Pattern in X

The formatting pattern allows a time number to be converted to a highly user-configurable, plain text format. When a time number is formatted, elements in the result are copies of the format pattern with format sequences replaced by the elements they represent.

The format sequences are intended to be visually reminiscent of the generated text. They use alphabetic characters easily associated with the substitution (e.g. D, M and Y for Day, Month and Year respectively) repeated one or more times to indicate format. As noted below, some sequences allow the first character to be replaced by a _, or the casing to be altered.

Format letter	Length	Meaning	Variations	Example
Year	YY	Without century	YY	19
	YYYY	With century	YYYY	2019
Month	M	1 or 2 digit numeric	M	3
	MM	2 character numeric	MM _M	03 3
	MMM	Abbreviated name	MMM Mmm mmm _mm[1]	MAR Mar mar Mar
	MMMM	Full name	MMMM Mmmm mmmm _mmm[1]	MARCH March march March

Format letter	Length	Meaning	Variations	Example
Day of month	D	1 or 2 digit numeric	D	4
	DD	2 character numeric	DD _D	04 4
hours	h	1 or 2 digit numeric	h	8
	hh	2 character numeric	hh _h	08 8
minutes	m	1 or 2 digit numeric	m	5
	mm	2 character numeric	mm _m	05 5
seconds	s	1 or 2 digit numeric	s	0
	ss	2 character numeric	ss _s	00 0
fractional seconds	f	1 digit precision	f	5
	ff	2 digit precision	ff	55
	fff	3 digit precision	fff	555
	ffff	4 digit precision	ffff	5555
	fffff	5 digit precision	fffff	55555
	ffffff	6 digit precision	ffffff	555555
day of week	d	Numeric (1-7)	d	1
	ddd	Abbreviated name	DDD Ddd ddd _dd[1]	MON Mon mon Mon
	dddd	Full name	DDDD Dddd dddd _ddd[1]	MONDAY Monday monday Monday
ISO week number	w	1 or 2 digit numeric	w	10
	ww	2 character numeric	ww _w	10 10
year of ISO Week number[2]	WW	Without century	WW	19
	WWWW	With century	WWWW	2019

Format letter	Length	Meaning	Variations	Example
day of year	y	1 to 3 digit numeric	y	63
	yy	3 character numeric	yy _y	063 63
Ordinal indicator[3] for day of month	O	Short	O o	T t
	OO	Full	OO Oo oo	TH Th th
hours in twelve hour clock	t	1 or 2 digit numeric	t	8
	tt	2 character numeric	tt _t	08
AM/PM Indicator	P	Short	P p	A a
	PP	Full	PP pp	AM am

Footnotes

1. Natural sentence case, which may be specified for M (month name) and d (day name) only, causes the text to be substituted in the case which is natural for the language; some languages (e.g. English) always capitalise the first letter of day and month names whereas others (e.g. French) do not.
2. Dates at the start of the year may be in the final week of the previous year, and dates at the end of the year may be in the first week of the following year.
3. An ordinal indicator is a character or group of characters following a numeral, such as (in English) the suffixes -st, -nd, -rd, -th as in 1st, 2nd, 3rd, 4th.

1. Natural sentence case, which may be specified for M (month name) and d (day name) only, causes the text to be substituted in the case which is natural for the language; some languages (e.g. English) always capitalise the first letter of day and month names whereas others (e.g. French) do not.
2. Dates at the start of the year may be in the final week of the previous year, and dates at the end of the year may be in the first week of the following year.
3. An ordinal indicator is a character or group of characters following a numeral, such as (in English) the suffixes -st, -nd, -rd, -th as in 1st, 2nd, 3rd, 4th.

The upper and lower case letters, underscore _, dollar $ and percent % are all reserved for introducing format sequences, even though not all currently have meaning. The remaining, non-reserved, characters are copied to the result unchanged, thus the format string hh:mm represents the hour of the day and minute of the hour with a colon between (e.g. 12:00). All characters or sequences of characters may be delimited by " or ' at any point in the format string to prevent them being interpreted as a part of a format sequence, and, within these delimiters, two adjacent delimiter characters produce a single delimiter.

Note: The characters AaaaBbbb consist of two adjacent format sequences because there is a sequence of As followed by a sequence of Bs. The characters AaaaAaaa consist of one format sequence because it only contains As. It can be separated into two format sequences by insering an empty " or ' - delimited string, e.g. Aaaa""Aaaa.

Language

Unless overridden, English is used for text substitutions. Different languages may be selected using the **Language** variant option and/or the use of language specifiers within the format pattern. In either case, the language is specified as either a two letter ISO 639-1 language code in lower case (e.g. en) or as a five character language with an additional underscore and two character region in upper case (e.g. en_GB). Within the format pattern, __xx__ (where xx is the two or five character specifier) will switch the language of the subsequent generated text. Dictionaries for the following languages are built in:

ISO 639-1	Language
da	Danish
de	German
el	Greek
en	English
es	Spanish
fi	Finnish
fr	French
it	Italian
ja	Japanese
nb	Norwegian Bokmål
nl	Dutch
nn	Norwegian Nynorsk

ISO 639-1	Language
pl	Polish
pt	Portuguese
ru	Russian
sv	Swedish
zh	Chinese

Predefined patterns

Any pattern can contain (in part or in whole) a named predefined pattern, which allows common date and time formats to be specified in abbreviated form. Predefined patterns may be specified on a per-language basis, allowing patterns to be tailored for the selected language.

Predefined patterns are included in a pattern using `%` delimiters. For example, `%ISO%` includes the named predefined pattern `ISO`.

The following global predefined pattern is built in:

Name	Substitutes as
`ISO`[1]	`YYYY-MM-DD"T"hh:mm:ss`

1. An ISO 8601 extended format calendar date and time with no time zone designator.

This list may be expanded in future.

Additional predefined patterns may be defined using the **Dictionary** variant option. Predefined patterns must not contain references to other predefined patterns.

Variant Options

The **Language** variant option specifies the language used for formatting datetimes and defaults to `'en'` (English). The option value is a two or five character name (e.g. `'en'` or `'en_GB'`). The setting may be explicitly overridden in the format pattern.

The **Dictionary** variant option specifies a namespace which contains additional or replacement names for the months etc. and/or predefined patterns, for languages and language regions.

At the top level there may be zero or more sub-namespaces with two or five character names, according to the rules for language and language regions. Within each of these, month names etc. are defined as follows:

Named item	Description
`MonthNames`	A twelve-element vector of character vectors containing the full names corresponding to January to December, respectively.
`ShortMonthNames`	A twelve-element vector of character vectors containing the short names corresponding to Jan to Dec, respectively.
`WeekdayNames`	A seven-element vector of character vectors containing the full names corresponding to Monday to Sunday, respectively.
`ShortWeekdayNames`	A seven-element vector of character vectors containing the full names corresponding to Mon to Sun, respectively.
`MorningAfternoon`	A two-element vector of character vectors containing the names corresponding to AM and PM, respectively.
`Ordinals`	A character vector containing the one ordinal used for all numbers in the range 1 to 31, or a thirty one-element vector of character vectors containing the ordinals for 1 to 31, respectively.

Also at the top level of the dictionary namespace there may be a sub-namespace named Patterns and within this further sub-namespaces named `Global` and/or two or five character language names, containing definitions of predefined patterns. Predefined patterns are defined in the same way as the formatting pattern except that they may not contain references to other predefined patterns.

If the namespace contains a definition which is supplied built into the interpreter, it replaces the built-in one.

If a dictionary is incomplete (e.g. is missing one of the expected named items, or one of the named items contains too few elements) an error is signalled only if the missing content would actually be needed.

Example dictionary

The following creates a dictionary defined by the namespace `dict` using JSON text. See the formatting examples below for uses of this dictionary.

```
      dict_json
{
  "Patterns": {
    "Global": {
      "ISOweek": "YYYY-'W'ww",
      "DateCompact": "D-MMM-YYYY",
      "DateVerbose": "'the date is' DD _mm YYYY"
    },
    "fr": {
      "DateVerbose": "'la date est le' DD mmm YYYY"
    },
    "en_US": {
      "DateVerbose": "'the date is' Mmm DD, YYYY"
    }
  },
  "en_US": {
    "ShortMonthNames": [
      "Jan.", "Feb.", "Mar.", "Apr.", "May", "June",
      "July", "Aug.", "Sept.", "Oct.", "Nov.", "Dec."
    ]
  },
  "cy": {
    "MonthNames": [
      "Ionawr", "Chwefror", "Mawrth", "Ebrill", "Mai", "Mehefin",
      "Gorffennaf", "Awst", "Medi", "Hydref", "Tachwedd",
"Rhagfyr"
    ],
    "ShortMonthNames": [
      "Ion", "Chw", "Maw", "Ebr", "Mai", "Meh",
      "Gor", "Awst", "Medi", "Hyd", "Tach", "Rhag"
    ],
    "WeekdayNames": [
      "Dydd Sul", "Dydd Llun", "Dydd Mawrth", "Dydd Mercher",
      "Dydd Iau", "Dydd Gwener", "Dydd Sadwrn"
    ],
    "ShortWeekdayNames": [
      "Sul", "Llun", "Maw", "Mer", "Iau", "Gwen", "Sad"
    ],
    "MorningAfternoon": [
      "yb", "yh"
    ],
    "Ordinals": [
      "af", "il", "ydd", "ydd", "ed", "ed", "fed", "fed", "fed",
      "fed", "eg", "fed", "eg", "eg", "fed", "eg", "eg", "fed",
      "eg", "fed", "ain", "ain", "ain", "ain", "ain", "ain",
      "ain", "ain", "ain", "ain", "ain"
    ]
  }
}
      dict←⎕JSON dict_json
```

Note the following:

- In the example, the predefined pattern `ISOweek` is defined globally and is not redefined. It therefore has the same value for all languages. Similarly, `DateCompact` has the same value for all languages, but although the definition is global, it contains the pattern `MMM` and this will be substituted with the month name in the selected language.
- The predefined patterns `DateVerbose` is defined globally, and redefined for languages `fr` and `en_US`. The global definition will be used when any language other than `fr` and `en_US` is selected. If there was not global definition it would only be defined for `fr`, all regional variations of `fr`, and `en_US`.
- There is no explicit definition of patterns or names for language region `en_GB`. If this language is selected the definitions for `en` will be used.
- There is an explicit definition for `ShortMonthNames` for language region `en_US`. If this language is selected the definition of `ShortMonthNames` is as defined, and as for `en` for other names. As `en` is not defined in the dictionary, the built-in defaults are used.

In the following examples:

```
      tn←1 ⎕DT ⊂2019 2 13 10 16 56
      tn
43508.42843
```

English

```
      'Dddd, DDoo Mmmm YYYY; hh:mm:ss' (1200⌶) tn
Wednesday, 13th February 2019; 10:16:56

      '__en__Dddd, DDoo Mmmm YYYY; hh:mm:ss' (1200⌶) tn
Wednesday, 13th February 2019; 10:16:56

      '"ISO date": %ISO%' (1200⌶) tn
ISO date: 2019-02-13T10:16:56

      '%DateVerbose%'(1200⌶⍠'Dictionary'dict) tn
the date is 13 Feb 2019
```

English (US)

```
     fmt←'%DateVerbose%'
     fmt (1200⌶⍠('Dictionary'dict)('Language' 'en_US'))tn
the date is Feb. 13, 2019
```

Danish

```
      '__da__Dddd, DDoo mmmm YYYY; hh:mm:ss' (1200⌶) tn
Onsdag, 13. februar 2019; 10:16:56

      fmt←'Dddd, DDoo mmmm YYYY; hh:mm:ss'
      fmt(1200⌶⍠'Language' 'da') tn
Onsdag, 13. februar 2019; 10:16:56
```

Welsh (using the dictionary defined above)

```
      fmt←'__cy__Dddd, DDoo mmmm YYYY; hh:mm:ss'
      fmt (1200⌶⍠'Dictionary' dict) tn
Dydd Mercher, 13eg chwefror 2019; 10:16:56


      '__cy__%DateVerbose%' (1200⌶⍠'Dictionary' dict) tn
the date is 13 Chw 2019
```

Chapter 4:

Object Reference Changes

ExecuteJavaScript Method 839

Applies To: HTMLRenderer

Description

This method is used to execute JavaScript in a HTMLRenderer object.

The argument to ExecuteJavaScript is a single item as follows:

[1]	Code	character vector containing JavaScript code

Example

```
hr.ExecuteJavaScript 'alert("Hello")'
```

InterceptedURLs Property

Applies To: HTMLRenderer

Description

The InterceptedURLs property is a 2-column matrix that specifies whether the HTMLRenderer will attempt to satisfy a request for a resource from the workspace or, via the CEF, from the internet. If directed to the workspace, the request will trigger an HTTPRequest event if the protocol is `http`, or a WebSocketUpgrade event if the protocol is `ws`.

The first column is a wild-carded character scalar or vector containing a pattern to match. The second column is numeric indicating whether or not the HTMLRenderer should trigger an event as shown in the table below. InterceptedURLs may contain any number of rows.

Value	Meaning
0	Ignore request; pass to CEF for fulfilment
1	Trigger an HTTPRequest or WebSocketUpgrade (automatic) in the workspace
2	Trigger an HTTPRequest or WebSocketUpgrade (manual) in the workspace

If the requested url is a relative rather than an absolute URL, it is prepended by the string `http://dyalog_root/`. So, for example, if the HTML property contains :

```
<link rel="stylesheet" href="style.css">
<script src="app.js"></script>
```

the HTMLRenderer will request `http://dyalog_root/style.css` and `http://dyalog_root/app.js` respectively.

When the value of InterceptedURLs is its default (`(0 2⍴'')`) it is treated as if it were set to `((1 2⍴'*://dyalog_root/*' 1)`. So by default, requests for a relative URL will fire an event in the workspace while absolute URL will be directed by the CEF to the internet.

Note that if code in the page creates a web socket intended for internal use, with anything other than `dyalog_root` as the URL, the URL must match a pattern in InterceptedURLs with 1 in the second column. The following example does not require a matching pattern in InterceptedURLs.

```
// Create a new WebSocket.
 window.socket = new WebSocket('ws://dyalog_root/');
```

Examples:

The following will trigger an event for all requested URLs.

```
InterceptedURLs ← 1 2⍴'*' 1
```

The following will attempt to retrieve from the net URLs containing `'.dyalog.com'` and trigger an HTTPRequest event for all other requested URLs.

```
InterceptedURLs ← 2 2⍴'*.dyalog.com*' 0 '*' 1
```

WebSocketUpgrade Event 841

Applies To: HTMLRenderer

Description

This event is reported when the client component of an HTMLRenderer object opens a WebSocket and the requested URL matches a pattern specified by the InterceptedURLs property. If there is no match, the connection request is processed as an external request by the Chromium Embedded Framework (CEF)[1].

The event message reported as the result of □DQ, or supplied as the right argument to your callback function, is a 6-element vector as follows:

[1]	Object	ref or character vector
[2]	Event	'WebSocketUpgrade' or 841
[3]	ID	Character vector containing the ID of the WebSocket
[4]	URL	The requested URL of the WebSocket
[5]	Headers	ASCII including CRLF
[6]	Type	Character vector 'auto' or 'manual'

The protocol for establishing the connection is defined by InterceptedURLs and is reported by the 6th element (Type) of the event message.

If Type is 'auto', the protocol is handled internally and this event is reported when the connection has already been made. Should the connection fail, a WebSocketError event will be reported instead.

If Type is 'manual', a callback function for WebSocketUpgrade is mandatory and is responsible for completing (or denying) the connection. This is achieved by setting the 5th element of the event message (Headers) to indicate an appropriate positive or negative response to the request[2] and returning the entire event message as its result. If a valid response is not generated in this way, the connection will time-out causing a WebSocketError event.

In both cases, the WebSocket ID is subsequently required to send a message by calling the WebSocketSend method or to close the connection using the WebSocketClose method.

Note that several WebSocket connections may be made concurrently.

[1]https://en.wikipedia.org/wiki/Chromium_Embedded_Framework

[2]https://tools.ietf.org/html/rfc6455#section-1.3

Example

Index

.

.NET Core 25

A

ActiveX control 33
APL_TextInAplCore parameter 25

B

beside operator 36
bind operator 37
bridge dll 31, 34
Bug Fixes 13

C

case convert 14
Classic Edition 36, 38
COM server
    in-process 32
    out-of-process 31
configuration files 4, 25
configuration parameters 23
constant operator 38
currying 37

D

date-time 43
dyadic primitive functions
    partitioned enclose 40
dyadic primitive operators
    beside 36
    bind 37
Dyalog_LineEditor_Mode 7
Dyalog_NETCore parameter 25
dyalognet dll 31, 34
DyalogStartSE Parameter 21-22
DyalogStartup parameter 21

E

environment variables 23
Events
    WebSocketUpgrade 61
ExecuteJavaScript 59

F

file associations 23
format date-time 50

G

GetEnvironment method 24
global assembly cache 31

I

i-beam
    format date-time 50
InterceptedURLs 59
Interoperability 16

K

Key Features 1
key operator 18

L

load parameter 12
lx parameter 12

M

Methods
    ExecuteJavaScript 59
mixe extension 8

monadic primitive functions
unique mask 39
monadic primitive operators
constant 38
multiline input 7

N

namespaces
serialisation 10
nest 18
Net assembly 33

O

over operator 18

P

partitioned enclose function 40
with axis 40
primitive operators
beside 36
bind 37
constant 38
Properties
InterceptedURLs 59

R

rank operator 18
regex 9
Regexp option 9
replace operator
Regexp 9
run-time
applications 29
bound 31
stand-alone 31
workspace based 30
run-time dll 33-34
run-time exe 30, 32

S

search operator
Regexp 9
session
initialisation 21
session initalisation 21
stencil operator 18
System Requirements 15

U

Unicode Edition 12
unique mask function 39

V

variant operator 18

W

WebSocketUpgrade 61
where 18
where extension 8