DYALOG

The tool of thought for expert programming

Dyalog™ for Windows

Release Notes

Version: 13.1

Dyalog Limited

email: support@dyalog.com

http://www.dyalog.com

Version: 13.1

Revision: 22185

Contents

Chapter 1:

Introduction

New Documentation

Previously, Dyalog maintained all product documentation in various different formats, including Microsoft Word and HTML, depending upon the target publication media. The duplication of source material lead to inconsistencies in the different forms of publication.

Dyalog has now invested in a single-source solution for the documentation and has converted the bulk of the existing documentation to the new platform from which the Version 13.1 documentation has been produced.

The new platform will enable Dyalog to improve the quality of the documentation and the range of formats in which it can be published. It will also allow the documentation to be more easily maintained and to be updated in a more timely fashion.

Due to the nature of the conversion process, there are currently a number of typographical errors, particularly formatting errors, which Dyalog is still working to correct. For this reason, the Version 13.1 manuals may not yet be ordered in *perfect bound* printed form from Lulu.

Key Features

Dyalog APL Version 13.1 provides the following enhancements and changes:

Enhancements

- New system function ⎕FHIST
- New system constant ⎕DMX
- New Random Number Generators
- Performance Improvements

Other Changes

- New defaults for component file properties
- Large Component Support
- Compatibility checking
- PrintList property
- Miscellaneous

New System Function ⎕FHIST

The new File History system function provides information about certain file events. This includes information about which user created the file and when, which user last tied the file and when, and which user updated the file and when.

This feature has been added to support users migrating from the SHAREFILE/AP system.

Extended Diagnostic Message: R←⎕DMX

Version 13.1 adds significant new error reporting functionality, in the form of a system variable which has been named ⎕DMX. ⎕DMX is designed to make error handling simpler, more complete and more accurate.

- **More Information:** ⎕DMX is a namespace with a number of properties, which provide much more information about errors than did the old variables (in version 13.1, only a few primitives and system functions take advantage of this; full support will evolve over the next several releases). Application code can also provide more error information, as ⎕SIGNAL has been extended accordingly.
- **Thread Safe:** ⎕DMX is local to each APL thread: An error occurring in one thread will not change the value of ⎕DMX in other threads. This is not the case for ⎕DM and ⎕EN.
- **Isolation of Handled Errors:** ⎕DMX is local to the code which is invoked to handle an error, in such a way that successfully handled (trapped) errors will leave no trace of the handled event.
- **Backwards compatibility:** The behavior of the existing variables ⎕DM (Diagnostic Message) and ⎕EN (Event Number) is unchanged. Dyalog recommends that applications which reference the old variables be modified to use ⎕DMX, as soon as this is convenient

More Information

The default output following an error has been changed in Version 13.1, to display most of this information. For example:

```
      'c:\no\such\folder\output.txt' ⎕FCREATE 0
FILE NAME ERROR: Unable to create file ("The system cannot find
the path specified.")
      'c:\no\such\folder\output.txt'⎕FCREATE 0
     ^
```

In addition to the `FILE NAME ERROR` which would also have been displayed by earlier versions, Version 13.1 displays the text `Unable to create file`, which is a more detailed message originating in the file system, and in parentheses `(The system cannot find the file specified.)`, which is a message originating in the underlying operating system.

The menu item *Options|Configure|Help / DMX* (See User Guide) allows you to decide whether this additional information should be displayed in the session (as above), and/or in the status window. Note that `⎕DM` is not affected by these settings; it only contains the original APL error message as in previous releases:

```
      ↑⎕DM
FILE NAME ERROR
      'c:\no\such\folder\output.txt'⎕FCREATE 0
     ^
```

Thread safe copies of the values of `⎕EN` and `⎕DM` are available as the properties `⎕DMX.EN` and `⎕DMX.DM`.

A number of properties within the `⎕DMX` space can be inspected, to extract the additional information. The detailed message is in a property called `Message`:

```
      ⎕DMX.Message
Unable to create file
```

The new messages also have “extended error numbers”, which are unique within each category of errors:

```
      ⎕DMX.(ENX Category Vendor)
10  Component file system  Dyalog
```

At present, `Vendor` will always contain the character vector `'Dyalog'`, but it is envisaged that external components will eventually be able to make use of this mechanism. Finally, if the error was a direct result of a problem reported by the operating system, the `OSError` property will be populated with this information:

```
      ⎕DMX.OSError
1 3  The system cannot find the path specified.
```

Currently the first item is either 0 or 1:

- 0 means that the error number is the value of `errno()`,
- 1 means that the error number is the result of `GetLastError()`.

The second contains the error number and the third element is the corresponding text.

Finally, the `InternalLocation` property identifies the line of the interpreter source code that issued the error, which may be useful for the Dyalog support team when analyzing particularly tricky errors:

```
      ⎕DMX.InternalLocation
qfile1.c  3614
```

Thread Safe

The old system variables `⎕EN` and `⎕DM` have workspace scope, which means that they are shared by all threads. This means that, if two (or more) threads encounter errors in rapid succession, error handling code which references these variables might pick up values which are unrelated to the error being handled. `⎕DMX`, on the other hand, has thread scope: Each thread has its own copy of the variable. As server applications become more common, it is rapidly becoming more important to write code which is "thread safe". Dyalog strongly recommends that all references to `⎕EN` and `⎕DM` in current application code be replaced by references to `⎕DMX.EN` and `⎕DMX.DM`, respectively.

Isolation of Handled Errors

In version 13.1, `⎕DMX` cannot be explicitly localised in the header of a function. However, for all trapped errors, the interpreter creates an environment which effectively makes the current instance of `⎕DMX` local to, and available only for the duration of, the trap-handling code.

In particular `⎕DMX` is localised within:

- Any function which explicitly localises `⎕TRAP`
- The `:Case[List]` or `:Else` clause of a `:Trap` control structure.
- The right hand side of a D-function Error-Guard.

This localisation uses the standard shadow/un-shadow mechanism, so that an existing value of `⎕DMX` will be hidden on entry to the trap-handling code (or localisation of `⎕TRAP`) and restored afterwards. The benefit of the localization strategy is that code which uses error trapping as a standard operating procedure (such as a file utility which traps `FILE NAME ERROR` and creates missing files when required) will not pollute the environment with irrelevant error information.

In particular, it becomes easier to implement tools to handle errors by logging them or displaying additional information to end users, without worrying that trapped errors within the tool code will interfere with current state of the application.

Examples

The effect of automatic localisation of ⎕DMX is illustrated by the following examples. In each case, assume the existence of a "Global" value of ⎕DMX, whose EN field is 0, following a)RESET.

Example[1] :Trap control structure:

```
 ⎕DMX.EN=0        ⍝ Global ⎕DMX visible here
:Trap 11          ⍝ Catch error 11
    ⎕DMX.EN=0     ⍝ Global ⎕DMX visible here
    ÷0            ⍝ Generate error 11
:Else             ⍝ Error-handling code:
    ⎕DMX.EN=11    ⍝ Local ⎕DMX for ÷0 error visible here
:EndTrap          ⍝ ⎕DMX unshadowed after Else-clause
⎕DMX.EN=0         ⍝ Global ⎕DMX visible here
```

Example[2] ⎕TRAP system variable:

```
    ∇ foo;⎕TRAP
[1]  ⍝ ⎕DMX is shadowed and un-shadowed @ the same time as ⎕TRAP
[2]  ⍝ The local value of ⎕DMX is initialised to a copy of the
[3]  ⍝ calling function's value ("pass-through localisation").
[4]  ⍝ This local value will be changed if an error occurs. On
[5]  ⍝ exit from the function, the calling environment's value
[6]  ⍝ will be restored (along with that of ⎕TRAP) as normal.
[7]   ⎕DMX.EN=0           ⍝ passed-through from calling env
[8]   ⎕TRAP←0 'c' '→next' ⍝ sets trap; does not change or
[9]                       ⍝ localise ⎕DMX
[10]  ⎕DMX.EN=0           ⍝ passed-through from calling env
[11]  ⍎'÷0'               ⍝ updates local ⎕DMX; jumps to next:
[12] next: ⎕DMX.EN=11     ⍝ Error number has been set to 11
[13]  ⎕TRAP←0 'c' '→end'  ⍝ sets trap; does not change or
[14]                      ⍝ localise  ⎕DMX
[15]  ⎕DMX.EN=11          ⍝ Error number is 11, as before
[16] end:                 ⍝ on return, calling envt's ⎕DMX.EN
[17]                      ⍝ will be 0
    ∇
```

Example[3] Dfn Error-guard:

```
⎕←⎕DMX.EN=0          ⍝ Global ⎕DMX visible here
{
    ⎕←⎕DMX.EN=0      ⍝ Global ⎕DMX visible here
    11               ⍝ trap DOMAIN
}⍬::{                ⍝ Error :: Guard
    ⎕←⎕DMX.EN=11     ⍝ Local ⎕DMX for ÷0 error visible here
}⍵
⎕←⎕DMX.EN=0          ⍝ Global ⎕DMX visible here
÷0                   ⍝ Generate error 11
```

Example[4] Nested localisation of ⎕DMX

```
⎕DMX.EN=0              ⍝ Global ⎕DMX
:Trap 11               ⍝ Trap DOMAIN ERROR
    :Trap 5            ⍝ Trap LENGTH ERROR
        2 3+4 5 6      ⍝ Generate LENGTH ERROR
    :Else              ⍝ ⎕DMX localised for following clause:
        ⎕DMX.EN=5      ⍝ EN set to LENGTH in local ⎕DMX
        ÷0             ⍝ Generate DOMAIN ERROR
    :End               ⍝
:Else                  ⍝ ⎕DMX localised for following line
    ⎕DMX.EN=11         ⍝ EN set to DOMAIN in local ⎕DMX
:End                   ⍝ ⎕DMX un-shadowed here
⎕DMX.EN=0              ⍝ Global ⎕DMX restored.
```

Clarification

Whenever ⎕TRAP is localised explicitly, ⎕DMX will be localised at the same time. This means only that any subsequent changes to ⎕DMX will be reverted as ⎕TRAP is unshadowed. In particular, the assignment of ⎕TRAP has no effect on ⎕DMX; only its localisation. Explicit localisation means the appearance in the header of a trad-fn or as the argument of ⎕SHADOW. It does not include the implicit localisation, which takes place at :Trap or dfn error-guard setting time. The examples above illustrate these cases.

The timing of the setting of the values within the ⎕DMX mechanism is important:

1. When an error occurs, the values and properties needed to populate ⎕DMX are collected
2. If a trap is in effect, the stack is cut back appropriately.
3. Within the error-handling code, ⎕DMX is populated with the values collected in step 1.

This means that lexically and dynamically nested error handling code will behave "as expected". See example[4] above.

New Random Number Generators

The Version 13.0 random number generator, that is used by Roll and Deal, is based upon the Lehmer linear congruential generator. This has several limitations, most notably that it has a limited value range of `(2*31)`. Mindful of the need to support applications that rely on the current mechanism, and the ability to generate specific repeatable random series using `⎕RL`, Dyalog has decided to provide two additional random number generators in Version 13.1. Both the new algorithms support 64-bit values and both may be considered to be an improvement (in terms of randomness) over the current mechanism. The new mechanisms are:

- Mersenne Twister random number generator. This algorithm produces 64-bit values with good distribution.
- Operating System random number generator. Under Windows APL this uses the `CryptGenRandom()` function. Under Unix/Linux it uses `/dev/urandom[3]`.

You may select the random number generator in use using `16807⌶`. This allows you to switch dynamically between the different algorithms if required.

Performance Improvements

Empty-Line Processing

The interpreter now requires less time to process empty lines in functions, i.e. those that contain only comments or have no content at all. Functions documented with a block of leading comment lines, in particular, will benefit from this optimisation.

This improvement is also implemented in Version 13.0 from 3rd January 2012 onwards.

Dyadic Iota

The code for dyadic iota (Index Of) has been optimised, giving a small performance improvement in most cases. This improvement is independent of the use of retained hash tables, which delivers additional performance benefits as before.

This improvement is also implemented in Version 13.0 from 3rd January 2012 onwards.

New Idioms

Two new idioms have been added. These are:

`0=⊃⍴`	Is first dimension empty `(⎕ML<2)`
`0≠⊃⍴`	Is first dimension not empty `(⎕ML<2)`

Note that these idioms were also added to Version 13.0 from 22nd December 2011 onwards.

Take

The performance of an expression such as `(N↑1)` has been improved significantly in Version 13.1 compared to previous versions.

New Defaults for File Properties

When you create a new component file, the default levels of checksum and journaling are now both 1. These defaults may be changed using the **APL_FCREATE_PROPS_J** and **APL_FCREATE_PROPS_C** parameters (see User Guide).

The new default will cause the creation of component files which are significantly more robust in the face of network and other system failures, and often more repairable using ⎕FCHK. However, depending upon the way in which the application uses component files, there may be a noticeable reduction in system performance. Dyalog recommends the use of the new settings, however it is possible to configure the system to use a different default if necessary.

Support for Larger Components

In Version 13.1, the maximum size of a component on a component file is 2^{64} bytes. Previously it was 2^{32} bytes.

Note: All builds of Version 12.0, 12.1 and 13.0 created after 24th November 2010 will refuse to read components greater than 2^{32} bytes in size. Builds prior to this date will not handle large components correctly.

Compatibility Checking

The mechanism used to avoid backwards compatibilty issues has been improved. Prior to this change, it would have been possible for an older version of Dyalog APL to attempt to fix the ⎕OR of a function from a later version that contained a "new" system function or idiom. This would typically cause a system error at that point, or worse still, some time later in the application.

The new mechanism, which has been implemented in Version 13.1, prevents this happening by changing the internal format of a ⎕OR so that it is unrecognisable by older Versions.

This means that henceforth if you attempt to access an incompatible ⎕OR the operation will fail with a `DOMAIN ERROR`, rather than potentially cause a system error. (Note that builds of Version 12.1 dated before November 11th 2011 report `FILE COMPONENT DAMAGED` instead of `DOMAIN ERROR`).

Change to PrintList

The mechanism used to generate the contents of the PrintList property has been updated to better support network printers and the current versions of Windows. The new mechanism uses a different set of Windows API calls. This means that in Version 13.1, the list is likely to include printers which were not included in previous versions and the names of the printers reported may be slightly different.

APL_CODE_E_MAGNITUDE Parameter

Version 13.0 introduced decimal floating point numbers which have greater precision than IEEE floating point numbers. This increased the maximum allowable print precision from 17 to 34 and this had the side effect of changing the way numbers in function bodies are descanned[1]. For example, the number one sextillion (10^{21}) in a function is descanned by Version 12.1 as `1E21` and by Version 13.0 as `1000000000000000000000`.

Note that only numbers `X` in the range `(10*17) ≤ X <(10*34)` are affected.

Whilst this change has no other deleterious effect, it means that code that contains such numbers is harder to read, and the result of `⎕CR` (and other character representations) of the same function may have changed between Version 12.1 and Version 13.0 causing undesired affects in code management systems.

The **APL_CODE_E_MAGNITUDE** parameter allows the user to choose between current (Version 13.0 and onwards) and earlier behaviour.

If the **APL_CODE_E_MAGNITUDE** parameter is undefined or set to 0 (the default), numbers are descanned and displayed as normal.

If **APL_CODE_E_MAGNITUDE** is `¯1`, numbers greater than or equal to 10^{17} will be displayed using exponential format, as in Version 12.1.

The effect of setting this parameter to any other value is undefined.

[1]Descanning refers to the internal process used to convert the internal representation of APL code into a character array. For numbers in function statements, this process uses the maximum value of Print Precision.

Miscellaneous

Component Checksum Validation

There is a new function `3002⌶` which controls whether or not components read by `⎕FREAD` are subject to checksum validation.

Core to APLCore

There is a new function `685⌶` that extracts an `aplcore` file from a UNIX `core` file.

Windows: Associating workspaces etc. with specific Version

Under Windows, when Dyalog APL is installed, all Dyalog-specific files are associated with that version. A new User Command, `]EFA`, has been included which allows the user to select which of the versions of Dyalog APL installed on the PC should be associated with Dyalog files. See the *SALT-UCMD release notes* for more details.

GetAvailableWorkspace

This undocumented method of Root has been removed. It is superceded by `2000⌶`.

Change to :Hold

If APL detects a deadlock situation, it now executes the `:Else` clause in preference to generating an error.

COM Interface Changes

In Version 13.0 and earlier the Dyalog APL COM (OLE) interface converted incoming values of type VT_CY (a 64 bit value, sometimes referred to as VT_CURRENCY) to a two element vector containing the high 32 bits of the value and the low 32 bits of the value.

As Version 13.1 supports the DECF element type the OLE interface now converts incoming VT_CY values to DECFs, which means the value can be stored as a single scalar numeric value.

In Version 13.0 and earlier, VT_DECIMAL types were converted to 8 byte floating point values, which could have resulted in a loss of precision. Version 13.1 converts VT_DECIMALS to DECFs, which will not result in a loss of precision.

Note that `⎕FR` should be set appropriately if arithmetic is to be performed on the resultant arrays.

Component Numbers

Previously, the component number passed as an argument to a component file function was not properly validated, and was simply rounded downto an integer. In Version 13.1 the component number must be an integer, or the function will report `DOMAIN ERROR`.

Enlist and Selective Assignment

Enlist (`∈` with `⎕ML>0`) may be used in selective assignment expressions.

Example:

```
      ⎕ml←1
      names←'Andy' 'Karen' 'Liam'
      (('a'=∈names)/∈names)←'*'
      names
 Andy  K*ren  Li*m
```

URL Strings

In Windows versions valid URLs are identified when in the session or in the editor/tracer. When the mouse pointer is over a URL, the URL is underscored and the following items appear in the context menu:

- *Open link*: this causes the URL to be opened in the default application appropriate for the URL. Ctrl+Left Mouse performs the same operation
- *Copy link to clipboard*: this causes the URL to be copied to the clipboard

This feature can be enabled or disabled by selecting or unselecting the *Underline URLs and links* checkbox on the General tab in the Configure box. This is saved as the registry entry **URLHighlight**.

Issue 8146: (⍬⎕mat)←0 should not change the shape of mat

In Version 13.0

```
      mat←2 3⍴⍳6
      (⍬⎕mat)←0
```

resulted in

```
      mat
0
      ⍴mat
```

This is incorrect. In Version 13.1 this had been corrected so that

```
      mat
0 0 0
0 0 0
```

Dyalog Script Compiler

The default value of ⎕WX for the script compiler is now 3. If you require a different value for ⎕WX you can specify it on the command line using the /wx option, or assign it in your script. See the DotNet Interface Guide.

Change to path with ⎕USING and :Using

In previous versions if a .Net Assembly was referred to without a path, it would only be loaded if it was located in the .Net installation directory. With 13.1 Dyalog APL will first look in the directory in which the Dyalog program (or host application) is located, and then the .Net installation directory.

Support for changing between .Net versions

Dyalog APL has improved support for changing between .Net versions; see the *.Net Framework Tab* in the User Guide for more information.

Clipboard Format support

Dyalog APL now includes support (currently inbound only) for XMLSpreadSheet format.

2-digit years and yy_window

Dyalog APL now adheres to the 2-digit year rules which appear in the Windows Region and Language settings. These can be overridden using the *yy_window* parameter.

Enhancement to ⎕MONITOR

⎕MONITOR has been enhanced so that it now correctly allocates time to diamondised lines where the line start with :Case and similar constructs.

System Requirements

Microsoft Windows

Dyalog APL Version 13.1 supports all current versions of Windows from Windows 2000 up to and including Windows 7 and Windows Server 2008.

Dyalog APL Version 13.1 is not supported for versions of Windows prior to Windows 2000, such as Windows 95, Windows 98, Windows ME and Windows NT4.

Microsoft .Net Interface

Dyalog APL Version 13.1 .Net Interface requires Version 2.x or greater of the Microsoft .Net Framework. It does *not* operate with .Net Version 1.0.

Unix and Linux

For an up-to-date list of supported Unix and Linux platforms, please contact support@dyalog.com.

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 13.1 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, which is fully inter-operable.

The following sections describe other limitations in inter-operability:

Code

Code which is saved in workspaces, or embedded within `⎕OR`s stored in component files, can generally only be read by the version which saved them and later versions of the interpreter. In the case of workspaces, a load (or copy) into an older version would fail with the message:

```
this WS requires a later version of the interpreter.
```

Every time a `⎕OR` object is read by a Version later than that which created it, time may be spent in converting the internal representation into the latest form. Dyalog recommends that `⎕OR` 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 which 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 (see the following section), and for small-span component files created by Version 11.0 or later.

32 vs. 64-bit Component Files

Large-span (64-bit-addressing) component files are inaccessible to versions of the interpreter that pre-dated their introduction (versions earlier than 10.1).

The second item in the right argument of ⎕FCREATE determines the addressing type of the file.

```
'small'⎕fcreate 1 32 ⍝ create small-span file.
'large'⎕fcreate 1 64 ⍝ create large-span file.
```

If the second item is missing, the file type defaults to 64-bit-addressing. In versions prior to 12.0, the default was 32-bit-addressing.

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 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.

Unicode vs. Classic Editions

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

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

When a Unicode edition writes to a component file which may not contain Unicode data, character data is mapped using `⎕AVU`, and can therefore be read without problems by Classic editions.

A `TRANSLATION ERROR` will occur if a Unicode edition writes to a non-Unicode component (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 which are not in `⎕AVU`.

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

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

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

AVU changes

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

DECFs and Complex numbers

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

Very large array components

The maximum size (in bytes) of a component written by Version 12.1 and prior is 2GB. This is the size of the component as held on disk which may be different than the size reported by `⎕SIZE`. In Version 13.0 the maximum size of a component written by a 64-bit interpreter is 4GB. From Version 13.1 onwards, the limit on the size of arrays or components is so large that for most practical purposes, there is effectively no limit.

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

File Journaling

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

TCPSockets

TCPSockets used to communicate between differing versions of Dyalog APL are subject to similar limitations to those described above for component files. In particular TCPSockets with `'Style' 'APL'` will only be able to pass arrays that are supported by both versions.

Auxiliary Processors

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

Session Files

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

Chapter 2:

New Language Features

File History: R←⎕FHIST Y

Access code 16384

Y must be a simple integer vector of length 1 or 2 containing the file tie number and an optional passnumber. If the passnumber is omitted it is assumed to be zero.

The result is a numeric matrix with shape (5 2) whose rows represent the most recent occurrence of the following events.

1. File creation (see note)
2. (Undefined)
3. Last update of the access matrix
4. Last tie (See User Guide: **APL_FHIST_TIE** parameter)
5. Last update performed by ⎕FAPPEND, ⎕FCREATE, ⎕FDROP or ⎕FREPLACE

For each event, the first column contain the user number and the second a timestamp. Like the timestamp reported by ⎕FRDCI this is measured in 60^{th}s of a second since 1st January 1970 (UTC).

Currently, the second row of the result (undefined) contains (0 0).

Note: ⎕FHIST collects information only if journaling and/or checksum is in operation. If neither is in use, the collection of data for ⎕FHIST is disabled and its result is entirely 0. If a file has both journaling and checksum disabled, and then either is enabled, the collection of data for ⎕FHIST is enabled too. In this case, the information in row 1 of ⎕FHIST relates to the most recent enabling ⎕FPROPS operation rather than the original ⎕FCREATE.

In the examples that follow, the FHist function is used below to format the result of ⎕FHIST.

```
     ∇ r←FHist tn;cols;rows;fhist;fmt;ToTS;I2D
[1]    rows←'Created' 'Undefined' 'Last ⎕FSTAC'
[2]    rows,←'Last Tied' 'Last Updated'
[3]    cols←'User' 'TimeStamp'
[4]    fmt←'ZI4,2(⊂-⊃,ZI2),⊂ ⊃,ZI2,2(⊂:⊃,ZI2)'⍕
[5]    I2D←{+2 ⎕NQ'.' 'IDNToDate'⍵}
[6]    ToTS←{d t←1 1 0 0 0⊂⍉⌊0 24 60 60 60⊤⍵
[7]        ↓fmt ⍕⎕FMT(0 ¯1↓↑I2D¨25568+,d),0 ¯1↓t}
[8]    fhist←⎕FHIST tn
[9]    fhist[;2]←ToTS fhist[;2]
[10]   fhist[;1]←⍕¨fhist[;1]
[11]   r←((⊂''),rows),cols⍪fhist
     ∇
```

Examples

```
     'c:\temp'⎕FCREATE 1 ⋄ FHist 1
               User  TimeStamp
Created        0     2012-01-14 12:29:53
Undefined      0     1970-01-01 00:00:00
Last ⎕FSTAC    0     2012-01-14 12:29:53
Last Tied      0     2012-01-14 12:29:53
Last Updated   0     2012-01-14 12:29:53

     (⍳10)⎕FAPPEND 1  ⋄ FHist 1
               User  TimeStamp
Created        0     2012-01-14 12:29:53
Undefined      0     1970-01-01 00:00:00
Last ⎕FSTAC    0     2012-01-14 12:29:53
Last Tied      0     2012-01-14 12:29:53
Last Updated   0     2012-01-14 12:29:55

     ⎕FUNTIE 1

     'c:\temp'⎕FCREATE 1 ⋄ FHist 1
               User  TimeStamp
Created        0     2012-01-14 12:29:53
Undefined      0     1970-01-01 00:00:00
Last ⎕FSTAC    0     2012-01-14 12:29:53
Last Tied      0     2012-01-14 12:29:57
Last Updated   0     2012-01-14 12:29:55
```

Random Link: ⎕RL

⎕RL establishes a base or *seed* for generating random numbers using Roll and Deal, and returns the current state of such generation.

Three different random number generatators are provided, which are referred to here as *RNG0*, *RNG1* and *RNG2*. These are selected using (16807⌶). See "Random Number Generator:" on page 25. ⎕RL is relevant only to *RNG0* and *RNG1* for which repeatable pseudo-random series can be obtained by setting ⎕RL to a particular value first.

Using *RNG0* or *RNG1*, you can *set* ⎕RL to any integer in the range 1 to ¯1+2*31 or ¯1+2*63 respectively. The latter case requires ⎕FR to be 1287.

In a clear ws, ⎕RL is initialised to the value defined by the **default_rl** parameter which itself defaults to 16807 if it is not defined.

Using *RNG0*, ⎕RL *returns* an integer which represents the *seed* for the next random number in the sequence.

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

Internally, APL maintains the current state separately for *RNG0* and *RNG1*. When you switch from one Random Number Generator to the other, the appropriate state is loaded into ⎕RL.

RNG2 does not permit access to the *seed*, so in this case ⎕RL is not relevant and is not used by Roll and Deal. It will accept any value but will always return zilde.

Examples

```
      16807⌶1 ⍝ Select RNG1
0
      ⎕RL←16807
      10?10
4 1 6 5 2 9 7 10 3 8
      5↑⎕RL
10 0 16807 1819658750 ¯355441828
      X←?1000⍴1000
      5↑⎕RL
100 ¯465541037 ¯1790786136 ¯205462449 996695303

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


      16807⌶0 ⍝ Select RNG0
1
      ⎕RL
16807
      ?9 9 9
2 7 5
      ?9
7
      ⎕RL
984943658

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

      16807⌶1 ⍝ Select RNG1
0
      5↑⎕RL
100 ¯465541037 ¯1790786136 ¯205462449 996695303
```

Random Number Generator: R←16807⌶Y

Specifies the random number generator that is to be used by Roll and Deal.

`Y` is an integer that specifies which random number generator is to be enabled and must be one of the numbers listed in the first column of the table below.

`R` is an integer that identifies the previous random number generator in use.

The 3 random number generators are as follows :

Id	Algorithm
0	Lehmer linear congruential generator.
1	Mersenne Twister.
2	Operating System random number generator.

Under Windows, the Operating System random number generator uses the `CryptGenRandom()` function. Under Unix/Linux it uses `/dev/urandom[3]`.

The default random number generator in a `CLEAR WS` is 0 (Lehmer linear congruential). The default is likely to be changed to 1 (Mersenne Twister) in a future release of Dyalog APL. In preparation for this change, avoid writing code which assumes that `⎕RL` will be a scalar integer.

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

The Mersenne Twister algortithm *RNG1* produces 64-bit values with good distribution.

The Operating System algorithm *RNG2* does not support a user modifiable random number seed, so when using this scheme, it is not possible to obtain a repeatable random number series.

For further information, see "Random Link: " on page 23.

Extended Diagnostic Message: R←⎕DMX

⎕DMX is a system object that provides information about the last reported APL error. ⎕DMX has *thread scope*, i.e. its value differs according to the thread from it is referenced. In a multi-threaded application therefore, each thread has its own value of ⎕DMX.

⎕DMX contains the following Properties (name class 2.6). Note that this list is likely to change. Your code should not assume that this list will remain unchanged. You should also not assume that the display form of ⎕DMX will remain unchanged.

`Category`	character vector	The category of the error
`DM`	nested vector	Diagnostic message. This is the same as ⎕DM, but *thread safe*
`EM`	character vector	Event message; this is the same as ⎕EM ⎕EN
`EN`	integer	Error number. This is the same as ⎕EN, but *thread safe*
`ENX`	integer	Sub-error number
`HelpURL`	character vector	URL of a web page that will provide help for this error. Version 13.1 identifies and has a handler for URLs starting with *http:*, *https:*, *mailto:* and *www*. This list may be extended in future
`InternalLocation`	nested vector	Identifies the line of interpreter source code (file name and line number) which raised the error. This information may be useful to Dyalog support when investigating an issue
`Message`	character vector	Further information about the error
`OSError`	see below	If applicable, identifies the error generated by the Operating System
`Vendor`	character vector	For system generated errors, `Vendor` will always contain the character vector `'Dyalog'`. This value can be set using ⎕SIGNAL

OSError is a 3-element vector whose items are as follows:

1	integer	This indicates how the operating system error was retrieved. 0 = by the C-library `errno()` function 1 = by the Windows `GetLastError()` function
2	integer	Error code. The error number returned by the operating system using `errno()` or `GetLastError()` as above
3	character vector	The description of the error returned by the operating system

Example

```
      1÷0
DOMAIN ERROR
      1÷0
     ^
      ⎕DMX
 EM        DOMAIN ERROR
 Message   Divide by zero
 HelpURL   http://help.dyalog.com/dmx/13.1/General/1

      ⎕DMX.InternalLocation
 arith_su.c  554
```

Isolation of Handled Errors

`⎕DMX` cannot be explicitly localised in the header of a function. However, for all trapped errors, the interpreter creates an environment which effectively makes the current instance of `⎕DMX` local to, and available only for the duration of, the trap-handling code.

With the exception of `⎕TRAP` with Cutback, `⎕DMX` is implicitly localised within:

- Any function which explicitly localises `⎕TRAP`
- The `:Case[List]` or `:Else` clause of a `:Trap` control structure.
- The right hand side of a D-function Error-Guard.

and is implicitly un-localised when:

- A function which has explicitly localised ⎕TRAP terminates (even if the trap definition has been inherited from a function further up the stack).
- The :EndTrap of the current :Trap control structure is reached.
- A D-function Error-Guard exists.

During this time, if an error occurs then the localised ⎕DMX is updated to reflect the values generated by the error.

The same is true for ⎕TRAP with Cutback, with the exception that if the cutback trap event is triggered, the updated values for ⎕DMX are preserved until the function that set the cutback trap terminates.

The benefit of the localisation strategy is that code which uses error trapping as a standard operating procedure (such as a file utility which traps FILE NAME ERROR and creates missing files when required) will not pollute the environment with irrelevant error information.

Example

```
      ∇ tie←NewFile name
[1]     :Trap 22
[2]         tie←name ⎕FCREATE 0
[3]     :Else
[4]         ⎕DMX
[5]         tie←name ⎕FTIE 0
[6]         name ⎕FERASE tie
[7]         tie←name ⎕FCREATE 0
[8]     :EndTrap
[9]     ⎕FUNTIE tie
      ∇
```

⎕DMX is cleared by)RESET, .

```
       )reset
       ⍴⎕FMT ⎕DMX
0 0
```

The first time we run NewFile 'pete', the file doesn't exist and the ⎕FCREATE in NewFile[2] succeeds.

```
      NewFile 'pete'
1
```

If we run the function again, the ⎕FCREATE in NewFile[2] generates an error which triggers the :Else clause of the :Trap. On entry to the :Else clause, the values in ⎕DMX reflect the error generated by ⎕FCREATE. The file is then tied, erased and recreated.

```
 EM        FILE NAME ERROR
 Message   File exists
 HelpURL   http://help.dyalog.com/dmx/13.1/Componentfilesystem/9
1
```

After exiting the :Trap control structure, the shadowed value of ⎕DMX is discarded, revealing the orignal value that it shadowed.

```
      ⍴⎕FMT ⎕DMX
0 0
```

Example

The EraseFile function also uses a :Trap in order to ignore the situation when the file doesn't exist.

```
     ∇ EraseFile name;tie
[1]    :Trap 22
[2]        tie←name ⎕FTIE 0
[3]        name ⎕FERASE tie
[4]    :Else
[5]        ⎕DMX
[6]    :EndTrap
     ∇
```

The first time we run the function, it succeeds in tieing and then erasing the file.

```
      EraseFile 'pete'
```

The second time, the ⎕FTIE fails. On entry to the :Else clause, the values in ⎕DMX reflect this error.

```
      EraseFile 'pete'
 EM        FILE NAME ERROR
 Message   Unable to open file
 OSError   1 2  The system cannot find the file specified.
 HelpURL   http://help.dyalog.com/dmx/13.1/Componentfilesystem/11
```

Once again, the local value of ⎕DMX is discarded on exit from the :Trap, revealing the shadowed value as before.

```
      ⍴⎕FMT ⎕DMX
0 0
```

Example

In this example only the error number (EN) propery of ⎕DMX is displayed in order to simplify the output:

```
      ∇ foo n;⎕TRAP
[1]     'Start foo'⎕DMX.EN
[2]     ⎕TRAP←(2 'E' '→err')(11 'C' '→err')
[3]     goo n
[4]    err:'End foo:'⎕DMX.EN
      ∇

      ∇ goo n;⎕TRAP
[1]     ⎕TRAP←5 'E' '→err'
[2]     ⍎n⊃'÷0' '1 2+1 2 3' '∘'
[3]    err:'goo:'⎕DMX.EN
      ∇
```

In the first case a DOMAIN ERROR (11) is generated on goo[2]. This error is not included in the definition of ⎕TRAP in goo, but rather the the Cutback ⎕TRAP definition in foo. The error causes the stack to be cut back to foo, and then execution branches to foo[4]. Thus ⎕DMX.EN in foo retains the value set when the error occurred in goo.

```
      foo 1
Start foo  0
End foo:  11
```

In the second case a LENGTH ERROR (5) is raised on goo[2]. This error is included in the definition of ⎕TRAP in goo so the value ⎕DMX.EN while in goo is 5, but when goo terminates and foo resumes execution the value of ⎕DMX.EN localised in goo is lost.

```
      foo 2
Start foo  0
goo:  5
End foo:  0
```

In the third case a `SYNTAX ERROR` (2) is raised on `goo[2]`. Since the `⎕TRAP` statement is handled within `goo` (although the applicable `⎕TRAP` is defined in `foo`), the value `⎕DMX.EN` while in `goo` is 2, but when `goo` terminates and `foo` resumes execution the value of `⎕DMX.EN` localised in `goo` is lost.

```
      foo 3
Start foo  0
goo:  2
End foo:  0
```

Index