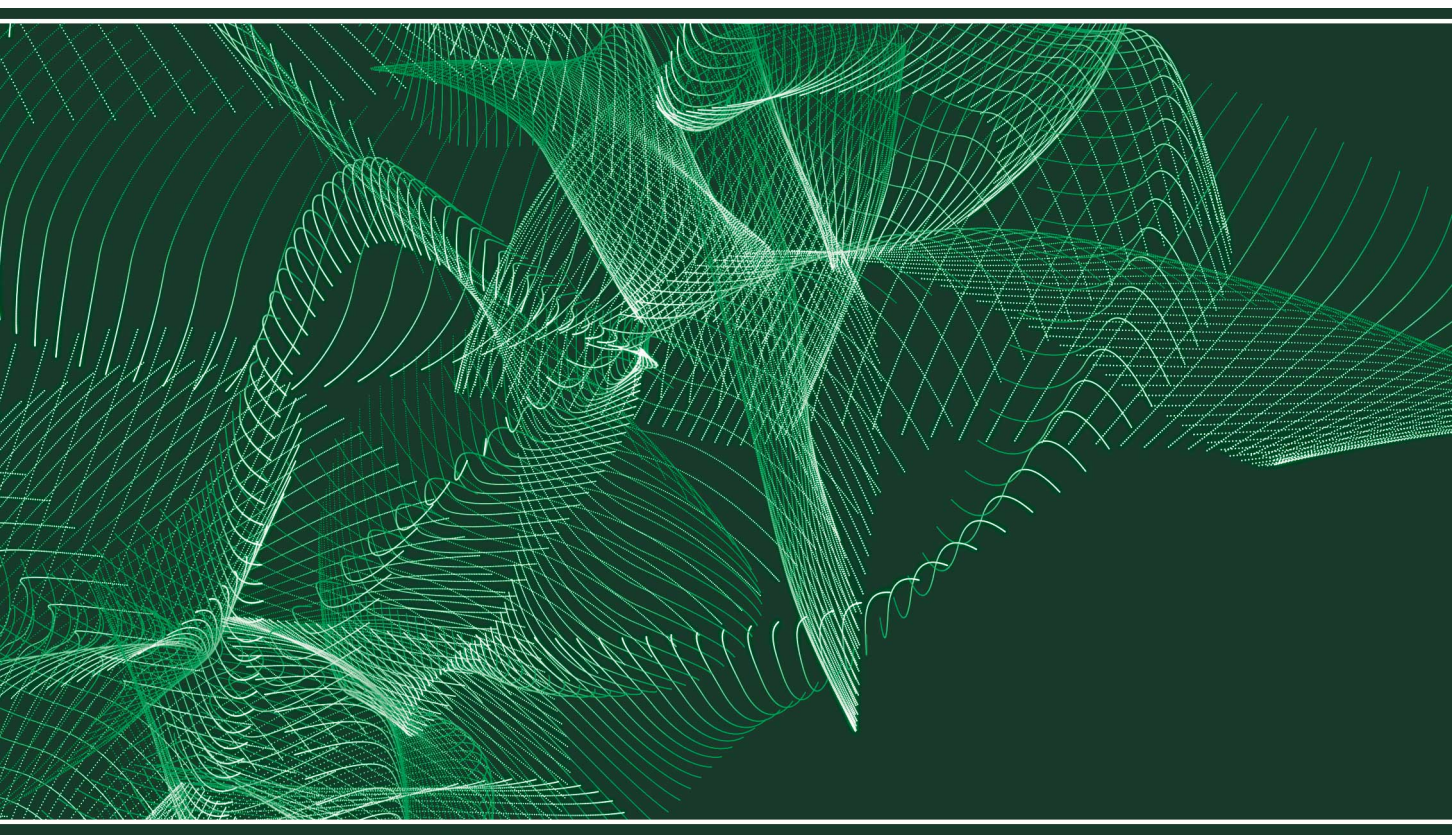
Dyalog
Release Notes

Dyalog version 16.0



DYALOG

The tool of thought for software solutions

Version: 17.0

Revision: 2341 dated 20180530

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

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

Contents

Chapter 1:

Introduction

Key Features

Dyalog APL Version 17.0 provides the following new features, enhancements and changes:

Performance Improvements

As part of the ongoing Performance Quality Assurance project[1], Version 17.0 includes a considerable amount of research and development work designed to substantially improve speed of execution[2].

Language Enhancements

New Language Features

- New monadic ⍸ function. See *Where* on page 35.
- New dyadic ⍸ function. See *Interval Index* on page 28.
- New monadic ⊆ function. See *Nest* on page 36.
- New dyadic ⊆ function. See *Partition* on page 37. This function is independent of ⎕ML, but is identical to dyadic ⊂ when 3≤⎕ML.
- New system function ⎕CSV. See *Comma Separated Values* on page 51.
- New system function ⎕JSON. See *JSON* on page 65. This provides improved and extended functionality to the existing I-beam functions 7159⌶ and 7160⌶ which are deprecated. The JSON Name Mangling rules have changed. See *JSON Name Mangling* on page 74 and *JSON Translate Name* on page 79.
- New @ operator. See *At* on page 40.
- New ⍠ operator. See *Stencil* on page 44.

---

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

[2] https://www.dyalog.com/dyalog/dyalog-versions/160/performance.htm

Enhancements

- Improvement to ⎕DR. You may now specify 0 as the first element of the left argument, to mean convert "as is". See *Language Reference Guide: Data Representation (Dyadic)*.
- ⎕SIGNAL now returns a result, and calling ⎕SIGNAL 0 now causes error-related system constants to be reset. See *Changes to ⎕SIGNAL* on page 22.
- Error guards (dfns) have been changed to use dynamic scope rules. See *Dfn Error-Guards* on page 17.
- There is a new option for ⎕R and ⎕S. See *UCP Option for ⎕R and ⎕S* on page 23.
- The default value for ⎕RL has been changed to (⍬ 1).
- Certain System Commands now take parameters. See *System Command Parameters* on page 75.
- ⎕PROFILE 'start' has a new 'coverage' option that avoids the cost (of memory) in counting the number of times lines of code are executed.
- Version 17.0 introduces trigger functions that apply to assigments to all global variables in the same namespace. See *Global Triggers* on page 26.

IDE Enhancements

- Text searches in the Session and Editor are now performed using PCRE. See *PCRE for Text Searches* on page 19.
- The Editor toolbar has two new option buttons. See *Editor Toolbar* on page 19.
- The edit boxes used in Session GUI components now have a clear button ✕ button. Clicking this button clears the contents of the edit field.
- New Options/Configure dialog. See *Configuration Dialog: Saved Responses Tab* on page 20.

GUI Enhancements

- There is a new HasClearButton property that applies to objects that have edit boxes. See *HasClearButton* on page 83.
- The DevCaps property has a new, 4th, element which reports the DPI scaling factor. See *DevCaps* on page 82.

Distributing Run-Time Components

Under Windows, many of the Dyalog APL run-time components (.EXE and .DLL) are linked dynamically with the Microsoft Universal C Runtime library (the UCRT) which is supplied and installed as part of the normal Dyalog development installation.

At execution time it is important that the Dyalog runtime components bind with a version of the UCRT that is compatible with (i.e. the same as or newer than) the one with which they were built.

Windows 10

If the end-user of the Dyalog application is known to be running Windows 10, the Dyalog application will pick up the system-wide UCRT which is part of Windows 10. There is therefore no need to include the UCRT with a Dyalog run-time application.

Other Versions of Windows

The UCRT is not supplied with versions of Windows prior to Windows 10. On these platforms, it is therefore necessary to install the UCRT as part of the installation of the Dyalog run-time application. There are two ways to achieve this which are referred to herein as the VCRedist installation and App-local installation. Dyalog recommends the former.

VCRedist Installation (Recommended)

The VCRedist package, which includes the UCRT, is supplied with the Dyalog development package.

Simply copy the `vc_redistx86.exe` (32-bit version) or `vc_redistx64.exe` (64-bit version) program from the Dyalog development package into your own installation package and execute it as part of the installation of your Dyalog run-time application. This installs the UCRT into a shared Windows location; in effect the UCRT becomes part of the Windows system. The installation therefore requires Administrator privileges.

App-local Installation

An alternative is to install the UCRT components into the same directory as your Dyalog run-time application. There are two ways to obtain these files.

Either:

Install the Dyalog development package (ideally onto a separate system just for this purpose) without administrator rights. This will perform an App-local installation of Dyalog itself. Then copy the UCRT files into your installation package. These files are:

- those beginning with `api-ms*`
- `ucrtbase.dll`
- `vcruntime140.dll`

Or:

Download and install the Windows 10 SDK from: https://developer.microsoft.com/en-us/windows/downloads/windows-10-sdk, and follow the instructions in the link below.
https://blogs.msdn.microsoft.com/vcblog/2015/03/03/introducing-the-universal-crt

Finally, modify your installer to add these files to the same folder as your Dyalog run-time application.

System Requirements

Microsoft Windows

Dyalog APL Version 17.0 is supported on versions of Windows from Microsoft Windows 7 up to and including Microsoft Windows 10 and Microsoft Windows Server 2016.

Microsoft .NET Interface

Dyalog APL Version 17.0 .NET Interface requires Version 4.0 or greater of the Microsoft .NET Framework. It does *not* operate with earlier versions of .NET.

For full Data Binding support (including support for the `INotifyCollectionChanged` interface[1]), and Syncfusion, Version 17.0 requires .NET Version 4.5.

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

AIX

For AIX, Version 17.0 requires AIX 7.2 or higher, and a POWER7 chip or higher.

Raspberry Pi

On the Raspberry Pi, Dyalog 32-bit Unicode supports Raspbian Jessie or later.

Non-Pi Linux

For non-Pi Linux, Version 17.0 only exists as 64-bit interpreters - there are no 32-bit versions. It is built on Debian 7, and QAed on RedHat 6; it runs on all recent distributions, including Ubuntu 14.04 and openSUSE Leap 42.3. Contact Dyalog for information about other distributions.

macOS/Mac OS X

Version 17.0 requires Mac OS X Yosemite or El Capitan or macOS Sierra or later. The target Mac must have been introduced in 2010 or later.

---

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

Inter-operability

Introduction

Workspaces and component files are stored on disk in a binary format (illegible to text editors). This format differs between machine architectures and among versions of Dyalog. For example, a file component written by a PC may well have an internal format that is different from one written by a UNIX machine. Similarly, a workspace saved from Dyalog Version 17.0 will differ internally from one saved by a previous version of Dyalog APL.

It is convenient for versions of Dyalog APL running on different platforms to be able to *interoperate* by sharing workspaces and component files. From Version 11.0, component files and workspaces can generally be shared between Dyalog interpreters running on different platforms. However, this is not always possible, for example:

- Component files created by Version 10.1 can often not be shared across platforms, even when used by later versions.
- *Small-span* (32-bit) component files become read-only when opened on a different architecture from that on which they were created.

Note however that the system function ⎕FCOPY can be used to make a logically identical copy of an old file, but the copy will be fully inter-operable.

The following sections describe other limitations in inter-operability:

Code and ⎕ORs

Code that is saved in workspaces, or embedded within ⎕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 inter-operability for arrays that only contain (nested) character and numeric data. Such arrays can be stored in component files - or transmitted using `TCPSocket` objects and Conga connections, and shared between all versions and across all platforms.

As mentioned in the introduction, full cross-platform interoperability of component files is only available for large-span component files.

Null Items (⎕NULL)

⎕NULLs created in Version 17.0 can be brought into Versions 14.0, 14.1 and 15.0 provided that the interpreters have been patched to revision 29846 or higher. Attempts to bring ⎕NULL into earlier versions of Dyalog APL or lower revisions of the aforementioned versions will fail with a `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* small-span (32-bit) files; however it is still currently possible to *read* and *write* small span files. Setting the second item of the right argument of ⎕FCREATE to anything other than 64 will generate a `DOMAIN ERROR`.

Note that *small-span* (32-bit-addressing) component files cannot contain Unicode data. Unicode editions of Dyalog APL can only write character data which would be readable by a Classic edition (consisting of elements of ⎕AV).

External Variables

External variables are implemented as small-span (32-bit-addressing) component files, and are subject to the same restrictions as these files. External variables are unlikely to be developed further; Dyalog recommends that applications which use them should switch to using mapped files or traditional component files. Please contact Dyalog if you need further advice on this topic.

32 vs. 64-bit Interpreters

From Dyalog APL Version 11.0 onwards, there are two separate versions of programs for 32-bit and 64-bit machine architectures (the 32-bit versions will also run on 64-bit machines running 64-bit operating systems). There is complete inter-operability between 32- and 64-bit interpreters, except that 32-bit interpreters are unable to work with arrays or workspaces greater than 2GB in size.

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

Unicode vs. Classic Editions

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

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

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

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

Likewise, a Classic edition 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)LOADs or)COPYs a workspace containing Unicode data that cannot be mapped to ⎕AV using the ⎕AVU in the recipient workspace.

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

The symbols ⊆, ⍳, ⍥, ⌹, ⌸ and ⌺ used for the Nest (Interval Index) and Where (Partition) functions, the Rank, Variant, Key and Stencil operators respectively are available only in the Unicode edition. In the Classic edition, these symbols are replaced by ⎕U2286, ⎕U2378, ⎕U2364, ⎕U2360, ⎕U2338 and ⎕U233A respectively. In both Unicode and Classic editions Variant may be represented by ⎕OPT.

Very large array components

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

TCPSockets and Conga

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

Auxiliary Processors

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

Session Files

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

Announcements

Withdrawal of Support for Version 14.0

The supported Versions of Dyalog APL are now Version 16.0, 15.0 and 14.1. Version 14.0 and earlier are no longer supported.

Planned Operating System Requirements for the next version

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

Operating System	Version
Microsoft Windows	Windows 7 or Server 2008
AIX	7.1 on POWER 7
Linux	RedHat/Centos 6 or equivalent
OS X	OS X Yosemite 10.10.x
Raspberry Pi	Raspbian Jessie or above

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

Planned Hardware Requirements for next version

The same as Dyalog 16.0.

Withdrawal of ability to disable Native Look and Feel

In versions 15.0 and 16.0 Native Look and Feel (NLF)is enabled by default but it is possible to disable it. Dyalog expects that the ability to disable NLF will be removed from the next major release of Dyalog APL. Not only do non-NLF applications look more and more dissimilar to "standard" applications, but it is becoming increasingly impractical to support non-NLF code in the interpreter.

Deprecation of small-span component files

Dyalog intends that version 16.0 will be the last that will be able to write to small-span component files.

Since Version 10.1, Dyalog APL has supported large-span (64-bit) component files, and since Version 12.0 ⎕FCREATE has created these by default. From Version 14.0 onwards it has not possible to create small-span component files, although you may still continue to read and write components on existing small-span component files.

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

Deprecation of JSON I-Beam Functions

7159⌶ and 7160⌶ are replaced by ⎕JSON and 7161⌶ has been found to be not required. These three I-beam functions are deprecated (and no longer documented) and will be removed in the next major release. 7162⌶ continues to be supported.

math.dws

The utility math.dws and its associated files fftw.dll and lapack.dll have been removed and are not part of 17.0.

More recent versions of these files for both 32 and 64 bit Windows are now available from github.com/dyalog/math, although math.dws has been replaced with math.dyalog. You can download a zip file for Windows, and it would be possible to download the source and compile the shared libraries for any platform as long as there is a Fortran compiler as well as a C compiler installed on the system.

Withdrawal of the default_rl parameter

17.0 no longer checks for the value of **default_rl**; it no longer appears in the Configure Box, nor appears in the registry. If set as an environment variable, Dyalog ignores its value.

Withdrawal of 16807⌶

As announced in the *Dyalog Version 15.0 Release Notes*, 16807⌶ has been withdrawn from 17.0.

A simple replacement for "16807⌶1" is "⎕RL←⍬ 1" (or "⎕RL←0 1" if you need to be able to generate repeatable sequences of "random" numbers).

Note that, since 14.0, random number generator 1 is the default in a CLEAR WS so you may not need to perform these steps unless the workspace was created in a version earlier than 14.0.

Conga and Windows

Conga 3 requires Windows Vista or higher, or Windows Server 2008 or higher. Note that this is now also true of Conga 2.6 and higher.

RIDE DLLs/shared libraries

In previous versions of Dyalog separate DLLs/shared libraries were supplied for Conga and for RIDE. In Version 16.0 the two sets of shared libraries/DLLs have been merged; the Conga libraries are required for RIDE to function.

Bug Fixes

A number of bug fixes implemented in Version 17.0 may change the way that existing code operates and are therefore documented in this section.

Namespace serialisation (14144)

Version 16.0 is stricter than 15.0 about serialising namespaces. "Serialising" happens every time you write a namespace to a component file or send it over a socket. (There is also an explicit "Serialise/Deserialise Array" I-beam, `220⌶`.)

In 16.0, when you serialise a namespace `A` it can only refer to other namespaces that are children of itself. If it contains a ref to some other namespace (e.g. a parent or sibling of `A`) then you will get an error:

```
      A←⎕NS''     ⍝ Create two sibling namespaces
                  ⍝ (both are children of #)
      B←⎕NS''
      A.x←B       ⍝ Create a ref from A to B
      {}1(220⌶)A ⍝ Try to serialise A
DOMAIN ERROR: Namespace is not self contained
```

15.0 allowed you to serialise non-self-contained namespaces like `A`, but was inconsistent about where it would recreate the namespaces when you deserialised them (e.g. when reading the namespace back from a component file); typically `B` would be recreated as a child of `A`, thus breaking the original namespace hierarchy.

Control Structure Parameters (13633)

Prior to 17.0, parameters supplied to control structure expressions that do not expect a parameter were ignored. From 17.0 onwards any control structure that should not have a parameter **must not** have a parameter. For example, the expression:

```
   :Continue 88
```

now generates a `SYNTAX ERROR`; whereas previously the unwanted parameter 88 was ignored.

Dfn Localisation Issue (7945)

There was a dfn bug whereby in executing an expression such as `f←{(⍴)var←⍵}`, where a simple assignment is immediately preceded by a function in parentheses, the system failed to localise `var`, so it would be established as a global variable.

The fix for this issue causes some dfns, that previously appeared to work, to fail. For example:

```
      foo←+ ⍝ foo is defined as any function
      f←{(left argument expression)foo var←⍵}
```

15.0 behaviour:

The expression `f 0` would assign `ω` to `var` and then apply `foo` dyadically. However, the system failed to localise `var`, but instead established it as a spurious global variable. And this isn't how dfns are supposed to work anyway; an expression like `foo var←ω` is supposed to assign to both `foo` and `var`, regardless of how/whether they were previously defined.

16.0 behaviour:

The expression generates a `SYNTAX ERROR` or `LENGTH ERROR`. The fix is to rewrite the expression in one of these two ways, according to taste:

```
f←{(left argument expression)foo ⊢ var←ω} ⍝ add ⊢
f←{(left argument expression)foo (var←ω)} ⍝ add parens
```

Invalid Control Structure Syntax

Invalid control structure, such as in the following example, now generates a SYNTAX ERROR as it should. In previous versions it was wrongly accepted.

```
      ∇ foo;a;b
[1]     a←b←1
[2]     :If a :AndIf b
[3]         'a and b are true'
[4]     :EndIf
[5]     ∇
      ∇
```

Scope of System Variables

The scope of system variables ⎕DCT and ⎕FR has been changed from workspace scope to namespace scope.

Chapter 2: Miscellaneous

Dfn Error-Guards

Prior to Version 16.0, in common with regular names, error-guards used "lexical scope" rules. This meant that when an error occurred, the interpreter would look outwards through enclosing nesting levels to find an appropriate error-guard. In particular, a function could not set an error-guard for a called function unless the called function was nested within the calling function.

With Version 16.0, error-guards have been changed to use "dynamic scope" rules. This means that when an error occurs, the interpreter looks back along the calling stack to find an appropriate error-guard: a calling function can set an error-guard for a called function, irrespective of their relative nesting levels.

```
main←{              ⍝ from V16:
    0::'error'      ⍝ this error-guard will catch ...
    sub ⍵
}

sub←{÷0}            ⍝ ... this error
```

Here is an example of a piece of code that will behave differently in Version 16.0:

```
{
    11::'old behaviour'
    {
        ÷0
    }{
        11::'new behaviour'
        ⍺⍺ ⍵
    }⍵
}
```

Note:

Following the setting of an error-guard, subsequent function calls will disable tail call optimisation:

```
{
    22::'Oops!'       ⍝ this error-guard means that ...
    tie←⍵ ⎕ftie 0
    subfn tie         ⍝ ... tail call not optimised
}
```

One way to maintain the tail call optimisation in the presence of an error-guard is to isolate it within an inner function:

```
{
    tie←{
        22::0          ⍝ error-guard local to inner fn
        ⍵ ⎕ftie 0
    }⍵
    tie=0:'Oops!'
    subfn tie          ⍝ ... so this is a tail call
}
```

IDE Enhancements

PCRE for Text Searches

Text searches in the *Workspace Search Tool* and *Find/Replace* dialogs are now performed using PCRE. If the *Use Regular Expressions* box is checked, the full range of regular expressions provided by PCRE are available for use.

See *Language Reference Guide: Appendix A*. This is the same mechanism as used by ▯R and ▯S.

Editor Toolbar

The Editor and Tracer toolbar contains two new option buttons which affect how searches are performed in the Edit window.

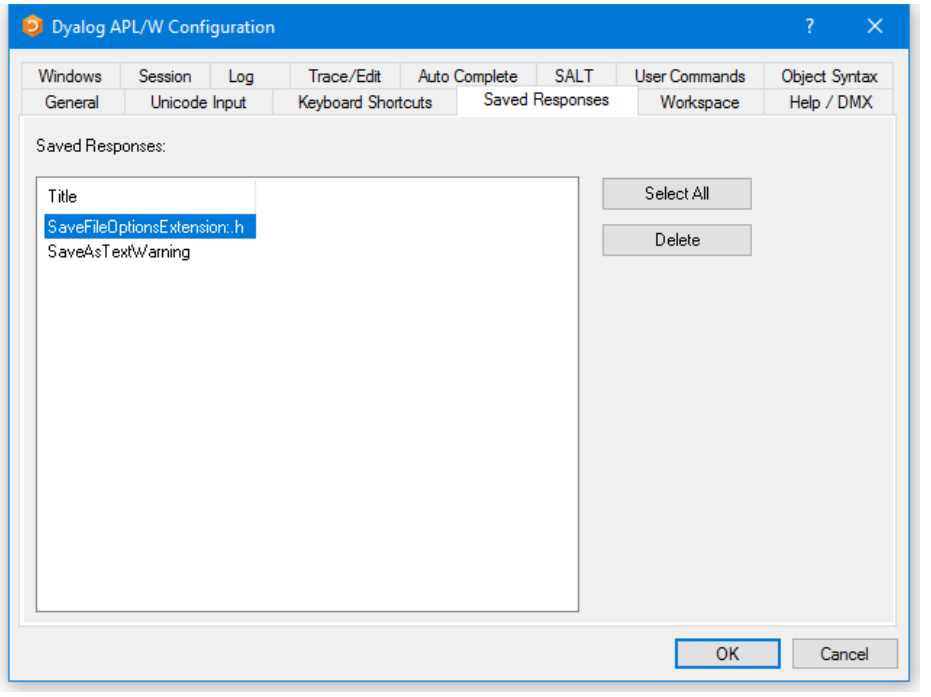
Button	Description
Aa aA Match whole word	Specifies whether or not the search matches a whole word
.* .* Use Regular Expressions	Specifies whether or not the search uses PCRE regular expressions

Clear Button

Where appropriate, an ✕ (clear) button has been added to edit boxes in the Session dialog boxes. Clicking the ✕ button clears the data in the field.

Configuration Dialog: Saved Responses Tab

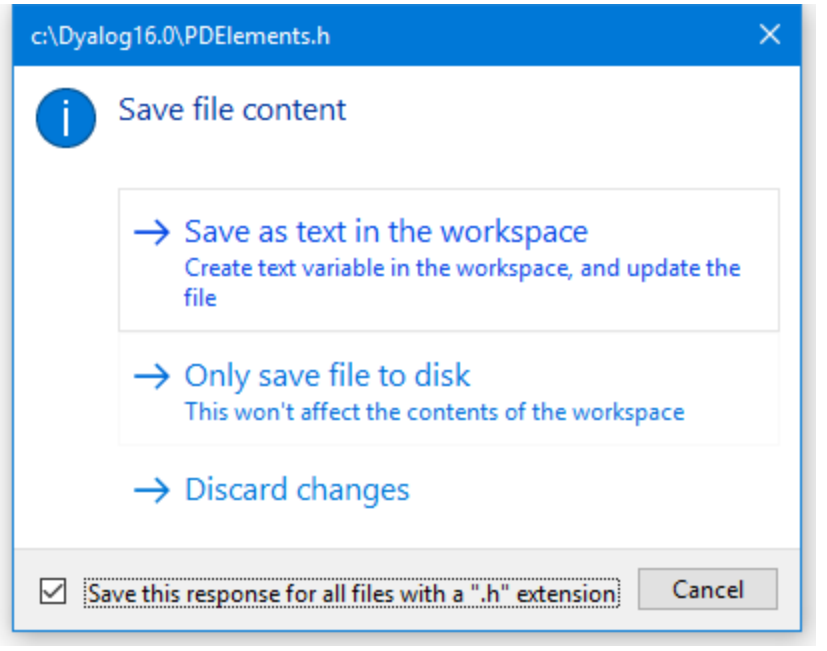
There is now a *Saved Responses* tab on the *Configuration Dialog*. This was actually added in Version 15.0 but after the completion of the documentation.

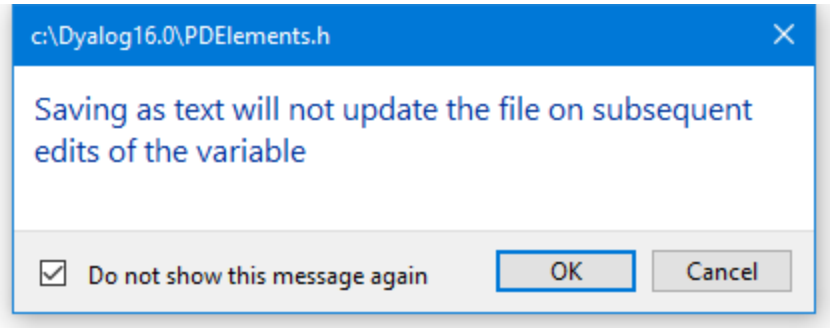


The *Saved Responses* tab of the *Configuration* dialog is used to remove preferences that the user has previously established.

In the example illustrated above, the user has at some point chosen to save a text file with a `.h` extension as text in the workspace and, by checking the option *Save this response for all files with a ".h" extension*, saved this as a preference for all such text files. Similarly, the user has checked the option *Do not show this message again* when responding to the warning dialog *Saving as text will ...*.

If the user wishes to reverse these decisions, even temporarily, it is necessary to select the corresponding option /preference name(s) and click *Delete*. The names are intended to be self-explanatory and are not listed here.


c:\Dyalog16.0\PDElements.h
Save file content
Save as text in the workspace
Create text variable in the workspace, and update the file
Only save file to disk
This won't affect the contents of the workspace
Discard changes
Save this response for all files with a ".h" extension
Cancel


c:\Dyalog16.0\PDElements.h
Saving as text will not update the file on subsequent edits of the variable
Do not show this message again
OK
Cancel

Other Changes

)LIB

Under Windows, the names of directories listed by)LIB were displayed in upper-case. This has been changed and)LIB now reports the names of directories as they are displayed by the Operating System.

MAXWS

With the exception of the Raspberry Pi, the default value of MAXWS has increased from 64Mb to 256Mb.

CONTINUE workspace

The handling of the continue workspace has been changed as follows:

- By default the automatic saving of a CONTINUE workspace (such as when a run-time violation occurs) is disabled. There is a new I-Beam function to control the autosave of a CONTINUE workspace; i.e. other than when directed to do so by the)CONTINUE system command. See *Continue Autosave* on page 78.
  Note that Dyalog will not introduce a configuration parameter to mimic the setting of 2704⌶; this is considered a security issue and therefore is one that must be under APL program control.
- Dyalog no longer loads a CONTINUE workspace automatically on start-up. The only way a CONTINUE workspace may be loaded is by using)LOAD or ⎕LOAD or by specifying `continue.dws` on the command line

Command Line option

The -F flag has been removed as a command-line option for starting Dyalog. It has been ignored since Version 14.0.

Changes to ⎕SIGNAL

⎕SIGNAL now returns a shy result which is the same as the right argument.

⎕SIGNAL 0 is a special case of ⎕SIGNAL, and results in ⎕DM, ⎕EN, ⎕DMX and ⎕EXCEPTION being reset.

⎕SIGNAL 0 is the only form of ⎕SIGNAL that can be used to reset the aforementioned system variables; including a left argument or using a name/value pair right argument of ⎕SIGNAL will result in a DOMAIN ERROR.

Shy Result for ⎕PROFILE

⎕PROFILE'start', 'stop', 'clear' and 'calibrate' now return shy results.

UCP Option for ⎕R and ⎕S

There is a new UCP variant option for ⎕R and ⎕S which is either 0 or 1.

This affects the way PCRE that processes \B, \b, \D, \d, \S, \s, \W, \w, and some of the POSIX character classes.

If UCP is 0 (the default), only ASCII characters are recognized. If UCP is 1, Unicode properties are used instead to classify characters.

Examples

By default, the character ø (which is not an ASCII character) is considered to be a "non-word" character, so:

```
      ('\w'⎕S'\0')'Bjørn'  ⍝ identify "word" characters
B  j  r  n
      ('\W'⎕S'\0')'Bjørn'  ⍝ non-word" characters
ø
```

When UCP is set to 1, Unicode characters are matched as "word" characters (\w) too.

```
      ('\w'⎕S'\0' ⍠'UCP' 1)'Bjørn'
B  j  ø  r  n
```

System Error Dialog Box

The System Error Dialog box has changed and the option *Generate complete image core* has been replaced by *Create a process dump file*. See *Programming Reference Guide: System Error Dialog Box*.

Copy, Cut and Paste Options

The configuration option *Use Ctrl-X,C,V for clipboard* option has been moved from the interpreter to the IME. See *UI Guide: Configuration Dialog, Unicode Input Tab*.

Changes to German locale files

Session files and Classic input translate tables for the German locale have been renamed so that they are now referred to as "DE" rather than "GR". The BuildSe workspace has been updated to reflect these changes.

Changes to French keyboard layout in the IME

The French keyboard supplied with the IME has been changed so that it now has the same APL character overlay as other languages (for example, `⍺` appears on the "q" key rather than the "a" key). The previous version appears in the aplkeys subdirectory of the IME installation directory as fr-FR_legacy.din.

F1 Help

Under Windows, F1 help is now case-insensitive, so if you place the cursor on a string such as `size` and press function key F1, you will obtain the help topic for the Size property.

Similarly, if you have a default Session (with *Use online help for non-Dyalog topics* checked, and the URL set to *http://social.msdn.microsoft.com/Search/en-US?query=%s*) typing `com` and pressing F1 will launch the appropriate msdn page with "com" as the search string.

Native File Functions and APLX

The native file functions `⎕NREAD` and `⎕NREPLACE` have been enhanced for compatibility with APLX. Specifically, the current file position may be specified by the value `¯1`. This enhancement was added to Version 15.0.

Chapter 3:

Language Reference Changes

Language Changes

The following table summarises the main changes to language features in Version 16.0.

Function	Description	Change
⍸ (monadic)	Where	New function
⍸ (dyadic)	Interval Index	New function
⊆ (monadic)	Nest	New function
⊆ (dyadic)	Partition	Equivalent to dyadic ⊂ but independent of ⎕ML
@	At	New operator
⌹	Stencil	New operator
⎕CSV	Comma Separated Values	New system function
⎕JSON	JSON Import and Export	New system function. Replaces 7159⌶ and 7160⌶
⎕R and ⎕S		New UCP option
dfns & dops	Error Guards	Error guards have been changed to use dynamic scope rule
⎕RL	New default	The default value for ⎕RL has been changed to ⍬ 1
⎕SIGNAL		⎕SIGNAL 0 causes error-related system constant to be reset; ⎕SIGNAL now returns a shy result
Trigger *		See *Global Triggers* on page 26.

Global Triggers

Version 17.0 provides an enhancement for Triggers that allows a single function to trigger on any assignment to a global variable in the same namespace.

This is implemented by the function declaration statement:

```
:Implements Trigger *
```

Example:

```
      ∇ foo args
[1]     :Implements Trigger *
[2]     args.Name,' is: ',⍕args.Name
      ∇
       a←⍳10
a is:  1 2 3 4 5 6 7 8 9 10
      a[3]←⊂'pete'
a is:  1 2  pete  4 5 6 7 8 9 10
      a b←10 'pete'
a is:  10
b is: pete
```

Notes:

- like other Triggers, only the most recently fixed global trigger function will apply and be called on assignment to a global variable.
- global triggers do not apply to local names nor to semi-globals (names which are localised further up the stack).
- in the case of a global trigger, the argument to the trigger function (an instance of the internal class `TriggerArguments`) contains the single member `Name`. This may be extended in the future.
- an assignment to a global variable will fire both its specific trigger (if defined) and the global trigger. However, the order of execution is undefined.

Further Example

A potential use for a global trigger is to detect the unintended creation of global variables due to localisation omissions. Note however that the timing of the activation of the Trigger is unpredictable. In this example, the trigger for the assignment to `b` activates after funtion `hoo` has exited. When Threads are involved, timing becomes even less predictable.

```
      ∇ CatchGlobals arg
[1]     ⍝ Displays a warning when a global is assigned
[2]     :Implements Trigger *
[3]     '*** assigment to global variable: ',
         arg.Name,' from ',1↓⎕SI
      ∇
      ∇ foo
[1]     goo
      ∇
      ∇ goo
[1]     hoo
      ∇
      ∇ hoo
[1]     a←10
[2]     b←a
      ∇
       foo
*** assigment to global variable: a from  hoo  goo  foo
*** assigment to global variable: b from  goo  foo
```

Interval Index R←X⍸Y

Classic Edition: the symbol ⍸ (Iota Underbar) is not available in Classic Edition, and Nest is instead represented by ⎕U2378.

X is an ordered non-scalar homogeneous array that represents a set of intervals or ranges.

Note that the i th interval starts at X[i], then includes all subsequent values up to but not including X[i+1].

For example, if X is (1 3 5) it defines 4 intervals numbered 0 to 3 as follows.

0	less than 1	<1
1	between 1 and 3	(≥1)∧(<3)
2	between 3 and 5	(≥3)∧(<5)
3	greater than or equal to 5	≥5

If X is 'AEIOU' it defines 6 intervals numbered 0 to 5 as follows:

0	before A	⎕UCS[0,⍳64]
1	between A and E	ABCD
2	between E and I	EFGH
3	between I and O	IJKLMN
4	between O and U	OPQREST
5	U and after	UVWXYZ...

Y is an array of the same type (numeric or character) as X.

The result R is an integer array that identifies into which interval the corresponding value in Y falls.

Like dyadic ⍳ (see *Index Of* on page 1), Interval Index works with major cells. For a vector these are its elements; for a matrix its rows, and so forth.

X and Y are compared using the same logic as monadic ⍋ (see *Grade Up (Monadic)* on page 1) which is independent of ⎕CT and ⎕DCT.

⎕IO is an implicit arguments of Interval Index. In all the following examples, ⎕IO is 1.

Examples

```
      10 20 30⍳11 1 31 21
1 0 3 2
```

In the above example:

- 11 is between `X[1]` and `X[2]` so the answer is 1.
- 1 is less than `X[1]` so the answer is 0
- 31 is greater than `X[⍴X]` so the answer is 3
- 21 is between `X[2]` and `X[3]` so the answer is 2.

```
      'AEIOU' ⍳ 'DYALOG'
1 5 1 3 4 2
```

And in the alphabetic example above:

- "D" is between `X[1]` and `X[2]`, so the answer is 1
- "Y" is after `X[⍴X]` so the answer is 5
- "A" is between `X[1]` and `X[2]`, so the answer is 1
- as so on ...

Example (Classification)

Commercially, olive oil is graded as follows:

- if its acidity is less than 0.8%, as "Extra Virgin"
- if its acidity is less than 2%, as "Virgin"
- if its acidity is less than 3.3%, as "Ordinary"
- otherwise, as "Lampante"

```
      grades←'Extra Virgin' 'Virgin' 'Ordinary' 'Lampante'
      acidity←0.8 2 3.3

      samples←1.3 1.9 0.7 4 .6 3.2
      acidity⍳samples
1 1 0 3 0 2
      samples,⍪grades[1+acidity⍳samples]
```

1.3	Virgin
1.9	Virgin
0.7	Extra Virgin
4	Lampante
0.6	Extra Virgin
3.2	Ordinary

Example (Data Consolidation by Interval)

x represents some data sampled in chronological order at timestamps t.

```
      ρx
200000
      x
3984300 2020650 819000 1677100 3959200 2177250 3431800
...
      ρt
200000 3
      (10↑t) (¯10↑t)
┌─────┬─────────┐
│0 0 0│23 59 54 │
│0 0 0│23 59 55 │
│0 0 0│23 59 56 │
│0 0 0│23 59 56 │
│0 0 0│23 59 58 │
│0 0 2│23 59 58 │
│0 0 3│23 59 59 │
│0 0 3│23 59 59 │
│0 0 4│23 59 59 │
│0 0 5│23 59 59 │
└─────┴─────────┘
```

u represents timestamps for 5-minute intervals:

```
      ρu
288 3
      (10↑u) (¯10↑u)
┌─────┬─────────┐
│0  0 0│23 10 0 │
│0  5 0│23 15 0 │
│0 10 0│23 20 0 │
│0 15 0│23 25 0 │
│0 20 0│23 30 0 │
│0 25 0│23 35 0 │
│0 30 0│23 40 0 │
│0 35 0│23 45 0 │
│0 40 0│23 50 0 │
│0 45 0│23 55 0 │
└─────┴─────────┘
```

Therefore, the expression `(u⍸t){+/⍵}⌸x` summarises `x` in 5-minute intervals.

```
      u ⍸ t
1 1 1 1 1 1 1 1 1 1 ... 288 288 288 288 288 288

      (u⍸t) {+/⍵}⌸ x
1339083050 1365108650 1541944750 1393476000 1454347100
...

      (u⍸t) {(⍺⌷u),+/⍵}⌸ x
 0  0 0 1339083050
 0  5 0 1365108650
 0 10 0 1541944750
 0 15 0 1393476000
   ...
23 45 0 1388823150
23 50 0 1453472350
23 55 0 1492078850
```

Higher-Rank Left Argument

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

Example

```
      x ← ↑ 'Fi' 'Jay' 'John' 'Morten' 'Roger'
      x
Fi
Jay
John
Morten
Roger
      ⍴x
5 6
      y ← x ⍪ ↑ 'JD' 'Jd' 'Geoff' 'Alpha' 'Omega' 'Zeus
'
      y
Fi
Jay
John
Morten
Roger
JD
Jd
Geoff
Alpha
Omega
Zeus

      x ⍸ y
1 2 3 4 5 1 2 1 0 4 5
      y ,⍪ x⍸y
Fi     1
Jay    2
John   3
Morten 4
Roger  5
JD     1
Jd     2
Geoff  1
Alpha  0
Omega  4
Zeus   5
```

Further Example

```
      ⍴x
5 6
      ⍴y
3 3 6
      x
Fi
Jay
John
Morten
Roger
      y
Fi
Jay
John

Morten
Roger
JD

Jd
Geoff
Alpha
      x⍳y
1 2 3
4 5 1
2 1 0
```

Example

A card-player likes to sort a hand into suits spades, hearts, diamond, clubs (fortunately alphabetic) and high-to-low within each suit.

```
      suits←'Clubs' 'Diamonds' 'Hearts' 'Spades'
      pack←,(⊂¨suits)∘.,1↓14 ⍝ 11=Jack ... 14=Ace
      hand←↑(,pack)[7?52]
      hand←hand[⍒hand;]
      hand
```

Spades	12
Hearts	12
Hearts	7
Hearts	2
Diamonds	11
Diamonds	9
Clubs	8

Another card, the 10 of diamonds is dealt. Where must it go in the hand ?

```
      (⍋hand)⍸'Diamonds' 10 ⍝ left arg must be sorted up
2
      (¯2↓hand)⍪'Diamonds' 10⍪¯2↑hand
```

Spades	12
Hearts	12
Hearts	7
Hearts	2
Diamonds	11
Diamonds	10
Diamonds	9
Clubs	8

Note that if `(∧/Y∊X)` and `X` is sorted and `⎕CT=0` ,then `x⍸y` is the same as `x⍳y`.

Where R←⍸Y

Classic Edition: the symbol `⍸` (Iota Underbar) is not available in Classic Edition, and Where is instead represented by `⎕U2378`.

`Y` must be a simple Boolean array.

`R` is a vector of the indices of all the 1s in `Y`. If `Y` is all zeros, `R` is an empty vector.

`⎕IO` is an implicit argument of Where.

Examples

```
      ⎕IO
1
      ⍸ 1 0 1 0 0 0 0 1 0
1 3 8

      ⍸'e'='Pete'
2 4

      3 4⍴0 1 1
0 1 1 0
1 1 0 1
1 0 1 1

      ⍸ 3 4⍴0 1 1
┌───┬───┬───┬───┬───┬───┬───┬───┐
│1 2│1 3│2 1│2 2│2 4│3 1│3 3│3 4│
└───┴───┴───┴───┴───┴───┴───┴───┘

      ⍸2 3 4⍴0 0 0 1
┌─────┬─────┬─────┬─────┐
│1 2 1│1 3 2│2 1 3│2 2 4│
└─────┴─────┴─────┴─────┘

      ⍸3 1 4 2
DOMAIN ERROR
      ⍸3 1 4 2
     ^
```

Nest R←⊆Y

Classic Edition: the symbol ⊆ (Left Shoe Underbar) is not available in Classic Edition, and Nest is instead represented by ⎕U2286.

Y may be any array.

If Y is simple, R is a scalar array whose item is the array Y. If Y is a simple scalar or is already nested, R is Y unchanged.

Examples

```
      ⊆1 2 3
┌─────┐
│1 2 3│
└─────┘
      ⊆ 1 (1 2 3)
┌─┬─────┐
│1│1 2 3│
└─┴─────┘
      ⊆'Dyalog'
┌──────┐
│Dyalog│
└──────┘
      ⊆'Dyalog' 'APL'
┌──────┬───┐
│Dyalog│APL│
└──────┴───┘
```

Partition R←X⊆[K]Y

Classic Edition: the symbol ⊆ (Left Shoe Underbar) is not available in Classic Edition, and Partition is instead represented by ⎕U2286.

Y may be any non-scalar array.

X must be a simple scalar or vector of non-negative integers.

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

R is an array of the elements of Y partitioned according to X.

A new partition is started in the result whenever the corresponding element in X is greater than the previous one. Items in Y corresponding to 0s in X are not included in the result.

Note that if ⎕ML≥3, the symbol ⊂ means the same as ⊆.

Examples

```
      ⎕ML←3

      ]display 1 1 1 2 2 3 3 3⊆'NOWISTHE'
┌→──────────────────────┐
│ ┌→──┐ ┌→─┐ ┌→───┐    │
│ │NOW│ │IS│ │THE │    │
│ └───┘ └──┘ └────┘    │
└∊──────────────────────┘

      ]display 1 1 1 0 0 3 3 3⊆'NOWISTHE'
┌→──────────────────┐
│ ┌→──┐ ┌→───┐      │
│ │NOW│ │THE │      │
│ └───┘ └────┘      │
└∊──────────────────┘

      TEXT←'   NOW     IS      THE      TIME    '
      ]display (' '≠TEXT)⊂TEXT
┌→──────────────────────────────┐
│ ┌→──┐ ┌→─┐ ┌→───┐ ┌→────┐    │
│ │NOW│ │IS│ │THE │ │TIME │    │
│ └───┘ └──┘ └────┘ └─────┘    │
└∊──────────────────────────────┘
```

```
      ]display CMAT←⎕FMT(' ',ROWS),COLS⍪NMAT
┌→──────────────────────────────────────┐
↓             Jan   Feb  Mar             │
│ Cakes         0   100  150             │
│ Biscuits      0     0  350             │
│ Buns          0  1000  500             │
└───────────────────────────────────────┘

      ]display (∨⌿' '≠CMAT)⊆CMAT    ⍝ Split at blank cols.
┌→──────────────────────────────────────────────────────────┐
↓ ┌→──────────┐ ┌→───┐ ┌→────┐ ┌→───┐                      │
│ │           │ │Jan │ │ Feb │ │Mar │                      │
│ └───────────┘ └────┘ └─────┘ └────┘                      │
│ ┌→──────────┐ ┌→───┐ ┌→────┐ ┌→───┐                      │
│ │Cakes      │ │  0 │ │ 100 │ │150 │                      │
│ └───────────┘ └────┘ └─────┘ └────┘                      │
│ ┌→──────────┐ ┌→───┐ ┌→────┐ ┌→───┐                      │
│ │Biscuits   │ │  0 │ │   0 │ │350 │                      │
│ └───────────┘ └────┘ └─────┘ └────┘                      │
│ ┌→──────────┐ ┌→───┐ ┌→────┐ ┌→───┐                      │
│ │Buns       │ │  0 │ │1000 │ │500 │                      │
│ └───────────┘ └────┘ └─────┘ └────┘                      │
└∊──────────────────────────────────────────────────────────┘

      ]display N←4 4⍴⍳16
┌→──────────┐
↓ 1  2  3  4 │
│ 5  6  7  8 │
│ 9 10 11 12 │
│13 14 15 16 │
└~───────────┘

      ]display 1 1 0 1⊆N
┌→──────────────────────┐
↓ ┌→──┐   ┌→──┐         │
│ │1 2│   │ 4 │         │
│ └~──┘   └~──┘         │
│ ┌→──┐   ┌→──┐         │
│ │5 6│   │ 8 │         │
│ └~──┘   └~──┘         │
│ ┌→───┐  ┌→───┐        │
│ │9 10│  │12  │        │
│ └~───┘  └~───┘        │
│ ┌→────┐ ┌→───┐        │
│ │13 14│ │16  │        │
│ └~────┘ └~───┘        │
└∊──────────────────────┘
```

```
]display 1 1 0 1⊆[1]N
┌→──────────────────────────────────┐
↓ ┌→──┐ ┌→──┐ ┌→──┐ ┌→──┐
│ │1 5│ │2 6│ │3 7│ │4 8│
│ └~──┘ └~──┘ └~──┘ └~──┘
│ ┌→──┐ ┌→──┐ ┌→──┐ ┌→──┐
│ │13 │ │14 │ │15 │ │16 │
│ └~──┘ └~──┘ └~──┘ └~──┘
└∊──────────────────────────────────┘
```

At R←{X}(f@g)Y

This operator substitutes selected items in Y with new values or applies a function to modify selected items in Y.

The right operand g identifies which items of array Y are to be substituted or modified. It is either:

- an array that specifies a set of indices in Y. If g is a simple scalar or vector, it selects major cells in Y. If nested, it specifies indices for Choose or Reach indexing.
- or a function that when applied to Y returns a Boolean array of the same shape as Y (a *mask*) in which a 1 indicates that the corresponding item of Y is to be substituted or modified. Note that the *ravel* of the mask selects from the *ravel* of the right argument's index array.

The left operand f is either:

- an array that contains values to replace those items in Y identified by g
- or a function to be applied to those items, the result of which is used to replace them. If this function is dyadic, its left argument is the array X. Note that the function is applied to the sub-array of Y selected by g*as a whole* and not to each item separately.

The result R is the same as Y but with the items specified by g substituted or modified by f.

Examples (array @ array)

Replace the 2nd and 4th items of `⍳5`:

```
      (10 20@2 4)⍳5    ⍝ 1
1 10 3 20 5

      10 20@2 4⍳5
1 10 3 20 5
```

Replace the 2nd and 4th items of nested vector with `⍬`:

```
      (⊂⍬)@2 4 ⍳¨⍳5
┌─┐┌─────┐┌───────┐
│1│└┴─┴─┘│1 2 3 4 5│
└─┘      └─┴─┴─┴─┴─┘
```

Replace the 2nd and 4th rows (major cells) of a matrix:

```
      (2 3⍴10 20)(@2 4)4 3⍴⍳12
 1  2  3
10 20 10
 7  8  9
20 10 20
```

Replace first and last elements with 0 using Choose Indexing:

```
      (0@(1 1)(4 3))4 3⍴⍳12
 0  2 3
 4  5 6
 7  8 9
10 11 0
```

[1]Note that the expression does not require parentheses because without them, the array `2 4` binds anyway to the `@` operator rather than to the `⍳` function.

Replace nested items using Reach Indexing:

```
      G
┌──────┬──────┬──────┐
│ABC 1 │DEF 2 │GHI 3 │
├──────┼──────┼──────┤
│JKL 4 │MNO 5 │PQR 6 │
└──────┴──────┴──────┘
      G[((1 2)1)((2 3)2)]
┌───┐
│DEF│6│
└───┘
      ('' '*' @((1 2)1)((2 3)2)) G
┌──────┬──────┬──────┐
│ABC 1 │ │2   │GHI 3 │
├──────┼──────┼──────┤
│JKL 4 │MNO 5 │PQR * │
└──────┴──────┴──────┘
```

Examples (function @ array)

Replace the 2nd and 4th items of ⍳5 with their reciprocals:

```
      ÷@2 4 ⍳5
1 0.5 3 0.25 5
```

Replace the 2nd and 4th items of ⍳5 with their reversal

```
      ⌽@2 4 ⍳5
1 4 3 2 5
```

Multiply the 2nd and 4th items of ⍳5 by 10:

```
      10×@2 4⍳5
1 20 3 40 5
```

Replace the 2nd and 4th items by their totals:

```
      +/¨@2 4 ⍳¨⍳5
┌─┬─┬─────┬──┬─────────┐
│1│3│1 2 3│10│1 2 3 4 5│
└─┴─┴─────┴──┴─────────┘
```

Replace the 2nd and 4th rows (major cells) of a matrix with their accumulatives:

```
      (+\@2 4)4 3⍴⍳12
 1  2  3
 4  9 15
 7  8  9
10 21 33
```

Examples (array @ function)

Replace odd elements with 0:

```
      0@(2∘|)⍳5
0 2 0 4 0
```

Replace multiples of 3 (note that masked items are substituted in ravel order):

```
      'abcde'@(0=3|⊢) 4 4⍴⍳16
 1  2  a  4
 5  b  7  8
 c 10 11  d
13 14  e 16
```

Examples (function @ function)

Replace odd elements with their reciprocals:

```
      ÷@(2∘|)⍳5
1 2 0.3333333333 4 0.2
```

Replace odd items of `⍳5` with themselves reversed:

```
      ⌽@(2∘|)⍳5
5 2 3 4 1
```

Stencil R←(f⌺g)Y

Classic Edition: the symbol ⌺ is not available in Classic Edition, and the Stencil operator is instead represented by ⎕U233a.

Stencil is used in image processing, artificial neural networks, computational fluid dynamics, cellular automata, and many other fields of application. The computation is sometimes referred to as tessellation, moving window, or stencil code[1]. This operator applies the left operand function f to a series of (possibly overlapping) rectangles in the array Y.

In general, the right operand g is a 2- row matrix of positive non-zero integers with up to ⍴⍴Y columns. The first row contains the rectangle sizes, the second row the *movements* i.e. how much to move the rectangle in each step. If g is a scalar or vector it specifies the rectangle size and the movement defaults to 1.

The predominant case uses a rectangle size which is odd and a movement of 1.

Rectangles are *centred* on successive elements of Y and (unless the rectangle size is 1), padded with fill elements.

The first rectangle is centred on the first element of Y preceded by the appropriate number of fill elements. Subsequent rectangles are centred on subsequent elements of Y according to the size of the movement, and padded before or after as appropriate. When the movement is 1, each element of Y in its turn is the middle of a rectangle.

f is invoked dyadically with a vector left argument indicating for each axis the number of fill elements and on what side; positive values mean that the padding precedes the array values, negative values mean that the padding follows the array values.

[1]See https://en.wikipedia.org/wiki/Stencil_code

Example

```
      {⊂α ω}⌺3 3⊢3 3⍴⍳12
```

```
┌─────────┬─────────┬─────────┐
│ 1 1 │0 0 0│ │ 1 0 │0 0 0│ │ 1 ¯1 │0 0 0│
│     │0 1 2│ │     │1 2 3│ │      │2 3 0│
│     │0 4 5│ │     │4 5 6│ │      │5 6 0│
├─────────┼─────────┼─────────┤
│ 0 1 │0 1 2│ │ 0 0 │1 2 3│ │ 0 ¯1 │2 3 0│
│     │0 4 5│ │     │4 5 6│ │      │5 6 0│
│     │0 7 8│ │     │7 8 9│ │      │8 9 0│
├─────────┼─────────┼─────────┤
│ ¯1 1 │0 4 5│ │ ¯1 0 │4 5 6│ │ ¯1 ¯1 │5 6 0│
│      │0 7 8│ │      │7 8 9│ │       │8 9 0│
│      │0 0 0│ │      │0 0 0│ │       │0 0 0│
└─────────┴─────────┴─────────┘
```

```
      {+/,ω}⌺3 3⊢3 3⍴⍳12
12 21 16
27 45 33
24 39 28
```

In the first expression above, the left operand function `{⊂α ω}` simply displays its left and right arguments to illustrate the mechanics of the operation. The right operand `(3 3)` specifies that each rectangle contains 3 rows and 3 columns, and the movement is 1.

In order for the first element of `Y` (1) to be centred, the first rectangle is padded with a row above and a column to the left, as indicated by the left argument `(1 1)` to the function.

Another way to think about the way Stencil operates is that it portions the array into sections or neighbourhoods in which elements can be analysed with respect to their immediate neighbours. Stencil has uses in image processing applications.

Examples

```
      {⊂α ω}⌹(3 3,[.5]2)⊢3 3⍴⍳12
```

```
┌──────────────────────────────────────┬──────────────────────────────────────┐
│┌──────┬──────┐                       │┌──────┬──────┐                       │
││1 1   │0 0 0 │                       ││1 ¯1  │0 0 0 │                       │
││      │0 1 2 │                       ││      │2 3 0 │                       │
││      │0 4 5 │                       ││      │5 6 0 │                       │
│└──────┴──────┘                       │└──────┴──────┘                       │
├──────────────────────────────────────┼──────────────────────────────────────┤
│┌──────┬──────┐                       │┌──────┬──────┐                       │
││¯1 1  │0 4 5 │                       ││¯1 ¯1 │5 6 0 │                       │
││      │0 7 8 │                       ││      │8 9 0 │                       │
││      │0 0 0 │                       ││      │0 0 0 │                       │
│└──────┴──────┘                       │└──────┴──────┘                       │
└──────────────────────────────────────┴──────────────────────────────────────┘
```

```
      {⊂α ω}⌹(3 3,[.5]3)⊢3 3⍴⍳12
```

```
┌──────────────────────────────────────┐
│┌──────┬──────┐                       │
││1 1   │0 0 0 │                       │
││      │0 1 2 │                       │
││      │0 4 5 │                       │
│└──────┴──────┘                       │
└──────────────────────────────────────┘
```

```
      ⊢ A←5 5⍴0 0 1 0 0, 0 1 2 1 0, 1 2 3 2 1, 0 1 2 1 0
0 0 1 0 0
0 1 2 1 0
1 2 3 2 1
0 1 2 1 0
0 0 1 0 0

      ⊢ y←1=?10 10⍴4
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 1 0 0 0 0
1 0 0 0 1 1 0 0 0 1
1 1 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
1 0 1 0 0 1 1 0 0 1
0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 0 0
1 0 0 0 1 1 0 1 1 0

      {+/,A×⍵}⌺(⍴A) ⊢y
0 0 1 0 0 1 0 1 2 3
1 1 2 1 2 3 1 0 1 3
4 4 3 4 6 6 3 1 1 3
6 6 5 4 7 7 4 2 2 3
8 6 5 3 5 6 2 0 1 3
6 5 4 3 5 6 5 2 1 3
5 5 4 4 6 7 8 7 4 3
3 2 2 1 4 7 8 7 5 3
3 1 1 1 3 5 6 6 4 2
3 2 2 3 5 6 7 7 5 3
```

You can see that the result identifies where there are clusters in `y`.

Examples (odd rectangle, movement not 1)

If the movement is greater than one, corresponding portions are skipped as shown below.

```
      {⊂ω}⌹(¯3 2) ⍳8
┌─────┬─────┬─────┬─────┐
│0 1 2│2 3 4│4 5 6│6 7 8│
└─────┴─────┴─────┴─────┘
      {(¯2↑⍕α),' f ',⍕ω}⌹(¯3 2) ⍳8
 1 f 0 1 2
 0 f 2 3 4
 0 f 4 5 6
 0 f 6 7 8
⍝       ↑ middle
      {⊂ω}⌹(¯5 2) ⍳9
┌─────┬─────┬─────┬─────┬─────┐
│0 0 1 2 3│1 2 3 4 5│3 4 5 6 7│5 6 7 8 9│7 8 9 0 0│
└─────┴─────┴─────┴─────┴─────┘
      {(¯2↑⍕α),' f ',⍕ω}⌹(¯5 2) ⍳9
 2 f 0 0 1 2 3
 0 f 1 2 3 4 5
 0 f 3 4 5 6 7
 0 f 5 6 7 8 9
¯2 f 7 8 9 0 0
⍝         ↑ middle
```

Even Rectangle Size

For even rectangle sizes, the "middle" consists of two elements which are moved according to the movement parameter (equal to 1 in these examples).

Examples

```
      ⎕←s←{⊂⍵}⌺ 2 ⍳8
┌───┬───┬───┬───┬───┬───┬───┐
│1 2│2 3│3 4│4 5│5 6│6 7│7 8│
└───┴───┴───┴───┴───┴───┴───┘

      {(¯2↑⍕⍺),' f ',⍕⍵}⌺ 2⍳8
 0 f 1 2
 0 f 2 3
 0 f 3 4
 0 f 4 5
 0 f 5 6
 0 f 6 7
 0 f 7 8
⍝    ↑ ↑ middle
      ⎕←s←{⊂⍵}⌺ 4⍳8
┌───────┬───────┬───────┬───────┬───────┬───────┬───────┐
│0 1 2 3│1 2 3 4│2 3 4 5│3 4 5 6│4 5 6 7│5 6 7 8│6 7 8 0│
└───────┴───────┴───────┴───────┴───────┴───────┴───────┘
      ⍴s
7
      {(¯2↑⍕⍺),' f ',⍕⍵}⌺ 4⍳8
 1 f 0 1 2 3
 0 f 1 2 3 4
 0 f 2 3 4 5
 0 f 3 4 5 6
 0 f 4 5 6 7
 0 f 5 6 7 8
¯1 f 6 7 8 0
⍝      ↑ ↑ middle
```

Examples (even rectangle, movement not 1)

```
      {⊂ω}⌹(¯4 2) ⍳8
┌─────────┬─────────┬─────────┬─────────┐
│0 1 2 3  │2 3 4 5  │4 5 6 7  │6 7 8 0  │
└─────────┴─────────┴─────────┴─────────┘
      {(¯2↑⍕⍺),' f ',⍕ω}⌹(¯4 2) ⍳8
 1 f 0 1 2 3
 0 f 2 3 4 5
 0 f 4 5 6 7
¯1 f 6 7 8 0
⍝       ↑ ↑ middle
      {⊂ω}⌹(¯6 2) ⍳8
┌─────────────┬─────────────┬─────────────┬─────────────┐
│0 0 1 2 3 4  │1 2 3 4 5 6  │3 4 5 6 7 8  │5 6 7 8 0 0  │
└─────────────┴─────────────┴─────────────┴─────────────┘
      {(⍕⍺),' f ',⍕ω}⌹(¯6 2) ⍳8
2 f 0 0 1 2 3 4
0 f 1 2 3 4 5 6
0 f 3 4 5 6 7 8
¯2 f 5 6 7 8 0 0
⍝         ↑ ↑ middle
```

Comma Separated Values {R}←{X} ⎕CSV Y

This function imports and exports Comma Separated Value (CSV) data.

Monadic ⎕CSV imports data from a CSV file or converts data from CSV format to an internal format. Dyadic ⎕CSV exports data to a CSV file or converts data from internal format to a CSV format.

Internal Format

Arrays that result from importing CSV data or arrays that are suitable for exporting as CSV data are represented by 3 possible structures:

- A table (a matrix whose elements are character vectors or scalars, or numbers).
- A vector, each of whose items contain field (column) values. Character field values are character matrices; numeric field values are numeric vectors.
- A vector, each of whose items contain field (column) values. Character field values are vectors of character vectors; numeric field values are numeric vectors.

Note that when importing CSV data, all fields are assumed to be character fields unless otherwise specified (see *Column Types* below). A field that contains only "numbers" will not be converted to numeric data unless specified as being numeric.

MetaCharacters

Some characters in a CSV file are metacharacters which define the structure of the data; for example, the field separator character between fields. Characters which are not metacharacters are literal characters. The variant options **QuoteChar**, **EscapeChar** and **DoubleQuote** make it possible to interpret metacharacters as literal characters and thus permit fields to contain field separator characters, leading and trailing spaces, and line-endings.

Fixed-width fields do not require these options and they are ignored if fixed-width fields are being processed.

Monadic ⎕CSV

```
R←⎕CSV Y
```

Y is an array that specifies just the source of the CSV data (see below) or a 1,2,3 or 4-element vector containing:

[1]	Source of CSV Data
[2]	Description of the CSV data
[3]	Column Types
[4]	Header Row Indicator

Source may be one of:

- a character vector or scalar containing a file name
- a native tie number
- a character vector or scalar containing CSV data with embedded newline characters. To avoid this source being interpreted as a file name, Y[2] must be specified as 'S'.
- a vector of character vectors and/or scalars containing CSV data with implicit newlines after each character vector or scalar

Description

If Y[1] is a file name or tie number *Description* may be one of:

- a character vector specifying the file encoding such as 'UTF-8'.
- a 256-element numeric vector that maps each possible byte value (0-255) to a Unicode code point (1st element = Unicode code point corresponding to byte value 0, and so on). ¯1 indicates that the corresponding byte value is not mapped to any character. Apart from ¯1, no value may appear in the table more than once.

If omitted or empty, the file encoding is deduced (see below).

If Y[1] is a character array containing CSV data *Description* is a character scalar 'S' (simple) or 'N' (nested). The default is 'N'

Column Types

This is a scalar numeric code or vector of numeric codes that specifies the field types from the list below. If *Column Types* is zilde or omitted, the default is 1 (all fields are character).

0	The field is ignored.
1	The field contains character data.
2	The field is to be interpreted as being numeric. Empty cells and cells which cannot be converted to numeric values are not tolerated and cause an error to be signalled.
3	The field is to be interpreted as being numeric but invalid numeric vales are tolerated. Empty fields and fields which cannot be converted to numeric values are replaced with the Fill variant option (default 0).
4	The field is to be interpreted numeric data but invalid numeric data is tolerated. Empty fields and fields which cannot be converted to numeric values are returned instead as character data; this type is disallowed when variant option Invert is set to 1.
5	The field is to be interpreted as being numeric but empty fields are tolerated and are replaced with the Fill variant option (default 0). Non-empty cells which cannot be converted to numeric values are not tolerated and cause an error to be signalled.

Note that if *Column Types* is specified by a scalar 4, all numeric data in all fields will be converted to numbers.

Header Row Indicator

This is a Boolean value (default 0) to specify whether or not the first record in a CSV file is a list of column labels. If *Header Row Indicator* is 1, the first record (the *header row*) is treated differently from other records. It is assumed to contain character data (labels) regardless of `Y[3]` and is returned separately in the result.

Variant options

Monadic ⎕CSV may be applied using the Variant operator with the following options. The Principal option is **Invert**.

Name	Meaning	Default
Invert	0, 1 or 2 (see below)	0
Separator	The field separator, any single character. If Widths is other than ⍬, Separator is ignored.	','
Widths	A vector of numeric values describing the width (in characters) of the corresponding columns in the CSV source, or ⍬ for variable width delimited fields	⍬
Decimal	The decimal mark in numeric fields - one of '.' or ','	'.'
Thousands	The thousands separator in numeric fields, which may be specified as an empty character vector (meaning no separator is defined) or a character scalar	''
Trim	A Boolean specifying whether undelimited/unescaped whitespace is trimmed at the beginning and end of fields	1
Ragged	A Boolean specifying whether records with varying numbers of fields are allowed; see notes below	0
Fill	The numeric value substituted for invalid numeric data in columns of type 3	0
Records	The maximum number of records to process or 0 for no limit. This applies only to a file specified by a tie number.	0
QuoteChar	The field quote character (delimiter), which may be specified as an empty character vector (meaning none is defined) or a character scalar	"
EscapeChar	The escape character, which may be specified as an empty character vector (meaning none is defined) or a character scalar	''
DoubleQuote	A Boolean which indicates whether (1) or not (0) a quote character within a quoted field is represented by two consecutive quote characters	1

The **Separator**, **QuoteChar** and **EscapeChar** characters, when defined, must be different.

Other options defined for export are also accepted but ignored.

Invert Option

This option specifies how the CSV data should be returned as follows:

0	A table (a matrix whose elements are character vectors or scalars or numbers).
1	A vector, each of whose items contain field (column) values. Character field values are character matrices; numeric field values are numeric vectors.
2	A vector, each of whose items contain field (column) values. Character field values are vectors of character vectors; numeric field values are numeric vectors.

QuoteChar, EscapeChar and DoubleQuote Options

If **EscapeChar** is set then any character may be prefixed by the escape character. The escape character is typically defined as `'\'`. The escape character immediately followed by the character `c` is the literal character `c` even if `c` alone would have been a metacharacter.

If **QuoteChar** is set then fields may be delimited by the specified quote character. Within quoted fields all characters except the quote character, and the escape character if defined, are literal characters.

If **DoubleQuote** is set to 1 then two consecutive quote characters within a quoted field are interpreted as the single literal quote character.

Result

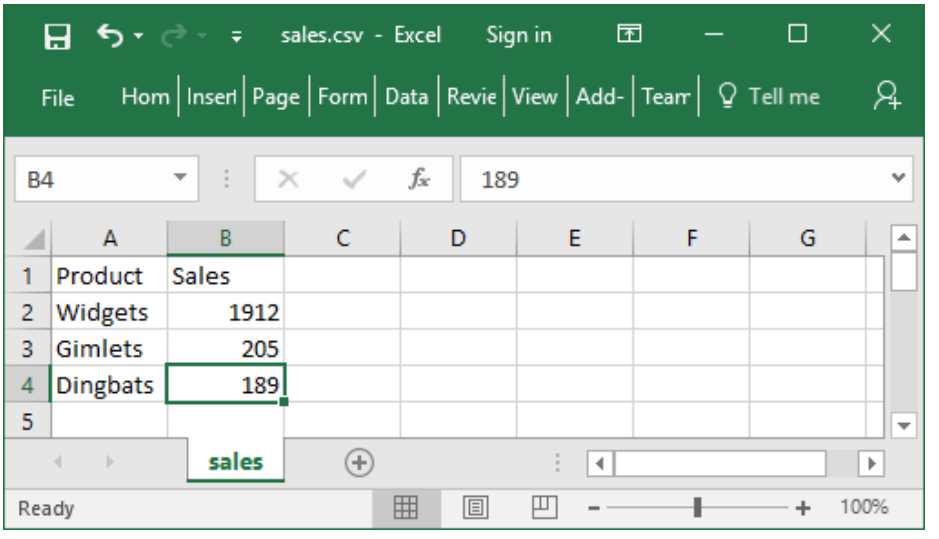
The result `R` contains the imported data.

If `Y[4]` does not specify that the data contains a header then `R` contains the entire data in the form specified by the Invert variant option.

If `Y[4]` does specify that the data contains a header then `R` is a 2-element vector where:

- `R[1]` is the imported data excluding the header.
- `R[2]` is a vector of character vectors containing the header record.

Examples

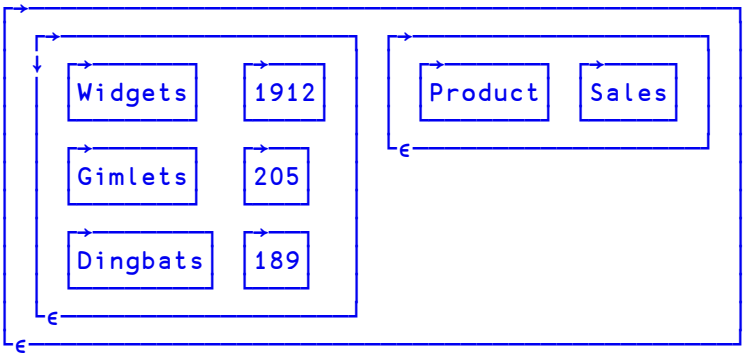


	A	B	C	D	E	F	G
1	Product	Sales					
2	Widgets	1912					
3	Gimlets	205					
4	Dingbats	189					
5							

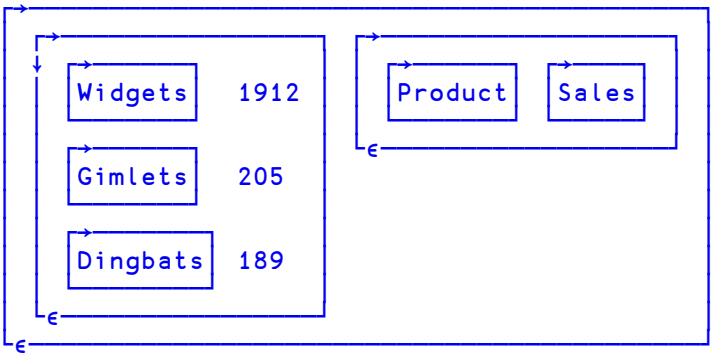
```
      ⊃⎕NGET CSVFile←'c:\Dyalog16.0\sales.csv'
┌→───────────────────────────────────────────────┐
│Product,Sales                                   │
│             Widgets,1912                       │
│                         Gimlets,205            │
│                                    Dingbats,189│
│                                                │
└────────────────────────────────────────────────┘
```

```
      ⎕CSV CSVFile
┌→───────────────────┐
↓ ┌→──────┐  ┌→────┐ │
│ │Product│  │Sales│ │
│ └───────┘  └─────┘ │
│ ┌→──────┐  ┌→───┐  │
│ │Widgets│  │1912│  │
│ └───────┘  └────┘  │
│ ┌→──────┐  ┌→──┐   │
│ │Gimlets│  │205│   │
│ └───────┘  └───┘   │
│ ┌→───────┐ ┌→──┐   │
│ │Dingbats│ │189│   │
│ └────────┘ └───┘   │
└∊───────────────────┘
```

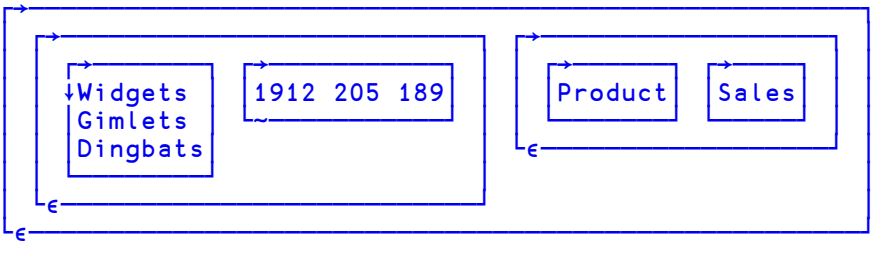
```
⎕CSV CSVFile'' ⍬ 1 ⍝ Header row
```




```
⎕CSV CSVFile''(1 2)1 ⍝ Fields are Char,Num
```




```
(⎕CSV⍠'Invert' 1)CSVFile'' (1 2) 1    ⍝ Invert 1
```




```
⊃(⎕CSV⍠'Invert' 2)CSVFile'' (1 2) 1    ⍝ Invert 2
```

Notes

- When `Y` specifies just the source of the CSV data, it does not need to be enclosed or ravelled to create a 1-element vector.
- `Y[2]`, the description of the source, distinguishes an otherwise ambiguous character vector source (which could contain either CSV data or a file name). The other source forms are unambiguous but the description, when given, must still match the given source type.
- Tab-separated fields may be imported by specifying `'Separator' (⎕UCS 9)`.
- Fields containing embedded new lines are supported (they must, of course, appear in quotes or be prefixed by the escape character). On import, line endings are always converted to a single line feed character.
- If **Ragged** is not set then all records must have the same number of fields (character delimited format) or same number of characters (fixed width field format).
- If **Ragged** is set:
  - The expected number of columns must be specified using the **Widths** variant option and/or the column types in `Y[3]`.
  - In character delimited format, all processed records are implicitly extended or truncated as required so that they contain the expected number of fields; implicitly added fields will be empty.
  - In fixed width format, all processed records are implicitly extended with spaces or truncated as required so that they contain as many characters as are specified in the **Widths** option declaration.

File handling

Data may be read from a named file or a tied native file. A tied native file may be read in sections by repeatedly invoking ⎕CSV for a specified maximum number of records (specified by the **Records** variant) until no more data is read.

In all cases the files must contain text using one of the supported encodings. The method used to determine the file encoding is as follows:

- If a Byte Order Mark (BOM) is encountered at the start of the file, it is used regardless of Y[2] (if specified). Note, however, that the BOM can only be encountered if the file is read from the start - specifically, if a native file is read in sections, any BOM present will only be encountered when the first section is read.
- Otherwise, the file will be read and decoded according to the file encoding in Y[2] if specified.
- Otherwise:
  - Native files will be decoded as if 'UTF-8' had been specified.
  - Files specified by name will be examined and the likely file encoding will be deduced using the same heuristics performed by ⎕NGET.

Note also:

- Native files are read from the current file position. On successful completion, the file position will be at the first unprocessed character (end of file if the **Records** variant option is not specified). If an error is signalled the file position is undefined.
- The result does not report the file encoding or line ending type as it does with ⎕NGET. If this information is required then it must be obtained by other means.

Dyadic ⎕CSV

```
{R}←X ⎕CSV Y
```

The left argument X is either:

- a matrix or a vector of vectors/matrices containing the data to be converted to CSV format.
- or a 2-element vector containing a matrix or vector of vectors/matrices containing the data to be converted to CSV format, and a vector of character vectors containing the header record.

Y is a 1 or 2-element vector containing:

[1]	Destination of CSV Data (see below)
[2]	Description of the CSV data (see below)

Destination - may be one of:

- a character vector or scalar containing a file name
- a native tie number
- an empty character vector, indicating that the CSV data is to be returned in the result R

Description

If Y[1] is a file name or tie number, *Description* may be:

- a character vector specifying the file encoding such as 'UTF-8'.
- a 256-element numeric vector that maps each possible byte value (0-255) to a Unicode code point (1st element = Unicode code point corresponding to byte value 0, and so on). ¯1 indicates that the corresponding byte value is not mapped to any character. Apart from ¯1, no value may appear in the table more than once.

If Y[1] is empty, *Description* may be a character scalar 'S' (simple) or 'N' (nested). If omitted, the default is 'S'

Variant options

Dyadic ⎕CSV may be applied using the Variant operator with the following options.

Name	Meaning	Default
IfExists	a character vector 'Error' or 'Replace' which specifies, when creating a named file which already exists, whether to overwrite it ('Replace') or signal an error ('Error')	'Error'
Separator	the field separator, any single character. If Widths is other than ⍬, Separator is ignored.	','
Widths	a vector of numeric values describing the width (in characters) of the corresponding columns in the CSV source, or ⍬ for variable width delimited fields	⍬
Decimal	the decimal mark in numeric fields - one of '.' or ','	'.'
Thousands	the thousands separator in numeric fields, which may be specified as an empty character vector (meaning no separator is defined) or a character scalar	''
Trim	a Boolean specifying whether whitespace is trimmed at the beginning and end of character fields	1
LineEnding	the line ending sequence	(13 10) on Windows; 10 on other platforms
QuoteChar	The field quote character (delimiter), which may be specified as an empty character vector (meaning none is defined) or a character scalar	"
EscapeChar	The escape character, which may be specified as an empty character vector (meaning none is defined) or a character scalar	''
DoubleQuote	A Boolean which indicates whether (1) or not (0) a quote character within a quoted field is represented by two consecutive quote characters	1

The **Separator**, **QuoteChar** and **EscapeChar** characters, when defined, must be different. Other options defined for import are also accepted but ignored.

The **Overwrite** variant option (Boolean) from Version 16.0 remains supported but is deprecated in favour of **IfExists**.

QuoteChar, EscapeChar and DoubleQuote options

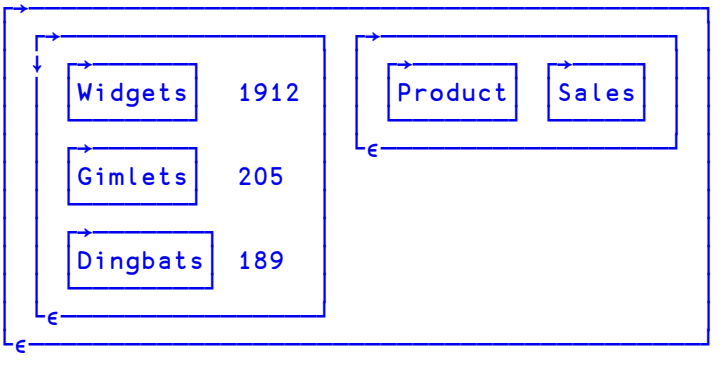
- The CSV text will be generated such that it can be read back according to the corresponding rules for import.
- If these options do not permit this (for example, a field contains the quote character and neither **DoubleQuote** or **EscapeChar** are set) an error is signalled.
- Quoting and Escaping is used as conservatively as possible.
- If both **QuoteChar** and **EscapeChar** are set, quoting is favoured.

If `Y` specifies that the CSV data is written to a file then `R` is the number of bytes (not characters) written, and is shy.

Otherwise, `R` is the CSV data in the format specified in Y, and is not shy.

Examples

```
      CSVFile←'c:\Dyalog16.0\sales.csv'
      ⎕←DATA HDR←⎕CSV CSVFile''(1 2)1
```

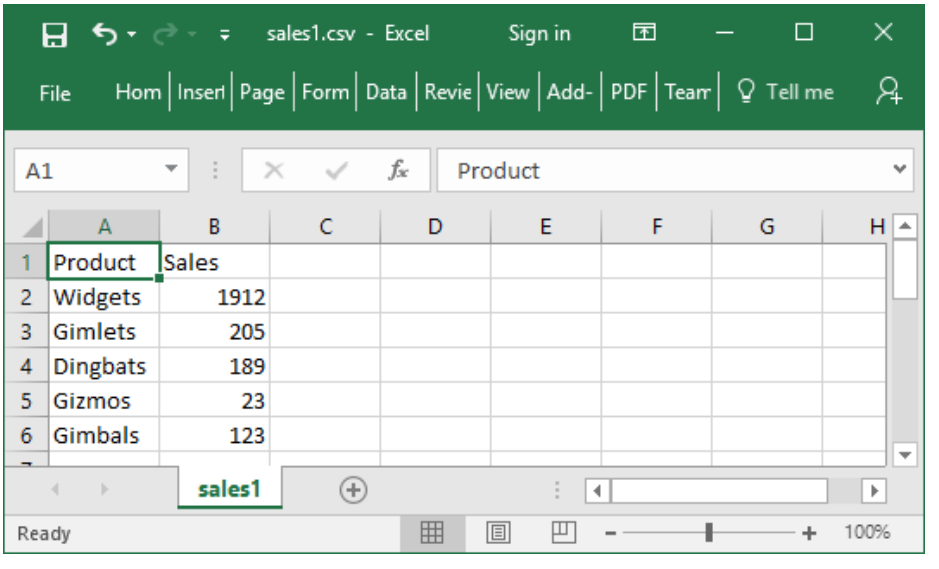


```
      DATA,←'Gizmos' 23
       DATA HDR ⎕CSV''
┌→──────────────────────┐
│Product,Sales          │
│                       │
│Widgets,1912           │
│                       │
│Gimlets,205            │
│                       │
│Dingbats,189           │
│                       │
│Gizmos,23              │
│                       │
└───────────────────────┘

       CSVFile1←'c:\Dyalog16.0\sales1.csv'
       ⎕←DATA HDR ⎕CSV CSVFile1

67
       DATA,←'Gimbals' 123
       ⎕←DATA HDR ⎕CSV CSVFile1
FILE NAME ERROR: Unable to create file ("The file
exists.")
       ⎕←DATA HDR ⎕CSV CSVFile1
      ^
       ⎕←DATA HDR(⎕CSV⍠'IfExists' 'Replace')CSVFile1

80
```



Notes

- When `Y` contains only the destination of the CSV data (i.e. omits the description in its second element) it does not have to be enclosed to form a single element vector.
- Native files are written from the current file position. On successful completion, the file position will be at the end of the written data. If an error is signalled the amount of data written is undefined.
- If the file encoding specifies that a BOM is required and output is to a native file, it will only be written if the file position is initially at 0 - that is, the start of the file is being written.
- When fixed width fields are written, character data shorter than the specified width is padded with spaces to the right and character data longer than the specified width signals an error. Numeric data is converted to character data as far as possible so that it fits into the specified width. If this is not possible, an error is signalled.
- Tab-separated fields may be exported by specifying `'Separator' (⎕UCS 9)`.
- Fields containing a single embedded new line are supported. On export, line feed characters are mapped back to the defined line ending sequence.

JSON R←{X}⎕JSON Y

This function imports and exports data in JavaScript Object Notation (JSON) Data Interchange Format[1].

If specified, X must be a numeric scalar with the value 0 (import JSON) or 1 (export JSON). If X is not specified and Y is a character array, X is assumed to be 0 (import); otherwise it is assumed to be 1 (export).

Other options for ⎕JSON are Format and Compact which are specified using the Variant operator ⍠. The Principle Option is Format.

JSON Import (X is 0)

Y is a character vector or matrix in JSON format. There is an implied newline character between each row of a matrix.

The content of the result R depends upon the Format variant which may be 'D' (the default) or 'M'.

If Format is 'D' (which stands for "data") the JSON described by Y is converted to APL object(s) and R is an array or a namespace containing arrays and sub-namespaces.

- JSON objects are created as APL namespaces.
- JSON null is converted to the enclosed character vector ⊂'null'.
- JSON true is converted to the enclosed character vector ⊂'true'
- JSON false is converted to the enclosed character vector ⊂'false'.
- If the JSON source contains object names which are not valid APL names they are converted to APL objects with mangled names. See *JSON Name Mangling* on page 74. 7162⌶ can be used to obtain the original name. See *JSON Translate Name* on page 79.

[1]IETF RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format - is a widely supported, text based data interchange format for the portable representation of structured data; any application which conforms to the standard may exchange data with any other.

If `Format` is `'M'` (which stands for "matrix") the result `R` is a matrix whose columns contain the following:

`[;1]`	depth
`[;2]`	name (for JSON object members)
`[;3]`	value
`[;4]`	JSON type (integer: see below)

- The representation of null, true and false are the same as for `Format 'D'`.
- Object names are reported as specified in the JSON text; they are not mangled as they are for `Format 'D'`.

JSON types are as follows:

Type	Description
1	Object
2	Array
3	Numeric
4	String
5	Null
6	Boolean (true / false)
7	JavaScript Object (export only)

Table 1: JSON data types

Duplicate Names

The JSON standard says that members of a JSON object should have unique names and that different implementations behave differently when there are duplicates. Dyalog handles duplicate names as follows:

- No error is generated
- For `Format 'D'`, the last member encountered is used and all previous members with the same name are discarded
- For `Format 'M'` all duplicate members are recorded in the result matrix

Examples

```
      ⍴JSON
18 19
      JSON
{
  "a": {
    "b": [
      "string 1",
      "string 2"
    ],
    "c": true,
    "d": {
      "e": false,
      "f⍺": [
        "string 3",
        123,
        1000.2,
        null
      ]
    }
  }
}
```

Import as Data (`Format 'D'`)

```
      j←⎕JSON JSON
      j
#.[JSON object]
      j.⎕NL 9
a
      j.a.⎕NL 2
b
c
      j.a.b
┌────────┬────────┐
│string 1│string 2│
└────────┴────────┘
      j.a.c
┌────┐
│true│
└────┘
      j.a.⎕NL 9
d
      j.a.d.⎕NL 2 ⍝ Note that fα is an invalid APL name
e
⍙f⍙9082⍙
      j.a.d.e
┌─────┐
│false│
└─────┘
      j.a.d.⍙f⍙9082⍙
┌────────┬───┬──────┬────┐
│string 3│123│1000.2│┌────┐│
│        │   │      ││null││
│        │   │      │└────┘│
└────────┴───┴──────┴────┘
```

Import as Matrix (`Format 'M'`)

```
(⎕JSON⍠'M')JSON
```

0			1
1	a		1
2	b		2
3		string 1	4
3		string 2	4
2	c	true	6
2	d		1
3	e	false	6
3	fα		2
4		string 3	4
4		123	3
4		1000.2	3
4		null	5

JSON Export (X is 1)

Y is the data to be exported as JSON and may be an array, a namespace or a matrix representation of JSON such as would have been produced by JSON Import with Format 'M'. Y is interpreted according to the Format variant which may be 'D' (the default) or 'M'.

⎕JSON will signal DOMAIN ERROR if Y is incompatible with the specified (or implied) value of Format.

If Format is M, the data values in Y[;3] must correspond precisely with the JSON types specified in Y[;4]as specified in the following table.

Y[;4] (Type)	Y[;3] (Value)
1	Empty array
2	Empty array
3	Numeric scalar
4	Character vector
5	Null
6	Enclosed character vector
7	Enclose character vector

R is a character vector whose content depends upon the value of the Compact variant.

If Compact is 0, the JSON text is padded with spaces and new lines for readability.

If Compact is 1 (the default) the JSON text is compacted into its minimal form.

The name of any namespace member that begins with ⍙ and otherwise conforms to the conversion format used for JSON object names will be demangled.

Example

```
      j                ⍝ See above
#.[JSON object]
      ⍴JS←1 ⎕JSON j
94
      JS
{"a":{"b":["string 1","string 2"],"c":true,"d":
{"e":false,"fα":["string 3",123,1000.2,null]}}}

      1(⎕JSON⍠'Compact' 0) j
{
  "a": {
    "b": [
      "string 1",
      "string 2"
    ],
    "c": true,
    "d": {
      "e": false,
      "fα": [
        "string 3",
        123,
        1000.2,
        null
      ]
    }
  }
}
```

If there are any mis-matches between the values in `Y[;3]` and the types in `Y[;4]`, `⎕JSON` will signal `DOMAIN ERROR` and report the first row where there is a mis-match (`⎕IO` sensitive) as illustrated in the following example.

Example

```
      M←(⎕JSON⍠'Format' 'M')'{"values": [ 75, 300 ]}'
      M
```

0			1
1	values		2
2		75	3
2		300	3

```
      M[3;3]←⊂'75'  ⍝ character not numeric

      M             ⍝ but looks the same as before
```

0			1
1	values		2
2		75	3
2		300	3

```
      1 (⎕JSON⍠ 'Format' 'M')M
DOMAIN ERROR: Value does not match the specified type in
row 3
      1(⎕JSON⍠'Format' 'M')M
     ^
```

Javascript Objects

The following example illustrates how Javascript objects may be exported.

In the example, the object is a Javascript function which is specified by the contents of an enclode character vector. Not that in this case Dyalog performs no validation of the code itself.

Example

```
     'Slider' ⎕NS ''
      Slider.range←⊂'true'          ⍝ Note the ⊂
      Slider.min←0
      Slider.max←500
      Slider.values←75 300

      fn1←' function( event, ui ) {'
      fn2←'$( "#amount" ).val( "$" + ui.values[ 0 ] +'
      fn2,←' " - $" + ui.values[ 1 ] );}'

      Slider.slide←,/fn1 fn2 ⍝ Enclosed character vec

      ⍴JS←1 ⎕JSON Slider
159
      JS
{"max":500,"min":0,"range":true,"slide": function( event,
ui ) {$( \"#amount\" ).val( \"$\" + ui.values[ 0 ] + \" -
$\" + ui.values[ 1 ] );},"values":[75,300]}
```

Restrictions and Limitations

The JSON standard describes a limited set of data types and JSON does not provide a general APL import/export mechanism. In particular:

Not all APL arrays are representable in JSON.

For example, arrays with more than one dimension cannot be represented in JSON. Of course, this does mean that applications using JSON are unlikely to use such objects; you probably will need rearrange your data into the format that is expected by the receiving application. In the case of a 2-dimensional matrix, a split will give you a vector of tuples that a JSON application is likely to expect:

```
      ⎕JSON 3 4⍴⍳12
DOMAIN ERROR: Array unsupported by JSON
      ⎕JSON 3 4⍴⍳12
     ^
      ⎕JSON ↓3 4⍴⍳12
[[1,2,3,4],[5,6,7,8],[9,10,11,12]]
```

Not all JSON types have exact APL equivalents

The JSON standard includes Boolean values true and false which are distinct from numeric values 1 and 0, and have no direct APL equivalent.

To represent JSON true and false types this implementation adopts the convention of using APL arrays `⊂'true'` and `⊂'false'` respectively. These arrays cannot otherwise be represented in JSON and allow true and false to be uniquely identified.

Not all names are valid APL names.

The names of JSON object members which would not be valid for APL are modified. See *JSON Name Mangling* below.

JSON Name Mangling

When Dyalog converts from JSON to APL data, and a member of a JSON object has a name which is not a valid APL name, it is renamed.

Example:

In this example, the JSON describes an object containing two numeric items, one named *a* (which is a valid APL name) and the other named *2a* (which is not):

```
{"a": 1, "2a": 2}
```

When this JSON is imported as an APL namespace using ⎕JSON, Dyalog converts the name *2a* to a valid APL name. The *name mangling* algorithm creates a name beginning with ⍙.

```
      (⎕JSON'{"a": 1, "2a": 2}').⎕NL 2
a
⍙2a
```

When Dyalog exports JSON it performs the reverse *name mangling*, so:

```
      1 ⎕JSON ⎕JSON'{"a": 1, "2a": 2}'
{"a":1,"2a":2}
```

Should you need to create and decode these names directly, 7162⌶ provides the same name mangling and un-mangling operations. See *JSON Translate Name* on page 79.

```
      0(7162⌶)'2a'
⍙2a
      1(7162⌶)'⍙2a'
2a
```

System Command Parameters

Certain system commands now accept parameters as shown in the table and examples below.

System Command	Description
`)SAVE -force ws`	**force** causes the specified file to be overwritten when the name specified by `ws` differs from `⎕WSID`.
`)SI n -tid=tn`	`n` restricts the display to the first or last `n` lines of the full output. **-tid=tn** specifies that the state indicator is to be displayed only for thread number `tn`.
`)RESET n`	Clears the top `n` suspensions on the stack.

State Indicator

```
      foo
DOMAIN ERROR: Divide by zero
loo[1] 1÷0
      ^
      )SI
#.loo[1]*
#.hoo[1]
#.goo[1]
#.foo[1]
      )SI 2
#.loo[1]*
#.hoo[1]
      )SI ¯2
#.goo[1]
#.foo[1]

       ⎕←foo&¨10 10 10 10
┌→──────────┐
│9 10 11 12 │
└~──────────┘

       )si
·   #.foo[1]
&9
·   #.foo[1]
&10
·   #.foo[1]
&11
·   #.foo[1]
&12
      )si -tid=11
#.foo[1]
```

Reset State Indicator

```
      )reset
      foo
DOMAIN ERROR: Divide by zero
loo[1] 1÷0
      ^
      foo
DOMAIN ERROR: Divide by zero
loo[1] 1÷0
      ^

      )si
#.loo[1]*
#.hoo[1]
#.goo[1]
#.foo[1]
#.loo[1]*
#.hoo[1]
#.goo[1]
#.foo[1]

      )reset 3

      )si
#.foo[1]*
#.loo[1]*
#.hoo[1]
#.goo[1]
#.foo[1]
```

Chapter 4:

I-Beam Reference Changes

I-beam Changes

I-beam functionality changed from Version 16.0.

A	Description	Change
2704	See *Continue Autosave* on page 78.	New function
16807	Random Number Generator. See *Withdrawal of 16807I* on page 11.	Removed
7159	Replaced by JSON. See *JSON* on page 65.	Deprecated
7160	Replaced by JSON. See *JSON* on page 65.	Deprecated
7161	JSON TrueFalse	Deprecated
7162	See *JSON Translate Name* on page 79.	Modified

Continue Autosave {R}←2704⌶Y

This function enables or disables the automatic saving of a CONTINUE workspace when Dyalog exits. By default this is disabled when Dyalog starts and must be explicitly enabled using this function.

Y is an integer defined as follows:

Value	Description
0	Disable the automatic saving of a CONTINUE workspace.
1	Enable the automatic saving of a CONTINUE workspace. This setting applies only to the current session or until disabled by 2704⌶0.

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

Circumstances when Dyalog automatically saves a CONTINUE workspace include:

- a run-time violation. This is most frequently caused by an untrapped APL error which causes Dyalog to return to session-input mode (i.e. an application programming fault).
- a hang-up signal.

JSON Translate Name R←X(7162⌶)Y

Converts between JSON names and APL names.

When ⎕JSON imports an entity from JSON text whose name would be an invalid APL name, the function converts the invalid name into a valid APL name using a *name mangling* algorithm. When ⎕JSON exports an APL namespace as JSON text, the process is reversed.

This function performs the same *name mangling* allowing the programmer to identify JSON entities as APL names, and vice-versa.

Y is a character vector or scalar.

X is a scalar numeric value which must be 1 or 0.

When X is 0, R is the name in Y converted, if necessary, so that it is a valid APL name. It performs the same translation of JSON object names to APL names that is performed when importing JSON.

When X is 1, R is the name in Y which, if mangled, is converted back to the original form.. It performs the same translation of APL names to JSON object names that is performed when exporting JSON.

Examples:

```
      0(7162⌶)'2a'
⍙2a
      1(7162⌶)'⍙2a'
2a

      0(7162⌶)'foo'
foo
      1(7162⌶)'foo'
foo
```

Note that the algorithm can be applied, even when mangling is not required. So:

```
      1(7162⌶)'⍙97'
a
```

For further details, see *JSON Name Mangling* on page 74.

Chapter 5:

Object Reference Changes

GUI Enhancements

The following table summarises the main changes to GUI features in Version 16.0.

Name	**Change**
DevCaps	Now returns 4 elements rather than 3
HasClearButton	New property

DevCaps Property

Applies To: Printer, Root

Description

This property reports the device capabilities of the screen or printer. It is a 4-element nested vector as follows:

[1]	Height and Width:2-element numeric vector of device in pixels
[2]	Height and Width:2-element numeric vector of device in mm
[3]	Number of colours or ¯1
[4]	Windows scaling factor as a percentage (100=no scaling). This value is the same as reported in the Display section of the Windows Control Panel

This property is useful if you want to make objects of a specific physical size. For example, to draw a 10mm square in a Form 'F' at (5,5):

```
Size ← 10× ⊃÷/2↑'.' ⎕WG 'DevCaps'
'F.R' ⎕WC 'Rect' (5 5) Size ('Coord' 'Pixel')
```

Notes

- the physical size reported for the screen is typically only a *nominal* size, because, if you use a generic video driver, Windows has no way to tell what size of screen is attached to your computer.
- The number of colours is reported only if the device has a colour depth of no more than 8 bits per pixel. For devices with greater colour depths, ¯1 is returned.
- new elements may be added to DevCaps in future releases.

HasClearButton Property

Applies To: ButtonEdit, Combo, ComboEx, Edit

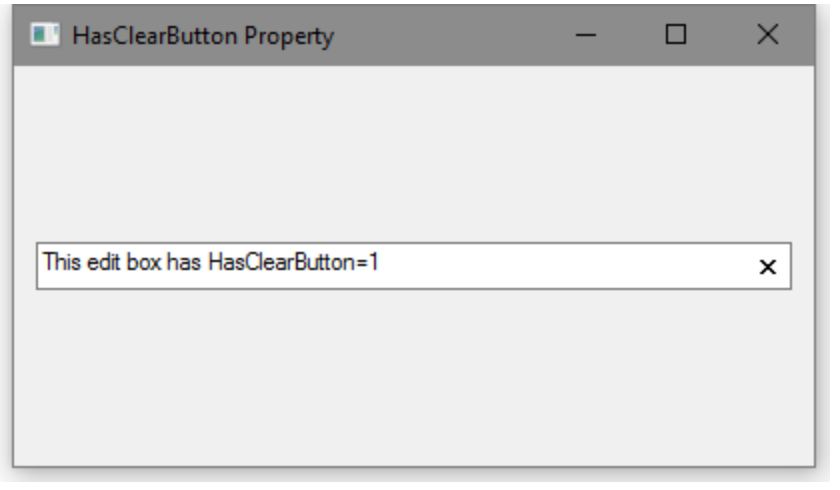
Description

Specifies whether or not an ✕ button is displayed in the right-hand end of an edit box. Clicking this button clears the text from the field.

Note that this feature only applies if Native Look and Feel **(see page 1)** **is enabled.**

HasClearButton is Boolean. 1 means that an ✕ button will be displayed; 0 (the default) means that the button will not be shown. It may only be specified when the object is created. If you subsequently attempt to change the value of HasClearButton, the operation will fail with `NONCE ERROR`.

HasClearButton is only effective for Edit objects with Style Single; it is silently ignored for other Styles of Edit objects.



Chapter 6:

Non-Windows Specific Features

Summary

This section summarises the changes specific to Dyalog APL Version 17.0 on non-Windows platforms. This list currently consists of:

- AIX
- Linux (including the Raspberry Pi)
- macOS/ Mac OS X

Hardware Requirements

AIX

For AIX, Version 17.0 requires AIX 7.2 or higher, and a POWER7 chip or higher.

Raspberry Pi

On the Raspberry Pi, Dyalog 32-bit Unicode supports Raspbian Jessie or later.

Non-Pi Linux

For non-Pi Linux, Version 17.0 only exists as 64-bit interpreters - there are no 32-bit versions. It is built on Debian 7, and QAed on RedHat 6; it runs on all recent distributions, including Ubuntu 14.04 and openSUSE Leap 42.3. Contact Dyalog for information about other distributions.

macOS/Mac OS X

Version 17.0 requires Mac OS X Yosemite or El Capitan or macOS Sierra or later. The target Mac must have been introduced in 2010 or later.

RIDE and Dyalog APL 17.0

Dyalog Version 17.0 supports RIDE 3 and RIDE 4 only; RIDE 2 is not supported. Dyalog recommends that RIDE 4 is used in preference to RIDE 3. RIDE 4 can be used with Version 15.0 too.

RIDE 4 is supported on Raspberry Pi models 2 and 3 only; models Zero and 1 are not supported (the underlying libraries which RIDE is build on are not available for the Pi Zero and 1). The *Dyalog RIDE Reference Guide* details how to configure the APL session to support the underscored alphabet; contact support@dyalog.com if you wish to be able to generate key-chords which result in the underscored alphabet being entered into APL.

Note that on Linux and Pi, if RIDE 4 is installed after Dyalog 16.0 an extra icon will be added to the window manager's start menu which will start Dyalog with a RIDE front end.

Linux Window Managers and APL characters

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

```
$ dyalog -kbd
```

Subsequent invocations of dyalog should not require this flag.

Location of configuration and log files

In Dyalog 16.0 the location of various configuration and log files has been changed so that they are all put in one directory. See the *UNIX Installation and Configuration Guide* for more information.

SQAPL on macOS

Dyalog 16.0 for macOS includes support for SQAPL. However, it is necessary to install iODBC and suitable drivers for your database before SQAPL can work. *The SQL Interface Guide* describes the steps that are typically necessary to get SQAPL connected to a MySQL database.

4000⌶ and 4002⌶

4000⌶ (Fork process) and 4002⌶ (Reap processes) have been withdrawn on all platforms except AIX. This is due to limitations imposed by the HTMLRenderer, and due to problems in the interaction of forking processes and using RIDE.

Index