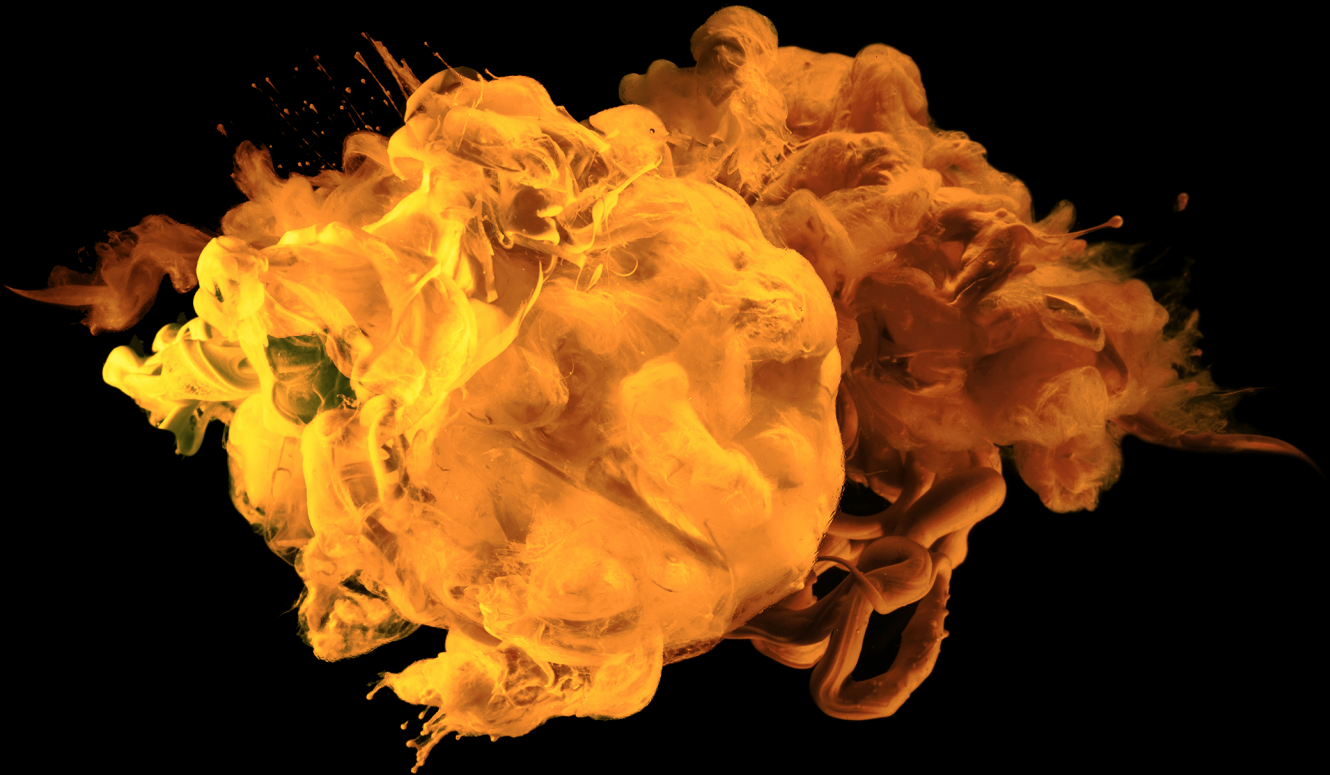


# Dyalog Release Notes

Dyalog version **18.2**



# DYALOG

The tool of thought for software solutions

*Dyalog is a trademark of Dyalog Limited  
Copyright © 1982-2022 by Dyalog Limited  
All rights reserved.*

## Dyalog Release Notes

Dyalog version 18.2  
Document Revision: 20240214\_182

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](mailto:support@dyalog.com)  
<https://www.dyalog.com>*

## TRADEMARKS:

*Array Editor is copyright of [davidliebtag.com](http://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

<b>Chapter 1: Highlights</b>	<b>1</b>
Key Features	1
Announcements	3
System Requirements	5
Interoperability	6
Load Parameter	10
File Associations	12
Shell Scripts	13
 <b>Chapter 2: Language Reference Changes</b>	 <b>17</b>
Extended Attributes	17
JSON Extension	20
Fix Script	24
Set Shell Script Debug Options	28
Read DataTable	28
Verify .NET Interface	32
Sample Probability Distribution	34
 <b>Chapter 2: Object Reference Changes</b>	 <b>37</b>
CornerTitleBCol	38
FireOnce	39
 <b>Index</b>	 <b>41</b>



# Chapter 1:

## Highlights

### Key Features

#### Upgrading from Version 17.1 to Version 18.2

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

#### Shell Scripts

It is now possible to run Dyalog APL from a script. See [Shell Scripts on page 13](#).

#### New Language Features

- `⎕ATX` is a new system function that provides information about a name in a workspace, including its usage, history, restrictions, role and origin. See [Extended Attributes on page 17](#). Dyalog Ltd recommends using `⎕ATX` in preference to `⎕AT`, `⎕NC`, `⎕NR`, `⎕SIZE` and `⎕SRC` (and some of the functionality of `5179⍤`).

#### Improved Language Features

- `⎕JSON` has been extended to recognise APL data structures commonly used to represent datasets. See [JSON Extension on page 20](#).
- `⎕DT` now supports the format used by the DateTime property of the DateTimePicker object.
- `⎕FIX` provides 3 new Variant options that control how Classes and Namespaces are fixed. See [Fix Script on page 24](#).
- `⎕R` and `⎕S` can now transform characters by case folding (`\f`) as well as by case conversion (`\l` and `\u`).
- `⎕KL` is now supported by the Unicode Edition.
- `2011⍤` has been extended to produce inverted tables and to provide an option for converting Int64 objects to DECF. See [Read DataTable on page 28](#).

## New I-Beam Functions

- **1010I** provides options for debugging Shell Scripts. See [Set Shell Script Debug Options on page 28](#).
- **2250I** provides information about the Dyalog interface to .NET. See [Verify .NET Interface on page 32](#).
- **16080I** generates random numbers of a given distribution. See [Sample Probability Distribution on page 34](#). Note that this is not currently implemented for AIX.

## Colour Schemes

Three third-party colour schemes have been added:

- Dracula theme (draculatheme.com) by Zeno Rocha: dark purple
- New Moon theme (taniarascia.github.io/new-moon) by Tania Rascia: dark grey
- Nord theme (nordtheme.com) by Arctic Ice Studio and Sven Greb: dark blue

## GUI Improvements

- The FireOnce property has been added to the Timer object. See [FireOnce on page 39](#).
- The Align property now applies to single-line Edit objects, specifies the vertical alignment of the text.
- The new CornerTitleBCol property specifies the background colour of the left corner rectangle in a Grid. See [CornerTitleBCol on page 38](#).

## User Command Framework version: 2.5

- The cache file is now called `UserCommand{UcmdMajor}{UcmdMinor}.{DyalogMajor}{DyalogMinor}{U|C}{bits}.cache`  
e.g. `UserCommand25.182U64.cache`
- When a user command signals an error, **DMX** is set to `SE.SALTutils.dmx`

## Link Integration

A number of enhancements have been made to further integrate Link into Dyalog. For more information about Link, see <https://dyalog.github.io/link/3.0/>.

- The **Load** parameter has been extended to load and run source code from a directory at start-up. See [Load Parameter on page 10](#).
- During installation, Dyalog establishes suitable file associations for Dyalog file types and adds menu items to the Windows Explorer context menu for directories. See [File Associations on page 12](#).

# Announcements

## Withdrawal of Support for Version 17.0

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

## Extended Support for Version 17.1

As previously announced, support for Version 17.1 has been extended; version 17.1 will remain on support along with 18.0 and 18.2 when the next version of Dyalog is released.

## APLScript Examples (documentation)

The APLScript examples provided in `Samples/aplscript` and `Samples/aplclasses` have `.apl` extensions in line with the new source code file naming conventions. In this respect the .NET documentation is out of date. Files with the extension `.apl` are no longer used.

## Forthcoming Removal of 819I

The system function `⍶` was introduced in version 18.0, at which point Dyalog announced that `819I` was deprecated. `819I` will still be present in the version which follows 18.2, but Dyalog intends that it will be removed from the version after that.

## 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.

## Removal of Syncfusion from Microsoft Windows installation images

The installation images for Dyalog APL for Microsoft Windows include the Syncfusion library of WPF controls. Dyalog version 18.2 will be the last version that includes these libraries in the installation images; in future Dyalog versions they will be made available using an alternative delivery mechanism. This change is the first step in supplying smaller installation images (and thus reduced disk requirements) by not including some of the less-used but large features of Dyalog APL for Windows.

## Chromium Embedded Framework (CEF)

The CEF is included with Dyalog APL, and therefore not a requirement in the usual sense. The version of CEF that is included with version 18.2 for Windows and macOS is #90, while Linux installations include #79. Dyalog is aware that these versions are already quite old, and aims to include much more up-to-date versions of the CEF in future releases. Please contact Dyalog support to inquire whether a more up-to-date version has become available since the first release of Dyalog version 18.2.

## Planned Hardware/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	AIX 7.2.5 on POWER 9
Linux	Versions of distributions which are in standard support for at least 3 months from when the next version of Dyalog is released. Note that Linux on POWER is not supported.
macOS	macOS Big Sur 11.6.1
Raspberry Pi	Raspbian Buster 32-bit

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



---

# System Requirements

## Microsoft Windows

Dyalog APL Version 18.2 is supported on versions of Microsoft Windows from Windows 8 or Windows Server 2012 upwards.

## Microsoft .NET Interface

Dyalog APL Version 18.2 .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<sup>1</sup>) Version 18.2 requires .NET Version 4.5. The Syncfusion libraries supplied with Version 18.2 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 18.2 requires AIX 7.2 or higher, and a POWER9 chip or higher.

## Raspberry Pi

On the Raspberry Pi, Dyalog 32-bit Unicode supports 32-bit Raspbian Buster or later (Bookworm requires Dyalog version 18.2.48479 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 Raspbian.

## Non-Pi Linux

For non-Pi Linux, Version 18.2 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. For further information, see [the Dyalog UNIX and Linux forum](#).

## macOS

Version 18.2 requires macOS Big Sur 11.6.1 or higher. There is no 32-bit version.

---

<sup>1</sup>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 18.2 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.
- 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.

- 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 or objects created by `WC`

or

- variables containing the `FOR` 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 `FOR` 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 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. 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 (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**.

## TCP.Sockets and Conga

TCP.Sockets 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.

# Load Parameter

This parameter is a character string that specifies the name of a workspace, or a directory or text file containing APL source code, to be loaded when Dyalog starts.

If **Load** specifies a text file, **FIX** is used to import the file contents and associate that file with each of the objects that have been fixed in the workspace.

If **Load** specifies a directory, **Link** is used to associate the directory with the active workspace and to import the code. For more information about **Link**, see <https://dyalog.github.io/link/3.0/>.

The **Load** parameter will normally be specified on the command line or in a Configuration file.

Having loaded the workspace, or fixed the code from the named file or directory, Dyalog executes the expression specified by the **LX** parameter if it is set.

If **LX** is not set, Dyalog checks whether or not the **-x** command line option was specified. If so, no further action is taken.

Otherwise, Dyalog executes an expression which is derived as follows.

If the value of **Load** is a directory, Dyalog will execute the expression:

```
Run ,<Load>
```

where **<Load>** is the value of the **Load** parameter.

If the value of **Load** is the name of a file, Dyalog determines whether or not the file is a workspace by its internal signature.

If the file is a workspace the expression to be executed is specified by its **LX**.

Otherwise, if the file extension is **.aplf**, **.aplc** or **.apl** the expression is shown in the table below, where **filename** is the file name specified by the **Load** parameter without its extension.

File Extension	Type	Expression
.aplf	Function source code	filename 0p<' '
.aplc	Class source code	filename.Run 0p<' '
.apl	Namespace source code	filename.Run 0p<' '

**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.
- The argument `Op=` may change in a future version of Dyalog.
- Nothing is executed when code is loaded from source files that define operators (`.aplo`) or Interfaces (`.apli`).

# File Associations

During installation, Dyalog establishes the following file associations:

Type	File Extension	Application
Shell Scripts	.apls	Dyalog script execution engine via Windows Power Shell
Sources	.aplc, .aplf, .apli, .aplnc, .aplo, .dyalog	Dyalog Editor
Configuration	.dcfg	Dyalog Editor
SALT apps	.dyapp	Dyalog
Workspaces	.dws	Dyalog

When you double-click on a file with one of the above extensions, the file is opened with the corresponding application.

In addition, two items are added to the Windows Explorer context menu for directories, namely *Load with Dyalog* and *Run with Dyalog*. Both these items start Dyalog and attempt to import code from the corresponding directory using `Link`. The *Run with Dyalog* option also calls the function named `Run` if it exists. See [Load Parameter on page 10](#).

For more information about `Link`, see <https://dyalog.github.io/link/3.0/>.

The `]fileAssociations` user command may be employed to alter these settings. For details, enter:

```
]fileassociations -?
```



# Shell Scripts

Shell scripts are typically executed from a terminal (or *shell*).

A script is executed by typing its name. User input is entered from the same terminal or shell and output is displayed on the terminal or shell.

## UNIX

On UNIX (and related) systems a Dyalog APL *shell script* is a text file with the following as the first line:

```
#!/usr/local/bin/dyalogscript
```

The script file must be executable. There are three execute bits relating to the user, the group and everyone else.

## Windows

On Windows systems a Dyalog APL shell script is a text file with a `.apls` file extension. An initial line beginning with `#!` is only required to include configuration parameters (see below), but if included it must include a file name even though that will be ignored. For portability it is recommended that you include the `#!` line.

### Note

Shell scripts are Unicode only and are not supported by the Classic Edition.

Any content that follows the `#!` line (if present) is used as input into a Dyalog session (as if the *Extended Multiline Input* feature has been enabled).

## Input and Output

`⎕` and `⎕` input are taken from characters typed by the user into the terminal or shell (Standard input or *stdin* for short). Anything assigned to `⎕` and `⎕` will be displayed in the terminal window using streams Standard output (*stdout*) and Standard error (*stderr*) respectively. Note that default output, that is, output to the session without assignment to `⎕` or `⎕` is NOT displayed. Redirections of *stdin*, *stdout*, and *stderr* are supported.

## Examples

The following then are all valid APL shell scripts:

```
#!/usr/local/bin/dyalogscript
'this text will not be seen'
⍵←2+2

#!/usr/local/bin/dyalogscript
∇r←l plus r
r←l+r
∇
⍵←2 plus 2

#!/usr/local/bin/dyalogscript
plus←{
  α+ω
}
⍵←2 plus 2
```

## Errors

Untrapped errors in a script will cause the termination of the process, further lines in the script will NOT be processed.

```
#!/usr/local/bin/dyalogscript
⍵←'this will be seen'
⍵←÷0
⍵←'this will NOT be seen'
```

However, the multiline input mechanism allows for **:Trap** statements, so the following will run to completion:

```
#!/usr/local/bin/dyalogscript
⍵←'this will be seen'
:Trap 0
  ⍵←÷0
:EndTrap
⍵←'this will ALSO be seen'
```

## Configuration Parameters:

Configuration parameters may be specified in a Configuration file located in the same directory as the script, or may be specified on the first line of the script. The name of the configuration file is derived from the name of the script file by replacing its file extension (if any) by the extension `.dcfg`. Configuration parameters specified in the Windows Registry or by environment variables are not honoured in Dyalog Shell Scripts.

### Example (first line of script)

```
#!/usr/local/bin/dyalogscript MAXWS=3GB
⎕←⎕WA
⎕←2 ⎕nq '#' 'GetEnvironment' ('MAXWS' 'WSPATH')
```

### Example (configuration file)

```
{ settings: {
    /* Maximum workspace size */
    MAXWS: "256M",
    /* wspath */
    WSPATH: ["c:/tmp", "f:/devt/tmp"]
}}
```

Note that the interpreter reads both of these locations, the command line in the script file overrides any setting in the .dcfg file.

## Debugging:

It is not currently possible to use RIDE to debug APL shell scripts. However there is an I-beam function, which can be used to provide some simple debugging/diagnostic information. See [Set Shell Script Debug Options on page 28](#).



# Chapter 2:

## Language Reference Changes

The following table summarises the main changes to language features in Version 18.2.

Function/Operator	Description	Change
<code>□ATX</code>	Extended Attributes	New system function
<code>□DT</code>	DateTime	New timestamp format <code>-30</code>
<code>□FIX</code>	Fix Script	New Variant options
<code>□JSON</code>	JSON export	New feature to export datasets
<code>1010I</code>	Set Shell Script Debug Options	New I-Beam function
<code>2011I</code>	Read DataTable	Enhanced I-Beam function
<code>2250I</code>	Verify .NET Interface	New I-Beam function
<code>16080I</code>	Sample Probability Distribution	New I-Beam function

### Extended Attributes

**R←X □ATX Y**

This function provides information about a name in a workspace, including its usage, history, restrictions, role and origin.

Note: To retrieve this information for an unnamed value, wrap `□ATX` in a dfn and use the name `ω`, for example `{60□ATX'ω'}`

**Y** can be a simple character scalar, a simple or enclosed character vector, or a vector of character scalars and vectors (as least one must be a character vector) of the name (s) for which information is required.

**X** can be a scalar or a vector indicating the information required:

Group	X	Meaning	Default
Identity	0	Name	' '
Syntax	10	Function result (0: none or not a function, 1: explicit, -1: shy)	0
	11	Function valence (0: niladic, 1: monadic, 2: dyadic, -2: ambivalent)	0
	12	Operator valence: (0: not an operator, 1: monadic, 2: dyadic)	0
Last edit	20	Author of last edit	' '
	21	Number of days passed between 1899-12-31 at 00:00 UTC and last edit (includes fractional days)	0
	22	Local timestamp at last edit (format is the 7-item vector described by <code>□TS.</code> )	⊖
	23	Number of bytes required for storage without sharing	0
Restrictions	30	Source can be displayed	-1
	31	Execution can be suspended mid-execution	-1
	32	Responds to weak interrupt	-1
Class*	40	Syntactic supra-class (-1: invalid name, 0: undefined, 1: label, 2: variable, 3: function, 4: operator, 8: event, 9: object)	-1
	41	Syntactic sub-class (0: none, 1: traditional/plain, 2: field/dynamic/instance, 3: property/derived/primitive, 4: class, 5: interface, 6: external, 7: external interface)	0
	42	Full syntactic class (sum of supra- and sub-class)	-1

Group	X	Meaning	Default
Source	50	File name	' '
	51	File encoding	' '
	52	File checksum	' '
	53	File line separators ( <b>13</b> : Carriage Return, <b>10</b> : Line Feed, <b>13 10</b> : Carriage Return followed by Line Feed, <b>133</b> : New Line, <b>11</b> : Vertical Tab, <b>12</b> : Form Feed, <b>8232</b> : Line Separator, <b>8233</b> : Paragraph Separator)	0
	54	Definition's offset from top	0
	55	Number of lines in definition	0
Definition	60	Verbatim source (as typed)	0p<' '
	61	Normalised source (with AUTOFORMAT=1 and TABSTOPS=4)	0p<' '
	62	Most precise available source (verbatim with fallback to normalised)	0p<' '

\* Names in the Class group that can return **-1** (meaning "invalid name") might return a different value in future versions of Dyalog, including values that are not currently possible and ones that deviate from the current **NC** values.

**R** depends on the combination of **X** and **Y**:

		<b>X</b>	
		Scalar	Vector
<b>Y</b>	<b>Simple character scalar/vector</b>	Requested value (not enclosed)	Vector of requested values
	<b>Enclosed character vector</b>	Requested value (enclosed)	Scalar containing vector of requested values
	<b>Vector of character scalars/vectors</b>	Vector of requested values	Outer shape from <b>pα</b> , inner shape from <b>pω</b>

**Examples:**

```

      Att
10 11 12 20 23 30 31 32 40 41 42 50 51 52 53 54 55

```

```
foo←{ω ω}
Att ⍴ATX 'foo'
```

1	-2	0	616	-1	-1	-1	3	2	3.2				0	0
---	----	---	-----	----	----	----	---	---	-----	--	--	--	---	---

```
x←42
Att ⍴ATX 'x'
```

0	0	0	32	-1	-1	-1	2	1	2.1				0	0
---	---	---	----	----	----	----	---	---	-----	--	--	--	---	---

```
10 11 12 30 31 32 40 41 42 ⍴ATX 'x' 'foo'
```

0	1	0	-2	0	0	-1	-1	-1	-1	-1	-1	2	3	1	2	2.1	3.2
---	---	---	----	---	---	----	----	----	----	----	----	---	---	---	---	-----	-----

```
2 ⍴FIX'foo ← {' 'ω ω }'
60 61 ⍴ATX 'foo'
```

foo ← { ω ω }	foo←{ ω ω }
---------------	-------------

```
src←':namespace c' ':endnamespace' '' 'range←
{α↓ιω}'
```

```
2 ⍴FIX src
55 54 ⍴ATX'c' 'range'
```

2	1	0	3
---	---	---	---

```
2 1↑''0 3↓''=src
```

:namespace c	:endnamespace	range←{α↓ιω}
--------------	---------------	--------------

## JSON Extension

Version 18.2 includes an extension to [JSON](#) that provides the ability to export APL arrays which have a specific data structure that would not otherwise be recognised. The structures that have been selected for this special treatment are ones that are often used to represent tables or datasets. The extension allows [JSON](#) to render APL table representations in a format that would be expected by many other programming languages including JavaScript.



## Datasets

The term dataset is used here to mean a collection of data, usually presented in tabular form. Each named column represents a particular variable. Each row corresponds to a given member of the dataset in question. It lists values for each of the variables, such as height and weight of an object.

Datasets are often represented in APL as a collection of variables.

```
Fields←'Item' 'Price' 'Qty'
Items←'Knife' 'Fork'
Price←3 4
Qty←23 45
```

As an aside, note that using this scheme each variable represents an inverted index into the dataset and enables rapid searches.

```
(Price<4)/Items
```

```
Knife
```

A conventional way to represent this dataset is as a matrix:

```
Fields;⍋ Items Price Qty
```

Item	Price	Qty
Knife	3	23
Fork	4	45

Another is as a 2-item vector containing the names of the fields and a matrix of their values:

```
(Fields (⍋Items Price Qty))
```

Item	Price	Qty	Knife	3	23
			Fork	4	45

A third way retains the inverted nature of the data structure, storing the values as a vector. The advantage of this structure is that it consumes significantly less memory compared to the matrix forms, because numeric columns are stored as simple numeric vectors.



In JSON, these three data structures are all expressed as follows:

```
[
  {
    "Item": "Knife",
    "Price": 3,
    "Qty": 23
  },
  {
    "Item": "Fork",
    "Price": 4,
    "Qty": 45
  }
]
```

The extension to **JSON** has been implemented by introducing a *wrapper*.

## Wrappers

A wrapper is an enclosed vector of the form:

**c**(code special)

The nature of the **special** data structure is identified within the wrapper by a leading numeric code. Code 1 is used to identify JSON values such as `null`, `true` and `false`. Codes 2, 3 and 4 are used to identify different forms of datasets.

This wrapper mechanism has been chosen to identify special treatment because a scalar enclosure cannot be represented in JSON/JavaScript.

A wrapper may be specified directly in the right argument to **JSON** and/or as part of the array structure specified by the right argument, as a sub-array or in a namespace. This allows a special array to be processed appropriately as part of a general data structure that is to be rendered in JSON notation.

## Examples

```
Fields,[1]↑[1]Items Price Qty
```

Item	Price	Qty
Knife	3	23
Fork	4	45

```
□JSON c 2 (Fields,[1]↑[1]Items Price Qty)
[{"Item":"Knife","Price":3,"Qty":23},
{"Item":"Fork","Price":4,"Qty":45}]
```

Note that if you omit the wrapper the operation fails:

```
□JSON Fields,[1]↑[1]Items Price Qty)
DOMAIN ERROR: JSON export: the right argument cannot be
converted (□IO=1)
□JSON Fields,[1]↑[1]Items Price Qty)
^
```

## Further Examples

```
□JSON c 3 ((↑[1]Items Price Qty)Fields)
[{"Item":"Knife","Price":3,"Qty":23},
{"Item":"Fork","Price":4,"Qty":45}]
```

```
□JSON c 4 ((Items Price Qty)Fields)
[{"Item":"Knife","Price":3,"Qty":23},
{"Item":"Fork","Price":4,"Qty":45}]
```

Note that if you omit the wrapper, the operation generates a different result.

```
□JSON ((Items Price Qty)Fields)
[[["Knife","Fork"],[3,4],[23,45]],["Item","Price","Qty"]]
```

## Selection

For codes 2, 3 and 4 the extension also provides the facility to optionally select elements of the dataset, so the array may contain 2, 3 or 4 items:

```
c<(code dataset {records} {fields})
```

where **records** and **fields** are integer indices that select which fields and which records are to be exported. The following example selects the first record and the first and third fields (**Items** and **Qty**)

```
□JSONc4 ((Items Price Qty)Fields)1(1 3)
[{"Item":"Knife","Qty":23}]
```

## Namespaces and Sub-Arrays

Wrappers in namespaces and sub-arrays are recognised for special treatment.

### Example

```

ns.Items←'Fork' 'Knife'
ns.Price←3 4
ns.Qty←23 45
ns.(ds←c4(φ('Item' 'Price' 'Qty')(Items Price
Qty)))
□JSON ns
{"Items":["Knife","Fork"],"Price":[3,4],"Qty":[23,45],
"ds":[{"Item":"Knife","Price":3,"Qty":23},
{"Item":"Fork","Price":4,"Qty":45}]}

a←'the' 'answer' 'is' 42
a[3]←cns.ds
□JSON a
["the","answer",[{"Item":"Knife","Price":3,"Qty":23},
{"Item":"Fork","Price":4,"Qty":45}],42]

```

## Wrappers for special JSON values

Previously, the enclosed character vectors `'null'`, `'false'` and `'true'` were used to represent the JSON values `null`, `false` and `true` respectively. This is still supported, and indeed remains the only way to represent these value on import, but for export the wrapper mechanism may be used instead.

### Example

```

□JSON←"(1 'null')(1 'true')(1 'false')
[null,true,false]"

```

## 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←{α ω}' '▽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←{' 'ω' '}'
':EndNamespace'
DOMAIN ERROR: There were errors processing the script
□FIX':Namespace iaK' 'a←1 ◇ adfn←{' 'ω' '}'
':EndNamespace'
^
```

### Variant Options

`□FIX` may be applied using the Variant operator with the options **Quiet**, **FixWithErrors** and **AllowLateBinding**. 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, <code>□FIX</code> fails with <code>DOMAIN ERROR</code> .
1	<code>□FIX</code> fixes all the namespaces and classes in the script regardless of any errors they may contain.
2	If the script contains errors, <code>□FIX</code> 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	<code>FIX</code> will only fix a Class whose Base class (if specified) is defined in the script or is present in the workspace.
1	<code>FIX</code> will fix a Class whose Base class is neither defined in the script nor present in the workspace.

## Set Shell Script Debug Options

$$R \leftarrow \{X\}(1010\pm)Y$$

This function sets options for debugging Shell Scripts.

`Y` is an integer that selects options as follows.

If `Y` is 1, the lines in the script to be echoed to stderr before they are executed.. The optional left argument `X` specifies a character scalar of vector that prefixes each line of output. If `X` is omitted, the default is `'+'`.

If `Y` is 2, the effect is as if `TRACE` was set for every line of every function in the script. In this case the left argument (if any) is ignored.

If `Y` is 3, it specifies a combination of the above.

The result `R` is the previous value of the debug options.

## Read DataTable

$$R \leftarrow \{X\}2011\pm Y$$

### .NET Framework only

This function performs a *block read* from an instance of the ADO.NET object `System.Data.DataTable`. This object may only be read using an explicit row-wise loop, which is slow at the APL level. `2011±` implements an *internal* row-wise loop which is much faster on large arrays. Furthermore, the function handles NULL values and the conversion of .NET datatypes to the appropriate internal APL form in a more efficient manner than can otherwise be achieved. These 3 factors together mean that the function provides a significant improvement in performance compared to calling the row-wise programming interface directly at the APL level.

`Y` is a scalar or a 1 or 2-item array containing:

1. A reference to an instance of `System.Data.DataTable`.
2. An optional vector which specifies the values to which a `System.DBNull` should be mapped in the corresponding columns of the result

The result `R` depends upon the value of the Variant option **Invert**. This the primary option with a default value of 0.



## Invert Option (Boolean)

0	The result <b>R</b> is a matrix with the same shape as the <code>DataTable</code> referenced by <b>Y</b> .
1	The result <b>R</b> is vector whose length is the same as the number of columns in the <code>DataTable</code> referenced by <b>Y</b> .

The optional left argument **X** is a numeric vector with the same length as the number of columns in the result in the `DataTable` referenced by **Y**:

1	Specifies that the corresponding column of the result should be converted to a string using the <code>ToString</code> method of the data type of column in question.
2	Specifies that numbers of type <code>System.Int64</code> in the corresponding column of the result should be converted to DECFs (NOT into .NET objects, which is the default)
4	Specifies that if the type of the corresponding column is <code>System.String</code> the entire column should be returned as a character matrix rather than a vector of character vectors. Any nulls will be replaced with a row of spaces. This applies only when <b>Invert</b> is 1.
5	Combines 1 and 4.

## Examples

```

[]USING←' ' 'System.Data,system.data.dll'

dt←[]NEW DataTable

add_col←{col←α.Columns.Add 0 ♦ col.DataType←ω}
dt add_col System.String
dt add_col System.Int32
dt add_col System.Int64

in←dt('One' 'Two')(1 2)(6401 6402)
2010I dt in

[]←out←2011I dt

```

One	1	6401
Two	2	6402

```

out[;3].GetType
System.Int64 System.Int64

```

0 0 2(2011I) dt A Convert 3rd col to DECF

One	1	6401
Two	2	6402

1 1 1(2011I)dt A Convert all values to text

One	1	6401
Two	2	6402

((2011I)I('Invert' 1)) dt

One Two		1 2	6401 6402
---------	--	-----	-----------

4 0 0((2011I)I('Invert' 1))dt

One	1 2	6401 6402
Two		

5 5 5((2011I)I('Invert' 1))dt A Convert to cmats

One	1	6401
Two	2	6402

Handling Nulls

2010Idt(1 3pNULL) A Add a row of nulls  
I←out←2011I dt

One	1	6401
Two	2	6402

out[3;].GetType  
System.DBNull System.DBNull System.DBNull

```
2011I dt ('this is null' 'this too' 'and this')
```

One	1	6401
Two	2	6402
this is null	this too	and this

## Performance Considerations

First for comparison is shown the type of code that is required to read a `DataTable` by looping:

```
t<3>[]AI ♦ data1<↑([]dt.Rows).ItemArray ♦ (3>[]AI)-t
191
```

The above expression turns the `dt.Rows` collection into an array using `[]`, and *mixes* the `ItemArray` properties to produce the result. Although here there is no explicit loop, involved, there is an implicit loop required to reference each item of the collection in succession. This operation performs at about 200 rows/sec.

`2011I` does the looping entirely in compiled code and is significantly faster:

```
GetDT<2011I
t<3>[]AI ♦ data2<GetDT dt ♦ (3>[]AI)-t
25
```

In the first case, `2011I` created 365 instances of `System.DateTime` objects in the workspace. If we are willing to receive the timestamps in the form of strings, we can read the data almost an order of magnitude faster:

```
t<3>[]AI ♦ data3<0 0 0 1 GetDT dt ♦ (3>[]AI)-t
3
```

The left argument to `2011I` allows you to flag columns which should be returned as the `ToString()` value of each object in the flagged columns. Although the resulting array looks identical to the original, it is not: The fourth column contains character vectors:

```
~2 4↑data3
364 even 4 18-01-2011 14:03:29
365 odd 5 19-01-2011 14:03:29
```

Depending on your application, you may need to process the text in the fourth column in some way – but the overall performance will probably still be very much better than it would be if `DateTime` objects were used.

Verify .NET Interface

R←2250IY

This function provides information about the Dyalog interface to .NET. The system attempts to load the Bridge DLL and reports the status of the .NET interface. It can be used to determine whether your .NET-related code can run, and also what sort of .NET support you have. It also means that you can suppress all messages that `⎕USING` would otherwise generate

The right argument `Y` is zero:

The result `R` is a 3-element nested array:

Item	Description
R[1]	Numeric. −1: .NET interface is not supported 0: .NET interface is not configured 1: the .NET interface is configured to use .NET Core 2: the .NET interface is configured to use the .NET Framework
R[2]	Boolean 0 or 1. 1 : the Bridge DLL was successfully loaded. 0 : the Bridge DLL failed to load, or was not attempted.
R[3]	A character vector containing error messages generated during load.

Examples (Windows)

```
⎕←2 ⎕NQ '.' 'GetEnvironment' 'Dyalog_NETCore'
⎕←2250I0
2 1

1 ⎕←2 ⎕NQ '.' 'GetEnvironment' 'Dyalog_NETCore'
⎕←2250I0
1 1
```

### Implementation Note

The underlying code is run once only and the results cached, so all subsequent calls to `2250±` will return the same result as the first time.

## Sample Probability Distribution

**R←X ( 16808± ) Y**

This function generates an array of random numbers from a named probability distribution. Note that this is not currently implemented for AIX and macOS.

**Y** is a 2-item vector containing the name of the probability distribution from those listed in the table below, and the shape of the result.

**X** is a scalar or 1 or 2 element numeric vector that specifies parameters.

For example to get an array with shape (3 5 7) of uniform random numbers for the interval from -17.3 to 12.7, you'd enter

```
-17.3 12.7 (16808 ±) 'Uniform' (3 5 7)
```

If you wanted a vector of 100,000 uniform random numbers for that interval, you'd enter

```
-17.3 12.7 (16808 ±) 'Uniform' 100000
```

The domain rules for the distributions currently implemented are as follows:

Distribution	X[1]	X[2]	Domain Rules
'Uniform'	a	b	$a < b$ ; A numeric interval. Example: 1.0 7.6
'Beta'	a	b	$a > 0$ AND $b > 0$
'Bernoulli'	probability		$\text{probability} \geq 0$ AND $\text{probability} \leq 1$
'Binomial'	trials	probability	trials is an integer $\geq 0$ ; $\text{probability} \geq 0$ AND $\text{probability} \leq 1$
'Cauchy'	location	scale	location unrestricted; $\text{scale} > 0$
'Chi Squared'	degree of freedom		$\text{degree of freedom} \geq 0$
'Exponential'	rate		$\text{rate} \geq 0$
'F'	a	b	$a \geq \text{eps}$ AND $b \geq \text{eps}$ ; where eps is smallest non- zero positive float number

Distribution	X[1]	X[2]	Domain Rules
'Gamma '	a	b	$a \geq 0$ AND $b \geq \text{eps}$ ; where eps is smallest non-zero positive float number
'Inverse Gamma '	a	b	$a \geq 0$ AND $b \geq 0$
'Laplace '	location	scale	location unrestricted; $\text{scale} \geq 0$
'Logistic '	location	scale	location unrestricted; $\text{scale} \geq 0$
'Log Normal '	location	scale	location unrestricted; $\text{scale} \geq 0$
'Normal '	location	scale	location unrestricted; $\text{scale} \geq 0$
'Poisson '	rate		$\text{rate} \geq 0$
'Student T '	degree of freedom		degree of freedom $\geq \text{eps}$ where eps is smallest non-zero positive float number
'Weibull '	a	b	$a \geq \text{eps}$ AND $b \geq \text{eps}$ ; eps is smallest non-zero positive float number

Each of those distributions has a corresponding Wikipedia entry with a description of its theoretical foundation and usually graphs of the probability density functions and cumulative distribution functions for interesting sets of parameter values.

### Example

The probability density function for the Beta distribution (see [https://en.wikipedia.org/wiki/Beta\\_distribution](https://en.wikipedia.org/wiki/Beta_distribution)) with the parameter vector (2 5) has an interesting shape.


**BucketCount** counts random numbers that fall into a sequence of evenly distributed bucket intervals:

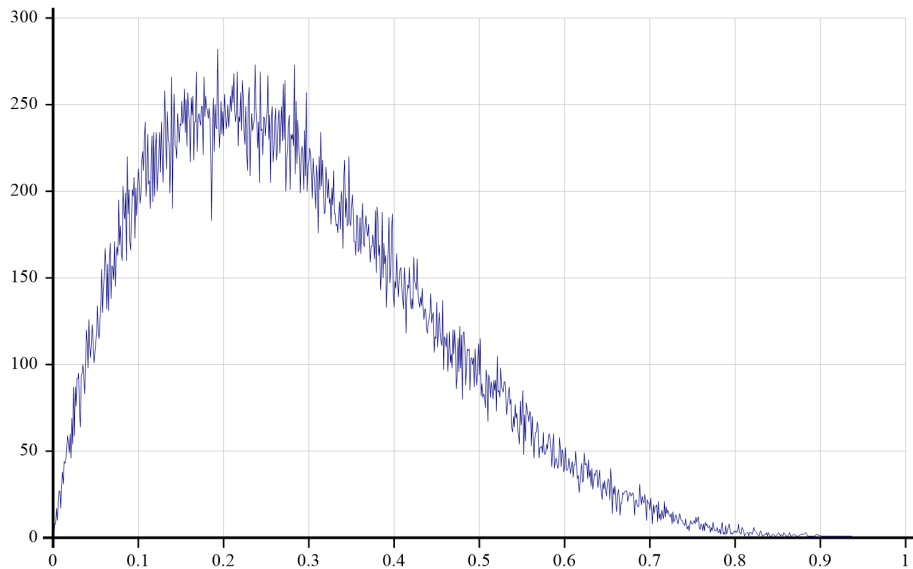
```

      BucketCounts←{
[1]      ir←[ω÷÷α
[2]      kir←{α(≠ω)}⊔ir
[3]      kir[;⊔IO]÷÷α
[4]      kir[⧸kir[;⊔IO];]
[5]      }
```

So then we can create 100,000 samples and calculate values for a density graph with 1,000 evenly spaced buckets by:

```
rv←2 5 (16808I)'Beta' 100000  
bc←1000 BucketCounts rv
```

Using the Chart Wizard  we can plot  $(\epsilon_2) \oslash bc$  against  $(\epsilon_1) \oslash bc$  to get the graph:





# Chapter 2:

## Object Reference Changes

**CornerTitleBCol****Property**

**Applies To:** Grid

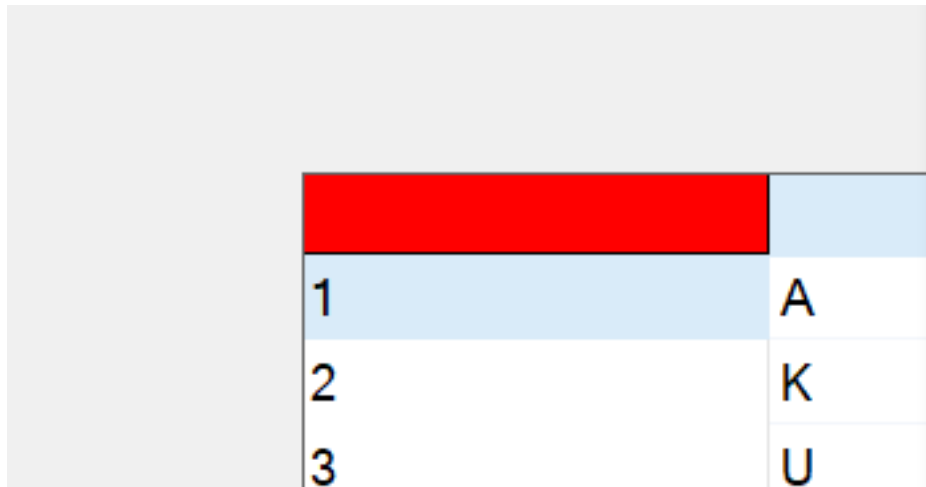
**Description**

This property specifies the colour used to fill the area in the left corner a Grid . This is the rectangle above the row titles and to the left of the column titles.

CornerTitleBCol may be a 3-element vector of integer values in the range 0-255 which refer to the red, green and blue components of the colour respectively, or it may be a scalar that defines a standard Windows colour element (see BCol for details). Its default value is 0 which means that the colour derives from your current Windows colour scheme.

**Example**

f.g.CornerTitleBCol←c255 0 0



The diagram illustrates a grid structure. A large light gray rectangle represents the grid's background. In the bottom-right corner, there is a smaller grid. The top-left cell of this smaller grid is red, representing the area defined by the CornerTitleBCol property. The other cells in this smaller grid are light blue. The first column of the smaller grid contains row numbers 1, 2, and 3. The first row of the smaller grid contains column letters A, K, and U.

1		A
2		K
3		U

**FireOnce****Property**

**Applies To:** Timer

**Description**

This property specifies one-off behaviour for a Timer object. It has the value 0, 1 or 2.

Setting FireOnce to 1, will cause the Timer to generate a single event and no more unless it is reset. After generating the single event, FireOnce is automatically set to 2 and this change occurs prior to the invocation of a callback function.

Setting FireOnce to 2 will cause the Timer to behave as if it has already raised its single event.

FireOnce honours the value of the Active property but does not change it. So setting FireOnce to 1 when Active is 0 will not immediately cause an event. If Active is subsequently set to 1, the single Timer event will then occur.



# Index

## A

AllowLateBinding option 28

## C

CEF 4

classes

fix script 24

CornerTitleBCol 38

## D

dyadic primitive operators  
variant 27-28

## E

Extended Attributes of operations 17

## F

file associations 12

FireOnce 39

fix script 24

FixWithErrors option 27

## I

i-beam

read dataTable 28

verify.NET Interface 32

Interoperability 6

Invert option 29

## J

JSON Extension 20

## K

Key Features 1

key operator 9

## L

load parameter 10

## N

nest 9

## O

over operator 9

## P

PCRE2 3

Principal option 27-28

Properties

CornerTitleBCol 38

FireOnce 39

## Q

Quiet option 27

## R

rank operator 9

read DataTable 28

## S

shell scripts 13

stencil operator 9

Syncfusion 3

System Requirements 5

## V

variant operator 9, 27-28

verify .NET Interface 32

## W

where 9