Native File Access

Native file access is provided through a set of system functions. These provide the ability to create, open, close, remove, resize, and add and retrieve serializable data to/from native files on the machines hard disk.

The File, Path, Directory, etc classes available in the System.IO assembly provides similar functionality, but with far greater detailed control.

To use the Native File System in your application, you will need to add a reference to the Visual APL Share/Native File System assembly. Here is an example of "referencing" and "using" the assembly by its strong name:

```
refbyname APLNext.APL.LegacyOps
using APLNext.Legacy.NativeFileSystem
```

□nappend

Appends data to a native file which is associated with the tie number. Any serializable object can be appended to a file using this system function.

```
      how are you  '□nappend ¯1 "
```

□ncreate

Creates a native file in the specified directory or in the current directory, if no directory is given. Tie numbers for referencing native files are negative to avoid a conflict with the positive tie numbers used by the component file system.

```
       c: \\mytest\\t'est.nf  □ncreate  -1 "
-1
       test1.nf  □nc'reate  -2 "
-2
      □nnums
-1  -2
      □nnames
 c: \\mytest\\t'est.nf    c: \\mydefau'lt"\\test1.nf              "
```

Note

The double \ as the \ is used as a delimiter in strings. To avoid having to double the \ you can use @ at the beginning of a string.

```
a = @ c: \mytest\tes't.nf              "
```

Which will place the raw string in the variable "a".

You can also create a file and let the system assign the tie number.

For instance:

```
      □ncreate   c: \\mytest\\t'est2.nf              "
-3
```

The system will also assign the next available tie number if you specify a tie number of 0.

For instance:

```
       c: \\mytest\\t'est.nf  □ncreate 0  "
-3
```

□nerase

This will delete the specified native file permanently.

```
        c: \\mytest\\t'est2.nf   □nerase ¯3   "
```

```
       □nerase   c: \\mytest\\t'est2.nf                "
```

nnames

Returns a string array of native file names which are currently tied.

□nnums

Returns an integer array of tie numbers associated with native files which are tied.

□nread

Reads data from a native file which is tied and associated with a given tie number.

```
a = □nread tn, convert, numberOfBytes, beginOffSet
```

The standard conversion values are:

Code	Description
11:	boolean (true/false, not bit)
81:	bytes
82:	chars (compatible with 82 in existing system)
83:	string (compatible with 82 in existing system)
163:	short (Int16, 16 bit integer)
164:	ushort (UInt16, unsigned short)
323:	int (Int32, 32 bit integer, default)
324:	uint (UInt32, unsigned int)
325:	float (Single, 32 bit real)
643:	long (Int64, 64 bit integer)
644:	ulong (UInt64, unsigned long)
645:	double (Double, 64 bit real, default)
1285:	Decimal (128 bit real)
807:	object (serialized object)

Example:

```
a = □nread ¯2 82 10 0
```

Reads back 10 characters.

convert can also be a TypeCode:

```
a = □nread ¯2 TypeCode.Char 10 0
```

convert can also be an intrinsic Type

```
a = □nread ¯2 Char 10 0
```

For reading matrices and arrays of heterogeneous data or any serialized object, use 807:

```
a = □nread ¯2 807 10 0
```

Note

□nsize tn returns a long, which may not be supported in your operator set. The default operator set does not include the long type. This can cause an error when catenating. So use spaces instead of commas, or cast the result to integer, as example:

```
a = □nread ¯2 82 (□nsize ¯2) 0
or
a = □nread ¯2,82,((int) □nsize ¯2), 0
```

nrename

Rename a native file currently tied and associated with a tie number.

```
new_filename nrename tn
```

Where new_filename is the new filename and tn is the existing tie number.

⎕nreplace

Replace existing data in a native file, beginning at the location given and continuing until all of the provided data is written.

```
      100.1 ⎕nreplace ¯3,10
      10L 10 ⎕nreplace ¯3,10
      10.1  test  10 ⎕nreplace ¯3,10
      (3 3⍴⍳9) ⎕nreplace ¯3,10
```

Note that a nested array or matrix is serialized and written to the file. To read serialized data back from the file use the 807 code.

□nresize

Resizes the native file associated with the tie number to a new size in bytes. The new size can be 0, smaller, larger or the same size as the existing size.

```
      0 □nresize ¯2
      10000 □nresize ¯2
```

The resize does not change the data, but making the file smaller will result in the loss of data that existed beyond the new file size.

nulls are used to pad the file when a resize makes the file larger □a v[□i o]

□nsize

Returns a long which represents the size of the file.

```
A = □nsize ¯3
```

If you want the size to be an Int32 use:

```
A = (int)□nsize ¯3
```

nuntie

Unties the native file associated with the tie number.

```
nuntie -3
```

ntie

Ties a native file in the specified directory or in the current directory, if no directory is given. Tie numbers for referencing native files are negative to avoid a conflict with the positive tie numbers used by the component file system.

```
      c: \\mytest\\test.nf  ntie -1    "
-1
      test1.nf  ntie -2    "
-2
     nnums
-1 -2
     nnames
 c: \\mytest\\test.nf    c: \\mydefault\\test1.nf              "
```

Note

The double \ as the \ is used as a delimiter in strings. To avoid having to double the \ you can use @ at the beginning of a string.

```
a = @ c: \mytest\test.nf                "
```

Which will place the raw string in the variable "a".

You can also tie a file and let the system assign the tie number.

For instance:

```
     ntie  c: \\mytest\\test2.nf              "
-3
```

The system will also assign the next available tie number if you specify a tie number of 0.

For instance:

```
      c: \\mytest\\test.nf  ntie 0     "
-3
```

Access Attributes

You can specify what permissions to tie the target file with by supplying a second element to the right argument.

This second argument is the sum of the permissions to request and allow for the file tie operation.

If the access attributes element is not specified, then the default value is 2 (Read/Write access, Exclusive tie)

Here is the list of the valid tie permission request values. The sum of the requested access attributes number can contain only one of these values:

Code	Description
0:	Request read access
1:	Request write access
2:	Request read and write access.

Here is the list of values which control what permissions are granted to future tie requests for the file being tied. The sum of the requested access attributes number can contain only one of these values:

Code	Description
0:	Compatibility Mode
16:	Exclusive Tie, no other ties can be made to the file.
32:	Read access is granted to future ties.
48:	Write access is granted to future ties.
64:	Read and Write access is granted to future ties.

Here is an example of tieing files with different Request and Granted permissions:

```
// tie with read access, and compatibility mode
 c: \\mytest\\test.nf  ntie -1 0  "

// tie with read/write access, and grant read/write
 c: \\mytest\\test1.nf  ntie -1 (2+64)

// tie with read/write access, and grant no permissions
 c: \\mytest\\test2.nf  ntie -1 (2+16)
```

ncopy

Copy the contents of the specified source file to the specified target path.

```
ncopy source_filepath target_filepath
```

Where source_filename is source file from which to copy data, and target_filename is the target path of the copy operation.

□nexists

Returns a value of 1 or 0 indicating if the specified file name exists. Specifying a directory name without a file returns 0 (false).

```
      A = @'c:\Windows\0.log'
      □nexists A
1
      B = @'c:\Windows\'
      □nexists B
0
```

□nstream

Returns the underlying .Net FileStream object for the associated tie number. This allows the use of all features provided by the FileStream object, while still maintaining compatibility with the Native File system.

```
      fs = □nstream ¯3

      fs.CanRead
true
      fs.CanWrite
true
```

Session Commands (Visual APL)

The Cielo Explorer includes a wide range of commands for managing the various aspects of the session.

These aspects include the listing of session contents, script management, and editing of variables.

)cd

Changes the current context of the session into or out of the specified object.

Syntax:

```
)cd obj
```

obj: The name of a class, variable, or path control string.

Remarks:

This session command provides the ability to explore classes. It is possible to explore either an instance of a class or the class itself.

When this session command is used without an argument it displays the current class being explored, for instance when at the top level, in the session, this is displayed

```
      )cd
session
```

The right argument to the)cd session command is either a classname, a variable or a relative path.

To navigate to a particular class:

```
      )cd classname
```

To navigate to an instance of a class:

```
      a = classname()
      )cd a
```

To navigate to a relative location:

```
      )cd ../../a/b/c
```

Or to navigate up one level:

```
      )cd ..
```

To return to the session or root:

```
```

```
      )cd /
```

Examples and narrative:

Once you have navigated into a class, you will see all of the methods, properties, events and fields in the class, regardless of member attributes. This means you can review members which are public, internal, private, etc.

As an example, consider an integer:

```
      a = 10
      )cd a
Loaded instance of:  System.Int32
```

Now if we look at the)fns in this instance of the ValueType Int32 we see:

```
      )fns
_dataRepresentation CompareTo           Equals              Finalize
          GetHashCode
GetType             GetTypeCode         MemberwiseClone     Parse
              ToString
TryParse
```

The)fns includes methods, functions and the methods associated with properties.

Now if we look at)vars we see:

```
     )vars

m_value  MaxValue MinValue
```

We can navigate back up to the session by entering)cd ..

```
     )cd ..
Loaded instance of:  APLNext.APL.Objects.module
```

If we try to navigate up again:

```
     )cd ..
Current instance is session
```

We see that we are already at the session level, and cannot navigate further up.

While we are back at the session level, let’s consider what is visible on the Int32 we have placed in the

variable a.

If we look at intellisense on an instance of an Int32, we see a small subset of those members we saw when we navigated into the instance of Int32 on the variable a.

Specifically, if we navigate back into the a variable:

```
     )cd a
Loaded instance of:  System.Int32
```

If we then check the variable m_value, which is not normally available to investigate, we find:

```
     m_value
10
```

So we see that the Int32 is an object, a ValueType in particular, and that the integer value is stored on the field m_value.

If we want to know where we are in our navigation, we can always do)cd without an argument:

```
     )cd
session/a
```

No matter how deep we have navigated we can always move back to the session by entering:

```
    )cd /
Current instance is session
```

To review, everything is an object, and we can navigate through those objects using)cd, in this example lets look at an Int32 and navigate down and up through this object.

```
     a = 10
     )cd a
Loaded instance of:  System.Int32

      )vars
m_value  MaxValue MinValue

      m_value
10

      )cd MaxValue
Loaded instance of:  System.Int32

      )vars
m_value  MaxValue MinValue

      m_value
```

```
2147483647

      )cd
session/a/MaxValue

      )cd ..
Loaded instance of:  System.Int32

      )cd
session/a

      )cd ..
Loaded instance of:  APLNext.APL.Objects.module

      )cd
session
```

)classes

Shows the current list of classes which have been defined in the session.

Syntax:

```
)classes
```

Remarks:

The *)classes* command shows the list of classes which have been created in the session.

Classes are most commonly created in the session by running a script file.

Example:

Here is an example script which contains the definition of two classes:

```
// Script: sc1

public class math {
    function add(a, b) {
        return a + b
    }

    function subtract(a, b) {
        return a - b
    }
}

public class useMath {
    function fn(a, b) {
        m = math()
        return m.add(a, b)
    }
}
```

Now lets run the script to create the classes in the session:

```
     // display the contents of the )classes list
     )classes

     // the list is empty

     // load and run the script 'sc1'
     )load sc1

     // display )classes again
     )classes
math usemath
     // the two classes now exist in the session.

     // run the useMath class:
     um = useMath()
     um.fn(10, 20)
30
```

)clear

Clears all variables, functions, etc, from the active Cielo Explorer.

Syntax:

```
)clear
```

Remarks:

When the clear session command is run in the Cielo Explorer, the .Net AppDomain which currently represents the Cielo Explorer is shutdown, thus removing from memory any variables, functions, UDF's, file ties, etc, from memory.

If an assembly is referenced into the session by the use of the refbyname or refbyfile directives, then the DLL which that reference represents is tied in memory by the session. This behavior is required by the .Net security model.

When the Cielo Explorer AppDomain is unloaded by the clear command, any assembly references are also removed from memory, meaning that any tied assemblies can again be modified, recreated, and moved on the disc.

Example:

```
     a = 10 20 30
     a
10 20 30

     )clear

     a
The variable 'a' does not exist
```

)edit

Opens a script file for editing in the current Session Project of the active Solution.

Syntax:

```
)edit script
```

script: The name of the script file to create or open.

Remarks:

When the edit command is run in the Cielo Explorer session, a script file of the specified name is opened for editing from the current Session Project in Visual Studio, and given the active window focus.

If the script file does not exist in the current Session Project, then a new script file is created in the Session Project by the supplied name, and opened for editing.

If the script file does exist in the current Session Project, then it is opened and focused for editing.

If the script file is already open in Visual Studio, then that script file is brought to the forefront and receives the window focus.

Saving a script

Once you have edited the contents of a script file, you can save and execute the script to the Cielo Explorer session by pressing the key sequence **Ctrl+E+E**. Once the script is saved and executed to the Cielo Explorer, a message is printed to the session stating that the script was modified and imported.

Example:

```
     // make a variable in the session
     a = 10 20 30
     a
10 20 30

     // edit a new script
     )edit sc

     // add this line to the script
     a = a + 100

     // save the script by pressing Ctrl+E+E.

     // this line is printed to the session
     Script 'sc' updated

     // now again display the contents of 'a' in the session.
     a
110 120 130
```

)fns

Shows the current list of functions which have been defined in the session.

Syntax:

```
)fns
```

Remarks:

The *)fns* command shows the list of functions which have been created in the session.

Here are a few examples of creating functions in the session:

- Entering its declaration directly in the session:

```
     // display the contents of the )fns list
     )fns

     // the list is empty

     // define a new function called 'add'
     function add(a, b) { return a + b }

     // display )fns again
     )fns
add
     // the function 'add' is now in the list.
```

- Using □def to declare a function from a text string:

```
     // display the contents of the )fns list
     )fns

     // the list is empty

     // define a function from a string
     □def  function add("a, b) { return a + b }              "
true

     // display )fns again
     )fns
add
     // the function 'add' is now in the list.
```

- Execute a script which contains the definition of one or more functions. Here is an example script which contains the definition of two functions:

```
// Script: sc1

function add(a, b) {
      return a + b
}
```

```
function subtract(a, b) {
      return a - b
}
```

Now lets run the script to create the functions in the session:

```
     // display the contents of the )fns list
     )fns

     // the list is empty

     // load and run the script 'sc1'
     )load sc1

     // display )fns again
     )fns
add subtract
     // the functions are now in the list.
```

These are only a few simple examples of creating functions in the session. Any valid expression or statement which creates a function can be run is the session, and once that command is run the resultant function will be present in the *)fns* list.

Created Function Time Stamping

When functions are dynamically created in the session, they receive an associated DateTime object which represents the moment that the function was created in the session. There are several system quad functions which allow the retrieval and modification of this time stamp.

Example:

```
     // check the contents of the )fns list
     )fns
mult sub
     // there are two functions currently defined

     // define a new function called 'add'
     function add(a, b) { return a + b }

     // check the )fns list
     )fns
add mult sub
     // the function 'add' is now in the list.

     // try running add:
     10 add 20
30
```

)load

Loads and executes a script from the current Session Project.

Syntax:

```
)load script
```

script: The name of the script file to load.

Remarks:

The)load command looks in the current Session Project for a script named the specified name. If the script file exists, it is added to the)scripts list in the session, and then the contents of the script are executed in the session.

This command has the same behavior as pressing **Ctrl+E+E** in an open script in Visual Studio.

Example:

```
     // check if 'a' exists
     a
name 'a' is not defined

     // check the contents of the )script list
     )scripts

     // the )scripts list is empty

     // load a script which defines 'a'
     )load sc

     // check if 'a' exists
     a
10 20 30

     // check the )scripts list
     )scripts
sc
     // the script 'sc' is now in the list.
```

)off

Clears all variables, functions, etc, from the active Cielo Explorer. Also closes the current open Solution in Visual Studio.

Syntax:

```
)off
```

Remarks:

The *off* command has the same effect as the *clear* command for the contents of the session, and also closes the currently open Solution in Visual Studio, and all associated projects.

If any open files are currently marked as unsaved in Visual Studio, then the Save File dialog is opened prompting for user action. This behavior is the built-in functionallity of Visual Studio, meaning that in relation to the currently open Solution, the off command has the same effect as the "File > Close Solution" menu item.

Example:

```
     a = 10 20 30
     a
10 20 30

     )off

     a
The variable 'a' does not exist
```

)run

Executes the contents of a script from the current Session Project.

Syntax:

```
)run script
```

script: The name of the script file to run.

Remarks:

The *)run* command looks in the *)scripts* list for a script named the specified name. If the script file exists in the list, the contents of the script are executed in the session.

The *)run* command is similar to the *)load* command, except that the specified script must already be listed in the *)scripts* list for the command to succeed.

Example:

```
     // check if 'a' exists
     a
name 'a' is not defined

     // check the contents of the )script list
     )scripts
sc
     // the 'sc' script is present in the session

     // run the script 'sc', which defines 'a'
     )run sc

     // check if 'a' exists
     a
10 20 30
```

)runf

Executes the contents of a script at the specified file path.

Syntax:

```
)runf scriptPath
```

scriptPath: The fully qualified file path of the script to run.

Remarks:

The *)runf* command takes a file path to a script file as its argument. When the *)runf* command is entered, the contents of the specified script file are run in the session.

The *)runf* command is similar to the *)run* command, except that the argument script file does not need to exist in the *)scripts* list.

Example:

```
     // check if 'a' exists
     a
name 'a' is not defined

     // check the contents of the )script list
     )scripts

     // the list is empty

     // runf the script 'sc', which defines 'a'
     )runf c:\sc.apl

     // check if 'a' exists
     a
10 20 30
```

)scripts

Shows a list of scripts which are currently loaded into the session.

Syntax:

```
)scripts
```

Remarks:

Script files can be loaded into the session by the *)load*, *)xload*, and *)edit* commands. Pressing **Ctrl+E+E** in an open script file also adds that script to the *)scripts* list.

Example:

```
     // check the contents of the )script list
     )scripts
sc1 sc2 sc3
     // there are three scripts currently loaded

     // xload a script called 'math'
     )xload math

     // check the )scripts list
     )scripts
sc1 sc2 sc3 math
     // the script 'math' is now in the list.
```

)vars

Shows the current list of variables which have been defined in the session.

Syntax:

```
)vars
```

Remarks:

The *)vars* command shows the list of variables which have been created in the session.

Example:

```
     // check the contents of the )vars list
     )vars
a b
     // there are two variables currently defined

     // define a new variable called 'c'
     c = 100 200 300

     // check the )vars list
     )vars
a b c
     // the variable 'c' now exists in the session.
```

)xload

Loads a script from the current Session Project.

Syntax:

```
)xload script
```

script: The name of the script file to xload.

Remarks:

The *)xload* command looks in the current Session Project for a script named the specified name. If the script file exists, it is added to the *)scripts* list in the session.

This command has a similar behavior to the *)load* command, except that the contents of the script are not executed.

Example:

```
     // check the contents of the )script list
     )scripts

     // the )scripts list is empty

     // xload a script called 'sc'
     )xload sc

     // check the )scripts list
     )scripts
sc
     // the script 'sc' is now in the list.
```

)xmlout

Exports a variable in XML format into the current Session Project in Visual Studio.

Syntax:

```
)xmlout var var var...
```

var: The name of a variable to export.

Remarks:

For each argument variable, the *xmlout* command creates an XML file in the current Session Project named "*var*.xml", where *var* is the name of the variable being exported.

If an XML file by that name already exists in the current Session Project, then that file is overwritten with the newly produced XML output.

Only objects which are serializable can be successfully exported to XML.

A variable is considered serializable if any of the following conditions are met:

- It is marked with the .Net Serializable attribute.
- Implements the .Net ISerializable interface.
- Implements the IXMLSerializable interface.
- Has a registered serializer in the Cielo Explorer.

If an object is encountered in a variable being exported with *xmlout* that does not meet any of the above serializable criteria, then a comment is placed in the generated XML at the location where the object would have appeared in the output XML, stating that the element could not be serialized. This behavior ensures that if you have a variable in the Cielo Explorer which contains mostly serializable data and only a few elements which cannot be serialized, the elements which are not serialzable will not prevent the serializable elements from being exported to XML format. Keep in mind that if you save the generated XML back to the Cielo Explorer by the use of **Ctrl+E+E**, that those elements which could not be serialized during the generation of the XML file will contain empty objects in the newly imported variable.

Once the *xmlout* command has been executed, the active window in Visual Studio is shifted to the newly created or updated XML file. If more than one variable was exported, the focus is placed on each XML file view as it is created, ultimately being placed on the XML file of the last variable being exported.

Saving changes to XML variables

Once a variable has been exported as XML, the generated XML can be modified in any way desired, and those changes can be saved back to the Cielo Explorer.

To save changes made to an XML file in the current Session Project, open that file and press the key sequence **Ctrl+E+E**

Once the sequence is pressed, focus is returned to the Cielo Explorer, and a message is printed to the session showing that an update was made to the variable.

When the changed XML is saved back to the session, the name of the variable into which the data is saved is taken from the name of the XML file. This means that if an XML file named "a.xml" is saved to the session using **Ctrl+E+E**, then the deserialized contents of that file will be saved as the variable "a" in the session. This means that you can not only save variables from the session in XML format, but you can also import entirely new variables from XML format by simply adding them to the Session Project, opening them, and

then pressing **Ctrl+E+E**

Additional Information

The *xmlout* command uses the XmlCvarSerialzier to perform the XML conversion to and from the current Session Project.

Example:

```
a = 10  test  (15)  "    "
)xmlout a
// a file has been created in the current Session Project.
```

Cielo Explorer Menu Reference

The Cielo Explorer includes a toolbar which allows various common session management activities to be easily performed at the click of a button.

Following is a listing of each button in the Cielo Explorer and their uses:

Toolbar buttons in Cielo Explorer

New

Clears the present session. This unloads the present domain, removing all references to assemblies and creates a new session domain.

Run Cielo Script

The user is prompted with the file selection dialog box. A script file is selected which is then defined and run in the current session. Scripts can contain any statement or expression; this includes control structures, function definitions, classes and simple statements. Scripts are dynamic, and functions, variables or classes defined in a script replace any dynamic members that exist in the current session with the same names.

For instance, the following script defines the function fn and then calls that function:

```
// Script: sc

function fn(a) {
           return a
}
fn( hello )
```

When this script is run in the session, the word hello is displayed in the session and the function fn is added to those available in the session.

Load Cielo File

This loads a Visual APL file which contains a formal assembly definition. The assembly is created and the resulting dll or exe can be referenced in the session or in the case of an exe, run from the OS.

Import Assembly

This adds a reference to an existing assembly to the session. If there is a namespace in the assembly which matches the name of the assembly, a using is also done.

Load Session Log

This prompts the user with a file selection dialog box. The user can choose any existing Visual APL log file, which is then displayed in the session, thus providing the user with all of the commands, definitions, etc which occurred in a previous session.

Save Session Log

This action saves all of the display content in the existing session to the file selected by the user. The user is prompted with the Save File dialog box.

Print

This prints the display contents of the existing session.

Cut

This removes the selected text from the session display and places it on the clipboard.

Copy

This copies the selected text which is placed on the clipboard.

Paste APL+Win

This pastes APL+Win code into the session explicitly converting from the legacy APL+Win text to APL Unicode.

:IF :ELSE

The tests for the :if and :elseif must evaluate to a single value which can be converted to a Boolean.

```
:if test
      if statement block
:elseif test1
      elseif statement block
:elseif test2
      elseif statement block
:else
      else statement block
:endif
```

The logical && and || are supported also.

In the example below the test2 is evaluated only if test returns a true

```
:if test && test2
      code block
:endif
```

In the example below the test2 is evaluated only if test returns a false

```
:if test || test2
      code block
:endif
```

:select :case

The :select control structure provides a mechanism for switching between multiple cases based on Identity comparison.

```
:select value
      :case value1
            code block
      :case value2
            code block
      :else
            else code block
:endselect
```

:while

The :continue keyword passes control to the :while test statement.

The :leave keyword branches to the first statement after the :while structure.

The test must return a value which will convert to Boolean

```
:while test
      statements
:endwhile
```

The logical && and || also works with the :while structure

In the example below the test2 is evaluated only if test returns a true

```
:while test && test2
      code block
:endwhile
```

In the example below the test2 is evaluated only if test returns a false

```
:while test || test2
      code block
:endwhile
```

:repeat :until

The :continue keyword passes control to the :until test statement.

The :leave keyword branches to the first statement after the :repeat structure.

The test must return a value which will convert to Boolean

The :repeat structure is repeated until the test evaluates to true.

```
:repeat
      code block
:until test
```

The logical && and || also works with the :repeat structure

In the example below the test2 is evaluated only if test returns a true

```
:repeat
      code block
:until test && test2
```

In the example below the test2 is evaluated only if test returns a false

```
:repeat
      code block
:until test || test2
```

:for :in

The :for control structure iterates across an iterable expression, placing the iterated values in the control variables.

```
:for i :in ⍳3
      print i
:endfor
0
1
2
```

The :continue keyword branches to the top of the for loop and the next value is assigned to the control variables.

The :leave keyword branches to the first statement after the :for loop.

The assignment of values into the variables follows the rules of variable assignment.

```
:for a b c :in (1 2 3) (4 5 6)
      print a
      print b
      print b
:endfor
```

The first time through the :for loop a:1, b:2, c:3 the second time a:4,b:5,c:6

It is also possible to assign based on depth of nested array

```
function fnf() {
    v = (1 (2 (3 4)) 5) (6 (7 (8 9)) 0)
    :for (a (b (c)) d) :in v
        print c
    :endfor
}
fnf()
3 4
8 9
```

: Label separator, switch case separator and legacy keyword indicator

Creates a label to which control can branch when used as follows:

```
function fn(a) {
      →L1
      print a
      L1:
      print  branch
}
```

Used to delimit legacy keywords

```
fuction fn(a) {
      :for i :in ι10
            print i
      :endfor
}
```

Used to delimit switch case statement

```
function fn(a) {
      switch (a) {
            case 10:
                  print  something
                  break
            default:
                  print  default
                  break
      }
}
```

→ Branch

The example below shows an unconditional branch to a label.

Example:

```
function fn(a) {
    print  one          "   "
    →L1
    print  two          "   "
    L1:
    print  three        "     "
}
```

:goto :return

:goto provides an unconditional branch to a label

:goto label

:return returns from the function

It is also possible to return data with the :return keyword

:return expression

Using :return with an expression returns a value without having to set the default return variable specified in the user defined function header.

: Label separator, switch case separator and legacy keyword indicator

Creates a label to which control can branch when used as follows:

```
function fn(a) {
      →L1
      print a
      L1:
      print  branch
}
```

Used to delimit legacy keywords

```
fuction fn(a) {
      :for i :in ι10
            print i
      :endfor
}
```

Used to delimit switch case statement

```
function fn(a) {
      switch (a) {
            case 10:
                  print  something
                  break
            default:
                  print  default
                  break
      }
}
```

Number sign

Delimits directives, such as region.

```
#region code
      function fn(a) {
            print a
      }
#endregion
```

This creates a collapsible region in Visual Studio.

: Label separator, switch case separator and legacy keyword indicator

Creates a label to which control can branch when used as follows:

```
function fn(a) {
      →L1
      print a
      L1:
      print  branch
}
```

Used to delimit legacy keywords

```
fuction fn(a) {
      :for i :in ι10
            print i
      :endfor
}
```

Used to delimit switch case statement

```
function fn(a) {
      switch (a) {
            case 10:
                  print  something
                  break
            default:
                  print  default
                  break
      }
}
```

; Axis Separator

When used inside of an indexer bracket block [ ] the axis separator identifies the values for each axis.

```
      a = 1 2 3
      a[1]
2
      a = 3 3⍴⍳9
      a[1 2;1 2]
4 5
 7 8
```

It is not required to use the axis separator to index an array, for instance:

```
      b = (1 2) (1 2)
      a[b]
 4 5
 7 8
            b = 1 2
      a[b]
5
```

Providing a single value will index the array as though it were a vector.

```
       a[1]
1
```

You can select all values in an axis by using null:

```
      b = (1 2) (1 2) null
      a[b]
 12 13 14
 15 16 17

 21 22 23
 24 25 26
```

This makes it possible to index an array without having to be concerned about the syntax of the number of semi colons.

; Statement Separator

Separates code expressions or statements on a line.

Example:

```
code1 ; code2
```

The diamond may also be used to delimit statements.

_ Underscore

A valid symbol to be used in a variable, method, function, property or other object name. It is also valid as the first character of a name.

In the session the _ contains the last information that was displayed to the screen or would have displayed to the screen if in a function.

Example:

```
      ⍳3
1 2 3
      _
1 2 3
```

This is very useful when reusing information in the session. Instead of having to copy and implement a long line of code, you can simply include _ on the next line.

Example:

```
       100+⍳10-3+5
100 101
      14+_
114 115
```

¯ High Minus

The high minus can be used to identify negative numbers in a vector or numbers being input.

For instance:

```
      10 - 5
5
```

However:

```
      10 ¯5
10  ¯5
```

This simplifies numeric input and reduces the need for parenthesis.

⍝ Comment

The ⍝ is the single line comment symbol. It can be used in conjunction with / to create a multi line comment.

Example:

```
        fn(a) {                    f
            b = a+1
            /⍝ this is a line
            another comment line
            yet another
            ⍝/
            print b
      }
      fn(10)
11
```

The double // also indicates a single comment line.

∇ Del

Delimiter used to identify the beginning of a user defined function.

Example:

```
∇ r←x add y {
      r←x+y
}
```

Notice that the beginning of the function block is started with a { and the end of the function block is terminated with a }.

∆ Delta

A valid symbol to be used in a variable, method, function, property or other object name. It is also valid as the first character of a name.

Note that objects that include the Δ will be difficult if not impossible to be consumed by other languages. This is included for legacy purposes.

≙ Delta underscore

A valid symbol to be used in a variable, method, function, property or other object name. It is also valid as the first character of a name.

Note that objects that include the ≙ will be difficult if not impossible to be consumed by other languages. This is included for legacy purposes.

◇ Statement Separator

Separates code expressions or statements on a line.

Example:

```
code1 ◇ code2
```

System Function Reference

This page contains a complete listing of all system quad functions currently available in Visual APL from APLNext.

Basic System Functions, Variables, etc.

System Function	Description
□DR	Data type and conversion
□ENLIST	Array to vector
□EXPAND	Array fill
□FI	Numeric format
□FIRST	First of an array
□FMT	Legacy Format
□FORMAT	New Array Formatter using .Net formatting specifiers
□MIX	Reduce nesting
□PENCLOSE	Array to nested vector
□REPL	Replicate array
□SPLIT	Increase nesting
□SS	String search
□TYPE	Numeric / character
□VI	Verify numeric
□FAVAIL	Returns a 1 if the share file system is available
□DM	Diagnostic message
□ERROR	Throw error
□dmx	Extended Diagnostic message
□DEF	Define function
□ERASE	Erase functions or variables
□EX	Erase functions or variables
□FX	Define function from □CR representation
□IDLIST	List objects in WS
□NC	List object types
□NL	List object names

□SIZE	Get size of object
□AT	Object attributes
□DL	Delay execution
□AI	Accounting information
□CT	Comparison tolerance
□IO	Index origin
□LIB	File directory
□LIBD	Set library to directory
□LIBS	List libraries and directories
□PP	Print precision
□RL	Random number seed
□TS	Timestamp
□reference	Adds a reference to an assembly
□using	Makes the namespace in a referenced assembly available
□AV	Atomic vector (character set)
□UCS	Returns index or Unicode character from index
□SYSID	APL system ID
□SYSVER	APL system version
□USERID	Workstation ID
□TCxx	Terminal control characters
□TC	contains a three-element vector of terminal control characters. □TC[1]= □TCBS (backspace) □TC[2]= □TCNL (newline) □TC[3]= □TCLF (linefeed).

Other Terminal Control Constants:

□TCBEL	Bell character
□TCBS	Backspace character
□TCDEL	Delete character
□TCESC	Escape character

□TCFF	Formfeed character
□TCHT	Horizontal Tab character
□TCLF	Linefeed character
□TCNL	Newline character
□TCNUL	Null character

State Functions

□ea	Executes either left or right arguments
□monadic	Indicates if an APL function was called monadically
□dyadic	Indicates if an APL function was called dyadically
□dbz	Divide By Zero
□dbzv	Divide By Zero Value
□nfi	NumberFormatInfo used by pattern format and when displaying to session

Argument Attributes

□arglist	Indicates argument is to used as list or arguments to the method
□argnames	Indicates argument is a matrix of named arguments and values

Application Shared DataStore (manages datastore created with svglobal keyword)

□svd	Remove a shared variable from the datastore
□svc	Check to see if a variable has been assigned since last assigned or referenced
□svs	Check to see if a variable is in the datastore
□svget	Sets an event method on a shared variable which runs when variable is referenced
□svset	Sets an event method on a shared variable which runs when variable is assigned

Windows Interface (legacy) loaded with Windows Interface Assembly

These quads have been deprecated in favor of the Windows Designer in Visual Studio and the new .Net System.Windows.Forms and related classes.

□wi	Windows Interface Legacy
□wself	The current or last reference wi object
□wres	Legacy wi wres
□warg	Legacy wi warg
□wsender	The actual object that created an event
□wievent	The actual event which was raised
□wevent	The legacy wi event

Loaded with the NativeFileSystem assembly

□NAPPEND	Add data to file
□NCREATE	Create file
□NERASE	Erase file
□NNAMES	Names of open files
□NNUMS	Numbers of open files
□NREAD	Read data
□NRENAME	Rename file
□NREPLACE	Replace data in file
□NRESIZE	Resize file
□NSIZE	Get file size
□NTIE	Tie (open) file
□NUNTIE	Untie file
□nexists	Deterimines if a file or directory exists
□ncopy	Copies a file to a new file
□nmove	Moves the file
□nstream	Returns he filestream associated with the tie number

Loaded with ShareFileSystem assembly

□FAPPEND	Append components
□FCREATE	Create file
□FDROP	Drops components from the beginning or end of a

	share file and renumbers the components
□FDUP	Duplicates a share file
□FERASE	Erase a share file
□FLIB	File directory
□FNAMES	Tied share file names
□FNUMS	Tied share file numbers
□FREAD	Read component
□FREPLACE	Replace component
□FSIZE	Get file size
□FSTIE	Tie share file
□FTIE	Tie share file
□FUNTIE	Untie share file
□fcatenate	Catenate a valuetype to a valuetype array stored in a component
□libdrw	Determines access to virtual share file directory
□libdcws	Changes access to virtual share file directory
□firead	Reads a specified range of valuetypes from a valuetype array in a component
□fireplace	Replaces a specified range of valuetypes in a valuetype array in a component
□falloc	Allocates contiguous space to a share file component
□fcnloc	Returns the physical location of a component in a share file
□fstream	Returns the filestream associated with the file tie number or virtual directory
□fremove	Removes a component from a share file and renumbers components

Loaded with either the Native File System or Share File System

□XLIB	Returns the directory or files in a directory
□CHDIR	Change current directory
□MKDIR	Create directory
□RMDIR	Delete directory

System Function Reference

This page contains a complete listing of all system quad functions currently available in Visual APL from APLNext.

Basic System Functions, Variables, etc.

System Function	Description
□DR	Data type and conversion
□ENLIST	Array to vector
□EXPAND	Array fill
□FI	Numeric format
□FIRST	First of an array
□FMT	Legacy Format
□FORMAT	New Array Formatter using .Net formatting specifiers
□MIX	Reduce nesting
□PENCLOSE	Array to nested vector
□REPL	Replicate array
□SPLIT	Increase nesting
□SS	String search
□TYPE	Numeric / character
□VI	Verify numeric
□FAVAIL	Returns a 1 if the share file system is available
□DM	Diagnostic message
□ERROR	Throw error
□dmx	Extended Diagnostic message
□DEF	Define function
□ERASE	Erase functions or variables
□EX	Erase functions or variables
□FX	Define function from □CR representation
□IDLIST	List objects in WS
□NC	List object types
□NL	List object names

□SIZE	Get size of object
□AT	Object attributes
□DL	Delay execution
□AI	Accounting information
□CT	Comparison tolerance
□IO	Index origin
□LIB	File directory
□LIBD	Set library to directory
□LIBS	List libraries and directories
□PP	Print precision
□RL	Random number seed
□TS	Timestamp
□reference	Adds a reference to an assembly
□using	Makes the namespace in a referenced assembly available
□AV	Atomic vector (character set)
□UCS	Returns index or Unicode character from index
□SYSID	APL system ID
□SYSVER	APL system version
□USERID	Workstation ID
□TCxx	Terminal control characters
□TC	contains a three-element vector of terminal control characters. □TC[1]= □TCBS (backspace) □TC[2]= □TCNL (newline) □TC[3]= □TCLF (linefeed).

Other Terminal Control Constants:

□TCBEL	Bell character
□TCBS	Backspace character
□TCDEL	Delete character
□TCESC	Escape character

□TCFF	Formfeed character
□TCHT	Horizontal Tab character
□TCLF	Linefeed character
□TCNL	Newline character
□TCNUL	Null character

State Functions

□ea	Executes either left or right arguments
□monadic	Indicates if an APL function was called monadically
□dyadic	Indicates if an APL function was called dyadically
□dbz	Divide By Zero
□dbzv	Divide By Zero Value
□nfi	NumberFormatInfo used by pattern format and when displaying to session

Argument Attributes

□arglist	Indicates argument is to used as list or arguments to the method
□argnames	Indicates argument is a matrix of named arguments and values

Application Shared DataStore (manages datastore created with svglobal keyword)

□svd	Remove a shared variable from the datastore
□svc	Check to see if a variable has been assigned since last assigned or referenced
□svs	Check to see if a variable is in the datastore
□svget	Sets an event method on a shared variable which runs when variable is referenced
□svset	Sets an event method on a shared variable which runs when variable is assigned

Windows Interface (legacy) loaded with Windows Interface Assembly

These quads have been deprecated in favor of the Windows Designer in Visual Studio and the new .Net System.Windows.Forms and related classes.

□wi	Windows Interface Legacy
□wself	The current or last reference wi object
□wres	Legacy wi wres
□warg	Legacy wi warg
□wsender	The actual object that created an event
□wievent	The actual event which was raised
□wevent	The legacy wi event

Loaded with the NativeFileSystem assembly

□NAPPEND	Add data to file
□NCREATE	Create file
□NERASE	Erase file
□NNAMES	Names of open files
□NNUMS	Numbers of open files
□NREAD	Read data
□NRENAME	Rename file
□NREPLACE	Replace data in file
□NRESIZE	Resize file
□NSIZE	Get file size
□NTIE	Tie (open) file
□NUNTIE	Untie file
□nexists	Deterimines if a file or directory exists
□ncopy	Copies a file to a new file
□nmove	Moves the file
□nstream	Returns he filestream associated with the tie number

Loaded with ShareFileSystem assembly

□FAPPEND	Append components
□FCREATE	Create file
□FDROP	Drops components from the beginning or end of a

	share file and renumbers the components
□FDUP	Duplicates a share file
□FERASE	Erase a share file
□FLIB	File directory
□FNAMES	Tied share file names
□FNUMS	Tied share file numbers
□FREAD	Read component
□FREPLACE	Replace component
□FSIZE	Get file size
□FSTIE	Tie share file
□FTIE	Tie share file
□FUNTIE	Untie share file
□fcatenate	Catenate a valuetype to a valuetype array stored in a component
□libdrw	Determines access to virtual share file directory
□libdcws	Changes access to virtual share file directory
□firead	Reads a specified range of valuetypes from a valuetype array in a component
□fireplace	Replaces a specified range of valuetypes in a valuetype array in a component
□falloc	Allocates contiguous space to a share file component
□fcnloc	Returns the physical location of a component in a share file
□fstream	Returns the filestream associated with the file tie number or virtual directory
□fremove	Removes a component from a share file and renumbers components

Loaded with either the Native File System or Share File System

□XLIB	Returns the directory or files in a directory
□CHDIR	Change current directory
□MKDIR	Create directory
□RMDIR	Delete directory

□ai Account Information

Legacy account information. Returns a four element vector, the second element of which is the time in milliseconds since the first time that □ai was referenced.

This is particularly useful when doing simple timing tests:

```
□io=1
ts = □ai[1]
for (I = 0;i<10000;i++) {
      b = 10×i
}
print □ai[1]-ts
```

This will display the time taken by the statements interposing the two references to □ai

The first element is always 1 and the last two elements are reserved.

□av Atomic Vector

This is provided for legacy reasons only.

Contains 256 characters and is a simple character vector. Visual APL is based on Unicode characters. □a v is a selection of commonly used Unicode characters.

cmd Command Window

This has been deprecated in favor of the System.Diagnostics.Process class.

Here is a simple example of how to use this:

```
using System.Diagnostics
a = Process()
a.StartInfo.FileName=  cmd.exe      "       "
a.StartInfo.UseShellExecute = false
a.StartInfo.Arguments =  /k dir *.*   "          "
a.Start()
```

This will open a cmd window and display the directory.

There are a wealth of options for this type and extensive documentation can be found for this .Net framework type at Microsoft.com, as well as the over 4,000 other .Net framework types.

□ct Comparison Tolerance

The comparison tolerance is the difference or fuzz allowed between two values when comparing them for equality. The default setting for □ct is double.Epsilon which is the chip dependent comparison tolerance.

Example:

```
      using System
      double.Epsilon
4.94065645841247E-324
       □ct
4.94065645841247E-324
```

The value of □ct can be set to alter the operation of the following operators.

```
⌊ floor
⍳ index of
⌈ ceiling
> ≥ ≈ ≤ <  numeric relation
∣ residue
∈ find
≡ match
~ without
⍳ membership
```

Note
the ≈, or approximately equal symbol is obtained by pressing the alt-5 key. This is not to be confused with the = symbol which is used for reference assignment. To perform an exact equal use == `a == b`

⎕dr

The data representation of intrinsic objects in .Net can be determined and manipulated using ⎕dr.

⎕dr can be used either monadically or dyadically.

Monadic:
When used monadically ⎕dr reports the type of an object based on legacy codes. These codes are:

Code	Description
11:	boolean (true/false, not bit)
81:	bytes
82:	chars (compatible with 82 in existing system)
83:	reserved.
162:	chars (compatible with 82 in existing system)
163:	short (Int16, 16 bit integer)
164:	ushort (UInt16, unsigned short)
323:	int (Int32, 32 bit integer, default)
324:	uint (UInt32, unsigned int)
325:	float (Single, 32 bit real)
643:	long (Int64, 64 bit integer)
644:	ulong (UInt64, unsigned long)
645:	double (Double, 64 bit real, default)
1285:	Decimal (128 bit real)
807:	object (serialized object)
99999:	no code available for data type

Example:

```
      ⎕dr 10
323
    ⎕dr 10L
643
    ⎕dr 20.1
645
    ⎕dr 10f
325
```

Dyadic:

The left argument to ⍈dr can be a legacy code listed above. When this is the case the data on the right is coerced to the new data type based on the bit representation of the data.

Example:

Converts a short to a Boolean representation:

```
      11 ⍈dr (short)32
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0

      323 ⍈dr 10.1
858993459 1076114227

      645 ⍈dr 323 ⍈dr 10.1
10.1

      645 ⍈dr (bool) 11 ⍈dr 10.1
10.1
```

Note

Requirement for casting to Boolean as the result of 11 ⍈dr 10.1 is an integer array of 1 and 0.

Often what is desired is to cast an int to a double, a double to an int, a short to and int, etc.

Using a type as the left argument to ⍈dr accomplishes this.

Example:

```
      int ⍈dr 10.1
10
      int ⍈dr 10.1 10.6
10 11
      ⍈dr double ⍈dr 10
645
```

It is also possible to serialize data using ⍈dr. This is accomplished using the text string "wrapl" as the left argument. To deserialize the data, use "unwrapl"

```
      a =  wrapl  ⍈dr 10" test" 20    "    "
      b =  wrapl  ⍈dr 10" test" 20    "    "
```

The result of the serialization is a string and the result is always identical for identical data. This means that the results can be compared for the purposes of checking equivalence.

Any object that supports serialization can be serialized either individually or as part of a nested structure.

If you understand the serialization of the object, you can even modify the string which will impact the object that you return.

This is useful for sending objects either over the internet or writing and object to file and then retrieving and reinstantiating the object at a later time.

⎕dbz Divide By Zero

This system function provides control over the way in which the system addresses divide by zero.

The default value is 0 to match .Net languages, however, you can set this to the following:

```
⎕dbz:
 0 :   1 0 = 0
       0 0 = 0
 1 :   1 0 = DOMAIN ERROR
       0 0 = 1
 2 :   1 0 = DOMAIN ERROR
       0 0 = DOMAIN ERROR
 3 :   1 0 = NaN or ⎕dbzv
       0 0 = NaN or ⎕dbzv
 4 :   1 0 = +-Infinity
       0 0 = NaN
```

You can set ⎕dbzv to any object, and that will be returned when ⎕dbz is set to 3

There are several new Double types which are valid doubles and therefore do not promote a double array to a heterogeneous array.

```
      double.NaN
NaN
      double.NegativeInfinity
-Infinity
      double.PositiveInfinity
Infinity

      ⎕dbzv← byzero       "       "
      ⎕dbz←3
      a←2 3⍴⍳6
      a 2 3⍴10 0
         0     byzero    0.2
    byzero        0.4 byzero

      ⎕dbzv←double.NaN
      a 2 3⍴10 0
       0 NaN     0.2
 NaN     0.4 NaN
  ÷    b = a 2 3⍴10 0
      ⎕dr b
645
```

⎕dyadic

Indicates if a user defined function was called with both a left and right argument. ⎕dyadic is false if the function was called with only the right argument.

```
∇r←a add b {
      if (⎕dyadic) {
            r←a+b
      } else {
            r←b
      }
}
```

Dynamically Referencing Assemblies

The **refbyfile**, **refbyname**, and **using** keywords are directives and are only referenced during creation of the dll or exe assembly.

For late binding to an assembly, Visual APL supplies two quad system functions:

□**reference**

□**using**

□reference adds a late bound reference to the specified assembly, whether it is given as a file or as a name, and does this during execution. Arguments to □reference can be any valid APL expression which produces a string. For instance:

```
a = @ c: \myproject's\myutils.dll            "
 □reference a
```

Or to reference by name:

```
□reference   System.Windows.Forms            "
```

Both will return true if successful and false if it fails to load the assembly.

Once an assembly has been loaded, you can then use the namespaces in that assembly, for example:

```
□using  myutils     "       "
 □using  System.Windows.Forms            "
```

You can also specify an alias for a using like this:

```
□using win =   System.Windows.Forms            "
```

The variable win will now contain the System.Windows.Forms assembly information.

Aliases are used to avoid name conflicts between assemblies.

As these are evaluated during execution, any valid APL expression can create the input to these system quad functions.

However, if you are using an alias it must be the first assignment in the expression before the □using.

□ex Expunge

Erases a global object or sets a local object to it's default value. Returns a 1 if successful, or a 0 if the object could not be erased or set to its default value.

Local variables that are dynamic are set to the default value for the data type which they contain when □ex is run on them. If they contain a ValueType they are set to the default for the particular value type, otherwise they are set to null.

Local variables that are strong typed are set to the default value for the data type which they must always contain. If a local variable is typed to int, then the erase will always set the value to 0, a Boolean type is set to false, etc. If a local typed variable is typed to a non ValueType then the value is set to null.

Setting a local variable to null will cause the garbage collector to remove the object to which the variable referred.

Erasing a global object removes the pointer to the object from the global dictionary and the object referenced is removed at the next garbage collection.

```
      a = 10
      b = 10 20 30
      □ex   a   b        " " " "
1 1
```

The .Net framework documentation has a large section on garbage collection and the garbage collection class is available on System.GC

You should read the documentation and examples available from Microsoft very carefully before using GC.

⎕io Index Origin

This is the index origin that the operators will use for indexing and numbering.

For instance, setting ⎕io to 0:

```
      ⍳10
0 1 2 3 4 5 6 7 8 9
      ⎕io←1
      ⍳10
1 2 3 4 5 6 7 8 9 10
```

Conversely, indexing with ⎕io set to 0, which is the default for .Net languages results as follows:

```
      a = 1 2 3 4 5
      a[1]
2
      ⎕io←1
      a[1]
1
```

Note that when a type has an indexer you must honor the ⎕io of that type. Setting ⎕io will always affect the operators and indexing of arrays, however, specific types with indexers will still have their own internal origin which must be honored.

⎕io is local to the class. There is a ⎕io for each instance of a class and also the static version.

□monadic

Indicates whether the user defined function was called with a left argument or not. □monadic is true if the function was called without a left argument.

```
▿r←a add b {
      if (monadic) {
            r←b
      } else {
            r←a+b
      }
}

      add 10
10
      10 add 20
20
```

⎕nc Name Class

Monadic:
Returns a vector of integers indicating the type of object identified within a string as the right argument. The valid identifiers are:

Identifier	Meaning
0:	Does not exist in present scope
2:	Variable, Field or property
3:	Function or method
4:	Other, most likely a class

Example:

```
      ⎕nc  a b c
2 3 0
```

This would indicate that a is a variable, b is a function and c does not exist.

One of the most common uses for ⎕nc is to identify if a left argument has been passed to a user defined function. See ⎕monadic to simplify and speed up this test.

```
∇ r←a add b {
      if (0 == ⎕nc  a ) {
            r←b
      } else {
            r←a+b
      }
}


∇ r←a add b {
      if (⎕monadic) {
            r←b
      } else {
            r←a+b
      }
}
```

□nl Name List

Returns a string array of objects that match the following numeric identifiers:

identifier	object type
2:	variable, property or field
3:	function or method

Example:

```
      □nl 2
a   b
      □nl 3
  fn
      □nl 2 3
a   b   fn
```

□nfi

□nfi provides the instance of the NumberFormatInfo class which is used by □fmt and pattern format (₸)
Changes the properties of this object are reflected in the subsequent formatting output.

```
      nfi = □nfi
     nfi.NegativeSign =  -            " "
       N2  □fmt -10 "   "
-10.00       "       "
       N2  □fmt -10 "-20".5
-10.00    -20.50     " "       "
```

This description by Microsoft of the way the NumberFormatInfo class is defined provides a rather complete layout of the different properties which can be set.

The values available on the NumberFormatInfo class are determined by the regional and culture settings of the computer.

There are additional members of the NumberFormatInfo class which are revealed either on the intellisense or in the detailed .Net framework information from Microsoft.

NumberFormatInfo Class

Defines how numeric values are formatted and displayed, depending on the culture.

Namespace: System.Globalization
Assembly: mscorlib (in mscorlib.dll)

This class contains information, such as currency, decimal separators, and other numeric symbols.

To create a **NumberFormatInfo** for a specific culture, create a CultureInfo for that culture and retrieve the CultureInfo.NumberFormat property. To create a **NumberFormatInfo** for the culture of the current thread, use the CurrentInfo property. To create a **NumberFormatInfo** for the invariant culture, use the InvariantInfo property for a read-only version, or use the **NumberFormatInfo** constructor for a writable version. It is not possible to create a **NumberFormatInfo** for a neutral culture.

The user might choose to override some of the values associated with the current culture of Windows through Regional and Language Options (or Regional Options or Regional Settings) in Control Panel. For example, the user might choose to display the date in a different format or to use a currency other than the default for the culture. If the CultureInfo.UseUserOverride property is set to **true**, the properties of the CultureInfo.DateTimeFormat instance, the **CultureInfo.NumberFormat** instance, and the CultureInfo.TextInfo instance are also retrieved from the user settings. If the user settings are incompatible with the culture associated with the **CultureInfo** (for example, if the selected calendar is not one of the OptionalCalendars), the results of the methods and the values of the properties are undefined.

Before .NET Framework version 2.0, if the **CultureInfo.UseUserOverride** property is set to **true**, then the object reads each user-overridable property only when it is accessed for the first time. Because **NumberFormatInfo** has more than one user-overridable property, that "lazy initialization" can lead to an inconsistency between such properties when the following occurs: the application accesses one property; then the user changes to another culture or overrides properties of the current user culture through Regional and Language Options in OS Control Panel; then the application accesses a different property. For example, in a sequence like this, CurrencyGroupSeparator could be accessed; then the user could change patterns in OS control panel, and CurrencyDecimalSeparator, when accessed, would follow the new settings. Similar inconsistency will happen when user change user culture in OS control panel.

In .NET Framework version 2.0 and later, **NumberFormatInfo** does not use this "lazy initialization". Instead, it reads all user-overridable properties when it is created. There is still a tiny window of vulnerability (neither

object creation nor the user override process is atomic, so the relevant values could change in the midst of object creation), but this should be extremely rare.

Numeric values are formatted using standard or custom patterns stored in the properties of a **NumberFormatInfo**. To modify how a value is displayed, the **NumberFormatInfo** must be writable so custom patterns can be saved in its properties.

The following table lists the standard format characters for each standard pattern and the associated **NumberFormatInfo** property that can be set to modify the standard pattern.

Format Character	Description and Associated Properties
c, C	Currency format. CurrencyNegativePattern, CurrencyPositivePattern, CurrencySymbol, CurrencyGroupSizes, **CurrencyGroupSeparator**, CurrencyDecimalDigits, **CurrencyDecimalSeparator**.
d, D	Decimal format.
e, E	Scientific (exponential) format.
f, F	Fixed-point format.
g, G	General format.
n, N	Number format. NumberNegativePattern, NumberGroupSizes, NumberGroupSeparator, NumberDecimalDigits, NumberDecimalSeparator.
r, R	Roundtrip format, which ensures that floating point numbers converted to strings will have the same value when they are converted back to numbers.
x, X	Hexadecimal format.

For details about these patterns, see Standard Numeric Format Strings and Custom Numeric Format Strings.

A DateTimeFormatInfo or a **NumberFormatInfo** can be created only for the invariant culture or for specific cultures, not for neutral cultures. For more information about the invariant culture, specific cultures, and neutral cultures, see the **CultureInfo** class.

This class implements the ICloneable interface to enable duplication of **NumberFormatInfo** objects. It also implements IFormatProvider to supply formatting information to applications.

□ print string representation

The □ does not take input from the keyboard. This is handled with streams in .Net. However, the □ is used to print data to the session. In particular, evaluated expressions do not produce output to the screen inside of a function. Using □ explicitly prints output to the session using the string representation of the object.

```
      function fn(a) {
            □←a+10
            a+10
      }
      fn(10)
20
```

The print keyword performs the same action:

```
      function fn(a) {
            print a+10
            a+10
      }
      fn(10)
20
```

□rl Random Link

The .Net framework provides a random number generator and the details of the generator can be found in the .Net framework documentation from Microsoft.

The roll and deal operations rely on `□rl`, which is the random link.

The default value for `□rl` is `16807`. However, the sequence of random numbers generated will be based on the random algorithm in the .Net framework.

Example:

```
      □rl
16807
      □rl←1230303
      □rl
1230303
      ?10
2
      ?10
9
      ?10
10
      □rl←1230303
      ?10
2
      ?10
9
      ?10
10
      □rl
872203611
```

⎕sysid System Identification

Returns a string with the name of the language.

```
      ⎕sysid
Visual APL for Windows

      or
      ⎕sysid
Visual APL for Linux

      or
      ⎕sysid
Visual APL for Macintosh
```

sysver System Version

Returns a string containing the information about the current build of the language.

```
       sysver
1.0.2400  on .Net 2.0.50727.42
```

⎕fi

Converts a string or character array to numeric data. Blanks are considered as delimiters and 0 is used to replace ill formed numbers.

```
      ⎕fi '3.6 2E2 ,1 THREE 0'
3.6 200 0 0 0

      ⎕fi '6.25 ¯6.25'
6.25 ¯6.25
```

Notice that the negative is shown as a middle minus. This is because the result of ⎕fi in this case is a native double vector.

If you use ravel you will see:

```
      ,⎕fi '6.25 ¯6.25'
6.25 ¯6.25
```

Which is the display for a Visual APL data type, which is created when the data is raveled. This can be cast back to native double by simply:

```
      (double) ,⎕fi '6.25 ¯6.25'
6.25 ¯6.25
```

In which case the data is now a native double again.

It is not required to use only a string with ⎕fi. You can use several strings or numbers.

```
        ⎕fi 10
10
       ⎕fi 10  10 10 10   10"0       " "   "
10 10 10 10 100
```

This reduces the cost of catenation and concern about data types as the input to ⎕fi.

⎕ts TimeStamp

Returns the current time stamp in a seven-element integer vector consisting of the year, month, day, hour, minute, second, and millisecond.

```
    ⎕ts
2006 7 28 18 42 2 304
```

This has been largely deprecated with the DateTime object in .Net

```
      using System
      DateTime.Now
7/28/2006 6:43:18 PM
      a = DateTime.Now
```

There are innumerable properties and methods on both the DateTime class and the instance of the DateTime.Now reference. In addition there are a wide range of formatters available for the DateTime class. See ⎕fmt for use of the DateTime format information.

It is also simple to do comparisons of time:

```
       DateTime.Subtract(DateTime.Now,a)
00:50:04.2198560
```

The DateTime.Subtract method returns a TimeSpan object which has numerous methods and properties which makes the analysis of the time difference very simple.

⎕ucs Universal Character Set

Translates between integers and Unicode characters.

Example:

```
      ⎕ucs  a←⍳10       "     "
97 8592 9075 49 48
     ⎕ucs ⎕ucs  a←⍳10       "     "
a ← ⍳ 1 0
```

If the right argument is a string or characters integers are returned

□userid User ID

Returns the name of the machine on which the system is running.

```
      □userid
workstation12
```

This has been deprecated in favor of the System.Environment object.

□vi

Returns an array of 1's and 0's which represent if the data, delimited by blanks, is a well formed number representation or not.

```
      □vi '3.6 2E2 ,1 THREE 0'
1 1 0 0 1
```

□vi also takes multiple strings or numeric data as an argument.

```
      □vi 10  10 10 10   10'0       " "   "
1 1 1 1 1
```

⎕format

⎕format uses all of the intrinsic .Net formatting and also includes control of widths, for all array sizes, for instance:

```
     10  N2  ⎕Format 2'3.3'4
     23.34
```

⎕format makes it possible to apply a format specifier across an array or singleton. It also adds the ability to specify width of format, as shown above.

For Example:

```
       10  N2  ⎕format 2" 2⍴'10 11
     10.00     11.00
     10.00     11.00
```

Without width specified:

```
       N2  ⎕format 2" 2⍴'10 11111.1 30.4
10.00 11,111.10
30.40 10.00
```

Notice that there are no pre set widths for the columns. This has the advantage of not losing data when formatting, but the disadvantage of not being able to control column widths.

For example:

```
        7  N2  ⎕format 2" 2⍴'10 12345.2 30.5
  10.00*******
  30.50  10.00

        N2  ⎕format 2" 2⍴'10 12345.2 30.5
10.00 12,345.20
30.50 10.00

Format can be applied by column:

       N2   C2  ⎕for'mat" 2" 2⍴'10 20 30 40
10.00 $20.00
30.00 $40.00
```

If there are more columns than format strings, then the string are reapplied in column order:

```
       N2   C2  ⎕for'mat" 2" 4⍴'10 20 30 40
10.00 $20.00 30.00 $40.00
10.00 $20.00 30.00 $40.00
```

The same applies for column widths and formats:

```
      7  N2  10  C2  ⎕'for'mat 2" 4⍴'10 20 30 40
  10.00    $20.00  30.00    $40.00
  10.00    $20.00  30.00    $40.00
```

If column widths are specified, they must be specified for all columns.

The formats can also be specified for each element in the array:

```
      a = (2 4⍴ N2   C2   C3 "  N3 "  C4 "  N3 "  N5 "  C6 ")  " "   " "   "
```

```
      a ⎕format 2 4⍴10 20 30 40
10.00    $20.00 $30.000  40.000
$10.0000 20.000 30.00000 $40.000000
```

The formats for each element in the array can also contain width settings:

```
      a = (2 4⍴(7  N2 ) (8  C2 )"(7"  C3 )") "     "  "
      a ⎕format 2 4⍴10 20 30 40
  10.00  $20.00$30.000  40.00
  10.00  $20.00$30.000  40.00
```

In .Net an object can contain its own format information. The DateTime object contains its own format information. With ⎕format you can apply the formatting to an object in an array, DateTime.Now returns an object with the current time information. We can format it like this:

```
       d  ⎕format DateTime.Now
 7/27/2006

      F  ⎕format DateTime.Now
 Thursday, July 27, 2006 12:33:47 PM
```

These can be applied using ⎕format to an array:

```
      d   N2   F  ⎕format '2 3⍴DateTime.Now 100 DateTime.Now
  7/27/2006  100.00  Thursday, July 27, 2006 12:34:54 PM
  7/27/2006  100.00  Thursday, July 27, 2006 12:34:54 PM
```

These formatting concepts apply to all objects in the .Net framework or objects created which contain their own formatting information.

One of the difficult problems with formatting is addressing comma delimiter by region, the high minus and other issues. These can be set using ⎕nfi.

For instance:

```
      nfi = ⎕nfi
      nfi.NegativeSign =  -          " "
       N2  ⎕format ¯10 "
-10.00       "      "
      N2  ⎕format ¯10 "¯20.5
-10.00   -20.50    " "      "
```

This shows changing the high minus to a middle minus. There are many regional and culture specific formatting options which are available to be set, which are shown in the documentation or with intellisense.

Setting regional setting will also affect the formatting. This means that when your formatting is performed on a machine with a different culture set, the correct currency, command and period delimiters will be used. Of course as we have shown these can be specifically overridden using ⎕nfi.

The following outlines how to use each of the formatting specifiers. These can be used with ⎕format or uniquely on a single scalar as shown below.

For additional information on the .Net formatting structure as provided by Microsoft see the related sections in this help or the Microsoft online help.

Composite Formatting

The .NET Framework composite formatting feature takes a list of objects and a composite format string as input. A composite format string consists of fixed text intermixed with indexed placeholders, called format items, that correspond to the objects in the list. The formatting operation yields a result string that consists of the original fixed text intermixed with the string representation of the objects in the list.

The composite formatting feature is supported by methods such as Format, AppendFormat, and some overloads of WriteLine and TextWriter.WriteLine. The String.Format method yields a formatted result string, the **AppendFormat** method appends a formatted result string to a StringBuilder object, the Console.WriteLine methods display the formatted result string to the console, and the TextWriter.WriteLine method writes the formatted result string to a stream or file.

Composite Format String

A composite format string and object list are used as arguments of methods that support the composite formatting feature. A composite format string consists of zero or more runs of fixed text intermixed with one or more format items. The fixed text is any string that you choose, and each format item corresponds to an object or boxed structure in the list. The composite formatting feature returns a new result string where each format item is replaced by the string representation of the corresponding object in the list.

Consider the following **Format** code fragment.

```
Visual APL
myName = Davin ;       "      "
String.Format( Name = {0}, hours = {1:hh} , myName, DateTime.Now);
```

The fixed text is "Name = " and ", hours = ". The format items are "{0}", whose index is 0, which corresponds to the object myName, and "{1:hh}", whose index is 1, which corresponds to the object DateTime.Now.

Format Item Syntax

Each format item takes the following form and consists of the following components:

{index[**,**alignment][**:**formatString]**}**

The matching braces ("{" and "}") are required.

Index Component

The mandatory *index* component, also called a parameter specifier, is a number starting from 0 that identifies a corresponding item in the list of objects. That is, the format item whose parameter specifier is 0 formats the first object in the list, the format item whose parameter specifier is 1 formats the second object in the list, and so on.

Multiple format items can refer to the same element in the list of objects by specifying the same parameter specifier. For example, you can format the same numeric value in hexadecimal, scientific, and number format by specifying a composite format string like this: "{0:X} {0:E} {0:N}".

Each format item can refer to any object in the list. For example, if there are three objects, you can format the second, first, and third object by specifying a composite format string like this: "{1} {0} {2}". An object that is not referenced by a format item is ignored. A runtime exception results if a parameter specifier designates an item outside the bounds of the list of objects.

Alignment Component

The optional *alignment* component is a signed integer indicating the preferred formatted field width. If the value of *alignment* is less than the length of the formatted string, *alignment* is ignored and the length of the formatted string is used as the field width. The formatted data in the field is right-aligned if *alignment* is

positive and left-aligned if *alignment* is negative. If padding is necessary, white space is used. The comma is required if *alignment* is specified.

Format String Component

The optional *formatString* component is a format string that is appropriate for the type of object being formatted. Specify a numeric format string if the corresponding object is a numeric value, a date and time format string if the corresponding object is a DateTime object, or an enumeration format string if the corrersponding object is an enumeration value. If *formatString* is not specified, the general ("G") format specifier for a numeric, date and time, or enumeration type is used. The colon is required if *formatString* is specified.

Escaping Braces

Opening and closing braces are interpreted as starting and ending a format item. Consequently, you must use an escape sequence to display a literal opening brace or closing brace. Specify two opening braces ("{{") in the fixed text to display one opening brace ("{"), or two closing braces ("}}") to display one closing brace ("}"). Braces in a format item are interpreted sequentially in the order they are encountered. Interpreting nested braces is not supported.

The way escaped braces are interpreted can lead to unexpected results. For example, consider the format item "{{{0:D}}}", which is intended to display an opening brace, a numeric value formatted as a decimal number, and a closing brace. However, the format item is actually interpreted in the following manner:

1. The first two opening braces ("{{") are escaped and yield one opening brace.
2. The next three characters ("{0:") are interpreted as the start of a format item.
3. The next character ("D") would be interpreted as the Decimal standard numeric format specifier, but the next two escaped braces ("}}") yield a single brace. Because the resulting string ("D}") is not a standard numeric format specifier, the resulting string is interpreted as a custom format string that means display the literal string "D}".
4. The last brace ("}") is interpreted as the end of the format item.
5. The final result that is displayed is the literal string, "{D}". The numeric value that was to be formatted is not displayed.

One way to write your code to avoid misinterpreting escaped braces and format items is to format the braces and format item separately. That is, in the first format operation display a literal opening brace, in the next operation display the result of the format item, then in the final operation display a literal closing brace.

Processing Order

If the value to be formatted is **null** (**Nothing** in Visual Basic), an empty string ("") is returned.

If the type to be formatted implements the ICustomFormatter interface, the ICustomFormatter.Format method is called.

If the preceding step does not format the type, and the type implements the IFormattable interface, the IFormattable.ToString method is called.

If the preceding step does not format the type, the type's **ToString** method, which is inherited from the Object class, is called.

Alignment is applied after the preceding steps have been performed.

Code Examples

The following example shows one string created using composite formatting and another created using an object's **ToString** method. Both types of formatting produce equivalent results.

```
Visual APL
FormatString1 = String.Format( {0:dddd MMMM}", DateTime.Now);
FormatString2 = DateTime.Now.ToString( dddd MMMM ); "          "
```

Assuming that the current day is a Thursday in May, the value of both strings in the preceding example is Thursday May in the U.S. English culture.

The following example demonstrates formatting multiple objects, including formatting one object two different ways.

```
Visual APL
myName = Davin ; String.Format( Name = {0}, hours = {1:hh}, minutes =
{1:mm} , myName, DateTime.Now);
```

The output from the preceding string is "Name = Fred, hours = 07, minutes = 23", where the current time reflects these numbers.

The following examples demonstrate the use of alignment in formatting. The arguments that are formatted are placed between vertical bar characters (|) to highlight the resulting alignment.

```
Visual APL
myFName = Davin ; string myLName = Opals ; int myInt = 100;
FormatFName = String.Format( First Name = |{0,10}| , myFName); "
FormatLName = String.Format( Last Name = |{0,10}| , myLName); "
FormatPrice = String.Format( Price = |{0,10:C}| , myInt); "
print String.Format(FormatFName);
print String.Format (FormatLName); Console.WriteLine(FormatPrice); FormatFName
= String.Format( First Name = |{0,-10}| , myFName); "
FormatLName = String.Format( Last Name = |{0,-10}| , myLName); FormatPrice =
String.Format( Price = |{0,-10:C}| , myInt); "
print String.Format(FormatFName);
print String.Format(FormatLName);
print String.Format(FormatPrice);
```

The preceding code displays the following to the console in the U.S. English culture. Different cultures display different currency symbols and separators.

```
First Name = |  Davin|
Last Name = |  Opals|
Price = |  $100.00|
First Name = |Davin |
Last Name = | Opals |
Price = | $100.00 |
```

There is a great section on .Net formatting at:

http://msdn2.microsoft.com/en-us/library/dwhawy9k(VS.80).aspx

http://msdn2.microsoft.com/en-us/library/241ad66z(VS.80).aspx

Make sure to checkout the NumberFormatInfo object, which we reveal through nfi

For instance, if you do:

```
a = nfi
a.NegativeSign = -            " "
```

Then formatting will use the middle minus for formatting instead of the high minus.

We also support the date and time formatting strings for a date/time object, which you can create with:

```
a = DateTime.Now
```

You can find this at:

http://msdn2.microsoft.com/en-us/library/az4se3k1(VS.80).aspx

http://msdn2.microsoft.com/en-us/library/hc4ky857(VS.80).aspx

You can place the DateTime object in a matrix and then when you format it will use the correct format, as:

```
              a = DateTime.Now 10.2
              'd' 'N2' format a
  6/15/2006  10.20
```

Standard DateTime Format Strings

A standard **DateTime** format string consists of a single standard **DateTime** format specifier character that represents a custom **DateTime** format stringCustom DateTime Format Strings

The format string ultimately defines the text representation of a DateTime object that is produced by a formatting operation. Note that any **DateTime** format string that contains more than one alphabetic character, including white space, is interpreted as a custom **DateTime** format string.

Standard DateTime Format Specifiers

The following table describes the standard **DateTime** format specifiers. For examples of the output produced by each format specifier, see Standard DateTime Format Strings Output Examples.

Format specifier	Name	Description
d	Short date pattern	Represents a custom **DateTime** format string defined by the current ShortDatePattern property. For example, the custom format string for the invariant culture is "MM/dd/yyyy".
D	Long date pattern	Represents a custom **DateTime** format string defined by the current LongDatePattern property. For example, the custom format string for the invariant culture is "dddd, dd MMMM yyyy".
f	Full date/time pattern (short time)	Represents a combination of the long date (D) and short time (t) patterns, separated by a space.
F	Full date/time pattern (long time)	Represents a custom **DateTime** format string defined by the current FullDateTimePattern property. For example, the custom format string for the invariant culture is "dddd, dd MMMM yyyy HH:mm:ss".
g	General date/time pattern (short time)	Represents a combination of the short date (d) and short time (t) patterns, separated by a space.
G	General date/time pattern (long time)	Represents a combination of the short date (d) and long time (T) patterns, separated by a space.
M or m	Month day pattern	Represents a custom **DateTime** format string defined by the current MonthDayPattern property. For example, the custom format string for the invariant culture is "MMMM dd".
o	Round-trip date/time pattern	Represents a custom **DateTime** format string using a pattern that preserves time zone information. The pattern is designed to round-trip **DateTime** formats, including the **Kind** property, in text. Then the formatted string can be parsed back using Parse or ParseExact with the correct **Kind** property value. The custom format string is "yyyy'-'MM'-'dd'T'HH':'mm':'ss.fffffffK". The pattern for this specifier is a defined standard. Therefore, it is always the same, regardless of the culture used or the format provider supplied.
R or r	RFC1123 pattern	Represents a custom **DateTime** format string defined by the current RFC1123Pattern property. The pattern is a defined standard and the property is read-only. Therefore, it is always the same regardless of the

		culture used or the format provider supplied. The custom format string is "ddd, dd MMM yyyy HH':'mm':'ss 'GMT'". Formatting does not modify the value of the **DateTime** object that is being formatted. Therefore, the application must convert the value to Coordinated Universal Time (UTC) before using this format specifier.
s	Sortable date/time pattern; conforms to ISO 8601	Represents a custom **DateTime** format string defined by the current SortableDateTimePattern property. This pattern is a defined standard and the property is read-only. Therefore, it is always the same regardless of the culture used or the format provider supplied. The custom format string is "yyyy'-'MM'-'dd'T'HH':'mm':'ss".
t	Short time pattern	Represents a custom **DateTime** format string defined by the current ShortTimePattern property. For example, the custom format string for the invariant culture is "HH:mm".
T	Long time pattern	Represents a custom **DateTime** format string defined by the current LongTimePattern property. For example, the custom format string for the invariant culture is "HH:mm:ss".
u	Universal sortable date/time pattern	Represents a custom **DateTime** format string defined by the current UniversalSortableDateTimePattern property. This pattern is a defined standard and the property is read-only. Therefore, it is always the same regardless of the culture used or the format provider supplied. The custom format string is "yyyy'-'MM'-'dd HH':'mm':'ss'Z'". No time zone conversion is done when the date and time is formatted. Therefore, the application must convert a local date and time to Coordinated Universal Time (UTC) before using this format specifier.
U	Universal sortable date/time pattern	Represents a custom **DateTime** format string defined by the current **FullDateTimePattern** property. This pattern is the same as the full date/long time (F) pattern. However, formatting operates on the Coordinated Universal Time (UTC) that is equivalent to the **DateTime** object being formatted.
Y or y	Year month pattern	Represents a custom **DateTime** format string defined by the current YearMonthPattern property. For example, the custom format string for the invariant culture is "yyyy MMMM".
Any other single character	(Unknown specifier)	An unknown specifier throws a runtime format exception.

Control Panel Settings

The settings in the **Regional and Language Options** item in Control Panel influence the result string produced by a formatting operation. Those settings are used to initialize the DateTimeFormatInfo object associated with the current thread culture, which provides values used to govern formatting. Computers using different settings will generate different result strings.

DateTimeFormatInfo Properties

Formatting is influenced by properties of the current **DateTimeFormatInfo** object, which is provided implicitly by the current thread culture or explicitly by the IFormatProvider parameter of the method that invokes formatting. Specify for the **IFormatProvider** parameter a CultureInfo object, which represents a culture, or a **DateTimeFormatInfo** object.

Many of the standard **DateTime** format specifiers are aliases for formatting patterns defined by properties of the current **DateTimeFormatInfo** object. Therefore, your application can change the result produced by some standard **DateTime** format specifiers by changing the corresponding **DateTimeFormatInfo** property.

Using Standard Format Strings

The following code fragment illustrates how to use the standard format strings with **DateTime** objects.

```
// This code example demonstrates the ToString(String) and
// ToString(String, IFormatProvider) methods for the DateTime
// type in conjunction with the standard date and time
// format specifiers.

using System;
using System.Globalization;
using System.Threading;

    function fn()
    {
    msgShortDate =  (d) Short date: . . . . . ;           "
    msgLongDate  =  (D) Long date: . . . . . ;            "
    msgShortTime =  (t) Short time: . . . . . ;           "
    msgLongTime  =  (T) Long time: . . . . . ;            "
    msgFullDateShortTime =
                        (f) Full date/short time: . . ;           "
    msgFullDateLongTime =
                        (F) Full date/long time: . . . ;          "
    msgGeneralDateShortTime =
                        (g) General date/short time: . ;          "
    msgGeneralDateLongTime =
                        (G) General date/long time (default): \n  +
                            . . . . ." . . . . . ;            "
    msgMonth    =     (M) Month: . ." . . . . . ;           "
    msgRFC1123 =      (R) RFC1123: ." . . . . . ;           "
    msgSortable =     (s) Sortable:" . . . . . ;            "
    msgUniSortInvariant =
                        (u) Universal" sortable (invariant): \n  +          "
                            . . . . ." . . . . . ;            "
    msgUniSort =      (U) Universal" sortable: . . . ;      "
    msgYear =         (Y) Year: . ." . . . . . ;            "

    msg1 =  Use ToString("String) and the current thread culture.\n ;
    msg2 =  Use ToString("String, IFormatProvider) and a specified culture.\n ;
    msgCulture  =  Culture: ;   "         "
    msgThisDate =  This date and" time: {0}\n ;           "

    thisDate  = DateTime.Now;
    utcDate  = thisDate.ToUniversalTime();

// Format the current date and time in various ways.
    print String.Format( Standard Date"Time Format Specifiers:\n );          "
    print String.Format(msgThisDate, thisDate);
    print String.Format(msg1);

// Display the thread current culture, which is used to format the values.
    ci = Thread.CurrentThread.CurrentCulture;
    print String.Format( {0,-30}{1}\n", msgCulture," ci.DisplayName);

    print String.Format(msgShortDate            + thisDate.ToString( d ));
    print String.Format(msgLongDate             + thisDate.ToString( D ));
    print String.Format(msgShortTime            + thisDate.ToString( t ));
    print String.Format(msgLongTime             + thisDate.ToString( T ));
```

```
    print String.Format(msgFullDateShortTime    + thisDate.ToString( f ));
    print String.Format(msgFullDateLongTime     + thisDate.ToString( F ));
    print String.Format(msgGeneralDateShortTime + thisDate.ToString( g ));
    print String.Format(msgGeneralDateLongTime  + thisDate.ToString( G ));
    print String.Format(msgMonth                + thisDate.ToString( M ));
    print String.Format(msgRFC1123              +  utcDate.ToString( R ));
    print String.Format(msgSortable             + thisDate.ToString( s ));
    print String.Format(msgUniSortInvariant     +  utcDate.ToString( u ));
    print String.Format(msgUniSort              + thisDate.ToString( U ));
    print String.Format(msgYear                 + thisDate.ToString( Y ));
    print String.Format();

// Display the same values using a CultureInfo object. The CultureInfo class
// implements IFormatProvider.
    print String.Format(msg2);

// Display the culture used to format the values.
    ci = new CultureInfo( de-DE );     "     "
    print String.Format( {0,-30}{1}\n", msgCulture,"ci.DisplayName);

    print String.Format(msgShortDate            + thisDate.ToString( d , ci));
    print String.Format(msgLongDate             + thisDate.ToString( D , ci));
    print String.Format(msgShortTime            + thisDate.ToString( t , ci));
    print String.Format(msgLongTime             + thisDate.ToString( T , ci));
    print String.Format(msgFullDateShortTime    + thisDate.ToString( f , ci));
    print String.Format(msgFullDateLongTime     + thisDate.ToString( F , ci));
    print String.Format(msgGeneralDateShortTime + thisDate.ToString( g , ci));
    print String.Format(msgGeneralDateLongTime  + thisDate.ToString( G , ci));
    print String.Format(msgMonth                + thisDate.ToString( M , ci));
    print String.Format(msgRFC1123              +  utcDate.ToString( R , ci));
    print String.Format(msgSortable             + thisDate.ToString( s , ci));
    print String.Format(msgUniSortInvariant     +  utcDate.ToString( u , ci));
    print String.Format(msgUniSort              + thisDate.ToString( U , ci));
    print String.Format(msgYear                 + thisDate.ToString( Y , ci));
    print String.Format();
    }
}
/A
```

This code example produces the following results:

```
Standard DateTime Format Specifiers:

This date and time: 1/9/2006 4:20:35 PM

Use ToString(String) and the current thread culture.

Culture:                      English (United States)

(d) Short date: . . . . . 4/17/2006
(D) Long date: . . . . . Monday, April 17, 2006
(t) Short time: . . . . . 2:38 PM
(T) Long time: . . . . . 2:38:09 PM
(f) Full date/short time: . . Monday, April 17, 2006 2:38 PM
(F) Full date/long time: . . . Monday, April 17, 2006 2:38:09 PM
(g) General date/short time: . 4/17/2006 2:38 PM
(G) General date/long time (default): . . 4/17/2006 2:38:09 PM
(M) Month: . . . . . April 17
(R) RFC1123: . . . . . Mon, 17 Apr 2006 21:38:09 GMT
(s) Sortable: . . . . . 2006-04-17T14:38:09
(u) Universal sortable (invariant): . . . 2006-04-17 21:38:09Z
(U) Universal sortable: . . . Monday, April 17, 2006 9:38:09 PM
(Y) Year: . . . . . April, 2006
(o) Roundtrip (local): . . . . 2006-04-17T14:38:09.9417500-07:00
```

```
(o) Roundtrip (UTC): . . . . . 2006-04-17T21:38:09.9417500Z
(o) Roundtrip (Unspecified): . 2000-03-20T13:02:03.0000000

Use ToString(String, IFormatProvider) and a specified culture.

Culture:                       German (Germany)

(d) Short date: . . . . . 17.04.2006
(D) Long date: . . . . . Montag, 17. April 2006
(t) Short time: . . . . . 14:38
(T) Long time: . . . . . 14:38:09
(f) Full date/short time: . . Montag, 17. April 2006 14:38
(F) Full date/long time: . . . Montag, 17. April 2006 14:38:09
(g) General date/short time: . 17.04.2006 14:38
(G) General date/long time (default): . . . . 17.04.2006 14:38:09
(M) Month: . . . . . 17 April
(R) RFC1123: . . . . . Mon, 17 Apr 2006 21:38:09 GMT
(s) Sortable: . . . . . 2006-04-17T14:38:09
(u) Universal sortable (invariant): . . 2006-04-17 21:38:09Z
(U) Universal sortable: . . . Montag, 17. April 2006 21:38:09
(Y) Year: . . . . . April 2006
(o) Roundtrip (local): . . . . 2006-04-17T14:38:09.9417500-07:00
(o) Roundtrip (UTC): . . . . . 2006-04-17T21:38:09.9417500Z
(o) Roundtrip (Unspecified): . 2000-03-20T13:02:03.0000000


A/
```

Standard DateTime Format Strings Output Examples

The following table illustrates the output created by applying some standard **DateTime** format strings to a particular date and time. Output was produced using the **ToString** method.

The Format string column indicates the format specifier, the Culture column indicates the culture associated with the current thread, and the Output column indicates the result of formatting.

The different culture values demonstrate the impact of changing the current culture. The culture can be changed by the settings in the **Regional and Language Options** item in Control Panel, or by passing your own DateTimeFormatInfo or CultureInfo class as the format provider. Note that changing the culture does not influence the output produced by the 'r' and 's' formats.

Short Date Pattern

Format string	Current culture	Output
d	en-US	4/10/2001
d	en-NZ	10/04/2001
d	de-DE	10.04.2001

Long Date Pattern

Format string	Current culture	Output
D	en-US	Tuesday, April 10, 2001

Long Time Pattern

Format string	Current culture	Output
T	en-US	3:51:24 PM
T	es-ES	15:51:24

Full Date/Time Pattern (Short Time)

Format string	Current culture	Output
f	en-US	Tuesday, April 10, 2001 3:51 PM
f	fr-FR	mardi 10 avril 2001 15:51

RFC1123 Pattern

Format string	Current culture	Output
r	en-US	Tue, 10 Apr 2001 15:51:24 GMT
r	zh-SG	Tue, 10 Apr 2001 15:51:24 GMT

Sortable Date/Time Pattern (ISO 8601)

Format string	Current culture	Output
s	en-US	2001-04-10T15:51:24
s	pt-BR	2001-04-10T15:51:24

Universal Sortable Date/Time Pattern

Format string	Current culture	Output
u	en-US	2001-04-10 15:51:24Z
u	sv-FI	2001-04-10 15:51:24Z

Month Day Pattern

Format string	Current culture	Output

m	en-US	April 10
m	ms-MY	10 April

Year Month Pattern

Format string	Current culture	Output
y	en-US	April, 2001
y	af-ZA	April 2001

An Invalid Pattern

Format string	Current culture	Output
L	en-UZ	Unrecognized format specifier; a format exception is thrown.

This is a more detailed description of the DateTime formatter.

Standard Numeric Format Strings

Standard numeric format strings are used to format common numeric types. A standard numeric format string takes the form **Axx**, where **A** is an alphabetic character called the format specifier, and **xx** is an optional integer called the precision specifier. The precision specifier ranges from 0 to 99 and affects the number of digits in the result. Any numeric format string that contains more than one alphabetic character, including white space, is interpreted as a custom numeric format string.

The following table describes the standard numeric format specifiers. For examples of the output produced by each format specifier, see Standard Numeric Format Strings Output Examples. For more information, see the notes that follow the table.

Format specifier	Name	Description
C or c	Currency	The number is converted to a string that represents a currency amount. The conversion is controlled by the currency format information of the current NumberFormatInfo (`□nfi`) object. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default currency precision given by the current NumberFormatInfo (`□nfi`) object.
D or d	Decimal	This format is supported only for integral types. The number is converted to a string of decimal digits (0-9), prefixed by a minus sign if the number is negative. The precision specifier indicates the minimum number of digits desired in the resulting string. If required, the number is padded with zeros to its left to produce the number of digits given by the precision specifier.
E or e	Scientific (exponential)	The number is converted to a string of the form "-d.ddd…E+ddd" or "-d.ddd…e+ddd", where each 'd' indicates a digit (0-9). The string starts with a minus sign if the number is negative. One digit always precedes the decimal point. The precision specifier indicates the desired number of digits after the decimal point. If the precision specifier is omitted, a default of six digits after the decimal point is used. The case of the format specifier indicates whether to prefix the exponent with an 'E' or an 'e'. The exponent always consists of a plus or minus sign and a minimum of three digits. The exponent is padded with zeros to meet this minimum, if required.
F or f	Fixed-point	The number is converted to a string of the form "-ddd.ddd…" where each 'd' indicates a digit (0-9). The string starts with a minus sign if the number is negative. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default numeric precision given by the current NumberFormatInfo (`□nfi`) object.
G or g	General	The number is converted to the most compact of either fixed-point or scientific notation, depending on the type of the number and whether a precision specifier is present. If the precision specifier is omitted or zero, the type of the number determines the default precision, as indicated by the following list. • **Byte** or **SByte**: 3 • **Int16** or **UInt16**: 5

		• **Int32** or **UInt32**: 10 • **Int64** or **UInt64**: 19 • **Single**: 7 • **Double**: 15 • **Decimal**: 29 Fixed-point notation is used if the exponent that would result from expressing the number in scientific notation is greater than -5 and less than the precision specifier; otherwise, scientific notation is used. The result contains a decimal point if required and trailing zeroes are omitted. If the precision specifier is present and the number of significant digits in the result exceeds the specified precision, then the excess trailing digits are removed by rounding. The exception to the preceding rule is if the number is a **Decimal** and the precision specifier is omitted. In that case, fixed-point notation is always used and trailing zeroes are preserved. If scientific notation is used, the exponent in the result is prefixed with 'E' if the format specifier is 'G', or 'e' if the format specifier is 'g'.
N or n	Number	The number is converted to a string of the form "-d,ddd,ddd.ddd…", where '-' indicates a negative number symbol if required, 'd' indicates a digit (0-9), ',' indicates a thousand separator between number groups, and '.' indicates a decimal point symbol. The actual negative number pattern, number group size, thousand separator, and decimal separator are specified by the current NumberFormatInfo object. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default numeric precision given by the current NumberFormatInfo object.
P or p	Percent	The number is converted to a string that represents a percent as defined by the NumberFormatInfo.PercentNegativePattern property if the number is negative, or the NumberFormatInfo.PercentPositivePattern property if the number is positive. The converted number is multiplied by 100 in order to be presented as a percentage. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default numeric precision given by the current NumberFormatInfo object.
R or r	Round-trip	This format is supported only for the Single and Double types. The round-trip specifier guarantees that a numeric value converted to a string will be parsed back into the same numeric value. When a numeric value is formatted using this specifier, it is first tested using the general format, with 15 spaces of precision for a **Double** and 7 spaces of precision for a **Single**. If the value is successfully parsed back to the same numeric value, it is formatted using the general format specifier. However, if the value is not successfully parsed back to the same numeric value, then the value is formatted using 17 digits of precision for a **Double** and 9 digits of precision for a **Single**. Although a precision specifier can present, it is ignored. Round trips are given precedence over precision when using this specifier.
X or x	Hexadecimal	This format is supported only for integral types. The number is converted to a string of hexadecimal digits. The case of the format specifier indicates whether to use uppercase or lowercase characters for the hexadecimal digits greater than 9. For example, use 'X' to produce "ABCDEF", and 'x'

		to produce "abcdef". The precision specifier indicates the minimum number of digits desired in the resulting string. If required, the number is padded with zeros to its left to produce the number of digits given by the precision specifier.
Any other single character	(Unknown specifier)	(An unknown specifier throws a runtime format exception.)

Notes

Control Panel Settings

The settings in the **Regional and Language Options** item in Control Panel influence the result string produced by a formatting operation. Those settings are used to initialize the NumberFormatInfo object associated with the current thread culture, and the current thread culture provides values used to govern formatting. Computers using different settings will generate different result strings.

NumberFormatInfo Properties

Formatting is influenced by properties of the current **NumberFormatInfo** object, which is provided implicitly by the current thread culture or explicitly by the IFormatProvider parameter of the method that invokes formatting. Specify a **NumberFormatInfo** or CultureInfo object for that parameter.

Integral and Floating-Point Numeric Types

Some descriptions of standard numeric format specifiers refer to integral or floating-point numeric types. The integral numeric types are Byte, SByte, Int16, Int32, Int64, UInt16, UInt32, and UInt64. The floating-point numeric types are Decimal, **Single**, and **Double**.

Floating-Point Infinities and NaN

Note that regardless of the format string, if the value of a **Single** or **Double** floating-point type is positive infinity, negative infinity, or Not a Number (NaN), the formatted string is the value of the respective PositiveInfinitySymbol, NegativeInfinitySymbol, or NaNSymbol property specified by the currently applicable **NumberFormatInfo** object.

Example

The following code example formats an integral and a floating-point numeric value using the thread current culture, a specified culture, and all the standard numeric format specifiers. This code example uses two particular numeric types, but would yield similar results for any of the numeric base types (**Byte**, **SByte**, **Int16**, **Int32**, **Int64**, **UInt16**, **UInt32**, **UInt64**, **Decimal**, **Single**, and **Double**).

This example provides an excellent example of discreetly formatting an individual scalar and accessing resource information about the formatting object. □fmt applies these formatting techniques to arrays as well as scalars.

```
// This code example demonstrates the ToString(String) and
// ToString(String, IFormatProvider) methods for integral and
// floating-point numbers, in conjunction with the standard
// numeric format specifiers.
// This code example uses the System.Int32 integral type and
// the System.Double floating-point type, but would yield
// similar results for any of the numeric types. The integral
// numeric types are System.Byte, SByte, Int16, Int32, Int64,
// UInt16, UInt32, and UInt64. The floating-point numeric types
// are Decimal, Single, and Double.

using System;
using System.Globalization;
using System.Threading;

    function fn()
```

```
    {
// Format a negative integer or floating-point number in various ways.
    integralVal = -12345;
    floatingVal = -1234.567d;

    msgCurrency =     (C) Currency:" . . . . . ;           "
    msgDecimal  =     (D) Decimal: ." . . . . . ;          "
    msgScientific =   (E) Scientific: . . . . . ;          "
    msgFixedPoint =   (F) Fixed point: . . . . . ;         "
    msgGeneral =      (G) General (default): . . ;          "
    msgNumber =       (N) Number: ." . . . . . ;           "
    msgPercent =      (P) Percent: ." . . . . . ;          "
    msgRoundTrip =    (R) Round-trip: . . . . . ;          "
    msgHexadecimal =  (X) Hexadecimal: . . . . . ;         "

    msg1 =  Use ToString("String) and the current thread culture.\n ;
    msg2 =  Use ToString("String, IFormatProvider) and a specified culture.\n ;
    msgCulture     =  Culture: ;   "        "
    msgIntegralVal =  Integral value: ;           "
    msgFloatingVal =  Floating-point value: ;           "

    CultureInfo ci;
    print  Standard Numeric Format Specifiers:\n ;          "
// Display the values.
    print msg1;

// Display the thread current culture, which is used to format the //values.
    ci = Thread.CurrentThread.CurrentCulture;
    print String.Format( {0,-26}{1} , "msgCulture," ci.DisplayName);

// Display the integral and floating-point values.
    print String.Format( {0,-26}{1} , "msgIntegralVal, integralVal);
    print String.Format( {0,-26}{1} , "msgFloatingVal, floatingVal);
    print              ""
// Use the format specifiers that are only for integral types.
    print ( Format specifiers only for integral types: );          "
    print String.Format(msgDecimal     + integralVal.ToString( D ));       " "
    print String.Format(msgHexadecimal + integralVal.ToString( X ));       " "
    print   ;          ""

// Use the format specifier that is only for the Single and Double
// floating-point types.
    print ( Format specifier only for the Single and Double types: );
    print String.Format(msgRoundTrip   + floatingVal.ToString( R ));       " "
    print   ;          ""

// Use the format specifiers that are for integral or floating-point //types.
    print String.Format( Format specifiers for integral or floating-point
types: );          "
    print String.Format(msgCurrency    + floatingVal.ToString( C ));       " "
    print String.Format(msgScientific  + floatingVal.ToString( E ));       " "
    print String.Format(msgFixedPoint  + floatingVal.ToString( F ));       " "
    print String.Format(msgGeneral     + floatingVal.ToString( G ));       " "
    print String.Format(msgNumber      + floatingVal.ToString( N ));       " "
    print String.Format(msgPercent     + floatingVal.ToString( P ));       " "
    print   ;          ""

// Display the same values using a CultureInfo object. The //CultureInfo
class
// implements IFormatProvider.
    print (msg2);

// Display the culture used to format the values.
// Create a European culture and change its currency symbol to  euro  //    "
```

```
because this particular code example uses a thread current UI // // culture
that cannot display the euro symbol ( ).
    ci = new CultureInfo( de-DE );     "     "
    ci.NumberFormat.CurrencySymbol =  euro ;       "    "
    print  String.Format( {0,-26}{1} , 'msgCulture," ci.DisplayName);

// Display the integral and floating-point values.
    print  String.Format( {0,-26}{1} , 'msgIntegral'Val, integralVal);
    print  String.Format( {0,-26}{1} , 'msgFloatingVal, floatingVal);
    print   ;           ""

// Use the format specifiers that are only for integral types.
    print  ( Format specifiers only for integral types: );           "
    print  String.Format(msgDecimal+ integralVal.ToString( D , ci));    " "
    print  String.Format(msgHexadecimal+integralVal.ToString( X , ci));    " "
    print   ;           ""

// Use the format specifier that is only for the Single and Double
// floating-point types.
    print  String.Format( Format specifier only for the Single and Double
types: );           "
    print  String.Format(msgRoundTrip+floatingVal.ToString( R , ci));    " "
    print   ;           ""

// Use the format specifiers that are for integral or floating-point types.
    print  String.Format( Format specifiers for integral or floating-point
types: );           "
    print  String.Format(msgCurrency+floatingVal.ToString( C , ci));    " "
    print  String.Format(msgScientific+floatingVal.ToString( E , ci));    " "
    print  String.Format(msgFixedPoint+floatingVal.ToString( F , ci));    " "
    print  String.Format(msgGeneral + floatingVal.ToString( G , ci));    " "
    print  String.Format(msgNumber + floatingVal.ToString( N , ci));    " "
    print  String.Format(msgPercent + floatingVal.ToString( P , ci));    " "
    print   ;           ""
    }

/A
```

This code example produces the following results:

Standard Numeric Format Specifiers:

Use ToString(String) and the current thread culture.

```
Culture:                  English (United States)
Integral value:           -12345
Floating-point value:     -1234.567

Format specifiers only for integral types:
(D) Decimal:. . . . . -12345
(X) Hexadecimal:. . . . . FFFFCFC7

Format specifier only for the Single and Double types:
(R) Round-trip: . . . . . -1234.567

Format specifiers for integral or floating-point types:
(C) Currency: . . . . . ($1,234.57)
(E) Scientific: . . . . . -1.234567E+003
(F) Fixed point:. . . . . -1234.57
(G) General (default):. . -1234.567
(N) Number: . . . . . -1,234.57
```

```
(P) Percent: . . . . . -123,456.70 %

Use ToString(String, IFormatProvider) and a specified culture.

Culture:                  German (Germany)
Integral value:           -12345
Floating-point value:     -1234.567

Format specifiers only for integral types:
(D) Decimal: . . . . . -12345
(X) Hexadecimal: . . . . . FFFFCFC7

Format specifier only for the Single and Double types:
(R) Round-trip: . . . . . -1234,567

Format specifiers for integral or floating-point types:
(C) Currency: . . . . . -1.234,57 euro
(E) Scientific: . . . . . -1,234567E+003
(F) Fixed point: . . . . . -1234,57
(G) General (default): . . -1234,567
(N) Number: . . . . . -1.234,57
(P) Percent: . . . . . -123.456,70 %

A/
```

This shows the formatting specifiers for the DateTime object.

□fmt

This documentation describes the supported feature set of the legacy □fmt system function.

□format in Visual APL provides support for all of the .Net formatting modifiers across arrays.

□fmt feature set:

□fmt - legacy formatter which returns character matrices with fixed width columns

The following elements of the legacy □fmt have been implemented for compatibility purposes.

Syntax:

```
res = 'fstring' □fmt data
```

'fstring' : character vector containing one or more editing phrases.

data : an array

Editing phrases:

rmAw	Character
rmEw.s	Exponential
rmFw.d	Fixed point
rmG<pattern>	Pattern
rmIw	Integer

d = Decimal positions

s = Significant digits

w = Field width

<pattern> = Example

Positioning and text phrases:

r = Repetition (optional)

m = Modifiers (optional)

Modifiers:

B Blank if zero (F,I)

C Comma insertion (F,I)

L Left justify (F,I)

M<text> Negative left decoration (F,G,I)

N<text> Negative right decoration (F,G,I)

P<text> Non-negative left decoration (F,G,I)

Q<text> Non-negative right decoration (F,G,I)

Z Zero fill (F,I)

Valid delimiters for text in decorations are:

<text> ⊂text⊃ ¨text¨

□text□ ⍰text⍰ /text/

[ ] Index

Many classes have indexers.

Array indexer:

When used inside of an indexer bracket block [ ] the **;** axis separator identifies the values for each axis.

```
      a = 1 2 3
      a[1]
2
      a = 3 3⍴⍳9
      a[1 2;1 2]
 4 5
 7 8
```

It is not required to use the axis separator to index an array, for instance:

```
      b = (1 2) (1 2)
      a[b]
 4 5
 7 8
            b = 1 2
      a[b]
5
```

Providing a single value will index the array as though it were a vector.

```
       a[1]
1
```

You can select all values in an axis by using null:

```
      b = (1 2) (1 2) null
      a[b]
 12 13 14
 15 16 17

 21 22 23
 24 25 26
```

This makes it possible to index an array without having to be concerned about the syntax of the number of semi colons.

Generic Type Indexer

Indexers also occur on Generic Types. To create a Generic Type you need to first use:

```
      using System.Collections.Generic
      a = Dictionary[string, int]()
```

This will create an instance of the generic Dictionary type which accepts only string as the key, and int as the value.

```
      a.Add( test , 10)
      a.Add(100, 20)
bad args for method
      a.Count
1
```

It is not possible to use a key other than string with this Dictionary.

Method Selection Indexer

The signature of a method includes not only the name of the method, but also the types and number of arguments of the method.

To pre-select a particular method, indexing is available. As an example, an instance of string has a method named IndexOf which has 9 overloads. To select a specific overload:

```
      a =  test         "    "
      a.IndexOf[string, int]( es ,1)       "  "
1
      a.IndexOf( es ,1)
1
```

In the vast majority of cases using the method indexer is not needed, but in some cases it can be quite beneficial. However, if the goal is to let the system select the best method for the dynamic values being used as arguments, then do not use the indexer.

← Assignment By Value and = Assign By Reference

The left assign arrow assigns data by value. This means that a copy of the data is made if possible. If it is not possible to make a copy of the data, a reference assignment is made.

Because this provides control over when assignment by value and assignment by reference will be made, discretion should be used when choosing to do assignment by value as copying all the data is considerably more expensive than assignment by reference. In general, there are relatively few occasions when assignment by value is required, which is one of the reasons it does not exist in other .Net languages.

For objects that are composed of ValueTypes, the copy is always made. However, for example, if an array contains an instance of a Form, then the Form is assigned by reference as creating another copy of the Form could have unintended consequences.

The = symbol is used for assign by reference, which matches the assignment behavior of other .Net languages. The ≈ symbol is used for comparison, or the double == symbol.

Example:

```
      a = ⍳10
      b←a
      a[3] = 100
      a
0 1 2 100 4 5 6 7 8 9
      b
0 1 2 3 4 5 6 7 8 9
      a = ⍳10
      b = a
      a[3] = 100
      a
0 1 2 100 4 5 6 7 8 9
      b
0 1 2 100 4 5 6 7 8 9
```

Simple assignment:

```
      a ← 10
      a b c ← 10 20 30
```

Assigns one value to each variable

```
      a b c ← ⊂10 20 30
```

Assigns the nested array 10 20 30 into each variable.

It is also possible to assign nested arrays by nesting shape.

```
      a (b c) d = 10 20 30
This makes a: 10, b: 20, c: 20 and d: 30

      a (b c) d = 10 (20 30) 40
In this case a: 10, b: 20, c: 30, d: 40
      x = 10 (20 (30 40)) 50
      a (b c) d = x
a: 10, b: 20, c: 30 40, d: 50
```

These assignment rules also apply when using for loops.

Matrix assignment:

```
      a←3 3⍴⍳9
      a
 0 1 2
 3 4 5
 6 7 8
      a[1 2;1 2]←2 2⍴10
      a
 0  1  2
 3 10 10
 6 10 10
```

Inline assignment works as follows:

```
      a ← 1+b ← 10+4
      a
15
      b
14
```

Selective assignment is also supported and is based on the original definition of selective assignment created by Jim Brown in his paper "Understanding Selective Assignment", 1989

"The notion of selective assignment is simple. If you can write an expression which selects some items at any depth in an array, then writing that same expression on the left of an assignment arrow requests replacement of the selected items."

This makes it possible to include user defined functions, the each operator, assign to more than one variable, etc.

For example:

```
      a = 1 2 3 4 5
      (1⊃a) = 10
      a = (1 2 3) (4 5 6)
      (1⊃¨a)=10
      (test a)=10
      (1 test a)=10
      a = 1 2 3 4
      b = 10 20 30 40
      ((1⊃a) (1⊃b)) = 100
      a
1 100 3 4
      b
10 100 30 40
      etc.
```

Execute

Compiles and runs a string which can be an expression or statement.

```
      1+1          "   "
2
      a = 10+3     "        "
13
      a
13
```

It is also possible to manage the executes use of local and global variables. Execute can only create global variables, local variables can not be created with execute.

```
function fn(a) {
      b = 10
      c = 20
      c = a+b      "       "
      print a
}
      fn(10)
30
```

When it is desired to pass only a subset of local variables to the execute domain:

```
function fn(a) {
    b = 10
    c = 10
    d = 20
    // only local variables a and b passed to the execute
    c = a+b  in ("a,b)    "
    print c
      // the value of c is not changed
      // a b and c are passed
    c = a+b  in ("a,b,c)  "
    print c
}
```

It is also possible to manage the global variables passed and have new variables created added to the provided Dictionary. In this example we are not passing any local variables to execute, but we could include those as well. Functions can also be localized to the excute by placing them in the dictionary. In the case the function associated with fn in the dictionary does not exist in the class or session, but only in the dictionary.

```
      d = Dictionary[object, object]()
      d.Add( var1 , 20)   "    "
      d.Add( var2 , 30)   "    "
      d.Add( x , 40)      " "
      q = var1+var2"+x  in (),d     "
false
      d.Count
5
      d[ q ]          " "
90
      q = var1+var2"+x  in (),d     "
false
      d[ var1 ] = 200 "    "
      q = var1+var2"+x  in (),d     "
false
```

```
      d[ q ]          " "
270
      d.Add( fn ,    r←(a,b"){r"←a+b})          f
      ⍎ q=fn(var1,var"2)  in (),d     "
false
      d[ q ]          " "
50
```

All of the variables used and created by the execute come from the Dictionary object. The Dictionary object inherits from IDictionary and you can create a class which inherits from IDictionary which can respond in any desired way to the execution of the code and the creation and modification of variables. For instance, you could have an event fire when a new variable is created or a value is changed, or any other action you might find useful.

This provides detailed control of the execute, and provides the ability to scope function and variables to a particular execute.

Empty numeric constant object.

This is displayed when the result of an expression evaluated in the session contains empty numeric data

⍕ Pattern format, Format

Simple formatter that provides simple width control and converts objects to their string representation. Relies on ⎕nfi

```
      ⍕2 3⍴⍳6
0 1 2
3 4 5
      (2 3⍴⍳6).ToString()
0 1 2
3 4 5
```

The ToString method in most cases is equivalent.

```
      1 0 4 1 6 2 ⍕ 2 3⍴⍳6
0 1.0  2.00
3 4.0  5.00
```

Notice that the width of each column was controlled by the left argument. The left argument is composed of value pairs, width and number of decimals.

Using a negative value for number of decimals formats objects in Exponential.

```
      10 ¯5 ⍕10 20 30 999.4
  1.0000E1  2.0000E1  3.0000E1  9.9940E2
```

The Share File System

The ShareFileSystem in Visual APL is a next generation component file system.

Not only does the ShareFileSystem support the legacy syntax common to share file systems, but it extends share file systems with virtual directories. This means you can place more than one share file in a single physical file.

To use the Share File System in your application, you will need to add a reference to the Visual APL Share/Native File System assembly. Here is an example of "referencing" and "using" the assembly by its strong name:

```
refbyname APLNext.APL.LegacyOps
using APLNext.Legacy.ShareFileSystem
```

The more Share Files that are placed in a virtual directory the better the space management becomes.

Additionally, because the ShareFileSystem uses the ISerializer .Net methodology for the IO of nested or object data types, shared and native files can read and write not only simple APL variables, but nested APL variables which even include Hashtables, Dictionaries, etc.

You can also write out the Hashtables or Dictionaries without including them in an APL variable.

Any class that inherits from ISerializable can be written to the share or native files and retrieved with the instance being automatically recreated.

□falloc

Pre-allocates a specific contiguous block in a component file as a single component.

```
      □falloc 12,1000
7
      ρ□fread 12,7
1000
```

Using this in conjunction with the index read (□firead) and index replace (□fireplace) you can easily manipulate text documents in a component.

It is also possible to retrieve the location of a component. This permits using other tools, such as □nread to access the data in a component. For instance, you could store a document in a component file, use □fcnloc to retrieve the starting point and then read the data using other tools:

```
      □fcnloc 12,7
54288
```

This is particularly useful to include images, documents and other data in a component file in a single virtual directory which needs to be accessed by other programs and tools.

□fappend

Appends a serializable object to a component file tied to the associated tie number. The append returns the component number into which the data was placed.

```
cn =  hello how are" you   □fappend 10"
cn = (3 3⍴⍳10) □fappend 10
```

□fcatenate

One of the new features of these component files is the ability to manipulate component data in place. This means that it is not necessary to read in a component and catenate data, then write the component back out. Since catenate is one of the most expensive operations, this can be very useful. Only homogenous intrinsic data types can be manipulated in place. For instance a vector of integers, doubles, chars, etc. can be modified. However, nested arrays can not.

Example:

```
      (⍳5) □fappend 12
4
      □fread 12,4
0 1 2 3 4
      10 11 12 □fcatenate 12,4
      □fread 12,4
0 1 2 3 4 10 11 12
```

□fdrop

□fdrop removes components from the beginning or end of a Share File.

Syntax:

```
□fdrop tn dropCount
```

tn: The tie number of the file to drop components from.

dropCount: An integer specifying the number of components to drop from the file.

Remarks:

□fdrop will remove the specified number of components from either the beginning or end of the specified share file.

If the *dropCount* is a positive number, that number of components will be removed from the beginning of the Share File. If the *dropCount* is a negative integer, then that number of components will be removed from the end of the Share File.

Legacy Considerations

□fdrop duplicates the syntax of the legacy □fdrop, but has one difference, when you drop components from the front of a file, the components that remain are renumbered from 1 instead of retaining their original numbers. Since the Share File System is structured to give data back to the virtual pool, artificially numbering component offsets after a drop would have introduced many unwanted exceptions to the Share File System.

Example:

```
// drop 5 components from the beginning of the share
// file at tie number 1.
□fdrop 1 5
```

ferase

Removes a specified component file from a virtual directory. This does not delete a physical file. The tie number must be the number associated with the file name to be erased.

```
filename   "ferase 10 "
```

□fcreate

Has two primary uses:

1. Create a component file and associated virtual directory of the same name.
For instance:

```
"some file name.extension" □fcreate 10
```

Or

```
tn = □fcreate "some file name.extension"
```

Which returns the next available tie number.

Both of these create a file in the current directory. You could also specify the entire path:

```
tn = □fcreate @"c:\mydir\subdir\some file name.extension"
```

This use primarily exists for legacy system support. All of the above examples create a vitual directory with the same name as the fileid specified. This example further illustrates the point:

```
@ c: \test \myfil"e  □fcreate 1  "
```

In the above example, a virtual directory is created with the same name as the fileid, "myfile", in the "c:\test" directory, and then creates a share file in that virtual directory with the same name.

Advantages over legacy file systems

One of the primary advantages of the Share File System is that not only can you place more than one share file in the virtual directories, but the share file system recovers data as it becomes available, thus avoiding the explosion of size common in some legacy share file systems.

Note
The use of the @ symbol to indicate a raw string, this obviates the need to use the \ as an escape character, as "c:\\mydir\\subdir\\some file name.extension"

2. When used with a library number, it creates a component file in the virtual directory associated with the library number.

```
"100 some file name.extension" □fcreate 10
```

Or

```
tn = □fcreate   10 some file name.extension"
// the system chose the tie number, as none was specified
```

Or

```
tn = "100 some file name.extension" □fcreate 0
// the system chose the tie number, as a 0 was specified.
```

□firead

This provides the ability to read a subset of an intrinsic array using index read. This reduces the need to read large amounts of data into memory for the purpose of indexing only a subset. Used in conjunction with □ fi replace and □fcatenate it makes the management of discrete data within a component file very simple.

```
      (⍳20) □fappend 12
6

      □firead 12,6,10,3
10 11 12

       how are you t'bday  □fappend 12 "
7
      □firead 12,7,4,3
are
```

□fireplace

The data in a component file can also be replaced in place using **index replace**: This obviates the need to read the data into memory, make the change, and then rewrite the data to disk. In this case the data is replaced on disk explicitly without reading the entire component into memory.

```
      (⍳20) □fappend 12
6
      100 200 300 □fireplace 12,6,10,3
      □fread 12,6
0 1 2 3 4 5 6 7 8 9 100 200 300 13 14 15 16 17 18 19
```

Catenating and modifying integers in place is extremely useful when updating pointers, such as are used as references. This significantly reduces the time and space required to maintain systems which require reading and modifying large arrays of integers, doubles, characters, etc.

In the event more data is provided than allocated for by the arguments, then only the first n elements of the data is used in the replacement:

```
      85 86 87 88 □fireplace 12,6,10,3

      □fread 12,6
0 1 2 3 4 5 6 7 8 9 85 86 87 13 14 15 16 17 18 19
```

□fnames

Returns a string array of strings. This is useful for manipulation with .Net classes such as generic List.

```
      a = □fnames
      a.GetType()
System.String[]
```

fnums

Returns an integer array of tie numbers indicating all of the files currently associated with a tie number.

□fread

Reads a component from a component file. The syntax for this is:

```
a = □fread tn cn
```

Any arbitrary serializable data can be returned from a component. The data will be deserialized and the original object will be returned.

□freplace

Replaces the data in an existing component with an arbitrary serializable object.

```
a = 1 20 30 40.5
a □freplace tn cn
```

□fsize

Returns a five element integer vector.

```
      □fsize
1 10  0  0  0
```

The first element is the starting component, the second element is the next component which will be used. The last component in use is this element less one.

□libdup

□libdup duplicates an entire virtual directory based on the associated library number, releasing any unused space from the virtual pool of the library.

Syntax:

```
newLibNo dupPath  □libdup libNo
```

newLibNo: The library number to which *dupPath* will be associated.

dupPath: The file path at which to create the newly duplicated library.

tn: The library number for the Share File library to duplicate.

Remarks:

The □libdup system function creates a copy of the specified library.

This newly created copy of the file library contains all components and data which were present in the source library.

The only difference between the source and newly created libraries, is that the newly created library has had all unused space released from the virtual pool of the Share File.

This process decreases the physical file size of the library, since all unused space in the library has been released back to the operating system.

The inclusion of the □libdup system function is primarily for completeness in the Share File System, as the Share File System by design reclaims space as necessary from the virtual pool.

Example:

```
 @ 2 c:\test\testnew  □libdup 101 "

Where 101 is the library number for the existing virtual directory. This
will duplicate all of the files in the 101 virtual directory and place
them in c:\test\testnew which is associated with the library number 2.
```

□fdup

Visual APL includes □fdup for legacy support.

Syntax:

```
filePath □fstream tn
```

filePath: The full file path of the tied share file.

tn: The tie number of the file to dup.

Remarks:

□fdup duplicates a single file. This will only duplicate share files whose name matches the virtual directory in which they reside, and the virtual directory contains only the file being duplicated.

Example:

```
c:\myfiles\filename  □fdup 3
```

□fremove

□fremove removes the specified component from a Share File, and renumbers the remaining components.

Syntax:

```
□fremove tn compNumber
```

tn: The tie number of the file to drop the component from.

compNumber: An integer specifying the component number to remove from the file.

Remarks:

□fremove removes a single component from a Share File, returning the space used by the removed component to the virtual pool.

Example:

```
// drop component 10 from the share
// file at tie number 2.
□fremove 2 10
```

□fstream

Returns the underlying .Net FileStream object for the associated tie number. This allows the use of all features provided by the FileStream object, while still maintaining compatibility with the Share File system.

```
      fs = □fstream 3

      fs.CanRead
true
      fs.CanWrite
true
```

□fstie

Ties an existing file and associates the file with either a given tie number or the next available tie number.

```
       c: \myfiles\fi"lename  □fstie 10   "
      or
      tn = □fstie  c: \myfiles\fi"lename             "
      or
      // if a tie number of 0 is specified, the system assigns the next
available tie number.
       c: \myfiles\fi"lename  □fstie 0    "
```

It is also possible to access component files within a virtual directory created either with `□libd` or `□fcreate` by using the associated library number for a virtual directory.

```
       101  filename " □fstie 10   "

      tn = □fstie  10 filename  "           "

       101  filename " □fstie 0    "
```

In this way many component files can reside in a virtual directory, or single physical file.

funtie

Removes the tie number associated with the existing component file.

□lib

To manage your files in their virtual directory, you have □fnums and □fnames as well as □lib and □libs:

```
            □lib 10

'my2file'   'myfile'   'another'
```

Which returns an array of file names found in the virtual directory.

To remove a file from a virtual directory, use:

```
       another   □fer'ase 12  "

      □lib 10
'my2file'   'myfile'
```

To untie a file use □funtie.

□libd

Since component files reside in a single physical file, to create the physical file or virtual directory you use □libd, for instance:

```
      □libd  10 c: \\tmysf "            "
true
```

Notice that the directory path has two backslashes, as the \ is the escape character. You could have also placed an @ symbol at the beginning for raw text, for instance:

```
      □libd @ 10 c: \tmysf  "           "
true
```

Which obviates the need for the double backslash.

Once the virtual directory has been created, you can use it just like you normally use a library.

For instance:

```
      □fcreate  10 myfile    "          "
1
      □fsize 1
1 1 0 0 0

      10 □fappend 1
1
```

The component file can also be tied or created by specifying the tie number:

```
       10 another  □"fcreate 12 "
12
      □fsize 12
1 1 0 0 0
```

The tie number can be changed at any time by simply retieing:

```
       10 another  □"ftie 10    "
10
```

As with all component files, you can store disparate data types in the components and retrieve them, as well as replace component data:

```
       test  □fappend 12 "
1

      10 11 12 □fappend 12
2

      □fread 12,2
10 11 12

       test  (10 11 "12)  "morestuff  □f"append 12 "
3

      □fread 12,3
 test  10 11 12 morestuff
```

```
      (3 3   9) □freplace 12,2
      □fread 12,2
 0 1 2
 3 4 5
 6 7 8
```

One of the new features of these component files is the ability to manipulate component data in place. This means that it is not necessary to read in a component and catenate data, then write the component back out. Since catenate is one of the most expensive operations, this can be very useful. Only homogenous intrinsic data types can be manipulated in place. For instance a vector integers, doubles, chars, etc. can be modified. However, nested arrays can not.

Example:

```
      ( 5) □fappend 12
4
      □fread 12,4
0 1 2 3 4
      10 11 12 □fcatenate 12,4
      □fread 12,4
0 1 2 3 4 10 11 12
```

This file system also uses blocks to minimize file size explosion as component sizes grow.

It is also possible to manage character data:

```
      hello how are"   □fappend 12 "
5
       you?  □fcatenate 12,5

      □fread 12,5
hello how are you?
```

It is also possible to read a subset of an intrinsic array using index read:

```
      ( 20) □fappend 12
6

      □firead 12,6,10,3
10 11 12
```

The data can also be replaced in place using index replace:

```
      100 200 300 □fireplace 12,6,10,3
      □fread 12,6
0 1 2 3 4 5 6 7 8 9 100 200 300 13 14 15 16 17 18 19
```

Catenating and modifying integers in place is extremely useful when updating pointers, such as are used as references. This significantly reduces the time and space required to maintain systems which require reading and modifying large arrays of integers, double, characters, etc.

In the event more data is provided than allocated for by the arguments, then only the first n elements of the data is used in the replacement:

```
      85 86 87 88 □fireplace 12,6,10,3

      □fread 12,6
0 1 2 3 4 5 6 7 8 9 85 86 87 13 14 15 16 17 18 19
```

It is also possible to allocate a contiguous block of space as a single component:

```
      □falloc 12,1000
7
```

```
      □fread 12,7
1000
```

Using this in conjunction with the index read and replace you can easily manipulate text documents in a component.

It is also possible to retrieve the location of a component. This permits using other tools, such as □nread to access the data in a component. For instance, you could store a document in a component file, use □ fcnloc to retrieve the starting point and then read the data using other tools:

```
     □fcnloc 12,7
54288
```

This is particularly useful to include images, documents and other data in a component file in a single virtual directory which needs to be accessed by other programs and tools.

libdcws

It is also possible to control access to the virtual directory, this is done with `libdrw` for setting read only or read/write access, and `libdcws` for checking write status. Use 0 to set read only and 1 for read/write.

```
      libdcws 10
1

      libdrw 10,0
0

      libdcws 10
0

      libdrw 10,1
1

      libdcws 10
1
```

□libdrw

It is also possible to control access to the virtual directory, this is done with □libdrw for setting read only or read/write access, and □libdcws for checking write status. Use 0 to set read only and 1 for read/write.

```
      □libdcws 10
1

      □libdrw 10,0
0

      □libdcws 10
0

      □libdrw 10,1
1

      □libdcws 10
1
```

□libs

This displays a matrix of all virtual directories and their associated library numbers.

```
□libs
   2 C:\mydir\test.mf
   3 C:\mydir\test
```