Dyalog
Release Notes

Dyalog version 19.0

DYALOG

The tool of thought for software solutions

Dyalog is a trademark of Dyalog Limited
Copyright © 1982-2024 by Dyalog Limited
All rights reserved.

Dyalog Release Notes

Dyalog version 19.0
Document Revision: 20240319_190

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:

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 U.S. 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 U.S. and other countries.
macOS® and OS X® (operating system software) are registered trademarks of Apple Inc. in the U.S. and other countries.

All other trademarks and copyrights are acknowledged.

Contents

Chapter 1: Highlights 1
Key Features 1
Extension to Native File Functions 5
Lexical Scope in Scripts 9
Session Gutter 12
Source as Typed 13
Bug Fixes 15
Announcements 16
System Requirements 18
Interoperability 19

Chapter 2: Configuration Parameters 23
DYALOG_GUTTER_ENABLE 23
DYALOG_DISCARD_FN_SOURCE 24
DYALOG_INITSESSION 24
DyalogLink 24
DyalogStartup_X 24
DyalogStartupSE 25
Log_File 26

Chapter 3: Language Reference Changes 27
File Hold 28
Fix Script 29
Allocate Token Range 37
Set aplcore Parameters 38
Memory Manager Statistics 39
Enable Compression of Large Components 43
Discard Source Information 43
Discard Source Code 44
Hash Table Size 44
Lookup Table Size 45

Chapter 4: Object Reference Changes 47
AllowContextMenu 47
ExecuteJavaScript 48
GetZoomLevel 48
IsLoading 48
LoadEnd 49

SetZoomLevel 50

Index **53**

Chapter 1:

Highlights

Key Features

Upgrading from Version 17.1 to Version 19.0

Please note that if you are upgrading from Version 17.1 to Version 19.0, you should read the Release Notes for Version 18.0 and the Release Notes for Version 18.2 in conjunction with this document.

Linux Restriction

There is a new restriction in Version 19.0 for Linux which will also apply in forthcoming new versions.

Under macOS and Linux, if the configuration parameter **ENABLE_CEF** is 1, Auxiliary Processors cannot be used (they hang on error). The default value is 1 unless you are not running under a desktop (for example, you are running Dyalog in a PuTTY session when the default is 0).

Dyalog APL on macOS

- Dyalog APL Version 19.0 is available as both a native Intel-based macOS version and a native ARM-based macOS version. Version 19.0 is expected to be the last version to be compiled for Intel-based Macs.

New Language Features

- Currently, multi-threaded applications rely on hand-picked token types and require coordination between developers in the choice of these tokens. There is a new system function designed to remove the need for hard-coded token numbers. See *Allocate Token Range* on page 37.

Improved Language Features

- A variant option `CharSet` is provided to restrict the result of `⎕JSON` export to ASCII characters. Non-ASCII characters are converted to encoded strings.

- The memory manager has been extended to give the programmer finer control over WS FULL errors. See *Memory Manager Statistics* on page 39.
- ⎕NCOPY and ⎕NMOVE now provide an option for an APL callback function to be invoked during execution. This allows the programmer to monitor and/or report progress and/or abort when processing a lot of data. See *Extension to Native File Functions* on page 5.
- The **Recurse** variant option for ⎕NINFO has been extended to allow a limit to the level of sub-directories that are searched.
- ⎕NINFO has been extended to provide file times as UTC Dyalog Date Numbers.
- The list of standard characters for the S qualifier of ⎕FMT has been extended to include the high minus symbol (¯).
- An option is provided to control the implementation of lexical scope in Namespace and Class scripts. This extension applies only to ⎕FIX; scripts fixed using the editor are unaffected by this change. See *Lexical Scope in Scripts* on page 9 and *Fix Script* on page 29.
- An option is provided to control whether or not source code is retained in the workspace exactly as it was typed. This is now the default. See *Source as Typed* on page 13, *Discard Source Code* on page 44 and *Discard Source Information* on page 43.
- Support is added for LZ4 *frames* which allows the LZ4 compression library to handle data > 2GB in size. Previously, extremely large file components could not be compressed because the LZ4 library could not process them and they were written uncompressed. Now, these large components will (in the absence of any other reason) be compressed as well. However, such components will be unreadable by previous Versions. 3012⌶ allows the user to specify that LZ4 frames should not be used for component compression, for when interoperability is required. See *Enable Compression of Large Components* on page 43. 219⌶ has been extended to allow arrays >2GB to be compressed.
- There is a new I-beam function to set the parameters for generating aplcore files dynamically. See *Set aplcore Parameters* on page 38.
- ⎕FHOLD now accepts an optional left argument to specify a time-out. See *File Hold* on page 28.
- The right argument to ⎕SIGNAL may include 1006 (TIMEOUT error).

GUI Improvements

- The HTMLRenderer provides a number of new Properties and Methods. See *AllowContextMenu* on page 47, *ExecuteJavaScript* on page 48, *GetZoomLevel* on page 48, *IsLoading* on page 48, *LoadEnd* on page 49, and *SetZoomLevel* on page 50.

Session Initialisation Improvements

- There is a new Boolean configuration parameter that determines whether or not the Session is initialised on start-up. See *DYALOG_INITSESSION* on page 24. By default this is 1 for the development version and 0 for the run-time. This must be 1 to use Link.
- Nested directory structures are now supported.
- Every top-level directory that is loaded as a namespace in ⎕SE can have a Run function which will be called after everything has been loaded. See *DyalogStartup_X* on page 24 for how to disable this.
- Note that Link is now required for Session initialisation. See *DyalogLink* on page 24 for how to use a non-default Link.
- The list of directories from which ⎕SE is populated can now be extended rather than just replaced. See *DyalogStartupSE* on page 25.

Session Improvements

- Multi-line input, which was introduced in 18.0, is now enabled by default. On Windows this can be changed by selecting/unselecting the "Enable Multiline Input" Checkbox on the Session tab of the Configure dialog, or on all platforms by setting the configuration parameter DYALOG_LINEEDITOR_MODE to 0 (disabled) or 1 (enabled).
- The log file used by the Session is now unique to the instance of Dyalog that is running and is reported by a new LogFile property of ⎕SE. Previously, multiple instances of the Dyalog program shared the same session log. See *Log_File* on page 26.
- Session log files are now saved in JSON format.
- The Session window (optionally) reserves the first column for information. See *DYALOG_GUTTER_ENABLE* on page 23. On TTY-versions it is hidden by default.
- Lines output to the Session which are associated with errors are now syntax coloured using the *error* colour for the selected Session colour scheme.
- The Caption property of the Session, which was previously read-only, can now be set. See below.
- When you edit an object by double-clicking the mouse or pressing the <ED> key, or executing)ED, and the name of the object is followed by [n], the Editor will position the cursor on line number n. Note that there must not be a space between the last character of the name and the "[".

Session Caption

The Caption property of the Session may be set dynamically to a character vector comprising free text and field names. Field names must be enclosed in braces and are replaced in-situ by corresponding values.

Field Name	Description
{TITLE}	the window specific text
{WSID}	⎕WSID
{NSID}	current namespace
{SNSID}	short version of namespace (no #.)
{PRODUCT}	e.g. Dyalog APL/W
{VER_A}	e.g. 19
{VER_B}	e.g. 0
{VER_C}	e.g. 47586 (SVN revision)
{PID}	process ID (decimal)
{CHARS}	"Classic" or "Unicode"
{BITS}	"32" or "64"

Table 1: Session Caption Fields

Example:

```
      ⎕SE.Caption←'Pete: {WSID} {Product} {VER_A}.{VER_B}'
```

The Session caption in a `CLEAR WS` will change to:

```
      Pete: CLEAR WS Dyalog APL/W-64 19.0
```

Note that Caption returns the codified string used to set it.

```
      ⎕SE.Caption
Pete: {WSID} {Product} {VER_A}.{VER_B}
```

Extension to Native File Functions

`⎕NMOVE` and `⎕NCOPY` now provide a feature to run an APL function as a callback during processing. This is implemented by the **ProgressCallback** Variant option.

ProgressCallback Option

Overview

If this option is enabled, the system function invokes an APL callback function as the file operation (move or copy) proceeds. A system object is used to communicate between the system function and the callback. The file operation has 4 distinct stages:

1. The start of the operation. The callback is invoked before any files are scanned or processed. This gives the application the opportunity to set parameters that control the frequency of callbacks during the operation itself.
2. The optional scan phase during which the system function enumerates the files that will be involved in the copy or move operation. The file count obtained is used to set the `Limit` field. The application may use this subsequently to indicate the degree of progress.
3. The main processing (move or copy) of the files.
4. The end of the operation.

The callback function is invoked once at the start of the operation, during the (optional) scan and processing stages, and finally once at the end of the operation. During the scan and processing stages, the `Skip` and `Delay` options provide alternative ways to control the frequency with which the callback is invoked.

If both options are 0, the callback will be invoked after every file is processed. However, if there are a large number of small files involved, and you simply want to update a progress bar, this may prove to be unnecessarily frequent, and will increase the total time required to complete the operation.

If you want to update a progress bar regularly (for example every second), the `Delay` option (1000 = 1 second) is the better choice. In other circumstances, you might choose to use `Skip`.

If you use both options, the callback will be invoked when *both* apply, so if you set `Skip` to 10 and `Delay` to 5000, the callback will be invoked after at least 10 files have been processed and at least 5 seconds have elapsed since the previous invocation of the callback.

The value of the ProgressCallback variant option may be:

`fn`	The name of the callback function.
`fn data`	The name of the callback function, and an array or namespace which is to be passed to the callback in its left argument.

The right argument given to the callback function is a 3-element vector:

[1]	Function	Character vector which identifies the function that caused the callback to be executed; either '⎕NCOPY' or '⎕NMOVE'.
[2]	Event	Character vector describing the event that lead to the callback being executed. See below.
[3]	Info	Reference to a namespace containing information about the event. See below.

Event

Event is a character vector which indicates the stage of the copy or move operation..

'Start'	Reported by the first invocation of the callback which occurs before any files are scanned or processed. This may be used to set the parameters that control the operation. See *Options* on page 8.
'Scan'	Indicates that the system function is in the initial phase of scanning the files in order to calculate Limit. See *ScanFirst* on page 8.
'Progress'	Indicates that the system function is at the main stage of the operation and is moving or copying the files.
'Done'	Indicates that all files have been processed.

Note that there will always be at least 2 invocations of the callback, to indicate the start and end of the operation.

Info

Info is a ref to a namespace that contains information about the event. This namespace persists for the duration of the execution of the system function and contains the following fields:

`Progress`	A number between `0` and `Limit`. When the event code is `'Start'`, `Progress` is `0`. Every time a file or directory is processed, `Progress` is increased by 1. Finally when the event code is `'Done'`, `Progress` will be equal to `Limit`.
`Limit`	The maximum value of `Progress`. This value might change during the file operation if it doesn't do a full discovery first (the `ScanFirst` option is 0), or if the file structure changes between the scan and the copy/move.
`Last`	A vector of file names which have been processed since the last invocation of the callback function. The user can specify the maximum length of this vector by setting the `LastFileCount` option. The names in this list are the source names, and not the destination names. The `Last` vector is always empty when the event is `'Start'`, and it is cleared when going from the `'Scan'` phase to the `'Progress'` phase, to avoid any confusion.
`Data`	A field that is reserved for the user to store data which persists between invocations of the callback. It could for example be used to keep a sequence number, to count the number of times the callback had been run.
`Options`	This is a namespace which contains the information that controls the future execution of the callback. The options persist between the calls to the callback, so there is no need to set them again unless they should be changed. The fields and their default values are described below.

Options

This is a namespace which contains options that control future invocations of the callback. The options persist between these invocations, so there is no need to set them again unless they should be changed. The fields and their default values are:

Field	Default	Description
`ScanFirst`	1	Specifies if the file operation should do a "scan pass" before moving/copying the files. This stage just enumerates the files to determine how many there are. This will ensure `Limit` has a realistic value when the actual processing of the files happens. The overhead is small in comparision with the time it takes to process the files. The `ScanFirst` field is only inspected right after the first invocation of the callback function, with the event code `'Start'`.
`Delay`	0	Specifies the number of milliseconds to wait, until the callback will be called again. If all file operations finish before this time, the callback function is called anyway, with the event code `'Done'`. If a slow file operation is happening (such as copying a big file), the actual delay before the callback is invoked might be longer than the value of `Delay`.
`Skip`	0	Specifies a number of files to skip between invocations of the callback function. If you are only interested in getting a callback for each 10th file, you should set this option to 9 for example.

Field	Default	Description
LastFileCount	1	An integer, specifying the maximum number of the latest filenames to be stored in the Last field. The default is to only store the last file processed, but if Delay or Skip are non-zero, multiple files could have been processed between calls to the callback function. A value of 5 for example, will make sure that the 5 last files processed before calling the callback, will have their names in the Last field. The Last field might have fewer elements than LastFileCount, if the number of files processed since the last call is less than LastFileCount. The special value ¯1 indicates that the Last field should contain **all** the last files since the last call (no limit).

The result of the callback function must be a Boolean scalar, indicating whether or not the ⎕NCOPY or ⎕NMOVE should continue or stop.

1: Execution should continue.

0: Execution should stop. In this case, an INTERRUPT (event 1003) is signalled.

Lexical Scope in Scripts

Historical Note

Lexical scope in scripts has been part of Dyalog since the implementation of Object Oriented Programming in Version 11.0, and is only partially documented. This section provides additional explanation and extends the discussion to Classes.

Introduction

Objects (Namespaces and Classes) that are defined using scripts, either in the workspace or in script files, may include nested objects (sub-namespaces and sub-classes). If so, Dyalog applies a form of lexical scope to all these objects to allow them to reference one another. Dyalog otherwise uses dynamic scope .

This feature makes it possible to implement a class structure, in which members of the class tree may access one another, and it provides a way for classes to share data stored in a namespace.

When Dyalog fixes nested classes and namespaces in a script, references between parent and child objects are inserted to allow them to reference one another, preventing what would otherwise be VALUE ERROR. For example:

```
:Class Parent
:Access Public

    :Namespace Data
    :EndNamespace

    ∇ new name
      :Access Public
      :Implements Constructor
      Data.Name←name
    ∇

    :Class Child
    :Access Public
        :Field Public Name

        ∇ new name
          :Access Public
          :Implements Constructor
          Name←name
          Name,' is a child of ',Data.Name
        ∇
    :EndClass
:EndClass

      pete←⎕NEW Parent 'Pete'
      andy←⎕NEW pete.Child 'Andy'
Andy is a child of Pete
```

In this example, the namespace Data is accessible from the Parent class, and from any sub-classes within it and can therefore be used to share information between them. A more realistic example might be to share the value of the tie number of a component file.

Note that this is not possible using variables or Fields; **data to be shared between nested classes must be stored in a namespace**.

Variant Options for ⎕FIX

Despite the essential benefits of lexical scope, there are circumstances in which it is undesirable and `⎕FIX` provides fine control over the insertion of references. See *InjectReferences Option* on page 33.

Note that the ability to control lexical scope in this way applies only to `⎕FIX`. When a nested script is fixed by the Editor, the default lexical scope (**InClasses**) is applied. If, after fixing a script from the Editor, you wish to apply a different option (**All** or **None**) it is necessary to re-fix the script using `⎕FIX 62 ATX 'name'`.

Session Gutter

The first column of the Session Window (the Session Gutter) is by default reserved to display the following information:

- A small red circle. This indicator is used on every line that is modified in the session, including old ones (e.g. if you move up the session and modify them, without pressing <ER>) . The indicators show which session lines will be re-executed when you subsequently press <ER>.
- A left bracket [to identify groups of default output. Note that other forms of output are not identified in this way.

```
      2+2
4
      ⍳⍳3
 1 1 1  1 1 2  1 1 3
 1 2 1  1 2 2  1 2 3
      ⍳⍳4
 1 1 1 1  1 1 1 2  1 1 1 3  1 1 1 4
 1 1 2 1  1 1 2 2  1 1 2 3  1 1 2 4
 1 1 3 1  1 1 3 2  1 1 3 3  1 1 3 4

 1 2 1 1  1 2 1 2  1 2 1 3  1 2 1 4
 1 2 2 1  1 2 2 2  1 2 2 3  1 2 2 4
 1 2 3 1  1 2 3 2  1 2 3 3  1 2 3 4
      ⎕←⍳⍳3
 1 1 1  1 1 2  1 1 3
 1 2 1  1 2 2  1 2 3
      2÷0
DOMAIN ERROR: Divide by zero
      2÷0
       ^
```

The Session Gutter may be enabled and disabled using the **DYALOG_GUTTER_ ENABLE** parameter. It is disabled by default in the TTY interface.

Source as Typed

Historical Introduction

When an object containing executable code such as a function, operator, class, or namespace is defined in a workspace either by an editor or by the system function ⎕FX, the object is tokenised into an internal form. Historically, this was the only form of the object, and both the editor and system functions like ⎕CR, ⎕VR, ⎕NR reconstitute the source code from the internal form. This reconstituted source lacks extraneous white space and the precise numerical formatting that the user originally entered, for example.

When classes and scripted namespaces were introduced, the source code was stored in text form for these objects, as it was typed, in addition to the tokens which were still used at runtime. The function ⎕SRC was added to return this text, and a new function ⎕FIX was added to define objects that also have source code.

Subsequently, ⎕FIX was extended to allow the definition of functions and operators which include source code, as well as the use of source files outside the workspace to store the source code of an object. However, unless a function or operator was defined using an external file, the editor continued to only store the tokenised form in the workspace, in order to save space.

Current Behaviour

From version 19.0 onwards, the default is that the editor stores source code *as it was typed in by the user* for **all** objects, in addition to the tokenised form. When an object is defined from an external source file using ⎕FIX, a copy of the source is also retained in the workspace.

In order to maintain backwards compatibility with applications that rely on the canonical representation returned by ⎕CR, ⎕VR , ⎕NR, these functions continue to reconstitute the source from tokens; and ⎕FX continues to only store the tokenised form. If you wish to access the source as typed, you should use ⎕SRC, or 60 ⎕ATX, and you should use ⎕FIX, to define not only namespaces and classes but functions and operators as well.

When the user opens an object in the Editor, the saved source code is presented if it exists. If the object was defined from a file and the source held in the workspace differs from the contents of the file, the user will be asked to decide whether to use the file or break the link and use the source in the workspace. If no source code is available, it is reconstituted from the internal form.

Note however, that there is no mechanism to reconstitute a script, as a whole, from its tokenised form. If there is no source code, the Namespace or Class appears as if it were created using `⎕NS` rather than having originated from a script. It cannot be opened in the Editor and the result of `⎕SRC` is empty. However, the source code for individual functions and operators within the Namespace or Class will be reconstituted from their individual tokenised code when required.

The functions `⎕SRC` and `62 ⎕ATX` (most precise available source) use the same logic as described above to generate a result.

Source code saved in the workspace is compressed to minimise space usage.

Note that the white space in comment statements is retained in both the compiled form and compiled form of a function.

The Boolean parameter **DYALOG_DISCARD_FN_SOURCE** (default 0) and `5172⌶` (Discard Source Information) allow the user to enable or disable this feature for functions and operators. The *AutoFormat Functions* option is automatically disabled if the **DYALOG_DISCARD_FN_SOURCE** parameter is 1. Note that the user can format code on demand).

`5171⌶` (Discard Source Information) discards source code and file information for scripted objects, namespaces, classes, functions, and operators that is saved in the workspace.

Note that, to ensure that they can be used by Classic Edition, the source code has been discarded from all the workspaces supplied by Dyalog as part of the distribution.

See also: *Discard Source Code* on page 44 and *Discard Source Information* on page 43.

Bug Fixes

A number of bug fixes implemented in Version 19.0 may change the way that existing code operates and are therefore documented in this section.

- When **APL_COMPLEX_AS_V12** is set, the circular functions `(X○Y)` with `(|X)>7`, generate `DOMAIN ERROR` if the result would be complex.
- Previously, if GetTextSize was given an invalid font name it would use the default for the window that the method was invoked in. Now, invalid font names correctly generate `DOMAIN ERROR`.
- `⎕FMT` using the `E` qualifier now behaves as intended.

```
      'E13.6' ⎕FMT ¯4.56789E¯12 ¯4.56789E¯123 ⍝ previous
 ¯4.56789E¯12
 ¯4.5678E¯123

      'E13.6' ⎕FMT ¯4.56789E¯12 ¯4.56789E¯123 ⍝ new
¯4.56789E¯12
¯4.56789E¯123  ⍝ NEW - note alignment of the 'E's!

⍝ Old behaviour - note ¯1.234 printed as ¯1.23
⍝ despite 4 digits requested

        '|',('E12.4' ⎕FMT ¯1.234E¯123),'|'
|  ¯1.23E¯123|


⍝ NEW behaviour - honour request for 4 digits
        '|',('E12.4' ⎕FMT ¯1.234E¯123),'|'
| ¯1.234E¯123|

⍝ Honouring request can now prevent fitting!

        '|',('E10.4' ⎕FMT ¯1.234E¯123),'|'  ⍝ Old
|¯1.23E¯123|

        '|',('E10.4' ⎕FMT ¯1.234E¯123),'|'  ⍝ NEW
|**********|
```

Announcements

Supported Versions

The supported versions of Dyalog are now versions 19.0, 18.2, 18.0, and 17.1. Version 17.0 and earlier versions are no longer supported.

Dyalog on macOS

Version 19.0 is expected to be the last version that will be available for Intel-based Macs. Version 19.0 is natively available for both Intel and ARM-based Macs.

Performance Issue with Namespaces

We have identified a namespace performance issue which is especially noticeable with JSON Import. We have a fixed planned for the next release of Dyalog. In the meantime, there is an easy workaround. For details, see *Language Reference Guide: JSON Convert*.

Hash and Lookup Tables

In the next major version of Dyalog the performance of the set functions will be improved. The new code will involve increasing the amount of workspace allocated to the internal tables used by these functions. These tables are described using the terms *hash table* and *lookup table*. The latter refers to internal tables that do not require hashing.

For more information, see *Programming Reference Guide: Search Functions and Hash Tables* and *Language Reference Guide: Hash Array*.

The proposed size increase may potentially cause `WS FULL` errors or may change the frequency of workspace compactions.

To allow the user to evaluate the effect of this future change on their applications, two new I-beam functions have been provided. These functions increase the space allocated to the internal tables for the sole purpose of testing these potential effects. The new I-beams may affect performance either directly or by triggering a change of algorithm, but should not be used for that sole purpose since performance degradation in some cases cannot be excluded. See *Hash Table Size* on page 44 and *Lookup Table Size* on page 45.

When the next major release is published, it is anticipated that few, if any users will notice negative effects from changing the internal table sizes. Rather, they will benefit from the improved performance that will result.

PCRE2 Upgrade

Dyalog uses the PCRE 8.x library to support regular expression searches in ⎕R, ⎕S and in the IDE. PCRE 8 is widely used, but future development and maintenance of PCRE will be based upon the newer PCRE2 (PCRE 10.x) library. Dyalog intends to switch to the new library in a forthcoming release.

Chromium Embedded Framework (CEF)

Version 19.0 is supplied with CEF version 121 on all supported platforms.

Forthcoming Removal of 819⌶

The system function ⎕C was introduced in Dyalog version 18.0, at which point we announced that 819⌶ was deprecated. 819⌶ is still present in Dyalog version 19.0, but it will be removed from the next version.

There is a temporary new configuration parameter **DYALOG_IBEAM819**.

If **DYALOG_IBEAM819** is set to 0, use of 819⌶ will signal an error and the DMX.Message will state that it has been withdrawn; in other words, the behaviour is what you would get with the next release. This is to help users prepare for its removal now if they want to.

819⌶ will only operate if either **DYALOG_IBEAM819** is not set, or **DYALOG_IBEAM819** is set to 1.

Forthcoming Removal of Array Editor

Version 19.0 is expected to be the last version that will include David Liebtag's Array Editor.

Forthcoming Removal of Syncfusion from Microsoft Windows installation images

Version 19.0 is expected to be the last version that will include the Syncfusion library of WPF controls; Dyalog Ltd will cease to offer support for the Syncfusion controls from the end of September 2024.

The Syncfusion licence provided with Dyalog 19.0 will continue to be valid for use with Dyalog 19.0 beyond this date, but later versions of Dyalog will not include this licence.

Removal of RConnect (R Interface)

RConnect (the R interface) and the R Interface Guide, are no longer included with Dyalog. Instead, Dyalog Ltd recommends *RSconnect - R connection for Dyalog APL with Rserve*, which can be obtained from https://github.com/kimmolinna/rsconnect.

System Requirements

Microsoft Windows

Dyalog version 19.0 is supported on versions of Microsoft Windows from Windows 10 or Windows Server 2016 upwards.

The Dyalog version 19.0 .NET Framework interface requires version 4.0 or greater of Microsoft .NET Framework. It does *not* operate with earlier versions of the .NET Framework. In addition:

- .NET Framework version 4.5 is needed for full Data Binding support (including support for the `INotifyCollectionChanged` interface, which 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).
- .NET Framework version 4.6 is needed to run the Syncfusion libraries supplied with Dyalog version 19.0.
- IIS needs to be installed before installing Dyalog APL in order to access the examples in the `Samples/asp.net` sub-directory – if IIS and ASP.NET are not present, the `asp.net` sub-directory will not be installed during the Dyalog installation.

Note that .NET Framework is specific to Microsoft Windows; the cross-platform .NET is also supported (see below).

AIX

Dyalog version 19.0 requires AIX 7.2 or higher, and a POWER9 chip or higher.

Raspberry Pi

Dyalog 32-bit Unicode supports 32-bit Raspberry Pi OS Buster or later but is not supported on the Raspberry Pi Pico. There is no 64-bit version of Dyalog for the Pi, nor will the 32-bit version run under 64-bit Raspberry Pi OS.

Non-Pi Linux

Dyalog version 19.0 only exists as 64-bit interpreters – there are no 32-bit versions. It is built on Ubuntu 20.04; it should run on all recent distributions. For further information, see the Dyalog UNIX and Linux forum.

macOS

Dyalog version 19.0 (64-bit version; there is no 32-bit version) is supported on both Intel and ARM processors. The macOS version required for Dyalog version 19.0 on each is:

- on Intel: macOS 11.6.1 (Big Sur) onwards
- on ARM: macOS 13.4.1 (Ventura) onwards

Dyalog for ARM is only supported on Macs with an ARM processor. Dyalog for Intel is supported on Macs with an Intel chip or Macs with an ARM chip and Rosetta enabled. Each has its own shared libraries. These, and any other customisations, must match the Dyalog installation.

Cross-platform Microsoft .NET Interface

The Dyalog version 19.0 .NET interface requires version 8.0 of Microsoft .NET or higher.

HTMLRenderer and Chromium Embedded Framework (CEF)

The HTMLRenderer is supported on the following platforms:

- Windows
- macOS (both Intel and ARM-based)
- Linux

It is not supported on the Raspberry Pi

To see which version of CEF was used when the HTMLRenderer was built, query the CEFVersion property of an instance of the HTMLRenderer:

```
      'hr' ⎕WC 'HTMLRenderer'
      hr.CEFVersion[2 3]⍝ CEF Maj Version and Commit No
121 3
```

Interoperability

Introduction

Workspaces and component files are stored on disk in a binary format. This format differs between machine architectures and among versions of Dyalog. For example, a file component written from Windows will have an internal format that is different from one written from AIX. Similarly, a workspace saved from Dyalog Version 19.0 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. 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.

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 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.
- Under Windows, a 32-bit version of Dyalog APL can only access 32-bit DLLs, and a 64-bit version of Dyalog APL can only access 64-bit DLLs. This is a Windows restriction.
- Objects saved in the workspace that are connected to external resources lose those connections when loaded or copied by an interpreter with different architecture.

In particular:

If a workspace containing:

- .NET objects, objects created by ⎕WC , or instances of built-in objects (excluding instances of user-defined classes) created by ⎕NEW.

or

- variables containing the ⎕OR of or refs to such objects

is loaded by an interpreter with differing architecture (32 vs 64) from the version that saved it, Dyalog displays:

```
GUI objects could not be recreated;
the file is from an incompatible architecture
```

The names of all incompatible objects are instantiated as plain namespaces, with any compatible contents (such as functions and variables) preserved.

If a component containing the ⎕OR of or refs to such objects is read by an interpreter with differing architecture (32 vs 64) from the version that wrote it, each incompatible object is instantiated as a plain namespace, preserving compatible contents as above.

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 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. Note that the problematic Unicode data may be in that part of a workspace which holds the information needed to generate ⎕DM and ⎕DMX, so calling)reset before)save in the Unicode interpreter may eliminate the TRANSLATION ERRORs.

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/Partition and Where/Interval Index functions and the Rank/Atop, 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.

TCPSocket Objects and Conga

TCPSocket objects 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.

Log Files

Log (.dlf) files can only be used by the version and edition with which they were created and saved.

Chapter 2:

Configuration Parameters

The following table summarises changes to configuration parameters in Version 19.0.

Parameter	Description	Change
DYALOG_GUTTER_ENABLE	Enable or disable Session Gutter	New
DYALOG_DISCARD_FN_SOURCE	Specifies whether source code is retained in the workspace	New
DYALOG_INITSESSION	Specifies whether or not the Session is initialised on start-up	New
DyalogLink	Specifies the directory for Link	New
DyalogStartupSE	Extended	Enhancement
DyalogStartup_X	Specifies whether the Run function is executed during Session startup	New
Log_File	Enhanced to support multiple Session log files	Enhancement

DYALOG_GUTTER_ENABLE

This Boolean parameter specifies whether (1) or not (0) a Gutter is displayed in the left-most column of the Session window. This gutter is used to display:

- A small red circle. This indicator is used on every line that is modified in the session, including old ones (e.g. if you move up the session and modify them, without pressing <ER>) . The indicators show which session lines will be re-executed when you subsequently press <ER>.
- A left bracket [to identify groups of default output. Note that other forms of output are not identified in this way.

The default value is 0 for the TTY interface, and 1 otherwise.

DYALOG_DISCARD_FN_SOURCE

This Boolean parameter specifies whether (1) or not (0) source code is discarded from the workspace when an object is fixed. The default value is 0 which means that source code is retained in the workspace and will subsequently be presented for editing as it had been saved previously.

For further information, see *Language Reference Guide: Discard Source Information* and *UI Guide: Source As Typed*.

DYALOG_INITSESSION

This Boolean parameter governs whether (1) or not (0) Dyalog performs Session Initialisation on start-up.

The default is 1 for development and shell script versions, and 0 for run-time versions.

Session initialisation makes Link, SALT and other things available. These features depend on DYALOG_INITSESSION being 1 (explicitly or by default).

DyalogLink

This parameter specifies the name of the directory containing the code for Link. The default is `[DYALOG]/StartupSession/Link`.

Note that Link is required for Session initialisation.

For further information, see https://dyalog.github.io/link/4.0/Usage/Installation.

DyalogStartup_X

During Session initialisation, code is loaded from the directories specified by the **DyalogStartupSE** parameter into a corresponding namespace tree in the Session namespace `⎕SE`. Optionally, the code is then executed.

If **DyalogStartup_X** is 0 (the default if not defined), the `Run` function (if it exists) in each *top-level* namespace loaded during Session start-up is executed. The namespaces are processed in alphabetical order.

If **DyalogStartup_X** is 1, the `Run` function is not executed.

Other values are reserved for future extension.

See also: *DyalogStartupSE* on page 25.

DyalogStartupSE

This parameter specifies one or more *Session initialisation* directories that contain APL code to be installed in ⎕SE. If this parameter is not specified, the default is a directory named `StartupSession` located in three standard locations.

Under Windows these might be:

1. `C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode`
2. `C:\Users\Pete\Documents\Dyalog APL Files`
3. `C:\Users\Pete\Documents\Dyalog APL-64 19.0 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

The parameter is a string containing the list of directory names separated by ";" on Windows, ":" elsewhere.

If DyalogStartupSE begins with the specified separator, the default list is *extended* rather than *replaced*.

Note that the effective sequence of directories specified by this parameter is converted to a vector of character vectors and stored in `⎕SE.Dyalog.StartupSession.AllPaths`.

If unset or extended (that is, starts with a : separator):

- the effective StartupSession directory in [DYALOG] is available as `⎕SE.Dyalog.StartupSession.Dyalog`.
- the StartupSession directory in the version-agnostic directory is available as `⎕SE.Dyalog.StartupSession.VerAgno`.
- the StartupSession directory in the version-specific directory is available as `⎕SE.Dyalog.StartupSession.VerSpec`.

Log_File

This parameter specifies the pathname to the Session log file; it can be absolute or relative to the working directory.

The Session log file is not interchangeable between different versions/editions/widths of Dyalog – this means that opening a new instance of Dyalog will overwrite any contents of the Session log file populated by an already-running instance. However, if the LOG_FILE parameter contains a '*' (e.g. `JD.*.dlf`) then at start-up Dyalog will attempt to open, and then **lock**, a file where the '*' has been replaced with an increasing integer value (starting with 000, so `JD.000.dlf`, `JD.001.dlf` etc). If said file cannot be opened and locked, the value will be incremented. The process will fail, and no log will be used if the extension number would exceed 999.

The default is `Users\<username>\Documents\Dyalog APL-<bits> <DyalogMajor>.<DyalogMinor> <Unicode|Classic> Files\default_*.dlf`, for example, `Users\Bob\Documents\Dyalog APL-64 19.0 Unicode Files\default_*.dlf`

Note that the LogFile property of `⎕SE` reports the name of the log file that is being used.

Chapter 3:

Language Reference Changes

The following table summarises the main changes to language features in Version 19.0.

Function/Operator	Description	Change
⎕TALLOC	Allocate Token Range	New system function
2000⌶	Memory Management Statistics	Extended I-beam function
219⌶	Compress Vector of Short Integers	Extended I-beam function
1302⌶	Set aplcore Parameters	New I-beam function
3012⌶	Enable Compression of Large Components	New I-beam function
5171⌶	Discard Source Information	New I-beam function
5172⌶	Discard Source Code	New I-beam function
9468⌶	Hash Table Size	New I-beam function
9469⌶	Lookup Table Size	New I-beam function
⎕NCOPY	Native File Copy	New ProgressCallback variant
⎕NMOVE	Native File Move	New ProgressCallback variant
⎕FHOLD	File Hold	New left argument to specify a time-out
⎕SIGNAL	Signal event	Now accepts 1006 (TIMEOUT error)

File Hold {R}←{X} ⎕FHOLD Y

Access code 2048

This function holds component file(s) and/or external variable(s). It is used to synchronise access to resources shared between multiple cooperating Dyalog processes. It is not intended to synchronise access between Dyalog threads; for this purpose you should use `:Hold`.

For a multi-threaded and multi-process application, a single `⎕FHOLD` is used to synchronise inter-process access, while `:Hold` is used in multiple threads to synchronise access between threads in the same process. See also *Programming Reference Guide: Hold Statement*.

If applied to component files, then `Y` is an integer scalar, vector, or one-row matrix of file tie numbers, or a two-row matrix whose first row contains file tie numbers and whose second row contains passnumbers.

If applied to external variables, then `Y` is a simple scalar character, a character vector, a non-simple scalar character vector, or a vector of character vectors that specifies one or more names of external variable(s) (NOT the file names associated with those variables). Note that when `Y` is simple, each character in `Y` is interpreted as a variable name. If applied to component files **and** external variables, `Y` is a vector whose elements are either integer scalars representing tie numbers, or character scalars or vectors containing names of external variables.

The effect is as follows:

1. **All** of the user's preceding holds (if any) are released, whether referenced in `Y` or not.
2. Execution is suspended until the designated files are free of holds by any other task.
3. When all the designated files are free, execution proceeds. Until the hold is released, other tasks using `⎕FHOLD` on any of the designated files will wait.

The optional left argument `X` is a non-negative integer that specifies a time-out in milliseconds. If step 2 (see above) does not complete before the time-out value specified by `X`, `⎕FHOLD` times out and signals a `TIMEOUT` error (1006) after releasing any holds that have succeeded.

A time-out value of 0 indicates that the `⎕FHOLD` should time out at once without waiting if it cannot immediately acquire all holds. If `X` is `¯1`, `⎕FHOLD` behaves as the monadic case, and does not time out.

If Y is empty, all of the user's preceding holds (if any) are released, and execution continues.

A hold is released by any of the following:

- Another ⎕FHOLD
- Untying or retying all the designated files. If some but not all are untied or retied, they become free for another task but the hold persists for those that remain tied.
- Termination of APL.
- Any untrapped error or interrupt.
- A return to immediate execution mode.

Note that a hold is not released by a request for input through ⎕ or ⍞.

⎕FHOLD is generally useful only when called from a defined function, as holds set in immediate execution (desk calculator) mode are released immediately.

If Y is a matrix, the shy result R is Y[1;]. Otherwise, the shy result R is Y.

Examples:

```
      ⎕FHOLD 1

      ⎕FHOLD ⍬

      ⎕FHOLD ⊂'XTVAR'

      ⎕FHOLD 1 2,[0.5]0 16385

      ⎕FHOLD 1 'XTVAR'

      3000 ⎕FHOLD 1
TIMEOUT
      3000 ⎕FHOLD 1
           ^
```

Fix Script {R}←{X}⎕FIX Y

⎕FIX establishes Namespaces, Classes, Interfaces and functions from the script specified by Y in the workspace.

In this section, the term *namespace* covers scripted Namespaces, Classes and Interfaces.

Y may be a simple character vector, or a vector of character vectors or character scalars. The value of X determines what Y may contain.

If Y is a simple character vector, it must start with file://, followed by the name of a file which must exist. The contents of the file must follow the same rules that apply to Y when Y is a vector of character vectors or scalars. The file name can be relative or absolute; when considering cross-platform portability, using "/" as the directory delimiter is recommended, although "\" is also valid under Windows.

If specified, X must be a numeric scalar. It may currently take the value 0, 1 or 2. If not specified, the value is assumed to be 1.

If X is 0, Y must specify a single valid *namespace* which may or may not be named, or a file containing such a definition. If so, the shy result R contains a reference to the *namespace*. Even if the *namespace* is named, it is not established *per se*, although it will exist for as long as at least one reference to it exists.

If X is 1, Y must specify a single valid *namespace* which may or may not be named, or a file containing such a definition. If so, the shy result R contains a reference to the *namespace*. If Y contains the definition of a named *namespace*, the *namespace* is established in the workspace.

If X is 2, Y is either a character vector containing the name of a script file, or a vector of character vectors that represents a script.

Y may specify a series of **named** *namespaces* or function definitions, or a combination of functions and namespaces.

- If the script contains more than one item, tradfn definitions must be delimited by ∇symbols.
- Derived and assigned functions may be specified only within namespaces.

In this case, the shy result R is a vector of character vectors, containing the names of all of the objects that have been established in the workspace; the order of the names in R is not defined. Currently 2 ⎕FIX is not certain to be an atomic operation, although this might change in future versions.

Example 1

In the first example, the Class specified by Y is *named* (MyClass) but the result of ⎕FIX is discarded. The end-result is that MyClass is established in the workspace as a Class.

```
      ⎕←⎕FIX ':Class MyClass' ':EndClass'
#.MyClass
```

Example 2

In the second example, the Class specified by `Y` is *named* (`MyClass`) and the result of `⎕FIX` is assigned to a different name (`MYREF`). The end-result is that a Class named `MyClass` is established in the workspace, and `MYREF` is a reference to it.

```
      MYREF←⎕FIX ':Class MyClass' ':EndClass'
      )CLASSES
MyClass MYREF
      ⎕NC'MyClass' 'MYREF'
9.4 9.4
      MYREF
#.MyClass
      MYREF≡MyClass
1
```

Example 3

In the third example, the left-argument of `0` causes the named Class `MyClass` to be visible only via the reference to it (`MYREF`). It is there, but hidden.

```
      MYREF←0 ⎕FIX ':Class MyClass' ':EndClass'
      )CLASSES
MYREF
      MYREF
#.MyClass
```

Example 4

The fourth example illustrates the use of un-named Classes.

```
      src←':Class' '∇Make n'
      src,←'Access Public' 'Implements Constructor'
      src,←'⎕DF n' '∇' ':EndClass'
      MYREF←⎕FIX src
      )CLASSES
MYREF
      MYINST←⎕NEW MYREF'Pete'
      MYINST
Pete
```

Example 5

In the final example, the left argument of 2 allows a script containing multiple objects to be fixed:

```
      src←':Namespace andys' '∇foo' '2' '∇'
      src,←':EndNamespace' 'dfn←{⍺ ⍵}' '∇r←tfn'
      src,←'r←33' '∇' ':Class c1' '∇goo' '1'
      src,←'∇' ':EndClass'
      ≢⎕←2⎕fix src
 c1  tfn  dfn  andys
4
```

Restriction

⎕FIX is unable to fix a namespace from Y when Y specifies a multi-line dfn which is preceded by a ⋄ (diamond separator).

```
      ⎕FIX':Namespace iaK' 'a←1 ⋄ adfn←{' '⍵' ' }'
':EndNamespace'
DOMAIN ERROR: There were errors processing the script
      ⎕FIX':Namespace iaK' 'a←1 ⋄ adfn←{' '⍵' ' }'
':EndNamespace'
      ^
```

Variant Options

⎕FIX may be applied using the Variant operator with the options **Quiet**, **FixWithErrors**, **AllowLateBinding** and **InjectReferences**. These options apply only to namespaces and classes specified by the script. There is no principal option.

Quiet Option

0	If the script contains errors, these are displayed in the Status Window.
1	If the script contains errors, the errors are not shown in the Status Window.

FixWithErrors Option

0	If the script contains errors, ⎕FIX fails with DOMAIN ERROR.
1	⎕FIX fixes all the namespaces and classes in the script regardless of any errors they may contain.
2	If the script contains errors, ⎕FIX displays a message box prompting the user to choose whether or not to fix all the offending namespaces and classes in the script.

AllowLateBinding Option

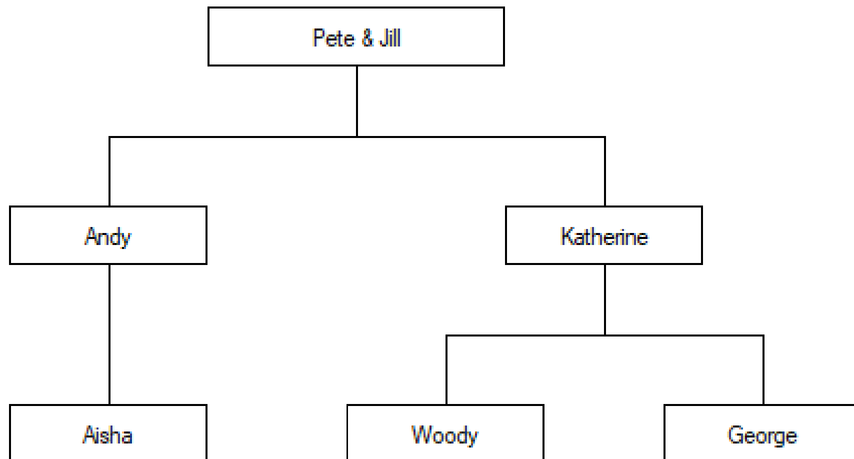
0	⎕FIX will only fix a Class whose Base class (if specified) is defined in the script or is present in the workspace.
1	⎕FIX will fixes a Class whose Base class is neither defined in the script nor present in the workspace.

InjectReferences Option

'All'	In order to implement lexical scope, ⎕FIX will insert internal references into all objects in the script.
'InClasses'	In order to implement lexical scope, ⎕FIX will insert internal references ONLY into Classes and sub-classes in the script, but not into namespaces.
'None'	No internal references are inserted and lexical scope does not apply.

See *Lexical Scope in Scripts* on page 9.

The following examples illustrate how different values of the **InjectReferences** option affect the scope of objects in scripts. The examples are based on the following family tree:



Two scripts are defined to map this tree onto a structure of Classes and Namespaces. In this scheme, female family members are represented by Classes and male family members by Namespaces.

So the scripted tree for Pete has a parent Namespace:

```
:Namespace Pete
    :Namespace Andy
        :Class Aisha
        :Access Public
        :Endclass
    :EndNamespace

    :Class Katherine
    :Access Public
        :Namespace Woody
        :EndNamespace
        :Namespace George
        :EndNamespace
    :EndClass
:EndNamespace
```

While the scripted tree for Jill has a parent Class:

```
:Class Jill
:Access Public
    :Namespace Andy
        :Class Aisha
        :Access Public
        :Endclass
    :EndNamespace

    :Class Katherine
    :Access Public
        :Namespace Woody
        :EndNamespace
        :Namespace George
        :EndNamespace
    :EndClass
:EndClass
```

Using the Pete Namespace, after executing the expression:

```
      2(⎕FIX⍠'InjectReferences' 'All')⎕SRC Pete
```

- Code in Pete may refer to Aisha , Andy , George , Katherine, and Woody
- Code in Andy may refer to Aisha and Katherine
- ... and so forth.

But after executing:

```
      2(⎕FIX⍠'InjectReferences' 'InClasses')⎕SRC Pete
```

- Code in Pete may refer only to Andy and Katherine
- Code in Andy may refer only to Aisha
- ... and so forth.

The following tables show which objects in Namespace `Pete` can *see* (i.e. refer to) which other objects representing members of the family, in each case; `All`, `InClasses` and `None`.

`'All'`	Pete	Andy	Aisha	Katherine	Woody	George
Pete		✓	✓	✓	✓	✓
Andy			✓	✓		
Aisha	✓	✓	✓			
Katherine	✓	✓		✓	✓	✓
Woody						✓
George					✓	

`'InClasses'`	Pete	Andy	Aisha	Katherine	Woody	George
Pete		✓		✓		
Andy			✓			
Aisha	✓	✓	✓			
Katherine	✓	✓		✓	✓	✓
Woody						
George						

`'None'`	Pete	Andy	Aisha	Katherine	Woody	George
Pete		✓		✓		
Andy			✓			
Aisha						
Katherine						
Woody						
George						

Whilst the next set of tables show the same for Class `Jill`.

`'All'`	Jill	Andy	Aisha	Katherine	Woody	George
Jill	✓	✓	✓	✓	✓	✓
Andy			✓	✓		
Aisha	✓	✓	✓			
Katherine	✓	✓		✓	✓	✓
Woody						✓
George					✓	

`'InClasses'`	Jill	Andy	Aisha	Katherine	Woody	George
Jill	✓	✓	✓	✓	✓	✓
Andy			✓			
Aisha	✓	✓	✓			
Katherine	✓	✓		✓	✓	✓
Woody						
George						

`'None'`	Jill	Andy	Aisha	Katherine	Woody	George
Jill						
Andy			✓			
Aisha						
Katherine						
Woody						
George						

Allocate Token Range {R}←{X} ⎕TALLOC Y

Y is either a single integer or a 2-element vector. The first (or only) item in Y is 0, 1, 2 or ¯1 and indicates the type of operation to perform. If it is 1, then the optional second item is a character vector.

The optional left argument X identifies an existing allocated range of token numbers n. X must be a scalar greater than or equal to n, but must be less than n+1.

Allocation (First element of Y is 1)

If the first element of Y is 1, the result R is a positive integer that identifies a range of numbers that may be used as token types for ⎕TPUT and ⎕TGET. That range is defined as the set of floating-point numbers between R and R+1 (but not the integer end-points). Negated values of these number may also be used.

In this case, the optional Y[2] is an arbitrary character vector that serves as a description for the allocated range of tokens.

De-allocation (Y is ¯1)

If Y is ¯1, ⎕TALLOC releases a previously allocated range of tokens identified by the left-argument X. The result R is a shy ⍬.

To succeed, this range must have previously been allocated, not freed by de-allocation, and must be inactive, i.e. its tokens must not currently be in the token pool or in use by a ⎕TGET. If not, ⎕TALLOC will signal a DOMAIN ERROR.

A de-allocated range becomes free for subsequent re-allocation by ⎕TALLOC.

Querying a description (Y is 0)

Y is 0, ⎕TALLOC returns a non-shy result R containing the description for a currently allocated range of tokens identified by the left-argument X.

If X does not represent a currently allocated range, ⎕TALLOC will signal a DOMAIN ERROR.

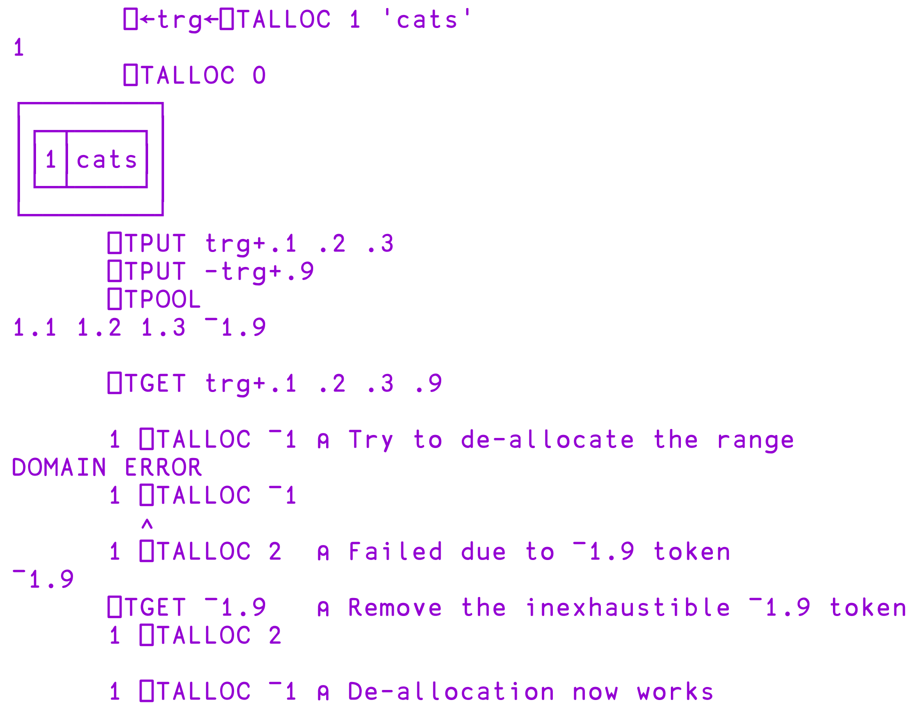
If X is omitted, the result R is a vector of 2-element vectors identifying the range and description of all currently allocated ranges.

Descriptions that were not defined are returned as empty character vectors.

Querying the Token Pool (Y is 2)

Y is 2, ⎕TALLOC returns a non-shy result R containing the list of tokens in the token pool that fall in the range specified by the left-argument X.

Examples



```
      ⎕←trg←⎕TALLOC 1 'cats'
1
      ⎕TALLOC 0
┌────────┐
│┌─┬────┐│
││1│cats││
│└─┴────┘│
└────────┘
      ⎕TPUT trg+.1 .2 .3
      ⎕TPUT -trg+.9
      ⎕TPOOL
1.1 1.2 1.3 ¯1.9

      ⎕TGET trg+.1 .2 .3 .9

      1 ⎕TALLOC ¯1 ⍝ Try to de-allocate the range
DOMAIN ERROR
      1 ⎕TALLOC ¯1
        ∧
      1 ⎕TALLOC 2  ⍝ Failed due to ¯1.9 token
¯1.9
      ⎕TGET ¯1.9   ⍝ Remove the inexhaustible ¯1.9 token
      1 ⎕TALLOC 2

      1 ⎕TALLOC ¯1 ⍝ De-allocation now works
```

Set aplcore Parameters R←1302⌶Y

Sets the aplcore parameters **AplCoreName** and/or **MaxAplCores** for the current process.

Y may be:

- a simple character vector that specifies AplCoreName
- a simple integer that specifies MaxAplCores
- a 2-element nested vector containing new values for AplCoreName and MaxAplCores in that order
- an empty vector

R is a 2-element nested vector containing the old values.

If Y is empty, the function simply returns the values of these parameters without changing them.

See also: *Installation & Configuration Guide: APLCoreName and MaxAplCores parameters*.

Memory Manager Statistics R←{X}(2000⌶)Y

This function returns information about the state of the workspace and provides a means to reset certain statistics and to control workspace allocation. This I-Beam is provided for performance tuning and is VERY LIKELY to change in the next release. See also *Installation & Configuration Guide: Workspace Management*.

Y is a simple integer scalar or vector containing values listed in the table below.

If X is omitted, the result R is an array with the same structure as Y, but with values in Y replaced by the following statistics. For any value in Y outside those listed below, the result is undefined.

Value	Description
0	Workspace available (a "quick" ⎕WA).
1	Workspace used.
2	Number of compactions since the workspace was loaded.
3	Number of garbage collections that found garbage.
4	Current number of garbage pockets in the workspace.
9	Current number of free pockets in the workspace.
10	Current number of used pockets in the workspace.
12	Sediment size.
13	Current workspace allocation, i.e. the amount of memory that is actually being used.
14	Workspace allocation high-water mark, i.e. the maximum amount of memory that has been allocated since the workspace was loaded or since this count was reset.
15	Limit on minimum workspace allocation.
16	Limit on maximum workspace allocation.
19	The number of calls to ⎕WA or 2002⌶ since the last time 2000⌶ was called, or when the process started.
20	The requested size of the WS Full Buffer, i.e. the amount of workspace requested for handling WS FULL errors.
21	The actual size of the WS Full Buffer.
22	The number of WS FULL handlers that are currently running.

Value	Description
23	The total number of WS FULL errors that have occurred.
24	The total number of WS FULL errors that have been trapped.

Note: While all other operations are relatively fast, the operation to count the number of garbage pockets (4) may take a noticeable amount of time, depending upon the size and state of the workspace.

Examples

```
      2000⌶0
55414796
      2000⌶0,⍳16   ⍝ with MAXWS=95G
1.02004292E11 1181312 1 1 0 ¯1 ¯1 ¯1 ¯1 78 13280 ¯1
1180800 1595016496 1595042464 0 1.020054733E11
```

If X is specified, it must be either a simple integer scalar, or a vector of the same length as Y, and the result R is ⍬. In this case, the value in Y specifies the item to be set and X specifies its new value according to the table below.

Value	Description
2	0 resets the compaction count; no other values allowed.
3	0 resets the count of garbage collections that found garbage; no other values allowed.
14	0 resets the workspace allocation high-water mark; no other values allowed. This should be called following a call to ⎕WA (which compacts the workspace and returns unused memory to the operating system).
15	Sets the minimum workspace allocation to the corresponding value in X; must be between 0 and the current workspace allocation.
16	Sets the maximum workspace allocation to the corresponding value in X; 0 implies **MAXWS** otherwise must be between the current workspace allocation and **MAXWS**.
19	0 resets the compaction count; no other values allowed.
20	Sets the requested size of the WS Full Buffer to the value specified by X. The actual space allocated may be less than that requested.

Notes:

- The workspace allocation high-water mark indicates a minimum value for **MAXWS**.
- Limiting the maximum workspace allocation can be used to prevent code that reserves as much workspace as it can from skewing the peak usage result.
- Limiting the minimum workspace allocation can avoid repeatedly committing and releasing memory to the Operating System when memory usage is fluctuating.

Examples

```
      2000⌶2 3
6 0 33216252
      0 (2000⌶)2 3 14 ⍝ Reset compaction count

      2000⌶2 3
0 0
      30000000 40000000(2000⌶)15 16 ⍝ Restrict min/max ws

      (2000⌶)15 16
30000000 40000000

      0 (2000⌶)15 16 ⍝ Reset min/max ws

      (2000⌶)15 16
0 65536000

      (2000⌶)13 14 ⍝ Current, peak WS allocation
4072532 4072532

      a←10e6⍴'x' ⍝ Increase WS allocation

      (2000⌶)13 14 ⍝ Current, peak WS allocation
15108580 15108580

      ⎕ex 'a' ⋄ {}⎕wa ⍝ Decrease current WS allocation

      (2000⌶)13 14 ⍝ Current, peak WS allocation
1962856 15108580

      0 (2000⌶) 14 ⍝ Reset High-water mark

      (2000⌶)13 14 ⍝ Current, peak WS allocation
1962856 1962856
```

WS Full Handling

Potentially, a `WS FULL` error represents a terminal condition that would prevent a program from continuing because the process has, quite literally, run out of memory.

To alleviate the problem,. Dyalog reserves a special *WS Full Buffer* for handling `WS FULL` errors. The default size of this buffer is `(1MB)⌊(0.01×⎕WA)`.

In simple terms, when a `WS FULL` error occurs that triggers a handler, i.e. an expression executed via `⎕TRAP` or `:Trap`, the reserved workspace in the *WS Full Buffer* is released to provide additional memory space for that expression to execute. When the expression terminates, the system removes the memory that it had previously released, reserving it once more for another potential `WS FULL`.

Note that until a `WS FULL` handler starts, the memory allocated to the *WS Full Buffer* is unavailable and inaccessible for any other purpose, thereby reducing the amount of active workspace available (`⎕WA`).

Further considerations are:

- Multiple `WS FULL` handlers can run concurrently as a result of muti-threading or nesting (when a `WS FULL` handler itself generates a `WS FULL` error).
- When the `WS Full Buffer` is restored when the handler (more accurately, the last handler) terminates, or when a saved workspace is re-loaded, there may be insufficient memory available. In these circumstances, the system allocates a reduced amount, without reporting an error. However, the system will later try to reclaim more (up to the desired amount), if more workspace has become free. The desired and actual sizes of the `WS Full Buffer` are reported by `(2000⌶)20` and `(2000⌶)21` respectively.
- When a `WS FULL` handler is activated and the `WS Full Buffer` is freed, `(2000⌶)21` will return 0 until the handler terminates.

Enable Compression of Large Components {R}←3012⌶Y

Specifies whether large components (>2GB) may be compressed.

Y is an integer defined as follows:

Value	Description
0	Large components will not be compressed
1	Large components will be compressed if Z property is 1 , but versions of Dyalog prior to 19.0 will not be able to read them.

The shy result R is the previous value of this setting.

Discard Source Information R←5171⌶Y

This function discards source code and file information for scripted objects, namespaces, classes, functions, and operators that is saved in the workspace. See also *Discard Source Code* on page 44.

Y is a vector or scalar containing zero or more references to # or ⎕SE, and specifies from which namespaces the information is removed.

R is an integer. A non-zero value indicates that some information was removed. 0 means nothing was discarded.

- The expression 5171⌶ # discards source code and file information from the workspace, but not from ⎕SE.
- 5171⌶⎕SE discards source code and file information from ⎕SE but not from the workspace.
- 5171⌶ # ⎕SE discards source code and file information from the workspace and from ⎕SE.

For further information, see *Source as Typed* on page 13.

Discard Source Code R←5172⌶Y

This specifies whether source code is discarded for functions and operators when they are created by the editor or by ⎕FIX. See also *Discard Source Information* on page 43.

Y is 0 or 1.

If Y is 0 (the default), source code is retained in the workspace when an object is fixed.

If Y is 1, source code is not retained in the workspace when an object is fixed (source code already retained in the workspace is not discarded).

In all case the result R is the previous setting (0 or 1).

For further information, see *Source as Typed* on page 13.

Hash Table Size {R}←8468⌶Y

Increases the amount of workspace allocated to internal hash tables. These tables are created when a set primitive is executed or by the Hash Array function (1500⌶).

Note:

The purpose of this function is to allow the user to evaluate potential side-effects of the proposed increase in table size in the next major version of Dyalog.

Y may be ⍬, or an integer 0, 1, 2, or 3.

If Y is 1, 2 or 3 the hash table size is increased by the factor 2*Y. If Y is 0, the hash table size is reset to its default value. In these cases, the shy result R is the previous value of the scale factor.

If Y is ⍬ the size is unaffected and the (non-shy) result is the current value of the scale factor.

It is recommended that users test their code using the maximum value 3.

For more information, see *Programming Reference Guide: Search Functions and Hash Tables* and *Language Reference Guide: Hash Array*.

Lookup Table Size R←8469⌶Y

Increases the maximum amount of workspace allocated to internal lookup tables. These tables are created when a set primitive is executed. Lookup tables are faster than hash tables, and are used when hashing is not required.

Note:

The purpose of this function is to allow the user to evaluate potential side-effects of the proposed increase in table size in the next major version of Dyalog.

Y may be ⍬, or an integer from 0 to 16777216.

If Y is between 1 and 16777216 the function sets the lookup table size in bytes to that value. If Y is 0, the lookup table size is reset to its default value. In both cases, the shy result R is the previous value of the table size.

If Y is ⍬ the size is unaffected and the (non-shy) result is the current value of the scale factor.

It is recommended that users test their code using the maximum value.

For more information, see *Programming Reference Guide: Search Functions and Hash Tables* and *Language Reference Guide: Hash Array*.

Chapter 4:

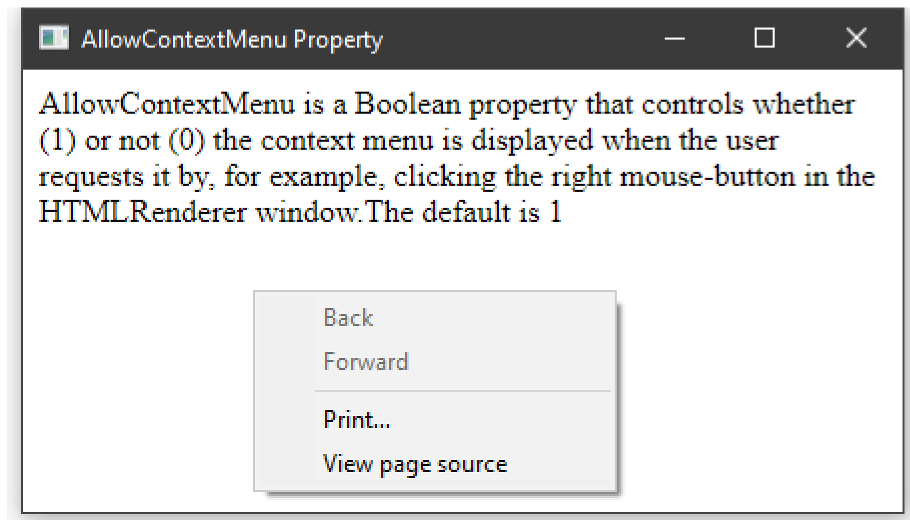
Object Reference Changes

AllowContextMenu Property

Applies To: HTMLRenderer

Description

This is a Boolean property that controls whether (1) or not (0)the context menu. is displayed when the user requests it. The default is 1



ExecuteJavaScript Method 839

Applies To: HTMLRenderer

Description

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

The argument to ExecuteJavaScript is a single item as follows:

`[1]`	Code	character vector containing JavaScript code

The shy result of ExecuteJavaScript is currently 1; this may change.

Example

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

GetZoomLevel Method 838

Applies To: HTMLRenderer

Description

This method is used to retrieve the current CEF ZoomLevel of the HTMLRenderer.

See *SetZoomLevel* on page 50.

IsLoading Property

Applies To: HTMLRenderer

Description

IsLoading is a Boolean property whose value is 1 if the browser is currently loading, or 0 when the frame content is completely loaded.

LoadEnd Event 836

Applies To: HTMLRenderer

Description

A LoadEnd event is raised when a particular frame has finished loading. Multiple frames may be loading at the same time. Sub-frames may start or continue to load even after the main frame has finished loading.

A common technique is to wait for the main frame to finish loading before further interaction with the HTMLRenderer instance. In this case, you should set up an event handler on the LoadEnd event and check the 4th element which indicates if the loaded frame is the main frame.

You may use the IsLoading property to check if the HTMLRenderer is still loading.

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

[1]	Object	ref or character vector
[2]	Event	'LoadEnd' or 836
[3]	url	The URL of the loaded frame
[4]	Flag	1 if the loaded frame is the "main" frame, 0 otherwise
[5]	Code	The HTTP status code as a result of loading the frame

SetZoomLevel Method

Description

Sets the CEF ZoomLevel. The default (unzoomed) level is 0. Setting a positive value will increase the zoom, whereas setting a negative will decrease the zoom. The zoom scale is not linear; rather the effective scaling is approximately 1.2*level, so, setting the ZoomLevel to 1 will result in an approximate 20% size increase. ZoomLevel affects all instances of HTMLRenderer windows; it is not possible to have different ZoomLevels for individual windows.

The argument to SetZoomLevel is a single numeric value:

[1]	ZoomLevel	Numeric

Examples

```
     ∇ hr Zoom level;lb;in
[1]    hr.SetZoomLevel level
[2]    level←⍕hr.GetZoomLevel
[3]    ((level='¯')/level)←'-'
[4]    lb←'<label>Zoom Level is </label>'
[5]    in←'<input type="number" value="',level,'"></input>'
[6]    hr.HTML←lb,in
     ∇

      hr.⎕WC'HTMLRenderer' ('Caption' 'ZoomLevel Method')
      hr Zoom 0
```

```
hr Zoom 1
```

```
hr Zoom 2
```

```
hr Zoom ¯1
```

Index

A

AllowContextMenu 47
AllowLateBinding option 33

B

Bug Fixes 15

C

CEF 17
classes
    fix script 29

D

dyadic primitive operators
    variant 32
DYALOG_DISCARD_FN_SOURCE parameter 24
DYALOG_GUTTER_ENABLE parameter 23
DYALOG_INITSESSION parameter 24
DyalogLink parameter 24
DyalogStartup_X parameter 24
DyalogStartupSE parameter 25

E

Events
    LoadEnd 49
ExecuteJavaScript 48

F

file
    hold 28
fix script 29
FixWithErrors option 32

G

GetZoomLevel 48

H

hash 16
holding component files 28

I

i-beam
    memory manager statistics 39
InjectReferences option 33
Interoperability 19
IsLoading 48

K

Key Features 1
key operator glyph 22

L

LoadEnd 49
Log_File parameter 26

M

MAXWS parameter 40
memory manager statistics 39
Methods
    ExecuteJavaScript 48
    GetZoomLevel 48
    SetZoomLevel 50

N

nest/partition function glyph 22

O

over operator glyph 22

P

PCRE2 17
Principal option 32
ProgressCallback option 5
Properties
    AllowContextMenu 47
    IsLoading 48

Q

Quiet option 32

R

rank operator glyph 22
RConnect 17
releasing component files 28

S

session ghutter 12
SetZoomLevel 50
source as typed 13
stencil operator glyph 22
Syncfusion 17
System Requirements 18

T

tokens
    allocate token numbers 37

V

variant operator 32
variant operator glyph 22

W

where/interval index function glyph 22