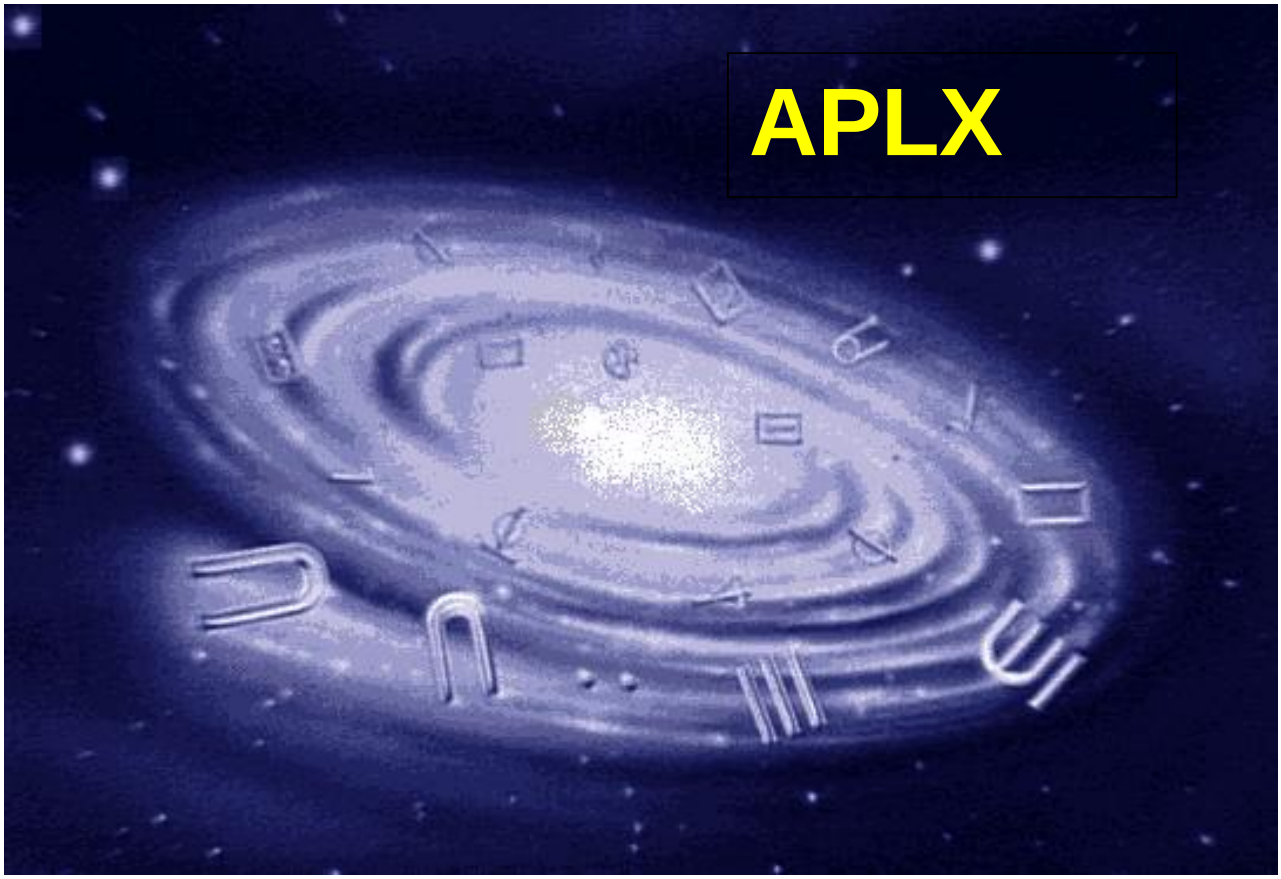


APLX64 and the Client-Server Architecture

MICRO APL

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

APLX64 Version 1.0 July 2006

Introduction to APLX64

Overview

APLX64 is a fully 64-bit version of APLX, which will overcome all interpreter-imposed limitations in workspace and array size for the foreseeable future. It is currently available for Linux and Windows.

In APLX64, all array dimensions are 64-bit, so there are effectively no limits other than available memory on the number of elements in any array, or the maximum length along any dimension. Integers are also 64-bit (otherwise you would have to use floating-point numbers to index arrays!). Booleans remain as one bit per element, making it possible to handle huge Boolean arrays without excessive memory requirements. Huge native file and component files are of course fully supported, and APLX64 includes built-in 64-bit access to SQL databases.

Key features

- Full 64-bit APL interpreter
- 64-bit integer type
- No limit on workspace size other than operating-system maximum (up to a theoretical 8,589,934,592 GB)
- Available in desktop editions and in client-server editions
- Fully compatible with the 32-bit APLX product range
- Automatically converts 32-bit workspaces on)LOAD
- Share component files with 32-bit APLX implementations
- Full user-friendly development environment and built-in charting
- Full GUI programming capability
- 64-bit access for native files and databases

The Client-Server Architecture

Internally, the APLX64 product comprises two separate programs. The 64-bit APL interpreter itself runs as a 64-bit application (called `aplx64_server` on Linux, or `aplx64_server.exe` on Windows). This is the *Server*. The front-end, which is the program you use to edit, run, and debug APL workspaces, and which implements all of the user-interface elements and `⎕WI`, is a 32-bit program (called `APLX.exe` on Windows). This is the *Client*. The Client and the Server can run on the same physical machine, or on separate machines connected by a TCP/IP network. Typically, the Client runs on a desktop Windows system, and the Server runs on a Windows or Linux server system, although other combinations are possible.

A given Server can support any number of Clients (each of which may be running more than one APL session on the Server), subject to having sufficient memory and CPU resources and the license agreement in force. Also, a given Client can connect to multiple Servers, so you can run several 64-bit sessions simultaneously on different servers.

As well as the 64-bit interpreter, APLX64 also includes a 32-bit version of the interpreter, which is part of the Client program. This allows you to develop and test 32-bit APLX applications as well as full 64-bit applications. A given Client can run both 32-bit and 64-bit APL sessions simultaneously.

Note: APLX64 Server Edition also includes a 'dumb-terminal' version of the 64-bit interpreter, as a simple 64-bit console application . This can be used from a `telnet` session, or can take input from `stdin` and send output to `stdout`, so it is useful for running scripts, batch jobs, and web-server applications.

Communication between the Client and Server

The Client and the Server communicate with each other using the TCP/IP network protocol (this is true even if they are physically on the same machine). The official IANA port number allocated to APLX is 1134.

Security and Firewalls

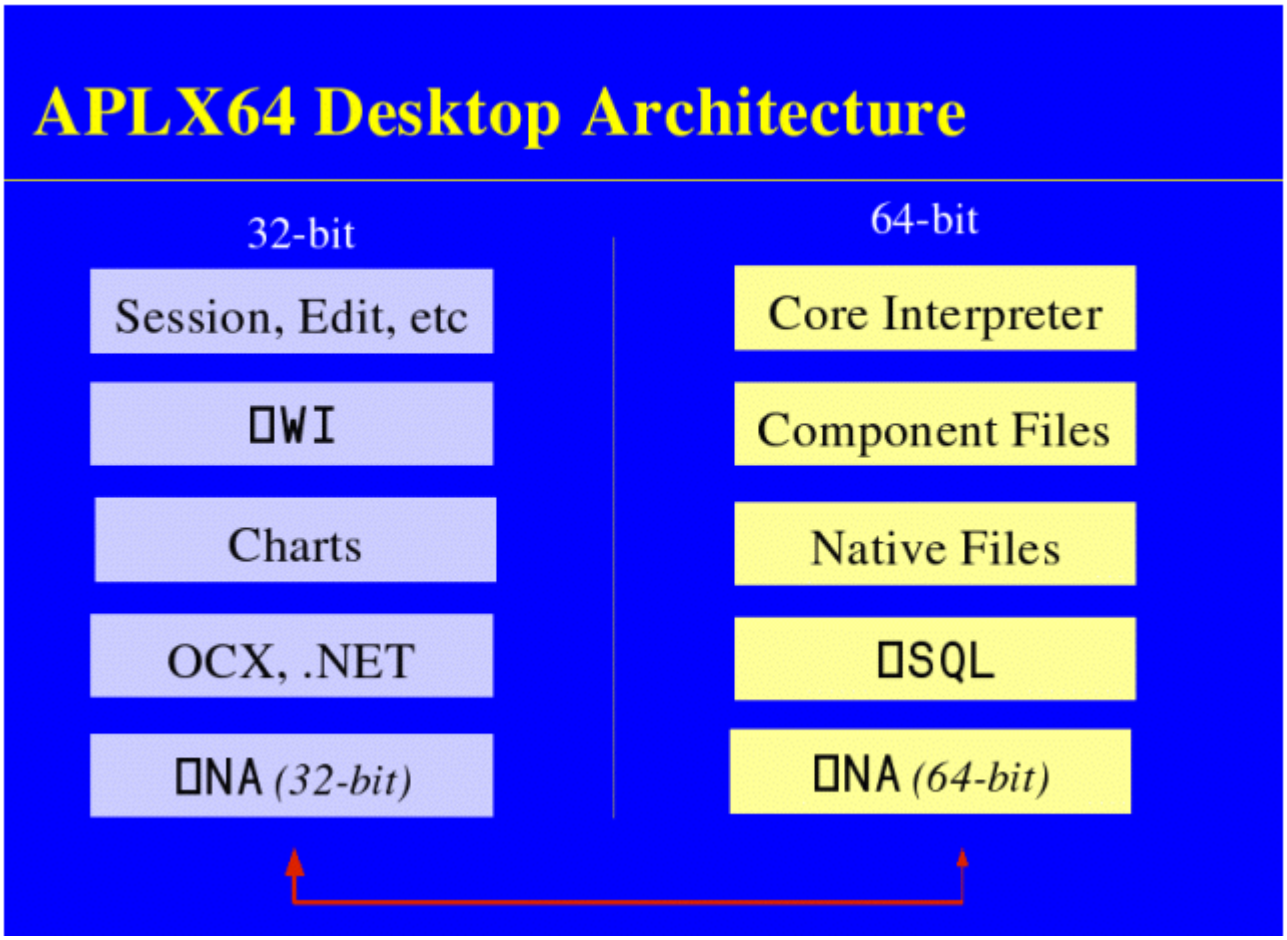
The network communication used by the APLX client-server architecture is not encrypted, and could in theory be snooped on, or used to run malicious APL code over the network. For this reason, we strongly recommend that the Server should be protected by a firewall so that it is not exposed to attacks from untrusted sites. The firewall should normally be set up to disallow all traffic on port 1134 except between the Server and authorised Client machines on an internal network.

It is possible to run the Client remotely from the Server (for example, for an employee to run the Client on a machine at his or her home, accessing the Server in the corporate data centre over the internet). However, the only safe way to do this is to use a secure VPN (Virtual Private Network), which has been correctly set up to fully protect traffic between the two machines.

Running APLX64 on a 64-bit Windows Desktop system

If you have purchased APLX64 Desktop Edition, the Client and the Server run on the same machine, usually under Windows XP64 or Vista. (The license agreement disallows connecting to the Server from different machines). When you start the Client program, normally the Server program is started automatically, so the fact that there are two separate programs running is transparent to the user. When the last APL session ends and the Client program exits, the Server program will also terminate automatically.

The first time you run APLX64, the Windows Firewall program will usually pop up and ask whether it should allow APLX64 to accept connections - you can either restrict it to just the local machine, or allow connections over the network. For security reasons, you should restrict it to allow access only on the local machine.

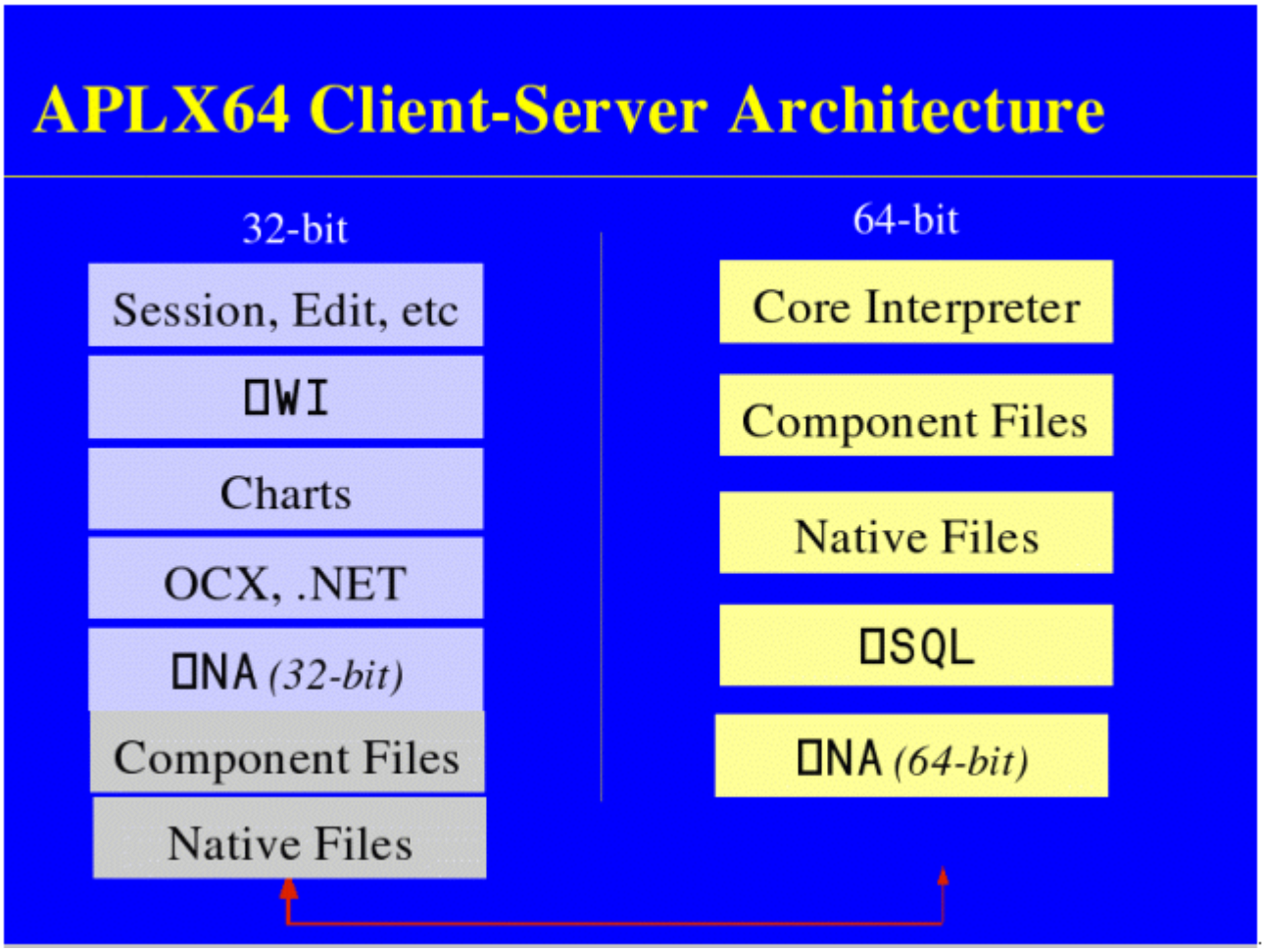


Because most of the program is 32-bit, it installs by default in the 'Program Files (x86)' directory. This is true even of the 64-bit interpreter itself. Also the Registry entries are 32-bit, sitting in the 'WOW6432Node' area of the Registry.

Running the Client and Server on separate machines

If you have purchased APLX64 Server Edition, the Client and the Server can run on separate machines. The Client usually runs under Windows, whereas the Server can run under 64-bit versions of Windows or Linux (or potentially other 64-bit operating systems). When you start the Client program, normally it will try to connect to the APLX Server program running on the server machine specified in your Preferences (see below).

The Server program must already be running when the Client tries to connect to it. The Server starts as a small 'listener' program which waits for a connection. When it receives a connection request from an APLX Client, it starts another process which is the actual APL interpreter associated with that connection. By default, when all clients have exited, the listener program will also exit. However, you will normally want to start the Server with the command-line option "-stay_alive", which prevents it exiting when the last Client has disconnected.



Checking the status of the Server with the 'aplxstatus' utility

The **aplxstatus** program (supplied with APLX64) is a utility which checks if the APLX server is running on a given host. You can optionally give it arguments:

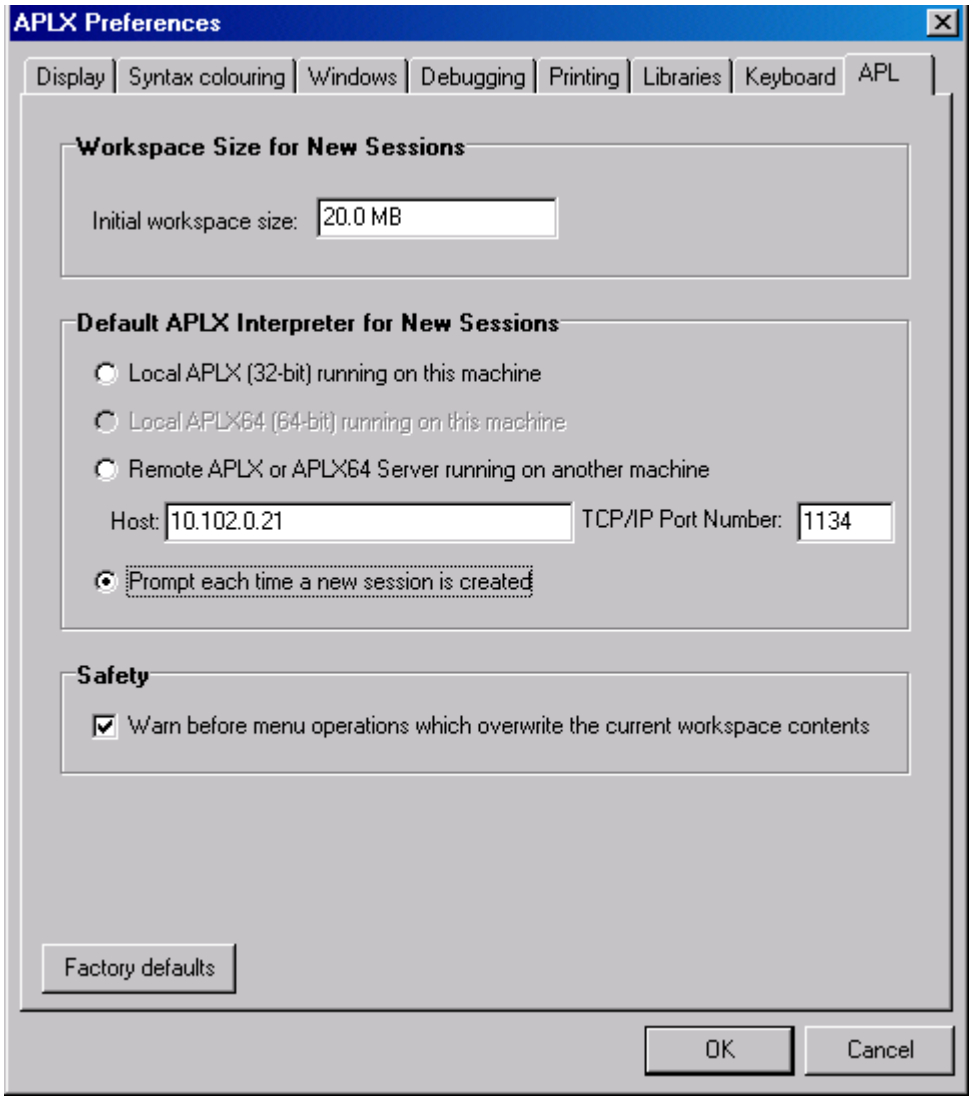
-host HOSTNAME	(default localhost)
-port PORTNUM	(default 1134)
-retry N	(number of retries, once per second. Default 10)

It displays (on standard console output) the number of APLX64 tasks running, or -1 if the server is not running or cannot be contacted. It also sets the exit code to 0 if successful, or 1 on error (you can use this in a script).

32-bit and 64-bit APL Tasks

Customizing the creation of new APL tasks created from the menus

Using the APL tab of the Tools->Preferences dialog, you can alter the way in which new APL sessions, including the initial session at start-up, are started:



The choices are:

(a) Use the 32-bit APL built-in to the Client program.

(b) Use the 64-bit APL Server running on the same machine as the client. If the Server is not already running, it will be started automatically. On a 32-bit Client system, or if you do not have the 64-bit interpreter installed on the same machine as the Client, this option is not available and will be greyed out.

(c) Connect to a remote APLX interpreter over the network, in which case you need to specify the host and port in the normal way (the default port is 1134). You can specify the host as either the IP address (for example, 10.102.0.21), or as the network name of the server machine (for example, server23@bigcorp.com), or as 'localhost', which always means the same machine as the client. Note that the APLX Server program must already be running and accepting connections on the specified system - it will not be started automatically.

You can also say you want to be prompted each time you start a new session. This applies even to the initial session which opens when the APLX Client starts up.

Creating tasks under program control

You can also create new tasks under program control, using the `⎕WI APL` object in the same way as in standard 32-bit APLX. The default is that the new APL session starts in the same execution environment as its parent, so if you create a new APLX task from a 64-bit task, the child will also be a 64-bit APL on the same server.

However, you can tune this by setting the `host` (and optionally `port`) property of the `APL` object before calling the `Open` method. If the `host` property is an empty string, the task will be created a 32-bit APL on the Client system. If it is set to the string '`localhost`', it will be a 64-bit APL on the client machine (assuming the Client is running on a 64-bit system), and the Server program will be started automatically if necessary. If it is anything else, the front-end will attempt to create the new APL task by connecting to the specified remote machine (which must already be running the APLX Server program).

This example will create a new 32-bit APL session:

```
'Session32' ⎕WI 'New' 'APL' ('host' '')
'Session32' ⎕WI 'Open'
```

This example will create a new 64-bit APL session running on the local machine (assuming it is a 64-bit machine with APLX64 installed):

```
'Session64' ⎕WI 'New' 'APL' ('host' 'localhost')
'Session64' ⎕WI 'Open'
```

This example will create a new 64-bit APL session running on a remote machine:

```
'SessionRemote' ⎕WI 'New' 'APL' ('host' 'server23@bigcorp.com')
'SessionRemote' ⎕WI 'Open'
```

If you do create child APL tasks in this way using `⎕WI`, they can share APL data by using the `data` property of the Child object (in the parent task) and the `data` property of the System object in the child task, or by using property names beginning with delta. However, these are held as 32-bit objects, and (like all `⎕WI` properties) will be converted to 32-bit variables before they are sent from the 64-bit APL task to the front-end.

Operations on Client and Server

Where the Client and the Server run on different machines, you can specify where you want certain operations to take place, by prefixing the file name or command string with either an up arrow (↑ Take), meaning the Client, or a down arrow (↓ Drop) meaning the Server. For example, you may want to `)SAVE` a workspace either on the Client machine, or on the Server machine.

The choice of where an operation occurs applies to:

- `)LOAD )SAVE` etc where a library number is used. In this case, the corresponding line of the `⎕MOUNT` table is used to determine the library path, and the first character of this path can be either ↑ or ↓ to indicate which machine is being referenced.
- `)LOAD )SAVE` etc where you specify the full file name
- Native files
- Component files
- `⎕NA`
- `⎕HOST` and `)HOST`

If you do not specify, the operation will take place on the Client. (This may seem a bit surprising, but it means that file selector dialogs still work and give the expected result).

See the on-line Help for the above system functions and system commands for more information.

Note: Menu file load/save will always be on the Client. Currently `⎕SQL` is always run on the Server.

This all makes no difference if you are running the Client and the Server programs on the same machine, except for `⎕NA`, which now allows you to call either a 32-bit or 64-bit DLL (see below for more detail on this).

The ⎕WI sub-system

The `⎕WI` sub-system is part of the Client program, so it always runs as 32-bit code. When you make a call from a 64-bit interpreter, the request is converted to 32-bit form and sent over to the Client program for execution. Any `⎕WI` windows and dialogs, therefore, appear on the Client system. In addition, any references in `⎕WI` objects to files and directories are from the viewpoint of the Client system.

The ⎕NA system function

`⎕NA` has been extended so that it allows you to call either a 32-bit DLL (from the Client program), or a 64-bit DLL (from the Server program. The implementation is as follows:

For clarity, we will assume that the 64-bit Server is running on one machine, and the 32-bit Client is running on a different machine. (In fact, they might be different operating systems, e.g. a Linux 64-bit server and a Windows 32-bit client).

When you use ⎕NA, you might want to call a function on either end. For example, it would make sense to make a 32-bit call to Windows to discover something about the screen or registry on the Client. Equally, you might want to invoke an OS service or library on the Server, for example to call a Linux file-encryption API.

APLX64 use the same conventions as for file names to allow you to specify which you want. If you prefix the ⎕NA specification (i.e. the right argument) with an up arrow (↑ Take), the call takes place on the Client. If you prefix it with a down arrow (↓ Drop), it takes place on the Server. The default (if you do not specify either) is that it takes place on the Client. (This is for compatibility with existing 32-bit APLX Windows applications).

If you make the call on the Server side, the APL task directly calls the requested library. There is no special handling. Everything is 64-bit, and there are no extra tasks involved.

If you make the call on the Client side, the APL task bundles up the request and sends it over the network (which might be just an internal pseudo-network if the two are on the same machine). It then blocks and waits for a response. On the Client side, a 32-bit task picks up the request and makes the (32-bit) call. It returns the results over the network to the 64-bit Server, which wakes up and continues APL execution.

As a consequence of all this, Server-side calls will be faster, which may be a consideration for some kinds of operation.

To support 64-bit calls, new data types for 64-bit integers have been added (I8 and U8 for signed and unsigned 64-bit integers).

If you are running the Client and the Server on the same physical machine, the above all remains true. You can access the Windows 32-bit sub-system (what Microsoft call 'WOW64') as the Client side, and the native 64-bit environment as the Server side.

The 64-bit Interpreter

Overview

APLX64 is a full 64-bit APL. The theoretical maximum workspace size is 2^{63} bytes, and the same limit applies for number of elements in an array and for maximum length along any dimension. In other words, no practical limits at all, other than virtual memory size and what the OS will allow. (In practice, performance starts getting poor if you set the workspace size to be significantly bigger than physical RAM, and current hardware is typically limited to at most 32GB of RAM. This imposes a practical workspace size limit of 32GB as at 2006).

To address these large arrays, APLX64 uses 64-bit integers (otherwise you would have to use floats to index arrays). Floats remain as 64-bit, and binary arrays are still one bit per element.

Workspace layout

The workspace format is different from that used in 32-bit versions of APLX, but this is largely transparent to the user because APLX64 does the conversion on the fly when you `)LOAD` a 32-bit workspace. However, a 32-bit APL cannot `)LOAD` a 64-bit workspace.

Component files

Component files are currently limited to 1024GB because APLX uses a component-file format which is compatible with the 32-bit version.

You can share component files between 32-bit and 64-bit versions of APLX. When you write to a 32-bit component file, the 64-bit WS objects are converted to the 32-bit form (provided they are less than 2GB in size and the number of elements fits in 32-bits). If you write an integer array to a 32-bit component file, and the array contains numbers too large to fit in 32-bit form, the array gets converted to float form. You can also exchange APLX overlays built using `⎕OV`, so you can easily swap data and functions back to the 32-bit form.

Conversion between integers and floats

Up to 2*53, integers can be represented exactly as 64-bit floats. Above 2*53, the floats start to lose so much precision that a given float bit-pattern covers a range which includes more than one integer (maybe many thousands of integers).

Therefore, in APLX64 the Floor and Ceiling primitives have been modified so that, given a float number greater than or equal to 2*53, the number is considered to have overflowed precision, and hence the primitives return the float value unchanged (as a 64-bit float). This is effectively the same behaviour as already happens in 32-bit APLs at 2*31. The reasoning here is that it is wrong to appear to create a spurious precision by choosing one particular 64-bit integer to represent the floor or ceiling, when the interpreter could equally validly choose many other integers.

For the same reason, any float greater than 2*53 cannot be used in expressions which require an exact integer (for example, to index an array). A `DOMAIN ERROR` will be reported.

Default display of numbers

In APLX64 the rules for the default display of numbers have been changed. Numbers represented internally as integers are displayed in full precision irrespective of `⎕PP` (this is also true in most 32-bit APLs, although it may not be obvious because of the limited allowed range of `⎕PP`). In addition, numbers internally represent as floats which are less than 2*53, and which are 'exact' integers, are also displayed in full precision irrespective of `⎕PP`. The practical effect of this is that, at the point where the floats lose precision, the default display switches into E format. Below that, true 64-bit integers, and floats which are close to or exactly integers, both display in the same way (full precision).

Primitives modified to return integer types

The Power and Factorial primitives have been modified so that, if given integer arguments, they always try to return an integer result (previously these two primitives both always returned floats). Hence `X←2*62` now returns an exact 64-bit integer. The goal here is to keep as much precision as possible for results which, in a mathematical sense, really are integers.

Comparison tolerance

In APLX64 the default value of `⎕CT` has been reduced from `1E¯13` to `3E¯15`. This is a compromise between a value which is small enough to distinguish `X` from `X+1` at high values of `X`, and not giving false negatives for true float comparisons because of calculation and representational inaccuracies. The new default value gives means that, for `X` up to 2*48, the expression `X=X+1` always returns 0, irrespective of the internal representation of `X`.

Summary of integer-float issues

The practical effect of these design choices is that, for whole numbers below 2*48, the APL programmer does not need to know or care whether the number is internally represented as a float or as a 64-bit integer; it will behave and display in the same way, and comparisons will always give the expected result. Any conversion between the two internal forms loses no precision, and hence is reversible (eg using Floor or Ceiling). Either representation can be used to index an array, or represent a position in a huge native file.

For numbers between 2*48 and 2*52, the same is true, except that the APL programmer might need to reduce `⎕CT` to avoid comparison problems, or alternatively use Floor or Ceiling to force the numbers to integer before doing a compare.

Above 2*52, if the APL programmer needs exact integers (for example, for doing high-precision arithmetic, or if the integers are 64-bit database record numbers), APLX64 can correctly handle this requirement. However, in this case the APL programmer needs to be careful to ensure that the integers do not accidentally get converted to float (for example, by mixing record numbers and float values in a single N by 2 matrix, or by doing arithmetic operations which are intrinsically non-integer, such as divide). Fortunately, if this does happen, it should be obvious, because the display will flip into E format at the point where precision has been lost, and operations which require an integer will give `DOMAIN ERROR` rather than giving the wrong answer.