APLX
for Windows, Linux and MacOS

New features in Version 2.0

MICRO APL

MicroAPL welcomes your comments and suggestions.
Please visit our website: http://www.microapl.co.uk/apl

APLX Version 2.0.3 Upgrade notes: February 2004

Summary of APLX Version 2 Enhancements

Multi-tasking

- Create a new session from the menu, creates an independent APL task with its own session window
- Create a new APL task under program control, optionally with session window:
```
      'ChildTask' ⎕WI 'New'  'APL'  ('wssize' 100000) ('visible' 1)
```
- New properties and methods for controlling APL child tasks, eg `Execute` method causes child task to execute APL expression or system command
- New events allow parent task to be notified when child task starts or ends execution of an expression, hits an untrapped error, or terminates
- Signals for the child and parent task to talk to each other and pass commands/results
- Tasks can share data through shared variables and `⎕WI` delta-properties

Interpreter

- Removal of 64KB limit on executing token strings
- Increase of maximum array rank to 63
- `)CLEAR N` allows you to change WS size
- `)IN` maps APL*Plus zilde to `(⍳0)`
- Full path names on workspaces:
```
      )LIB C:\Workspaces
      )LOAD C:\Workspaces\myws.aws
```
- `⎕VI` and `⎕FI`
- `⎕TCxx` character constants for APL*Plus compatibility
- `⎕UCS` for converting chars to Unicode and vice-versa
- `⎕PFKEY` allows you to program the function keys
- Unicode conversion in native-file read/write
- Automatic allocation of tie numbers in native file system
- Capture the output from executed system commands: `X ← ⍎')WSID'`

`⎕FTIE`-family component files

- `⎕FTIE ⎕FCREATE` etc for compatibility with other APL interpreters
- Extensions to allow insertion or deletion of components anywhere in the file
- Multi-user and multi-tasking support *(not in Personal Editions of APLX)*
- Tie number can be allocated automatically by APLX if required
- Component files compatible across Windows, MacOS and Linux

Grid object

- Built-in Grid class for spreadsheet-like control with text or numeric cells
- Can be used for display, user-editing, or both
- Natural syntax, compatible with other `⎕WI` controls
- Set and read cell arrays directly as APL arrays

Other ⎕WI enhancements

- Enhancements to the `Draw` method including `TextSize`, `Seg` etc
- Delta-properties allow you to store as many APL variables within an object as you like
- `fonts` property of System and Printer objects give names of installed fonts
- `unicode` property of System object for reading/writing clipboard as Unicode
- `firstvisible` property of List object returns/sets first visible line
- `selection` and `seltext` properties for Combo
- Enhancements to OCX interface to allow returned LPDISPATCH handles to be used.

These enhancements are described in detail below.

Note: All of the enhancements in APLX Version 2 are upwards compatible, with the exception of changes to the specifying of path names in system commands, which could cause different behavior in a few cases. In particular, previous versions of APLX interpreted a blank row in the ⎕MOUNT table as meaning the 'currently-logged directory'. It is now always interpreted as the user's home directory. This could mean that workspaces appear to have 'disappeared' when you upgrade to APLX Version 2. The solution is to use the Preferences dialog to set up an explicit path, rather than a blank path, for the library numbers you use.

Multi-tasking Support

APLX Version 2.0 includes built-in support for multi-tasking. This means that a single instance of the APLX program can simultaneously run multiple APL tasks, each with its own workspace and optionally its own session window. (Technically, APLX runs as a single *process*, with multiple *threads*. One thread controls the user-interface, and each APL task has its own thread.)

APLX tasks are of two types, 'top-level' or 'parent' tasks, and 'child' tasks. When APLX starts up, it creates the first top-level task automatically. Further top-level tasks are created using the File menu on the APLX session window, and each has its own session window in which you can enter APL expressions. A top-level task keeps running until one of the following occurs:

- You type `)OFF` in the task's session window, or your program executes `)OFF`
- You select 'Close Session' from the File menu
- You close the session window

When a top-level task ends, other top-level tasks are not affected. When all top-level tasks have finished, the APLX program terminates. If you select 'Exit APLX' from the File menu, all tasks terminate.

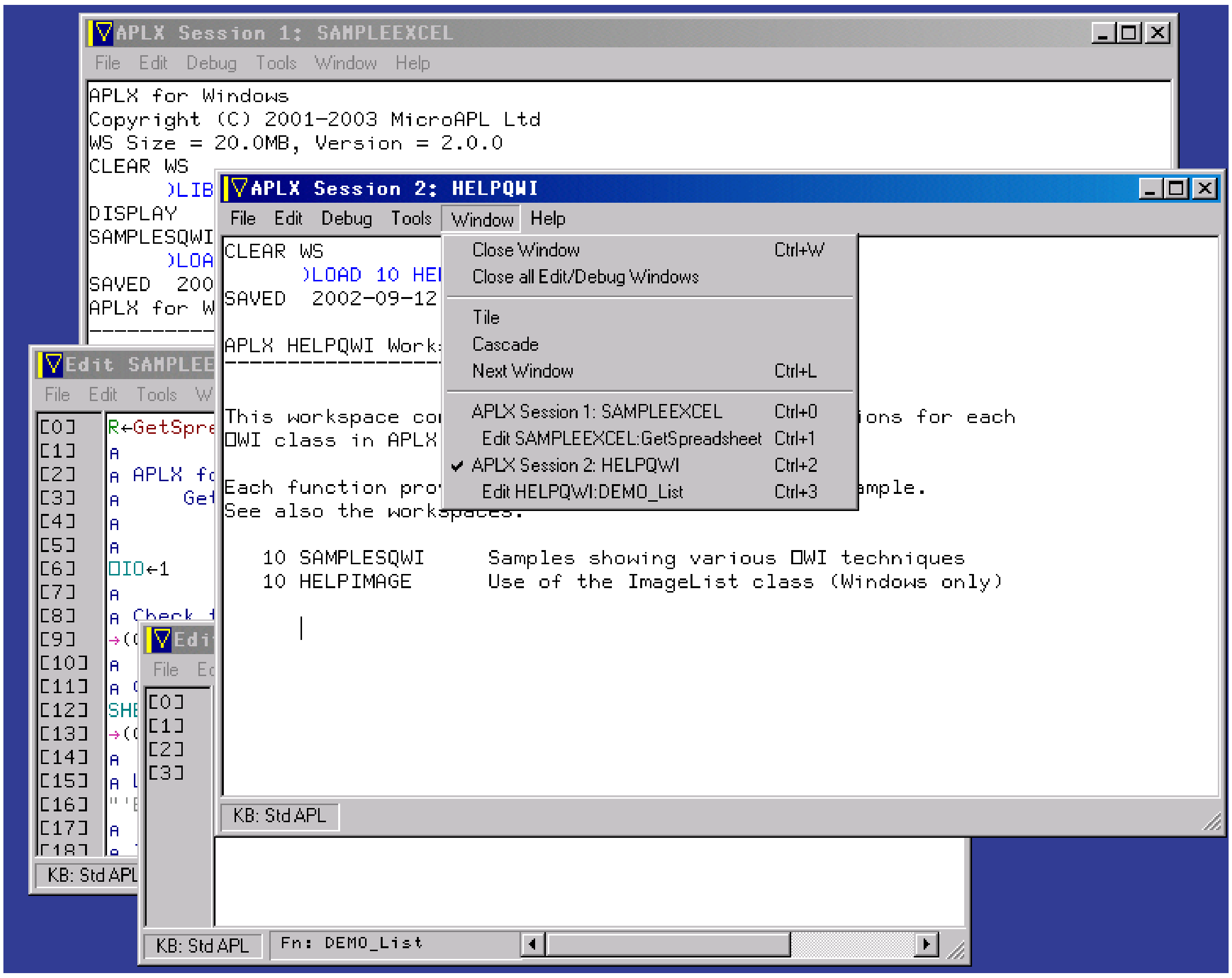
Child tasks are created under program control using `⎕WI`. They can either be *background* tasks (with no session window), or alternatively they can be ordinary tasks with their own session windows, although these might be hidden. Child tasks may be terminated in the same ways as top-level tasks, but in addition they can be terminated under program control by the parent task. They will also terminate automatically if their parent task terminates. Child tasks can themselves create further child tasks.

Each task has its own separate memory allocation for the workspace, of a size which you can specify. The tasks execute independently, but you can use *signals* to allow parent and child tasks to communicate with one another. In addition, variables can be shared between tasks. The niladic system function `⎕UL` returns the number of APLX tasks which are currently running.

Creating APL top-level tasks using the File menu

To create a new top-level task, just select 'New Session' from the File menu. A new session window will open, and you now have two APL tasks running. The window titles indicate which session is which.

You can execute APL expressions and)LOAD workspaces separately in each session window. You can also use the other APLX features such as Edit windows and the Workspace Explorer simultaneously in multiple tasks. To help you keep track of which window belongs to which task, the window title includes the workspace name. In addition, the Window menu shows the window titles associated with each session grouped together and indented. For example, this is how the screen might look if you have two top-level tasks running, and you are editing one function in each task:


APLX Session 1: SAMPLEEXCEL
File Edit Debug Tools Window Help
APLX Session 2: HELPQWI
File Edit Debug Tools Window Help
Close Window Ctrl+W
Close all Edit/Debug Windows
Tile
Cascade
Next Window Ctrl+L
APLX Session 1: SAMPLEEXCEL Ctrl+0
Edit SAMPLEEXCEL:GetSpreadsheet Ctrl+1
✔ APLX Session 2: HELPQWI Ctrl+2
Edit HELPQWI:DEMO_List Ctrl+3
Edit SAMPLEE
File Edit Tools W
KB: Std APL
Edi
File Ed
KB: Std APL
Fn: DEMO_List
KB: Std APL

```
APLX for Windows
Copyright (C) 2001-2003 MicroAPL Ltd
WS Size = 20.0MB, Version = 2.0.0
CLEAR WS
      )LIB
DISPLAY
SAMPLESQWI
      )LOA
SAVED  200
APLX for W
```

```
CLEAR WS
      )LOAD 10 HEL
SAVED  2002-09-12

APLX HELPQWI Work:
-------------------

This workspace co                                    ions for each
⎕WI class in APLX

Each function pro                                    ample.
See also the workspaces:

   10 SAMPLESQWI      Samples showing various ⎕WI techniques
   10 HELPIMAGE       Use of the ImageList class (Windows only)
```

```
[0]  R←GetSpre
[1]  ⍝
[2]  ⍝ APLX fo
[3]  ⍝      Get
[4]  ⍝
[5]  ⍝
[6]  ⎕IO←1
[7]  ⍝
[8]  ⍝ Check
[9]  →((
[10] ⍝
[11] ⍝ (
[12] SHE
[13] →((
[14] ⍝
[15] ⍝ L
[16] ''E
[17] ⍝
[18] ⍝
```

```
[0]
[1]
[2]
[3]
```

Setting the workspace size

When you create top-level task using the File menu, it is allocated the default workspace size which you have set using the Preferences dialog. You can change the workspace size using the)CLEAR system command, which optionally takes a parameter which is the desired workspace size in bytes, kilobytes (K), megabytes (M) or gigabytes (G):

```
      )CLEAR 800K
WS Size = 800KB
CLEAR WS
      )CLEAR 2M
WS Size = 2.0MB
CLEAR WS
```

The valid range is 50K to 2G. Depending on the operating system and its configuration, you are likely to be limited in the maximum size of workspace which you can allocate. *(Note: Under MacOS 9 and earlier, the total memory available to APLX is fixed in advance, so the limit may be quite small).*

Using multiple top-level tasks during APL development

During development, having multiple APL sessions available to you is very useful for comparing different versions of a workspace, or modifying an existing workspace whilst keeping the original version open as a reference point. It is also very useful when using utility functions and APL function libraries.

Creating APL child tasks under program control

You can create child tasks under program control using the 'APL' object under ⎕WI. This works as follows:

Creating a new APL task object

```
      'ChildTask' ⎕WI 'New' 'APL'
```

This creates a task object, but the APL task itself does not start running until you call the Open method. Before doing that, you can set various properties of the new task:

Specifying whether the task runs in the background

By default, child tasks you create will have their own session window. However, if you set the background property to 1, the task will not have a session window and all its output will be thrown away (you must set this property before calling the Open method).

Specifying the workspace size

By default, a new child task is allocated the amount of workspace which you have set as the default using the Preferences dialog. The wssize property allows you to specify a different size, as a value in bytes. If you ask for a very small value, a minimum value of around 100KB will be allocated. If you ask for a very big value, the operating system may allocate a smaller value. A value of 0 means 'use the default'. You must set the wssize property before calling the Open method, but you can read it back at any time.

Specifying whether the task's session window is visible

A task which is not a background task will have a session window associated with it, but it may not be visible. You can use the visible property (or the Show and Hide methods) to control whether it is shown, just as you would with any other window. This can be done before or after the Open method has been called. You can also use the where and size properties to set or read the position and size of the task's session window.

Starting the APL task

The actual APL task starts when you call the Open method. If the task has a session window, it will be created at this time, and it will appear unless the visible property is 0. If the task has a background property of 1, nothing will appear. The child task inherits the ⎕MOUNT table of its parent.

Example

This sequence creates a child task with its own visible session window, and a workspace size of 1MB:

```
      'ChildTask' ⎕WI 'New' 'APL' ('wssize' 1E6)
      'ChildTask' ⎕WI 'Open'
```

You can enquire about the visibility and coordinates of the session window as follows:

```
      'ChildTask' ⎕WI 'visible'
1
      'ChildTask' ⎕WI 'where'
12 32 24 64
```

Using the child task

If the child task has a visible session window associated with it, you can enter commands and APL expressions in it in the normal way. Alternatively, the parent APL task can use it as follows:

Checking the status of the child task:

The `status` read-only property indicates the execution state of the task. It returns one of the following codes:

`¯1`	The task is not running (i.e. it has not been opened, or has terminated)
`0`	The task is busy executing an APL expression or command
`1`	The task is in desk-calculator mode, awaiting input
`2`	The task is awaiting `⍞` input
`3`	The task is awaiting `⎕` input
`4`	The task is awaiting `⎕CC` input

Causing the child task to execute an expression or command

The `Execute` method allows the parent to cause the child task to execute an APL expression or system command. It takes as an argument a character string which the command to be executed (or passed as input to `⍞` or `⎕`). Note that the child task must be in awaiting input (`status` > 0), or else an error is generated. The `Execute` method returns immediately (it doesn't wait for the command to complete).

In this example, the child task is initially ready for input. It then executes the)LOAD command, and is busy for a short time (`status` property is 0). The `status` property then changes back to 1 when it is ready for further input:

```
      'ChildTask' ⎕WI 'status'
1
      'ChildTask' ⎕WI 'Execute' ')LOAD 10 SAMPLEEXCEL'
      'ChildTask' ⎕WI 'status'
0
      'ChildTask' ⎕WI 'status'
1
```

Note that in a real multi-tasking APLX application you would typically use 'signal' events to allow the parent and child tasks to communicate with each other - this is described below. The `Execute` method would typically be used only when starting up the child task.

Reading back the child task's session window contents

The `text` property of the APL object contains the text in the task's session window. For example, after executing the)LOAD above you might have:

```
      CONTENTS←'ChildTask' ⎕WI 'text'
      CONTENTS
CLEAR WS
      )LOAD 10 SAMPLEEXCEL
SAVED  2002-09-12 16.43.21
APLX for Windows and Excel Sample - 18th July 2002
--------------------------------------------------
This workspace contains some functions to demonstrate how to use APLX to
access Excel spreadsheets via the Microsoft Excel OLE server.
```
... *etc*

It is a text vector, usually with embedded carriage return (⎕R) characters. The `text` property returns the logical contents of the session window, even if the session window is not visible (however it is always an empty vector for a background task, which does not have a session window). You can also write to this property to change the text in the session window of a child task.

Uniquely identifying a child task

Every task has a unique identifier. You can read the task identifier of a child task using the read-only `taskid` property. It is a scalar integer, and can be used in event handling to identify the task:

```
      'ChildTask' ⎕WI 'taskid'
53429272
```

The `taskid` property will be zero if the task is not running.

The child task can read its own task identifier using the `taskid` property of its System object.

Interrupting a child task under program control

If a child task is executing (`status` property is 0), the `Interrupt` method can be called to interrupt it. It is equivalent to a keyboard interrupt (break).

Terminating a child task under program control

A child task may terminate for various reasons. For example, you might use the `Execute` method to cause it to execute an)OFF command, or it may run an APL function which ends by executing an)OFF. If it has a session window, you can end the task in the same ways as you would a top-level task, using the File menu or by closing the session window.

Alternatively, a parent task can terminate the child by using the `Close` method:

```
      'ChildTask' ⎕WI 'Close'
```

This closes any windows belonging to the task, releases the workspace memory, and ends the task. The `status` property reverts to `¯1`. If you wish you can then use the `Open` method to start it afresh, or the `Delete` method to destroy the task object altogether.

If a parent task terminates, all its child tasks are automatically terminated as well.

Communication between child and parent tasks

Once a child task is running, events are used to communicate between the child and its parent. These events can arise automatically, or more usually by explicit program action. They are used to trigger the execution of a callback function in the receiving task. (Note: As with all callbacks in APLX, the actual callback is run in a `⎕WE` (wait for events) statement).

The automatic events occur when the state of the child task changes. They are as follows (in each case, `⎕EV[6]` in index origin 1 will contain the unique task ID, as returned by the `taskid` property):

- When the state of the child task changes from executing to awaiting input, an `onReady` event is triggered in the parent's task object. You can associate a callback with this, for example to pass the next command to the child task:

```
      'ChildTask' ⎕WI 'onReady' 'NEXTCMD'
```

This would cause the `NEXTCMD` function to be run when the child task is ready for the next command to be sent to it using the `Execute` method.

- When an untrapped error occurs during function execution (but not desk-calculator mode) in the child task, an `onError` event is triggered in the parent's task object. You can associate a callback with this, for example to write the error to a diagnostic log:

  ```
        'ChildTask' ⎕WI 'onError' 'ERROR_HANDLER'
  ```

  This would cause the `ERROR_HANDLER` function to be run when an error is encountered in the child task. The error message (in the same format as `⎕ERM`) will be available in `⎕WARG` during the callback. (If you have an `onReady` callback defined as well, this will be triggered after the `onError`).

- When the child task is about to execute an expression or command (either typed by the user, or under program control) an `onExecute` event is triggered in the parent's task object. During the callback, `⎕WARG` contains the command or expression which is about to be executed as a character vector. This can be used for example to write an APL tutorial application when the parent task can see what the user is entering in the child-task session window.

- Finally, when the child task is about to terminate for any reason an `onClose` event is triggered in the parent's task object.

Signal events

The main method of communication between child and parent tasks is through explicit 'signal' events. This mechanism allows the child task to send a message to the parent, and vice versa. This works as follows:

- If the parent wants to send a signal to the child, it invokes the `Signal` method in the child task object. This causes an `onSignal` event to trigger in the System object of the child task.

- Conversely, if the child task wishes to send a signal to the parent, it invokes the `Signal` method in its own System object. This causes an `onSignal` event to trigger in the child task object of the parent.

In both cases, the `Signal` method optionally takes an argument, which is any APL array (or an APLX overlay created using `⎕OV`). This is typically used to send a command to the other task, or to return a result to it. Any argument to the `Signal` method is available to the receiving task as the `⎕WARG` system variable during the execution of the callback.

For example, suppose a child task is being used to carry out a time-consuming calculation. It starts by awaiting a signal to indicate that it should do the calculation, with the argument to the signal being the data to work on. When it has completed the calculation, it sends a signal back to the parent with the answer. The following sequence indicates how this might be done.

First we need to do some setup. The parent task attaches a callback to the child task object:

```
      ∇CALCDONE
[1]   'The answer is ' ⎕WARG
      ∇
```

```
      'ChildTask' ⎕WI 'onSignal' 'CALCDONE'
```

This will cause the CALCDONE function to be run when the child task signals that it has a result available, and the result will thus be printed.

Similarly, the child task attaches a callback to its System object, waiting for a signal to indicate that it should carry out the calculation:

```
            ∇DOCALC;RESULT;DATA
      [1]   DATA←⎕WARG
      [2]   RESULT←RUNMODEL DATA
      [3]   '⎕' ⎕WI 'Signal' RESULT
            ∇

      '#' ⎕WI 'onSignal' 'DOCALC'
      ⎕WE ¯1
```

This will cause the DOCALC function to be run when the parent signals the child, using the data passed as the argument to the **Signal** method. When the calculation is complete, the child signals back to the parent. The child then sits in the ⎕WE event loop waiting for signals to occur (this takes no CPU resource).

To carry out the calculation, the parent signals the child, and then processes events (if it is not already running under ⎕WE):

```
      'ChildTask' ⎕WI 'Signal' THEDATA
      ⎕WE ¯1
```

This triggers the child task to run the DOCALC function, and on completion the parent's CALCDONE function runs:

```
The answer is 42
```

In a real example, the parent task would be responding to other events (for example, user-interface events), and there might be a number of child tasks each simultaneously working on a different run of the model. If you have multiple child tasks running, you can use the task ID (⎕EV[6] in index origin 1) to distinguish between them in the callback function.

There might also be a queue of signals waiting to be processed on either side. Note that there is a limit to the number of events which can be queued, typically around 200 events. If this maximum is exceeded, the oldest events are thrown away.

Using 'delta' properties to share data between Parent and Child tasks

As discussed below in the discussion of new ⎕WI features, APLX now also supports 'delta' properties for all objects. Any property name which starts with the delta character ∆ can be used to store your own data in the object. Subject to available memory, you can have as many such properties as you like, and you can store any simple or nested array, or overlay, in each property.

This feature works in a special way for child task objects. A delta property for the System object of a child task is the same as the delta property of the same name in the parent task's child task object. Thus the parent task can assign arbitrary data to a delta property, and the child task can access it through its own System object. If the child task writes to a delta property of its System object, the parent can see the data which has been assigned in the child task object.

For example:

```
          'Task1' ⎕WI 'New' 'APL' ('wssize' 200000) ('background' 1)
          'Task1' ⎕WI '∆MESSAGE' 'Startup'
          'Task1' ⎕WI 'Open'
          'Task1' ⎕WI 'Execute' "'⎕' ⎕WI '∆MESSAGE' 'OK'"
          'Task1' ⎕WI '∆MESSAGE'
   OK
```

This technique can be used in a number of ways. When starting a task, it can be used to pass an overlay containing all the functions and variables which the child task will need to run. When the task is running, delta properties can be used to share state information, parameters and results between the parent and the child.

The delta property persists for as long as the child task object exists in the parent task. This means that a child task can write to a delta property, and then terminate. The data will still be available to the parent.

Sharing variables between all tasks

As well as passing data using the `Signal` method and delta properties, you can also share variables between all the APL tasks you have running (this includes all top-level tasks and all child tasks). You do this using Auxiliary Processor 800, which is built into APLX. All that is required is to share a variable with this processor:

```
      800 ⎕SVO 'TITLE'
2
```

This causes the variable `TITLE` to be held outside the workspace, in a common data area accessible to any other tasks which use `⎕SVO` to share the same variable. When you assign to a shared variable of this type, APLX allocates memory for it outside the workspace. If this memory cannot be allocated, an `IO ERROR` is reported to the APL task. Other than available memory, there are no limits on the size or number of such variables.

The value of the variable will be the last value written to it by any task. This can be any APL array, or an overlay created using `⎕OV` which contains multiple functions and variables. It persists for as long as any task has the variable shared, and is automatically deleted when it is no longer required (or when APLX exits).

For example:

Share a variable with all other tasks:

```
      800 ⎕SVO 'TITLE'
2
```

Assign some data to it:

```
      TITLE←"Lady Windermere's Fan"
```

Read it back:

```
      TITLE
Lady Windermere's Fan
```

Read it back again:

```
      TITLE
The Importance of Being Earnest
```

Another task has modified it!

APL Language enhancements

- The maximum rank for arrays in APLX is increased to 63.

- Previous versions of APLX had a limit of 64KB for the size of an executed token string. This limit has now been removed and the size of a token string is limited only by available workspace. (An individual function line is still limited to 64KB).

- As described earlier in this document, `)CLEAR` now optionally changes the workspace size. You can specify a parameter which is the workspace size you want. It must be an integer, and can be specified in bytes, or followed by K or KB for kilobytes, M or MB for megabytes, or G or GB for gigabytes. The valid range is 50KB to 2GB. Depending on the operating system and its configuration, and amount of memory already in use by APLX tasks, you are likely to be limited in the maximum size of workspace which you can allocate. Thus you may not get the full size requested. *(Note: Under MacOS 9 and earlier, the total memory available to APLX is allocated in advance, so the limit may be quite small. You can change the amount allocated using 'Get Info' in the Finder's 'File' menu).*

  For example:

  ```
        )CLEAR 1000000
  WS Size = 976KB
  CLEAR WS
        )CLEAR 1024KB
  WS Size = 1.0MB
  CLEAR WS
        )CLEAR 100M
  WS Size = 100MB
  CLEAR WS
        )CLEAR 2G
  WS Size = 484MB
  CLEAR WS
  ```

  Note that in the last example, the user requested 2GB but the operating system allocated only 484MB.

- The output from executed system commands can now be captured in a variable:

  ```
        X←⍎')SYMBOLS'
        X
  IS 1026, USED 21
  ```

- The `)IN` system command has been enhanced for better handling of transfer files created by APL*Plus. Where the APL*Plus transfer file contains the zilde character (which is not available in APLX), it is replaced by `(⍳0)`

- System commands for workspace and transfer-file manipulation (such as `)WSID )LOAD )SAVE )IN`) can now take a full pathname. The limits on the length and valid characters in a workspace name have been removed. See the separate section on pathnames for full details.

- The new monadic system functions `⎕VI` and `⎕FI` can be used to validate a text string and convert it to numeric form. In both cases, the argument is a character vector (or scalar), containing one or more sub-strings of characters separated by blanks. For each non-blank sub-string, `⎕VI` returns a 1 if the sub-string represents a valid number, and 0 if it does not. `⎕FI` returns the numeric representation of the sub-string, or 0 if it is not a valid number. Numbers are validated and converted in the same way as normal APL input. Scientific notation is supported, and negative numbers are prefixed by the high minus (¯) character. For example:

```
      ⎕FI '100.32 $4 2,,3 0 12.2 ¯3 +2 -2'
100.32 0 0 0 12.2 ¯3 0 0
      ⎕VI '100.32 $4 2,,3 0 12.2 ¯3 +2 -2'
1 0 0 1 1 1 0 0
```

`⎕VI` and `⎕FI` are usually used to validate and convert user input, or to convert text files to numeric form. To allow users to enter negative numbers using the ordinary (non-APL) minus sign, you can use `⎕SS` to translate minus to high-minus before using `⎕VI` and `⎕FI`. To allow comma-delimited input, use `⎕SS` to translate comma to space:

```
      STRING←'3,-2,45.5,-0.08'
      ⎕VI STRING
0
      ⎕SS (STRING; ('-';','); ('¯';' '))
3 ¯2 45.5 ¯0.08
      ⎕VI ⎕SS (STRING; ('-';','); ('¯';' '))
1 1 1 1
```

- The new system function `⎕UCS` translates Unicode values to the equivalent character in the APLX character set, and vice versa. See the separate section on Unicode support for details.

- The native file functions `⎕NREAD` and `⎕NWRITE` now provide the facility to read and write Unicode (or byte-reversed Unicode) text. See the separate section on Unicode support for details.

- The native file functions `⎕NTIE` and `⎕NCREATE` normally take a non-zero tie number as the first element of the right argument, leaving you to ensure that the tie number is unique. This has now been extended to allow you specify zero as the tie number. In this case, APLX allocates the next available free tie number and returns it as the explicit result of the function. (The tie number returned is equivalent to the expression `1+⌈/0,⎕NNUMS` before the file is tied/created). You use it as follows:

```
      TIE←FILENAME ⎕NTIE 0
      DATA←⎕NREAD TIE,0
```

- `⎕UL` now returns the number of APLX tasks which are currently running.

- In addition to the existing primitive functions `⍇ ⍈ ⍐ ⍗` for accessing APL component files, APLX now supports the `⎕FTIE` family of component-file system functions for compatibility with some other APL interpreters, with extensions to provide additional facilities such as inserting or deleting components anywhere in the file rather than just at the start or end. These system functions are described in detail in a separate section.

- The new system `⎕PFKEY` function allows you to associate your own strings with function keys. It takes a left argument which is the character string you want to associate with the key (with a closing `⎕R` or `⎕TCNL` character if you want to simulate pressing Return at the end). The right argument is a numeric scalar giving the key number. The range 1 to 15 corresponds to function keys F1 to F15. Add 15 to the value to represent the shifted key, and 30 to the value to represent the key with Control held down. For example, the following two commands would cause F5 to execute a `)WSID` command and Shift-F5 to execute a `)TIME` command, in the session window:

```
      (')WSID',⎕R) ⎕PFKEY 5
      (')TIME',⎕R) ⎕PFKEY 5+15
```

The strings you attach can be used for any purpose you like, either as short-cuts in APL development, or within your application. However, be aware that some function-key combinations may be reserved for use by the operating system or window-manager, or as menu short-cuts. In this case programming the function key yourself may not work (because the operating-system may intercept it before APLX sees the keystroke), or may be incompatible with the user-interface guidelines for the system you are using.

- To assist migration of APL code from APL*Plus, APLX now supports the following niladic system functions which return 'terminal-control' characters (these are in addition to the APL.68000-compatible `⎕L ⎕R ⎕B ⎕C`, and the APL2-compatible `⎕TC`):

`⎕TCBEL`	Bell character, equivalent to `⎕C[⎕IO+7]`
`⎕TCBS`	Backspace, equivalent to `⎕B` or `⎕TC[⎕IO+0]`
`⎕TCDEL`	Delete character, equivalent to `⎕C[⎕IO+32]`
`⎕TCESC`	Escape character, equivalent to `⎕C[⎕IO+27]`
`⎕TCHT`	Horizontal tab character, equivalent to `⎕C[⎕IO+9]`
`⎕TCFF`	Formfeed character, equivalent to `⎕C[⎕IO+12]`
`⎕TCLF`	Linefeed character, equivalent to `⎕L` or `⎕TC[⎕IO+2]`
`⎕TCNL`	Newline (carriage return) character, equivalent to `⎕R` or `⎕TC[⎕IO+1]`
`⎕TCNUL`	Null character, equivalent to `⎕AV[⎕IO+0]`

System-command pathnames

Traditionally, in APLX and its predecessor APL.68000, you have been able to identify workspaces and transfer files in two ways:

- From the menu, using a file-selector dialog to select a filename
- From the command line, by providing a simple undecorated name and an optional library number. Library numbers 0 through 9 were initially set using the Preferences dialog, and could be modified under program control using `⎕MOUNT`. Library 10 is set to the directory where the APLX support and sample workspaces are installed. File names were limited to 11 alphanumeric characters, and (except under MacOS), always had the default extensions appended to them (`.aws` for workspaces).

APLX Version 2 retains upwards compatibility with this model, but also allows you to specify full pathnames in system commands. It also removes the restrictions on workspace name length and valid characters. You can now if you wish use different (or no) file extensions in file names, although this is not usually recommended except when sharing files between MacOS and other systems.

Specifying workspace and transfer-file names in system commands

APLX now applies the following rules when interpreting a name in a system command such as `)LOAD` or `)OUT`:

1. If the name begins with a number followed by one of more spaces, the number is taken to be a library number and the text after the space(s) is the file name. The directory in which the file is located is taken from the appropriate row of the `⎕MOUNT` table. If the directory is blank, the user's home directory is assumed. Under Linux or Windows, the file extension is automatically appended to the supplied name.

2. If the name does not begin with a number followed by spaces, APLX looks at the name to see if it contains at least one directory separator character (/ under Linux, \ under Windows, or : under MacOS). If it does not, then it is treated as a simple filename in Library 0, and (under Linux or Windows) the file extension is automatically appended to the supplied name.

3. If the name does contain a directory-separator character, it is assumed to be a full pathname, **including the file extension if any**. APLX uses the name exactly as supplied.

In each case, the name is normally assumed to end at the first blank character. If you want to include blanks in the name, you can enclose the whole file name in single quotes.

Examples *(Windows):*

Suppose the first three rows of your `⎕MOUNT` table are set up as follows:

```
      3 40↑⎕MOUNT ''
c:\temp

G:\apl\historic\aplx
```

Note that the second row, library 1, is blank, so will correspond to the user's home directory. This might be something like: `C:\Documents and Settings\Jim\My Documents`.

Library of directory 0:

```
      )LIB 0
IF          JIM         PICTUREDEMO PLUSFNS     SC          TESTDISPLAY
```

Library of directory 0, implicit library number:

```
      )LIB
IF          JIM         PICTUREDEMO PLUSFNS     SC          TESTDISPLAY
```

Library of the same directory, specifying a full library path:

```
      )LIB c:\temp
IF          JIM         PICTUREDEMO PLUSFNS     SC          TESTDISPLAY
```

Library of directory 1. Because `⎕MOUNT` *table entry is blank, this is the user's home directory:*

```
      )LIB 1
ANOTHER     BERT        FRED
```

Load a workspace from library 0 (the 0 could be omitted):

```
      )LOAD 0 PICTUREDEMO
SAVED  2002-07-05 15.51.31
```

Full path is `c:\temp\PICTUREDEMO.aws`

Load a workspace from library 1:

```
      )LOAD 1 ANOTHER
SAVED  2003-11-18 10.49.51
```

Full name is `ANOTHER.aws` *in user's home directory*

Load a workspace using full explicit path, ***including file extension*** *(Note that Windows file names are not case-sensitive):*

```
      )LOAD C:\TEMP\PICTUREDEMO.AWS
SAVED  2002-07-05 15.51.31
      )WSID
C:\TEMP\PICTUREDEMO.AWS
```

Save under a name containing spaces - we need to enclose the name in quotes:

```
      )SAVE 'A nice name with spaces'
2003-12-10 13.44.16
      )LIB
A nice name with spaces IF          JIM         PICTUREDEMO PLUSFNS
SC          TESTDISPLAY
```

Re-load using a full pathname - again we need to enclose the name in quotes, and supply the file extension because we're using a full path:

```
      )LOAD 'c:\temp\A nice name with spaces.aws'
SAVED  2003-12-10 13.44.16
```

Notes

1. If you use full pathnames under Windows and Linux, you should normally supply the `.aws` file extension, otherwise the workspace will not show up in the `)LIB` listing. This is not true under MacOS, which keeps a file type separate from any file extension.
2. Some system commands (such as `)LOAD`) can take an optional colon and password. Under MacOS, this might be confused with a full pathname, so you must include a space before the colon to terminate the name.

`⎕FTIE`-family component file functions

Introduction

APLX (and its predecessor APL.68000) has always had a powerful, multi-user component file system accessed using the primitives ⍇ ⍈ ⍐ and ⍗. Whilst this system provides data storage facilities which allow you to write sophisticated APL applications, many other APL interpreters use a different component file system based on the system functions `⎕FCREATE` `⎕FTIE` `⎕FREAD` and so on. Version 2.0 of APLX implements this second system (with enhancements), making it easier for users to migrate applications from other APL interpreters. The original primitives are of course still available, so existing APLX and APL.68000 code will continue to work unchanged.

`⎕Fxxx` files are identified by a file name, and created using `⎕FCREATE`. For each APL component file, a separate operating-system file will be created. When you want to use an existing file, you first 'tie' (open) it using `⎕FTIE` or `⎕FSTIE`, and then you refer to the file by the tie number which you have specified or which has been automatically allocated by APLX. (This is in contrast to the ⍈ ⍇-based system, where a single 'dataspace' holds multiple APL component files, component files are always identified by number, and there is no need to 'tie' a file to use it.) Once the file has been tied, components are accessed by component number. When you have finished using a file, you must close it using `⎕FUNTIE`. (They are untied automatically when the APL task ends, but they are not untied automatically when you `)CLEAR` the workspace or `)LOAD` another workspace).

Components within a file are numbered sequentially, initially from 1 to N, where N is the number of components in the file. You read components from an existing file using `⎕FREAD`. You can write a component to the file using the `⎕FAPPEND` and `⎕FREPLACE` facilities implemented by other APL interpreters; these allow you to append to the end of the file, or to replace an existing component respectively. You can also delete components using `⎕FDROP`, but only from the start or the end of the file. Components are not re-numbered, so if you drop components from the start of the file, the first component will no longer be number 1.

APLX retains upwards compatibility with this simple model, but in addition provides the more general functions `⎕FWRITE` (which allows you to insert components anywhere within the range of existing components, or immediately before or after them), and `⎕FDELETE` (which allows you to delete a component anywhere in the file). When you use these extensions, components are automatically re-numbered so that they always comprise sequential integers from the first component M to the last component 1+M-N, where N is the number of components in the file.

Individual components may be any valid APL data, including nested arrays and overlays created using `⎕OV` (which can contain multiple functions and variables). The components keep their type and shape when stored and retrieved. When you replace a component, the new component does not have to be the same size as the original; the file system automatically expands the file if necessary to accommodate a larger component, and if possible releases space when you replace an existing component with a smaller one.

When using the file system in a multi-user or multi-tasking environment, you can optionally tie a file for exclusive use (`⎕FTIE`), or for shared access (`⎕FSTIE`). A file may be kept secure from other users by a pass number, and you can set an *access matrix* which determines what operations other users can perform. To facilitate concurrent use of shared files whilst maintaining data integrity, the file hold facility `⎕FHOLD` allows you to hold one or more files temporarily for exclusive use.

⎕FCREATE - Create a new file

The ⎕FCREATE function creates a new component file, and leaves it tied. The syntax is:

```
          FILENAME ⎕FCREATE TIENO
```

FILENAME is a character vector specifying the name of the file to create. The name may be specified in either of two ways. If the name contains a directory-separator character (: / or \), it is treated as a full host path name (including the file extension, by convention "`.aqf`"). Otherwise it is treated as basic file name only, optionally preceded by a volume number 0 to 9, separated by one or more spaces from the name. In the latter case, the file is created in the directory 0 to 9 specified (the actual path is set using the preferences dialog or ⎕MOUNT), and the file extension `.aqf` is added automatically.

TIENO is an arbitrary non-zero integer to be used in subsequent read/write operations to identify the file (the tie is exclusive). The tie number must not currently be in use to tie another file. Alternatively, you can provide a tie number of 0, in which case APLX automatically allocates the next available unused tie number, and returns it as the explicit result of the function.

For example, suppose APLX is running on a Windows machine and the ⎕MOUNT table is set (either under program control or using the Preferences dialog) so that library 0 is in `c:\temp` and library 1 is in `m:\budget\current`:

```
      2 30↑⎕MOUNT ''
c:\temp
m:\budget\current
```

You could create a new file called RUN3 in `c:\temp` as follows (no directory separator character appears in the name, so it is taken as a simple file name in library 0):

```
      'RUN3' ⎕FCREATE 2
```

(Using a left argument of `'0 RUN3'` would be equivalent). The file is created, and then exclusive-tied on tie number 2, so you can write to it immediately:

```
      ⎕FNUMS
2
      ⎕TS ⎕FWRITE 2
```

The full operating-system path would be `c:\temp\RUN3.aqf`.

In this second example, the user has specified 0 as the tie number, so APLX allocates and returns the next available tie number. The file is created in library 1, so the full operating-system path would be `m:\budget\current\RUN4.aqf`:

```
      '1 RUN4' ⎕FCREATE 0
3
      ⎕FNUMS
2 3
```

In this third example, a full path name has been supplied, with an explicit tie number of 8, so we now have three files tied:

```
      'c:\temp\RUN4.aqf' ⎕FCREATE 8
      ⎕FNUMS
2 3 8
```

```
      ⎕FNAMES
RUN3
1 RUN4
C:\TEMP\RUN4.aqf
```

(Under Linux or AIX, the full path name might be something like `'/usr/tmp/RUN4.aqf'`. Under MacOS, it might be something like `'Macintosh HD:temp:RUN4.aqf'`).

Note that, if you specify a full file name, you can use any file extension (or none). However, we recommend that you always use `.aqf` for APLX `⎕Fxxx` component files. If you do not use the `.aqf` extension, your component files will not show up in `⎕FLIB`, and you will not be able to access them using the library-relative syntax.

`⎕FTIE` and `⎕FSTIE` - Open an existing file

The `⎕FTIE` and `⎕FSTIE` functions open ('tie') an existing component file. The syntax is:

```
      FILENAME ⎕FTIE TIENO {PASS}
or    FILENAME ⎕FSTIE TIENO {PASS}
```

If you tie the file using `⎕FTIE`, it is tied for exclusive use and no other users or tasks will be able to tie it until you untie it. If you tie it using `⎕FSTIE`, other tasks and/or users can also tie it using `⎕FSTIE`.

`FILENAME` is a character vector specifying the name of the file to tie, following the same rules as `⎕FCREATE`. The name may be specified in either of two ways. If the name contains a directory-separator character (: / or \), it is treated as a full host path name (including the file extension, by convention "`.aqf`"). Otherwise it is treated as basic file name only, optionally preceded by a volume number 0 to 9, separated by one or more spaces from the name. In the latter case, the file is searched for in the directory 0 to 9 specified (the actual path is set the preferences dialog or `⎕MOUNT`), and the file extension `.aqf` is added automatically.

`TIENO` is an arbitrary non-zero integer to be used in subsequent read/write operations to identify the file (the tie is exclusive). The tie number must not currently be in use to tie another file. Alternatively, you can provide a tie number of 0, in which case APLX automatically allocates the next available unused tie number, and returns it as the explicit result of the function.

The optional `PASS` parameter is a pass number. If you use this parameter, it must match one of the valid pass numbers for your user ID set in the access matrix (see the discussion of `⎕FSTAC` below), and you must use the same pass number in all subsequent operations until you untie the file.

For example:

Exclusive-tie the file `RUN3.aqf` in library 0, under tie number 2:

```
      'RUN3' ⎕FTIE 2
```

Share-tie the file `RUN4.aqf` in library 1, allowing APLX to allocate the tie number:

```
      'RUN3' ⎕FSTIE 0
3
```

Share-tie the file `c:\temp\RUN3.aqf`, allowing APLX to allocate the tie number and using pass number 33421:

```
      'c:\temp\RUN3.aqf' ⎕FSTIE 0 33421
4
```

If you try to tie a file which is already tied for exclusive use by another user or task, APLX will retry for a few seconds in the hope that the tie will be released. If this does not occur, the operation will fail with error code `24` in `⎕LER`, or `6 5` in `⎕ET`:

```
      'RUN3' ⎕FTIE 2
FILE OR COMPONENT HELD
      'RUN3' ⎕FTIE 2
      ^
      ⎕LER
24 0
      ⎕ET
6 5
```

Files are untied automatically when an APLX task ends. They are NOT untied when you `)CLEAR` the workspace or `)LOAD` another workspace.

`⎕FWRITE` - Write a component anywhere in a file

The `⎕FWRITE` function allows you to write a component anywhere in the file, either inserting, replacing or appending. The syntax is:

```
          DATA ⎕FWRITE TIENO {COMPONENT} {PASS}
```

`DATA` is any APL array, or an overlay created using `⎕OV`. `TIENO` is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the `PASS` parameter.

`COMPONENT` is the component number at which the data should be written. This can be one of the following:

- If `COMPONENT` is zero, or is omitted, or is equal to the highest current component number plus 1, the data is written to the next available component number in the file (i.e. the highest current component number plus 1). If the file currently contains no components, the data will be written as component 1. This behavior is the same as that of `⎕FAPPEND`.

  For example, if the file currently contains components 1 to 6, the following are all equivalent; the data will be written as a new component 7:

  ```
  DATA ⎕FWRITE TIENO
  DATA ⎕FWRITE TIENO,0
  DATA ⎕FWRITE TIENO,7
  ```

- If `COMPONENT` is an integer which corresponds to an existing component, the existing component is replaced by the new data supplied. For example, if the file currently contains components 1 to 6, and you supply a `COMPONENT` parameter of 3, the third component will be replaced by the new data you write. The number of components is unchanged. If the new array is bigger than the previous data, the necessary space will be allocated automatically. If the new array is smaller, the released space will be kept in a pool of available space and re-used if possible in a future operation. This behavior is the same as that of `⎕FREPLACE`.

- If `COMPONENT` is a non-integral number between the first existing component minus 1, and the last component number plus 1, a new component is inserted and the data is written to the new component. The new component will be placed at the component number `⌈COMPONENT`. Any later components will be renumbered to allow for the inserted component.

For example, if the file currently contains components 1 to 6, the following statement would insert a new component between the existing fifth and sixth components, with the old sixth component becoming the new component 7:

```
      DATA ⎕FWRITE TIENO,5.5
```

If the file currently contains components 3 to 8, then the following statement would insert a new component (as the new component 3) before the current first component. The existing components would be renumbered 4 to 9:

```
      DATA ⎕FWRITE TIENO,2.5
```

If the number is between the current highest component number and the current highest component number plus 1, the operation is equivalent to appending a new component.

- Any other component number gives rise to an error `COMPONENT NOT IN FILE` (error code `20` in `⎕LER`, or `6 3` in `⎕ET`)

The following complete sequence illustrates the various possibilities:

Create a new file, and append four components to it in different ways:

```
      'EXAMPLE' ⎕FCREATE 2
      'First component' ⎕FWRITE 2 1
      'Second component' ⎕FWRITE 2
      'Third component' ⎕FWRITE 2 0
      'Fourth component' ⎕FWRITE 2 3.5
```

Read back the four components:

```
      ⎕DISPLAY ⎕FREAD ¨2,¨⍳4
┌→─────────────────────────────────────────────────────────────────────────┐
│ ┌→──────────────┐ ┌→───────────────┐ ┌→──────────────┐ ┌→───────────────┐ │
│ │First component│ │Second component│ │Third component│ │Fourth component│ │
│ └───────────────┘ └────────────────┘ └───────────────┘ └────────────────┘ │
└∊─────────────────────────────────────────────────────────────────────────┘
```

Drop the first component, components are now number 2 to 4:

```
      ⎕FDROP 2 1
      ⎕FSIZE 2
2 5 3584 0 256
      ⎕FREAD 2 1
COMPONENT NOT IN FILE
      ⎕FREAD 2 1
      ^
```

Replace component 3 with a new value:

```
      'Third rewritten' ⎕FWRITE 2 3
      ⎕FREAD 2 3
Third rewritten
```

Insert a new component before the existing first component, renumbering existing components to allow for it:

```
      'Inserted first' ⎕FWRITE 2 1.5
      ⎕FREAD 2 2
Inserted first
```

Insert a new component between the existing third and fourth components (it will become the new component 4).:

```
      'Interloper' ⎕FWRITE 2 3.5
```

See what we've ended up with:

```
      ⎕FSIZE 2
2 7 4096 0 0                    (First component number is 2, next available is 7)

      ⎕FREAD 2 2
Inserted first
      ⎕FREAD 2 3
Second component
      ⎕FREAD 2 4
Interloper
      ⎕FREAD 2 5
Third rewritten
      ⎕FREAD 2 6
Fourth component
```

We're done. Untie the file:

```
      ⎕FUNTIE 2
```

⎕FAPPEND - Write a new component at the end of a file

The ⎕FAPPEND function appends a new component to the file, returning the component number used. The syntax is:

```
          R ← DATA ⎕FAPPEND TIENO {PASS}
```

DATA is any APL array or an overlay created using ⎕OV. TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS parameter.

The explicit result is the component number to which the data was written (i.e. the highest existing component number plus 1).

The effect of ⎕FAPPEND is similar to ⎕FWRITE with a component number of 0 (or omitted).

⎕FREPLACE - Replace an existing component in a file

The ⎕FREPLACE function writes new data to an existing component in the file, or appends to the file. The syntax is:

```
          DATA ⎕FREPLACE TIENO COMPONENT {PASS}
```

DATA is any APL array or an overlay created using ⎕OV. TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS parameter.

The parameter COMPONENT is the component number at which you want to write the new data. This must be an integer in the range *Lowest current component number* to *Highest existing component number plus 1*.

The effect of ⎕FREPLACE is similar to that of ⎕FWRITE with an integral non-zero component number.

⎕FREAD - Read a component from a file

The ⎕FREAD function reads a component from the file. The syntax is:

```
          R ← ⎕FREAD TIENO COMPONENT {PASS}
```

TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS parameter.

The parameter COMPONENT is the component number you want to read. This must be an integer in the range *Lowest existing component number* to *Highest existing component number*.

The explicit result is the data (array or overlay) last written to that component by any user.

If the component number does not exist, APLX will generate an error COMPONENT NOT IN FILE (error code 20 in ⎕LER, or 6 3 in ⎕ET).

⎕FDELETE - Delete a component from a file

The ⎕FDELETE function deletes a component from the file, renumbering the remaining components accordingly. The syntax is:

```
          ⎕FDELETE TIENO COMPONENT {PASS}
```

TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS parameter.

The parameter COMPONENT is the component number you want to delete. This must be an integer in the range *Lowest current component number* to *Highest existing component number*. The component will be deleted, and any later components will be re-numbered so that they remain in integral sequence. For example, suppose the file currently has components numbered 1 to 5. If you delete component 3, then the old components 4 and 5 will be re-numbered 3 and 4 respectively. If you now delete component 1, the remaining components will be re-numbered 1 2 3 (corresponding to the original components 2 4 and 5).

See also the function ⎕FDROP which deletes components at the start or end of the file, but does not re-number the remaining components.

⎕FDROP - Drop components from the start or end the file

The ⎕FDROP function deletes one or more components from the start or end of the file, without renumbering the remaining components. The syntax is:

```
          ⎕FDROP TIENO N {PASS}
```

TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS parameter.

The parameter N is a positive or negative integer. If it is positive, the first N components of the file are deleted. If it is negative, the last -N components are deleted. In both cases, existing components are not re-numbered.

For example, suppose you have a file with components numbered from 1 to 12. After executing the two statements:

```
      ⎕FDROP TIENO,2
      ⎕FDROP TIENO, ¯3
```

the first component in the file will be component 3, and the last will be component 9. The original components 1, 2, 10, 11, 12 will no longer exist. The original components 3 through 9 still exist, and retain the same component numbers.

See also the function ⎕FDELETE which deletes a single component anywhere in the file, re-numbering the remaining components accordingly.

⎕FRESIZE - Set maximum file size

The ⎕FRESIZE function allows you to specify a maximum size for a file. The syntax is:

```
          NEWSIZE ⎕FRESIZE TIENO {PASS}
```

TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS parameter.

The left argument NEWSIZE is the maximum size (in bytes) to which the file will be allowed to grow. If NEWSIZE is 0, the file size is not limited by APLX, although the maximum size is still subject to any operating-system imposed limit and available disk space.

If you try to write to a file in a way which would cause the file size to exceed the limit, APLX will generate the error FILE ALLOCATION EXCEEDED.

The default is 0, meaning there is no limit.

⎕FRDCI - Read component information

The ⎕FRDCI function returns information about a component. The syntax is:

```
            R ← ⎕FRDCI TIENO COMPONENT {PASS}
```

`TIENO` is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the `PASS` parameter.

The parameter `COMPONENT` is the component number. This must be an integer in the range *Lowest existing component number* to *Highest existing component number*.

The explicit result is a three-element integer vector:

1. The free workspace required to read the component
2. The account number (`1↑⎕AI`) of the user who last wrote the component.
3. The time at which the component was last written, expressed in seconds since the start of the millennium (00:00:00, 1 Jan 2000)

`⎕FCSIZE` - Read component size information

The `⎕FCSIZE` function returns information about the size of a component. The syntax is:

```
            R ← ⎕FCSIZE TIENO COMPONENT {PASS}
```

`TIENO` is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the `PASS` parameter.

The parameter `COMPONENT` is the component number. This must be an integer in the range *Lowest existing component number* to *Highest existing component number*.

The explicit result is a two-element integer vector:

1. The size the variable would use if read into the workspace.
2. The size of the slot allocated to the component in the file, excluding the component header.

Both are expressed in bytes.

`⎕FSIZE` - Read file size and component-range information

The `⎕FSIZE` function returns information about the size of a file and the range of its components. The syntax is:

```
            R ← ⎕FSIZE TIENO {PASS}
```

`TIENO` is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the `PASS` parameter.

The explicit result is a five-element integer vector:

1. The number of the first component in the file
2. The next available component number (i.e. current highest + 1)
3. The file size currently used (including unused space), in bytes

4. The file size limit (set using ⎕FRESIZE) in bytes; 0 means unlimited
5. The approximate number of bytes of unused space that would be reclaimed by ⎕FDUP

As you write components to a file, and delete existing components, APLX attempts to make the freed-up space available for future use. However, in some cases the file may become fragmented, and the amount of unused space may increase over time. The fifth element of the result of ⎕FSIZE is useful in deciding whether to make a clean copy of the file (using ⎕FDUP), thus reclaiming the space.

⎕FRDFI - Read file information

The ⎕FRDFI function returns information about the creation and last update of a file. The syntax is:

```
          R ← ⎕FRDFI TIENO {PASS}
```

TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS parameter.

The explicit result is a 4 by 2 array of information about the file. The first column represents user numbers, and the second timestamps, expressed as seconds since the start of the millennium (00:00:00, 1 Jan 2000). The rows are:

1. User who created file, Timestamp file was created
2. Reserved (returns ¯1 ¯1)
3. Reserved (returns ¯1 ¯1)
4. User who last updated file, Timestamp file was last updated.

⎕FNUMS - Return file tie numbers in use

The niladic function ⎕FNUMS returns an integer vector of the tie numbers currently in use by this APL task.

⎕FNAMES - Return names of currently-tied files

The niladic function ⎕FNAMES returns a character matrix of the names of the files currently tied by this APL task, in the same order as the tie numbers returned by ⎕FNUMS. Names are padded to the right with blanks as necessary. The names are returned in the same form in which they were tied.

For example, under Linux you might have:

```
      'RUN3' ⎕FTIE 2
      '1 RUN4' ⎕FTIE 0
3
      '/home/jim/RUN5.aqf' ⎕FTIE 8
      ⎕FNUMS
2 3 8
      ⎕FNAMES
RUN3
1 RUN4
/home/jim/RUN5.aqf
```

⎕FLIB - Return names of component files in directory

⎕FLIB returns a character matrix of the names of the component files in a particular directory. Names are padded to the right with blanks as necessary. It takes a single argument, which can be either a library number (usually 0 to 9, corresponding to the rows of the ⎕MOUNT table), or a character string representing an operating-system path.

For example, under Windows you might have:

```
      ⎕FLIB 1
RUN4
JIM
COPY
      ⎕FLIB 'C:\TEMP'
RUN3
```

Only files which end in the `.aqf` extension will appear in the list. The extension is stripped from the names of the files returned.

⎕FHOLD - Hold/Release files for exclusive access

The function ⎕FHOLD is used for synchronizing access when multiple users and/or tasks are reading or writing the same file or files. It takes a single argument, which is usually a vector of tie numbers. If you need to supply pass numbers, the argument is a 2-row matrix of tie number/pass number pairs. The tie numbers should correspond to files you have share-tied using ⎕FSTIE, or should be an empty vector (meaning release all holds). The effect of ⎕FHOLD is as follows:

Firstly any existing file-holds belonging to this APL task are released. Secondly the system attempts to secure an exclusive lock on all the file numbers you have specified in the argument. Only one task can hold a given file at any time. If any of the files you try to hold are already held by another task, the operation waits until all files are available.

All file holds are automatically released when the APLX task reaches desk-calculator mode (this means you cannot experiment with ⎕FHOLD in desk-calculator mode), or when the APL task ends. For best performance in multi-user or multi-tasking applications, you should attempt to minimize the amount of time that your application holds files for exclusive use.

Technical Notes:

1. ⎕FHOLD is protected against deadlock. Supposing Task 1 tries to lock files A and B, and at the same time Task 2 tries to lock files B and A. You could get a situation where each task successfully locks the first file it tries, and then both tasks wait for ever for the second to be unlocked. APLX detects this situation, and automatically corrects for it by backing off (i.e. releasing any locks), waiting a random period, and then trying again.

2. ⎕FHOLD uses operating-system locks. Where you have files accessed over a network, some network systems do not honor locks correctly, particularly when mixing clients of different types (e.g. Windows and Linux tasks). We recommend testing the effect of ⎕FHOLD when implementing multi-user or multi-tasking file-systems across networks.

3. ⎕FHOLD may do nothing on some single-user versions of APLX, for example *APLX Personal Edition.*

⎕FDUP - Duplicate component file, reclaiming wasted space

The ⎕FDUP function makes a copy of an existing tied component file. Component ownership and timestamp information is retained in the copy of the file. Because the file is copied component-by-component, any wasted space caused by fragmentation of the original file is not reflected in the copy.

The syntax is:

```
          DESTNAME ⎕FDUP TIENO {PASS}
```

TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS parameter.

DESTNAME is a character vector specifying the name of the destination file. The name is specified in the same way as for ⎕FCREATE. If the name contains a directory-separator character (: / or \), it is treated as a full host path name (including the file extension, by convention "`.aqf`"). Otherwise it is treated as basic file name only, optionally preceded by a volume number 0 to 9, separated by one or more spaces from the name. In the latter case, the file is created in the directory 0 to 9 specified (the actual path is set the preferences dialog or ⎕MOUNT), and the file extension `.aqf` is added automatically.

Note that DESTNAME must not be the same as the original file name. If you want to achieve the effect of reclaiming space whilst keeping the file name unchanged, you should first make a copy using a temporary file name, then erase the original and rename the temporary file back to the original name.

⎕FRENAME - Rename component file

The ⎕FRENAME function allows you to rename a component file. You must have the correct host access permission to rename the file. If the file is currently tied, the name change will be reflected in the name list returned by ⎕FNAMES.

Two alternative syntax forms are supported for ⎕FRENAME:

```
          NEWNAME ⎕FRENAME OLDNAME
```
or
```
          NEWNAME ⎕FRENAME TIENO
```

OLDNAME is a character vector specifying the file to rename. The name is specified in the same way as for ⎕FCREATE. If the name contains a directory-separator character (: / or \), it is treated as a full host path name (including the file extension, by convention "`.aqf`"). Otherwise it is treated as basic file name only, optionally preceded by a volume number 0 to 9, separated by one or more spaces from the name. In the latter case, the file is created in the directory 0 to 9 specified (the actual path is set the preferences dialog or ⎕MOUNT), and the file extension `.aqf` is added automatically.

NEWNAME is a character vector specifying the new name of the file, specified in the same way.

The second syntax form is an alternative way to rename a file which has already been tied. TIENO is the tie number on which the file is tied.

Note that the host operating system may not allow you to rename a file across different file systems or physical disks.

⎕FERASE - Erase component file

The ⎕FERASE function allows you to erase a component file from the host file system. You must have the correct host access permission to erase the file. If the file is currently tied it will be untied before attempting to erase it.

The usual syntax of ⎕FERASE is :

```
      ⎕FERASE FILENAME
```

FILENAME is a character vector specifying the name of the file to erase. The name is specified in the same way as for ⎕FCREATE. If the name contains a directory-separator character (: / or \), it is treated as a full host path name (including the file extension, by convention "`.aqf`"). Otherwise it is treated as basic file name only, optionally preceded by a volume number 0 to 9, separated by one or more spaces from the name. In the latter case, the file is created in the directory 0 to 9 specified (the actual path is set the preferences dialog or ⎕MOUNT), and the file extension `.aqf` is added automatically.

For compatibility with some other APL interpreters which require the file to be tied before it can be erased, an alternate syntax for ⎕FERASE is supported :

```
     FILENAME ⎕FERASE TIENO
```

⎕FERROR - Return operating-system error information

When a component-file operation fails because the operating system reports an error, APLX usually generates a `FILE I/O ERROR` (error code 17 in ⎕LER, or `6 9` in ⎕ET). The niladic system function ⎕FERROR returns a character vector with further information (if available) from the operating system about what caused the error. For example:

```
      ⎕FLIB 'C:\JIM\REGIONS'
The system cannot find the path specified.
FILE I/O ERROR
      ⎕FLIB 'C:\JIM\REGIONS'
      ^
      ⎕ET
6 9
     ⎕FERROR
The system cannot find the path specified.
```

⎕FSTAC - Set access matrix

In multi-user versions of APLX, each user can be allocated a unique user number (shown by `1↑⎕AI)`. Individual APLX component files are tagged with a User Number, and have an associated File Access Matrix which indicates which users can access the file, what operations they may perform, and whether they need to specify a pass number to tie the file. Users will be allocated their user number by the logon procedure adopted by their system. Each user can thus 'own' a number of files, and can grant or deny access to these files.

The Access Matrix is three columns wide. The first column is a list of user numbers , with 0 being taken to mean ALL users. The second column is a list of integers which indicate the access privileges for the indicated user. The third column is the list of pass numbers which must be used by the given user to access the file with those rights (0 means no pass number is required). The access matrix may have a maximum of 19 rows. When a file is created, the default access matrix allows only the owner to access it, and grants the owner all rights, with no pass number.

When any operation on a file is attempted, APLX looks through the access matrix to find the first match for the user number (in the first column) and the pass number supplied when the file was tied (in the third column). A user number of 0 in the access matrix matches any user ID. If a match is found, the user is granted the permissions specified by the second element of the row. If no match is found, and the user is not the owner of the file, no permissions are granted. If no match is found, and the user is the owner of the file, all permissions are granted.

The access privileges can be specified in two ways. A positive privilege states what the user can do, and a negative privilege states what the user cannot do.

The privilege code is a number generated by adding various powers of 2 (1, 2, 4, 8, 16,....), each power of 2 corresponding to a particular privilege. Positive privilege codes are merely the sum of the individual privileges granted, whilst negative privilege codes are generated by adding `¯1` and the result of negating the sum of all the privileges denied.

Power of 2	*Value*	*Operation*
0	1	`⎕FREAD`
1	2	`⎕FTIE` *(exclusive)*
2	4	`⎕FERASE`
3	8	`⎕FAPPEND`
4	16	`⎕FREPLACE ⎕FWRITE`
5	32	`⎕FDROP ⎕FDELETE`
7	128	`⎕FRENAME`
9	512	`⎕FRDCI ⎕FCSIZE`
10	1024	`⎕FRESIZE`
11	2048	`⎕FHOLD`
12	4096	`⎕FRDAC`
13	8192	`⎕FSTAC`
14	16384	`⎕FDUP`

Permission to use `⎕FSTIE` (shared tie) is implicitly granted to any user who has any permission to use the file. `⎕FCREATE ⎕FLIB ⎕FNAMES ⎕FNUMS` and `⎕FUNTIE` do not need explicit APLX permissions (although the operating-system may restrict your rights). `⎕FSIZE` does not require explicit permission, but you must supply a pass number if the file was tied with one.

Examples of privilege codes are:

Privilege	*Meaning*
0	No access
1	`⎕FSTIE ⎕FREAD`
3	`⎕FSTIE ⎕FTIE ⎕FREAD`
17	`⎕FSTIE ⎕FREAD ⎕FREPLACE ⎕FWRITE`
¯33	Full access except for `⎕FDROP ⎕FDELETE`
¯1	Full access

`⎕FRDAC` - Read access matrix

The `⎕FRDAC` function returns the access matrix for a file. The syntax is:

```
          R ← ⎕FRDFI TIENO {PASS}
```

`TIENO` is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the `PASS` parameter.

The explicit result is the file's current N by 3 access matrix (which may be empty). See the description of `⎕FSTAC` for details.

`⎕FUNTIE` - Untie files

The function `⎕FUNTIE` unties zero, one or more currently-tied files. The syntax is:

```
            ⎕FUNTIE TIENOS
```

`TIENOS` is an integer vector of any length, containing the tie numbers of the files to be untied. If the tie number is not in use, no error is generated. You never need to supply a pass number to untie a file.

To untie all the files you currently have tied, use the expression:

```
      ⎕FUNTIE ⎕FNUMS
```

Files are automatically untied when the APL task ends. They are not automatically untied on `)CLEAR` or `)LOAD`.

The Grid object

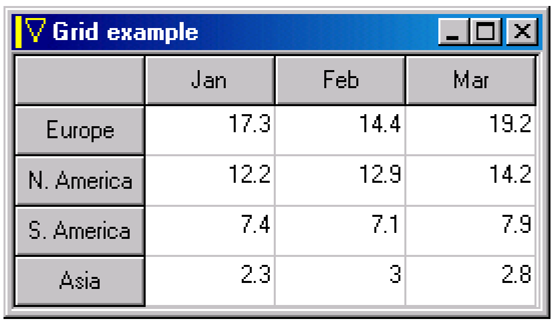
APLX Version 2 introduces a new `⎕WI` object class, the Grid. This is an object which allows the display, optionally with editing, of two-dimensional data in a form similar to that of a spreadsheet. By placing Grid objects on a Page object within a tabbed Selector, you can add further dimensions to the data.

The Grid is made up of a rectangular array of *cells*, which can contain either text or numeric data. At the top and left of the grid you can optionally have header (non-scrolling) cells.

In this example, we create a grid with 4 rows and 3 columns of scrolling (data) cells, and one row and column of header cells:

```
'W' ⎕WI 'New' 'Window' ('caption' 'Grid example')
'W.Grid' ⎕WI 'New' 'Grid' ('align' ¯1) ('rows' 4) ('cols' 3)
'W.Grid' ⎕WI 'text' (¯1) (⍳3) ('Jan' 'Feb' 'Mar')
'W.Grid' ⎕WI 'text' (⍳4) (¯1) ('Europe' 'N. America' 'S. America' 'Asia')
 DATA←4 3⍴17.3 14.4 19.2 12.2 12.9 14.2 7.4 7.1 7.9 2.3 3.0 2.8
'W.Grid' ⎕WI 'value' (⍳4) (⍳3) DATA
```

This produces the following effect:



	Jan	Feb	Mar
Europe	17.3	14.4	19.2
N. America	12.2	12.9	14.2
S. America	7.4	7.1	7.9
Asia	2.3	3	2.8

Cell Properties

As well as ordinary properties, a Grid object has properties which are associated with individual cells, known as *cell properties*. In the example above, we have used the `text` cell property to set the text within the header cells. We have also used the `value` cell property to set the contents of the data (scrolling) cells, in this case to be numbers. Cell properties are set with a special syntax:

<ControlName> `⎕WI` *<PropertyName>* *`Rows Cols Array`*

where Rows is a scalar or vector of the row indices you want to change, Cols is a scalar or vector of the column indices you want to change, and Array is an array of the data you want to assign to the cells. As with normal APL indexed assignment, the shape of Array must match the number of rows and columns specified, or be a scalar (in which case the data is assigned to all the cells specified). If either Rows or Cols has only one element, a vector can be used for Array instead of a one-row or one-column matrix.

The syntax for reading back cell properties is similar:

Result ← *<ControlName>* `⎕WI` *<PropertyName>* *`Rows Cols`*

The result will be an array of shape `(⍴Rows) (⍴Cols)`.

Rows and columns in the data (scrolling) region are numbered starting from 1. Rows and columns in the header (non-scrolling) region are numbered `¯1`, `¯2` etc, where `¯1` means the row or column nearest to the data area. In some cases, you can specify a row or column of 0, which means 'apply to the whole row/column'.

Using our example above:

```
     'W.Grid' ⎕WI 'Text' (1 2) (¯1)
 Europe
 N. America

      ⎕DISPLAY 'W.Grid' ⎕WI 'Text' (1 2) (¯1)
┌→──────────────┐
↓ ┌→─────┐      │
│ │Europe│      │
│ └──────┘      │
│ ┌→─────────┐  │
│ │N. America│  │
│ └──────────┘  │
└∊──────────────┘

      'W.Grid' ⎕WI 'value' (1 2) (1 3)
17.3 19.2
12.2 14.2
```

Numeric versus Text cells

Each cell can be of type Text or Numeric. Text cells contain simply character vectors; when you read either the **`text`** or **`value`** property of the cell, you get the text shown in the cell, possibly edited by the user, as a character vector.

Numeric cells contain numbers. If you read the **`value`** property of a Numeric cell, you get a numeric scalar, which is the current value of the cell. If the cell does not currently contain a valid number (because the user has edited it), you get a special numeric value to indicate an invalid number (by default this is a very large negative number; you can change it using the **`conversionerrorvalue`** property). If you read the **`text`** property of the cell, you get the actual text shown in the cell, which is usually the text representation of the cell's value, but may be different if it has been edited by the user and is not a valid value. You can also read the **`valid`** cell property, which tells you whether the cell contains a valid value.

With our example above:

```
       'W.Grid' ⎕WI 'valid' (1 2) (1 3)
1 1
1 1

      ⎕DISPLAY 'W.Grid' ⎕WI 'value' (1 2) (1 3)
┌→────────┐
↓17.3 19.2│
│12.2 14.2│                 (2 by 2 numeric array)
└~────────┘
```

```
      ⎕DISPLAY 'W.Grid' ⎕WI 'text' (1 2) (1 3)
┌→─────────────────┐
↓ ┌→───┐  ┌→───┐   │
│ │17.3│  │19.2│   │
│ └────┘  └────┘   │
│ ┌→───┐  ┌→───┐   │
│ │12.2│  │14.2│   │
│ └────┘  └────┘   │
└∊─────────────────┘
```

(2 by 2 nested array of character vectors)

Initially, all cells default to type Text. A cell becomes of type Numeric if you write a numeric value to its `value` property, or if you write a format string to its `format` property, which determines how it is formatted in the cell.

Properties of the Grid object

`style`: The `style` property is a set of flags which determine how the Grid behaves:

1	If set, user can select a range of cells *(Default: Off)*
2	If set, user can select a whole row *(Default: Off)*
4	If set, user can use the Tab key to move between cells *(Default: On)*
8	Reserved
16	If set, vertical scroll bar is shown if the Grid's contents are higher than the visible area *(Default: On)*
32	Reserved,
64	If set, horizontal scrollbar is shown if the Grid's contents are wider than the visible area *(Default: On)*
128	If set, user can re-size rows *(Default: Off)*
256	If set, user can re-size columns *(Default: On)*
512	If set, user can move rows to re-arrange their order *(Default: Off)*
1024	If set, user can move columns to re-arrange their order *(Default: Off)*
2048	If set, the contents of the grid update as the scroll-bar is moved *(Default: Off. Under MacOS, this flag is ignored since the grid always updates immediately).*

`borderstyle`: The `borderstyle` property is a set of flags which determine the appearance of the Grid (the default is all On):

1	Header cells have separator horizontal line
2	Header cells have separator vertical line
4	Data cells have separator horizontal line
8	Data cells have separator vertical line
16	Use 3D effect *(Windows and MacOS only, ignored under Linux)*

`autoeditstart`: (Integer) Determines whether the data in the Grid can be edited, and if so how the edit process starts. It can be one of:

0	Editing of a cell is possible only when the program calls the `Editstart` method.
1	Editing is automatically enabled when the cell is selected; the user can immediately type
2	The user must press Enter (or, under Windows and Linux, F2) to start editing a cell

The default is 1.

`color` or `colour`: (1, 2, 3 or 6 element numeric) The `color` or `colour` property for a Grid is set in the same way as for other controls. The foreground color you set determines the default color of the text shown in the Grid (you can change this for individual cells using the `colortext` cell property). The background color is used as the background for the data part of the Grid.

`colorhead` or `colourhead`: (1 or 3 element numeric) The `colorhead` or `colourhead` property for a Grid can be set as either a single RGB encoded integer (256 ⊥ Blue Green Red) or as three Red Green Blue values, each 0 to 255. It determines the background color used for header cells. The special value `¯1` means use the Grid control's foreground color. The special value `¯2` means use the default (light gray).

`font`: (Character vector or nested vector) The `font` for a Grid is set in the same way as for other controls. It determines the default font for the text shown in the Grid. You can change the font style for individual cells using the `fontstyle` cell property.

`conversionerrorvalue`: (Floating-point number) Value to return if a numeric cell cannot be converted to valid number. The default is a large negative number.

`gridlines`: (Boolean) Determines whether the grid lines within the data area are visible. (Default 1)

`headcols`: (Integer) The number of heading (fixed) columns (Default 1). You typically set this property when you create the grid (before setting the `cols` property).

`headrows`: (Integer) The number of heading (fixed) rows (Default 1). You typically set this property when you create the grid (before setting the `rows` property).

`cols`: (Integer) The number of data (scrolling) columns. You typically set this property when you create the Grid to fit the size of the data you want to display.

`rows`: (Integer) The number of data (scrolling) rows. You typically set this property when you create the Grid to fit the size of the data you want to display.

`col`: (Integer) The Column number (in index origin 1) of the currently-active cell. You can write to this property to change the current cell.

`row`: (Integer) The Row number (in index origin 1) of the currently-active cell. You can write to this property to change the current cell.

`activecell`: (2-element integer vector) The Row and Column of the currently-active cell. You can write to this property to change the current cell.

`text`: (Character vector) Although `text` is a cell property, as a convenience you can read (but not write to) `text` as a simple property of the Grid object. It returns the text of the currently-active cell.

`value`: (Character vector or numeric scalar) Although `value` is a cell property, as a convenience you can read (but not write to) `value` as a simple property of the Grid object. It returns the value of the currently-active cell. If the cell is a text cell, it will return a character vector. If it is a numeric cell, it will return the numeric value of the cell, or (if the cell is invalid) the current `conversionerrorvalue`.

`selection`: (4-element integer vector) The Row and Column of the top-left of the selected cell range, followed by the number of rows selected and the number of columns selected. If the `style` property is set so that only single-selection is possible (which is the default), the last two elements will both be one. You can write to this property to change the selection under program control.

`view`: (4-element integer vector) Determines the visible portion of the grid. The four elements are the First visible row, First visible column, Number of visible Rows, Number of visible columns. You can write to this property to scroll the grid under program control (only the first two elements are required).

`firstvisible`: (2-element integer vector) Determines the top-left position of the visible portion of the Grid. It is the same as the first two elements of the `view` property.

`imagesize`: (Read-only, 2-element numeric vector) Returns the total height and width, in the control's current scale, of all the cells in the Grid. This may be smaller than or larger than the height and width of the visible Grid control itself.

Cell properties of the Grid object

These are all set using the syntax:

<ControlName> `⎕WI` *<PropertyName>* *`Rows Cols Array`*

and read using the syntax:

Result `←` *<ControlName>* `⎕WI` *<PropertyName>* *`Rows Cols`*

`value`: (Each cell element is a character vector or numeric scalar) Read or set cell values.

When you read this property, it returns a character vector for each Text cell and a numeric scalar for each Numeric cell, for each cell in the list of rows and columns specified. If a Numeric cell is invalid, it will return the current `conversionerrorvalue`. If you read a set of cells some of which are Numeric and some Text, it will return a nested array containing a mixture of character vectors and numeric scalars. If you read a set of cells all of which are Numeric, it will return a simple numeric array.

When you write to this property, it sets the value for each cell in the list of rows and columns specified. If you supply a number for a cell, that cell becomes a Numeric cell. If you supply a character vector or scalar, it becomes a Text cell. Writing to a cell also causes its `valid` property to be set to 1.

`text`: (Each cell element is a character vector) Read or set cell text.

When you read this property, it returns a character vector containing the text displayed in the cell, irrespective of the cell's type, for each cell in the list of rows and columns specified. For multiple cells, it will therefore return a nested array of character vectors.

When you write to this property, the text displayed in the cell is set to the text you supply. To set multiple cells is one operation, provide a nested array of character vectors. The type of the cell, and the cell's `valid` property, are not changed. If the cell is Numeric, it will remain so, and its `value` will not change.

`valid`: (Each cell element is a Boolean scalar, read-only) Returns 1 if the cell is valid, and 0 if it is invalid. Text cells are always valid. Numeric cells may be invalid if the user has edited the text.

`allowselection`: (Each cell element is a Boolean scalar) If this is set to 1 (default), the cell can be selected. If it is set to 0, the cell cannot be selected.

`textalign`: (Each cell element is an integer scalar) Determines how the text is aligned within the cell. The value is the sum of two numbers. The horizontal alignment is 1 for left, 2 for right, or for 4 center. The vertical alignment is 8 for top, 16 for bottom, or 32 for center. The default value depends on the type of the cell. Header cells default to centered in both directions (4 + 32 = 36). Text cells in the data area default to top, left (1 + 8 = 9). Numeric cells default to top, right (2 + 8 = 10).

`colortext` or `colourtext`: (Each cell element is an integer scalar) Determines the color of text in the cell. The value is a color expressed as a numeric scalar (256 ⊥ Blue Green Red). The special value of means `¯1` use the Grid foreground color for this cell; this is the default.

`colorback` or `colourback`: (Each cell element is an integer scalar) Determines the background color for the cell (when it is not selected). The value is a color expressed as a numeric scalar (256 ⊥ Blue Green Red). The special value of means `¯1` use the Grid background color for this cell; this is the default.

`fontstyle`: (Each cell element is an integer scalar) Determines the style of the text within the cell. The value is the sum of:

0	Plain
1	Bold
2	Italic
4	Underlined
8	Hollow (supported under MacOS only)
16	Strikeout (not supported under MacOS)

The special value of `¯1` means use the default font style of the Grid.

`format`: (Each cell element is a character vector) Determines the format used to display numeric values. Setting this value automatically sets the cell type to Numeric. The format string is a character vector of up to three parts, where each part is separated by a semicolon. If you supply just one part, it applies to all numbers. If you supply two parts, the first applies to positive numbers or zero, and the second to negative numbers. If you supply all three parts, the first applies to numbers greater than zero, the second to less than zero, and the third to zero.

Each part comprises a set of characters giving the format for the number. These comprise:

0 If the number has a digit in the position where the '0' appears in the format, then that digit is displayed, otherwise 0 is displayed. Thus, the positions of the leftmost '0' before the decimal point and the rightmost '0' after the decimal point in the format determine the range of digits that are always displayed.

If the number has a digit in the position where the '#' appears in the format, then that digit is displayed. Otherwise, nothing is displayed in that position.

. Decimal point. The first '.' character in the format string determines the location of the decimal separator in the display. (The actual character used as a the decimal separator may depend on the locale set in the operating system).

, Thousands separator. If the format string contains one or more ',' characters, the output will have thousand separators inserted between each group of three digits to the left of the decimal point. The placement and number of ',' characters in the format string does not affect the output, except to indicate that thousand separators are wanted. (The actual character used may depend on the locale set in the operating system.)

E+ Scientific notation. If any of the strings 'E+', 'E-', 'e+', or 'e-' are contained in the format string, the number is formatted using scientific notation. A group of up to four '0' characters can immediately follow the 'E+', 'E-', 'e+', or 'e-' to determine the minimum number of digits in the exponent. The 'E+' and 'e+' formats cause a plus sign to be output for positive exponents and a minus sign to be output for negative exponents. The 'E-' and 'e-' formats output a sign character only for negative exponents.

`' "` Characters enclosed in single or double quotes are output unchanged.

Row and column sizes

You can set the size of the rows and columns using the `rowsize` and `colsize` properties, as follows:

```
<ControlName> ⎕WI 'rowsize' Rows Sizes
<ControlName> ⎕WI 'colsize' Cols Sizes
```

where Rows or Cols is a scalar or vector list of row/column numbers, and Sizes is a matching vector (or scalar) of the row/column sizes in pixels. A row or column number of 0 sets the default size for all rows or columns which are not explicitly set.

You can read back the current sizes using the syntax:

```
Result ← <ControlName> ⎕WI 'rowsize' Rows
Result ← <ControlName> ⎕WI 'colsize' Cols
```

Note that, depending on the flags you have set in the `style` property, the user may be able to resize rows and columns.

Methods for the Grid object

`EditStart`: This method takes no arguments. It initiates an editing session on the currently-active cell. It is useful mainly if you have set the `autoeditstart` property to 0. For example, you might initiate an edit session for certain cells only (in response to an `onSelChange` event).

`DeleteCols` and `DeleteRows`: These methods take an argument which is a vector of rows or columns to be deleted. After they have been deleted, the remaining rows or columns are re-numbered to form an integer sequence starting at 1.

`InsertCols` and `InsertRows`: These methods take an argument which is a vector of positions at which you wish to insert new rows or columns. You can specify integers (for example 5 means ‘insert before the current row/column 5’), or non-integers (for example 4.5 means ‘insert between the current row/column 4 and 5’). After the insertion, the rows or columns are re-numbered to form an integer sequence starting at 1. Thus, if you insert at 5 and 6, the new rows/columns become numbers 5 and 7. To insert three new rows/columns before the current second one, specify 2 2 2.

Callbacks for the Grid object:

`onChange`: This event is triggered when the user has edited the contents of a cell. The event-specific parameters in `⎕EV` are (in index origin 1):

```
⎕EV[6]         Row number of the edited cell
⎕EV[7]         Column number of the edited cell
⎕EV[8]         Flag to indicate whether the contents were valid
```

`onColMoved`: This event is triggered when the user has moved a column by dragging it to a different position in the grid. (This is possible only if the **`style`** property includes the flag 1024, column moving allowed). The event-specific parameters in `⎕EV` are (in index origin 1):

```
⎕EV[6]         Column number before the move
⎕EV[7]         The position of the column after the move
```

`onRowMoved`: This event is triggered when the user has moved a row by dragging it to a different position in the grid. (This is possible only if the **`style`** property includes the flag 512, row moving allowed). The event-specific parameters in `⎕EV` are (in index origin 1):

```
⎕EV[6]         Row number before the move
⎕EV[7]         The position of the row after the move
```

`onScroll`: This event is triggered when the user has scrolled the Grid control.

`onSelChange`: This event is triggered when the user selects a new cell. The event-specific parameters in `⎕EV` are (in index origin 1):

```
⎕EV[6]         Row number of the newly-selected cell
⎕EV[7]         Column number of the newly-selected cell
```

Other GUI programming enhancements: `⎕WI`

- The new read-only property **`fonts`** applies to the System object. It returns a nested vector containing the names of the screen fonts installed on the system:

  ```
        ⊃'#' ⎕WI 'fonts'
  Abadi MT Condensed
  Abadi MT Condensed Extra Bold
  Abadi MT Condensed Light
  Algerian
  American Uncial
  Andy
  APL385 Unicode
  APLX Upright
  Arial
  ```
  ... *etc*

  (Note that the fact that a font is installed does not guarantee that it is available in a particular size, style and character set).

  Under Windows only, the **`fonts`** property also applies to the Printer object. It returns a nested vector of the names of the fonts installed on the currently-selected printer, together with the TrueType fonts which can be downloaded to the printer:

  ```
        'Prn' ⎕WI 'New' 'Printer'
        ⊃'Prn' ⎕WI 'fonts'
  Goudy
  Carta
  Optima
  Oxford
  Bodoni
  Geneva
  Monaco
  Tekton
  ```
  ... *etc*

- The new **`firstvisible`** property applies to the List object. It is an integer scalar which represents the first visible item at the top of the list box. Thus, if the list box has been scrolled down three lines, it will have the value 4. You can write to this property to force a scroll so that a specific item is at the top of the visible portion of the list; however, because you the list will not scroll beyond the last item, you should read back the property if you need to know exactly which item ended up as the first visible item.

- The Combo object now supports the **`selection`** and **`seltext`** properties. These are similar to the equivalent properties for an Edit object. If the user has selected text within the Edit portion of the Combo, the **`selection`** property will have the start position and length of the selection, and **`seltext`** will contain the selected text. *(This applies to Windows and Linux only, since Combo boxes under MacOS do not include an editable field).*

- You have always been able to associate a single arbitrary APL array (or overlay) with any object by using the **`data`** property. APLX now also supports 'delta' properties for all objects. Any property name which starts with the delta character ∆ can be used to store your own data in the object. Subject to available memory, you can have as many such properties as you like, and you can store any simple or nested array, or overlay, in the property. Referencing a delta property to which no data has been assigned generates a `VALUE ERROR`, just like a standard APL variable:

```
      'W' ⎕WI 'New' 'Window'
      'W' ⎕WI '∆TYPE' 'SCATTER'
      'W' ⎕WI '∆VALS' 13.4 66.7 12.2
      'W' ⎕WI '∆TYPE'
SCATTER
      'W' ⎕WI '∆VALS'
13.4 66.7 12.2
      'W' ⎕WI '∆XPOS'
VALUE ERROR
      'W' ⎕WI '∆XPOS'
      ^
```

- The new `unicode` property of the System object allows you to access the clipboard as Unicode. See the separate section on Unicode support for full details.

- The `Draw` method has been enhanced to support the `Seg` keyword. This allows you to draw a number of connected lines (without polygon fill) in one operation:

  <ControlName> `⎕WI 'Draw' 'Seg'` *`Array [Array, Array...]`*

  The argument comprises one or more M by N matrices of: Y1 X1 Y2 X2... Each row of the matrix corresponds to one set of connected lines. Points are taken in pairs from each row, so N must be even, and the sets of lines in each array must have the same number of segments.

  This method is mainly provided for compatibility with APL*Plus/Win. You can achieve the same effect in a more flexible manner by supplying a nested argument to the `Line` keyword.

- The `Draw` method has been enhanced to support the `TextSize` (or `?Text`) keyword. This returns the size which is required to draw one or more strings in the current font:

  <ControlName> `⎕WI 'Draw' 'TextSize'` *`String [String, String...]`*

  The argument comprises one or more character scalars, vectors or matrices.   If only one string is supplied, the result is a 2-element numeric vector containing the height and width required to draw the string with the currently-selected font, in the current Draw scale. If more than one string is supplied, the result is a nested vector with one element per string. Each element is a two-element vector giving the height and width for the corresponding string.

  If a string is a matrix, or is a vector containing carriage-return, line-feed or backspace characters, the size returned will reflect this.

- *(Windows only)* The OCX interface now allows you to use numeric ‘LPDISPATCH’ handles which may be returned by some OCX/ActiveX controls or OLE Servers. This usually applies in the case where you create a new object in the external control, and need some way to refer to it. Windows returns a ‘handle’, which in APL is represented as an integer. You can now use the text representation of the integer as part of a hierarchical property or method name to access the newly-created object.

  For example, you can create a new Word document using the Add method of the Word application. This returns a handle. In APLX, you can now use this as follows:

```
      'W' ⎕WI 'New' 'Word.Application'
      MYDOC←⍎'W' ⎕WI 'Documents.Add'
      MYDOC
1797860
      'W' ⎕WI MYDOC,'.Words.Count'
1
```

Support for Unicode characters

APLX Version 2 includes a number of new features to provide support for Unicode text.

Unicode is a worldwide standard which encodes characters in two bytes, thus providing a total of 65,536 possible characters. This is enough to represent all the international and special characters used in modern computer applications. APL characters, including all those used in APLX, are defined in the Unicode standard, although there are some ambiguities about a few of them (see Adrian Smith's editorial and article in Vector 19.3 for a discussion of this). This makes it possible to exchange character data between APLX and other applications (including other APL interpreters which support Unicode), without encountering problems with character translation and 'code tables', provided that the text being transferred can be represented in the APLX character set.

The support provided for Unicode in APLX Version 2 is as follows:

- New menu items 'Copy as Unicode' and 'Paste as Unicode' have been added to Edit menu. The first of these copies text to the clipboard, representing it in the 16-bit Unicode standard rather than in the 8-bit APLX-specific character set. The second of these pastes text from the clipboard into the current APLX window, translating it from Unicode to the APLX internal representation. You can use these to exchange text (including international and APL characters) with non-APL applications which support Unicode, and also with other APL interpreters which provide a similar facility. For example, you can copy the text of an APLX function to the clipboard as Unicode, paste it into Notepad, and display it correctly using an APL-aware Unicode font such as Adrian Smith's 'APL385 Unicode' font (downloadable from http://www.vector.org.uk).

- The new system function ⎕UCS translates Unicode values to the equivalent character in the APLX character set (if there is one), and vice versa. It takes a right argument, which must be a simple character or integer array.

  If the argument is a character array, ⎕UCS returns an integer array of the same shape, containing the Unicode representation of each character. This will be a number in the range 0 to 65535.

  If the argument is an integer array, it must contain only numbers in the range 0 to 65535. ⎕UCS returns a character array of the same shape, containing the APLX character corresponding to each Unicode value provided. Unicode values which have no equivalent in the APLX character set are converted to a question mark.

  Examples:

```
      ⎕UCS 'X←⍳10'
88 8592 9075 49 48
      ⎕UCS 88 8592 9075 49 48
X←⍳10
      ⎕UCS 937 8364 209
?€Ñ
```

  In the last example, the Unicode value 937 (hex 03A9, representing the Greek capital omega character Ω) was translated to a question mark because it has no equivalent in the APLX character set. See below for a more detailed discussion of the character mapping.

- The new `unicode` property of the System object allows your program to exchange Unicode text with the clipboard. When you read the property, any Unicode text in the clipboard is returned as a vector of integers. (You can translate it to the APLX internal text representation by using `⎕UCS`). You can write either a vector of integers, each representing a valid Unicode character in the range 0 to 65535, or a character vector. If you write a character vector to this property, it is translated to Unicode and placed on the clipboard.

- The native file functions `⎕NREAD` and `⎕NWRITE` now support additional data-conversion types for reading and writing Unicode text files. When reading, the conversion type `5` will cause 16-bit Unicode characters to be read from the file, and converted to the equivalent APLX character using the same rules as described above for `⎕UCS`. (Each Unicode character takes two bytes in the file). When writing, the conversion type `5` will cause APLX characters to be written out as two-byte Unicode values. The conversion value `¯5` is the same, except the Unicode values are byte-reversed.

  Note that, by convention, Unicode plain-text files start with a 'byte-order' mark. This is the special hex value FEFF, represented as a two-byte value in the byte-ordering used to create the file. Thus, on 'big-endian' systems such as the Macintosh, the first two bytes of the file will normally be hex FE and FF (decimal 254 and 255). On a 'little-endian' system such as Windows or x86 Linux, numbers are represented backwards so the first two bytes will normally be FF and FE. You can use this information to determine whether to use the conversion type `5` or `¯5` when reading the contents of a Unicode text file, by reading the first element of the file as a 16-bit integer (conversion code 163 for `⎕NREAD`). If you get the value 65279 (hex FEFF), the Unicode file was written using the same byte-ordering as the machine you are running on, so no byte reversal is required and you can use conversion code `5` to read the Unicode characters from the remainder of the file. If you get the value 65534 (hex FFFE), the Unicode file was written using the opposite byte-ordering convention to that of the machine you are using, so you need to use conversion code `¯5`. For example:

```
      'c:\temp\uni.txt' ⎕NTIE 1     ⍝ Open a Unicode text file
      ⎕NREAD 1 163 1 0              ⍝ Read first two bytes as 16-bit integer
65279                               ⍝ This is the correct value for hex FEFF
      ⎕AF 4 ⎕DR 65279
0 0 254 255
      TEXT←⎕NREAD 1 5 ¯1            ⍝ Read the remainder of the file as Unicode
      ⎕NUNTIE 1
```

Unicode character set mapping used in APLX

The Unicode values used when translating APLX characters to Unicode follow the standard described by Adrian Smith in Vector 19.3 (January 2003). The full set of characters, and the values used when mapping from APLX characters to Unicode, are shown in the Appendix. The following points should be noted about the mapping from APLX characters to Unicode:

- The APLX quad symbol ⎕ is mapped to hex 2395, rather than the square shape in the 'Geometric Shapes' subset at 25AF.
- The APLX diamond statement-separator ⋄ is mapped to hex 22C4, rather than the diamond shape in the 'Geometric Shapes' subset at 25CA.
- The APLX tilde ~ ('Not'/'Without') is mapped to the ASCII tilde at 007E, not the 'APL tilde' at 223C. This is to ensure that tilde characters can be exchanged properly with non-APL applications. The same approach is taken with And ∧ Star * and Remainder |.

There may be several different Unicode characters which look similar or provide a similar function to an APLX character. In order to maximise the probability of being able to represent the text when converting from Unicode to internal representation, APLX accepts as input a number of alternative Unicode values for certain characters. These are shown in Appendix 2.

Appendix 1: Mapping from APLX characters to Unicode

Note: Non-printing control characters are not shown.

Character	*Position in* ⎕AV *(hex)*	*Unicode value (hex)*	*Description*
⍐	07	2350	File hold
	08	0008	Backspace
⍗	09	2357	File drop
	0A	000a	Line feed
⍇	0B	2347	File read
⍈	0C	2348	File write
	0D	000d	Carriage-return, Newline
⍱	0E	2371	Nor
⍲	0F	2372	Nand
⍒	10	2352	Grade down
⍋	11	234b	Grade up
⌽	12	233d	Reverse
⍉	13	2349	Transpose
⊖	14	2296	Circle bar
⍟	15	235f	Log
⌶	16	2336	I-Beam
⍫	17	236b	Del-Tilde
⍎	18	234e	Execute
⍕	19	2355	Format
⍀	1A	2340	Slope-Bar
⌿	1B	233f	Slash-Bar
⍝	1C	235d	Lamp
⍞	1D	235e	Quote-Quad
!	1E	0021	Exclamation
⌹	1F	2339	Domino
	20	0020	Space
¨	21	00a8	Dieresis, Each
)	22	0029	Right parenthesis
<	23	003c	Less than
≤	24	2264	Not greater than
=	25	003d	Equal
>	26	003e	Greater
]	27	005d	Right bracket
∨	28	2228	Or
^	29	005e	And
≠	2A	2260	Not equal
÷	2B	00f7	Divide
,	2C	002c	Comma
+	2D	002b	Plus
.	2E	002e	Period
/	2F	002f	Slash
0	30	0030	0
1	31	0031	1
2	32	0032	3
3	33	0033	3
4	34	0034	4
5	35	0035	5
6	36	0036	6
7	37	0037	7
8	38	0038	8
9	39	0039	9
(	3A	0028	Left parenthesis
[	3B	005b	Left bracket
;	3C	003b	Semi-colon
×	3D	00d7	Multiply
:	3E	003a	Colon
\	3F	005c	Slope

¯	40	00af	High minus
⍺	41	237a	Alpha
⊥	42	22a5	Decode
∩	43	2229	Cap
⌊	44	230a	Floor
∊	45	220a	Epsilon
_	46	005f	Underbar
∇	47	2207	Del
{	48	007b	Left brace
⍳	49	2373	Iota
∘	4A	2218	Jot
'	4B	0027	Single quote
⎕	4C	2395	Quad
|	4D	007c	Stile, Remainder
⊤	4E	22a4	Encode
○	4F	25cb	Circle
*	50	002a	Star
?	51	003f	Query
⍴	52	2374	Rho
⌈	53	2308	Ceiling
~	54	007e	Not
↓	55	2193	Drop
∪	56	222a	Cup
⍵	57	2375	Omega
⊃	58	2283	Right shoe
↑	59	2191	Take
⊂	5A	2282	Left shoe
←	5B	2190	Left arrrow
⊢	5C	22a2	Right tack
→	5D	2192	Right arrow
≥	5E	2265	Not less than
-	5F	002d	Minus
⋄	60	22c4	Diamond
A	61	0041	A
B	62	0042	B
C	63	0043	C
D	64	0044	D
E	65	0045	E
F	66	0046	F
G	67	0047	G
H	68	0048	H
I	69	0049	I
J	6A	004A	J
K	6B	004B	K
L	6C	004C	L
M	6D	004D	M
N	6E	004E	N
O	6F	004F	O
P	70	0050	P
Q	71	0051	Q
R	72	0052	R
S	73	0053	S
T	74	0054	T
U	75	0055	U
V	76	0056	V
W	77	0057	W
X	78	0058	X
Y	79	0059	Y
Z	7A	005A	Z
∆	7B	2206	Delta
⊣	7C	22a3	Left tack
⍪	7D	236a	Comma bar
$	7E	0024	Dollar
}	7F	007d	Right brace

┌	90	250c	Line draw top left
┐	91	2510	Line draw top right
└	92	2514	Line draw bottom left
┘	93	2518	Line draw bottom right
─	94	2500	Line draw horizontal
│	95	2502	Line draw vertical
┼	96	253c	Line draw cross
├	97	251c	Line draw join right
┤	98	2524	Line draw join left
┴	99	2534	Line draw join up
┬	9A	252c	Line draw join down
	9B	001B	Escape
	9C	001C	[Spare]
Í	9D	00cD	I acute
	9E	001E	[Spare]
	9F	001F	[Spare]
"	A0	0022	Double-quote
#	A1	0023	Hash
%	A2	0025	Percent
&	A3	0026	Ampersand
@	A4	0040	At
£	A5	00A3	Pound
`	A6	0060	Backquote
≡	A7	2261	Match
≢	A8	2262	Not match
⍷	A9	2377	Epsilon underbar
⍸	AA	2378	Iota underbar
⌻	AB	233b	Quad-Jot
⍂	AC	2342	Quote-Slope
⍤	AD	2364	Jot dieresis
⍥	AE	2365	Circle dieresis
⌷	AF	2337	Squad
Ä	B0	00c4	A umlaut
Å	B1	00c5	A ring
Ç	B2	00c7	C cedilla
É	B3	00c9	E acute
Ñ	B4	00d1	N tilde
Ö	B5	00d6	O umlaut
Ü	B6	00dc	U umlaut
á	B7	00e1	a acute
à	B8	00e0	a grave
â	B9	00e2	a circumflex
ä	BA	00e4	a umlaut
ã	BB	00e3	a tilde
å	BC	00e5	a ring
ç	BD	00e7	c cedilla
é	BE	00e9	e acute
è	BF	00e8	e grave
ê	C0	00ea	e circumflex
ë	C1	00eb	e dieresis
í	C2	00ed	i acute
ì	C3	00ec	i grave
î	C4	00ee	i circumflex
ï	C5	00ef	i dieresis
ñ	C6	00f1	n tilde
ó	C7	00f3	o acute
ò	C8	00f2	o grave
ô	C9	00f4	o circumflex
ö	CA	00f6	o umlaut
õ	CB	00f5	o tilde
ú	CC	00fa	u acute
ù	CD	00f9	u grave
û	CE	00fb	u circumflex
ü	CF	00fc	u dieresis

À	D0	00c0	A grave
Ã	D1	00c3	A tilde
Õ	D2	00d5	O tilde
	D3	0152	OE
	D4	0153	oe
Æ	D5	00c6	AE
æ	D6	00e6	ae
	D7	0000	[Spare]
Ø	D8	00d8	O /
ø	D9	00f8	o /
¿	DA	00bf	Inverted ?
¡	DB	00a1	Inverted !
β	DC	00df	Beta
ÿ	DD	00ff	y dieresis
	DE	0000	[Spare]
	DF	0000	[Spare]
	E0	0000	[Spare]
a	E1	0061	a
b	E2	0062	b
c	E3	0063	c
d	E4	0064	d
e	E5	0065	e
f	E6	0066	f
g	E7	0067	g
h	E8	0068	h
i	E9	0069	i
j	EA	006A	j
k	EB	006B	k
l	EC	006C	l
m	ED	006D	m
n	EE	006E	n
o	EF	006F	o
p	F0	0070	p
q	F1	0071	q
r	F2	0072	r
s	F3	0073	s
t	F4	0074	t
u	F5	0075	u
v	F6	0076	v
w	F7	0077	w
x	F8	0078	x
y	F9	0079	y
z	FA	007A	z
⍙	FB	2359	Delta underbar
È	FC	00c8	E grave
€	FD	20ac	Euro
	FE	0000	[Spare]
	FF	007f	Rubout

Appendix 2: Alternative Unicode characters

As well as the Unicode characters shown in Appendix 1, APLX also accepts the following alternative characters when mapping from Unicode to internal representation:

Unicode character (hex)	*Unicode meaning*	*Mapped to*
00a6	Broken vertical bar	| Remainder
2223	'Divides' in 'Mathematical operators'	| Remainder
2227	Logical And in 'Mathematical operators'	∧ And
223c	Tilde in 'Mathematical operators'	∼ Not
22c6	Star in 'Mathematical operators'	⋆ Star
2013	En dash	- Minus
2212	Minus in 'Mathematical operators'	- Minus
2044	'Fraction slash'	/ Slash
2215	'Division slash'	/ Slash
25AF	Square in 'Geometric shapes'	⎕ Quad
25ca	Diamond in 'Geometric shapes'	⋄ Diamond
203e	Overline	¯ High minus
2019	Left quote	' Single quote
2019	Right quote	' Single quote
201c	Left double quote	" Double quote
201d	Right double quote	" Double quote
03b1	Greek alpha	⍺ Alpha
03b2	Greek beta	β Beta-S
03b5	Greek epsilon	∊ Epsilon
03b9	Greek iota	⍳ Iota
03c1	Greek rho	⍴ Rho
03c9	Greek omega	⍵ Omega
2028	Line-separator character	Carriage-return
2029	Paragraph-separator character	Carriage-return