APL64

User Manual

Copyright

APL64

This edition published 2026.

Table of Contents

Chapter 1: Introduction

This APL64 User Manual documents the APL64 System, an APL application development system. This system operates under Microsoft Windows 11 and newer operating systems. The user interface is designed to take advantage of the benefits of Windows and allows you to develop true Windows applications. This product also includes control structures as a facility of the APL language for writing effective, understandable, but still concise code.

This chapter of this manual describes how to install APL64.

APL64 Prerequisites

- 650 MB free disk space
- Microsoft Windows 11 and newer
- Microsoft .Net 9 SDK version 9.0.308 (x64)
  https://dotnet.microsoft.com/en-us/download/dotnet/thank-you/sdk-9.0.304-windows-x64-installer

The Setup Program

The Setup program does the following:

- Installs the files necessary to run APL64 to your hard drive
- Installs the APL64 font
- Installs the APL64 Grid Control.
- Installs the Programming Tools workspaces.
- Installs the Examples workspaces.
- Configures the Windows Registry to start APL64 when opening a workspace with a .ws64 extension.
- Adds **APL64** to the Windows **Start** menu.

Installing APL64

Run the executable file you were instructed to download from the APL2000 Software Downloads area at APL2000 » Software Downloads. Follow the prompts in the Setup program starting with the following window:

APL64_XXXX.XX.XXXX.exe

Start APL64 from the shortcut in the APL64 program in the Windows Start menu.

Modifying the APL64 xml-format configuration and APLNow32.INI files

Optional command line arguments are available in APL64 to tailor your setup to your own configuration and preferences. Refer to **Help | Command Line Arguments** menu for detailed information.

Chapter 2: Working in an APL64 Session

When you start APL64, the system opens an APL64 session and places you in the History. The APL64 session has the standard title line you can drag to move the window, buttons to minimize and maximize the window, and a system menu. It has a menu bar with pull-down menus, an optional toolbar with buttons for the commands users need most often, and an optional status line at the bottom of the window.

This chapter describes the various menu choices on the menu bar and explains how you use the features they invoke. You can select menu choices in the standard manner by clicking on an item, by highlighting a choice and pressing Enter, or by pressing the Alt key to activate the menu bar and pressing a succession of menu access keys (letters that are underlined on the menu titles and menu choices).

Since APL64 uses the Alt key in combination with other keys to generate APL characters, you need to press and release the Alt key to start a sequence of access keys. Note that pressing the Alt key by itself highlights the first menu choice on the menu bar. At this point, you can use the cursor keys or access keys to select the menu options. For example, if you press Alt and release it and then press H, you see the Help menu. If, instead, you hold the Alt key down and press H, you get the delta symbol (∆) in your History Pane.

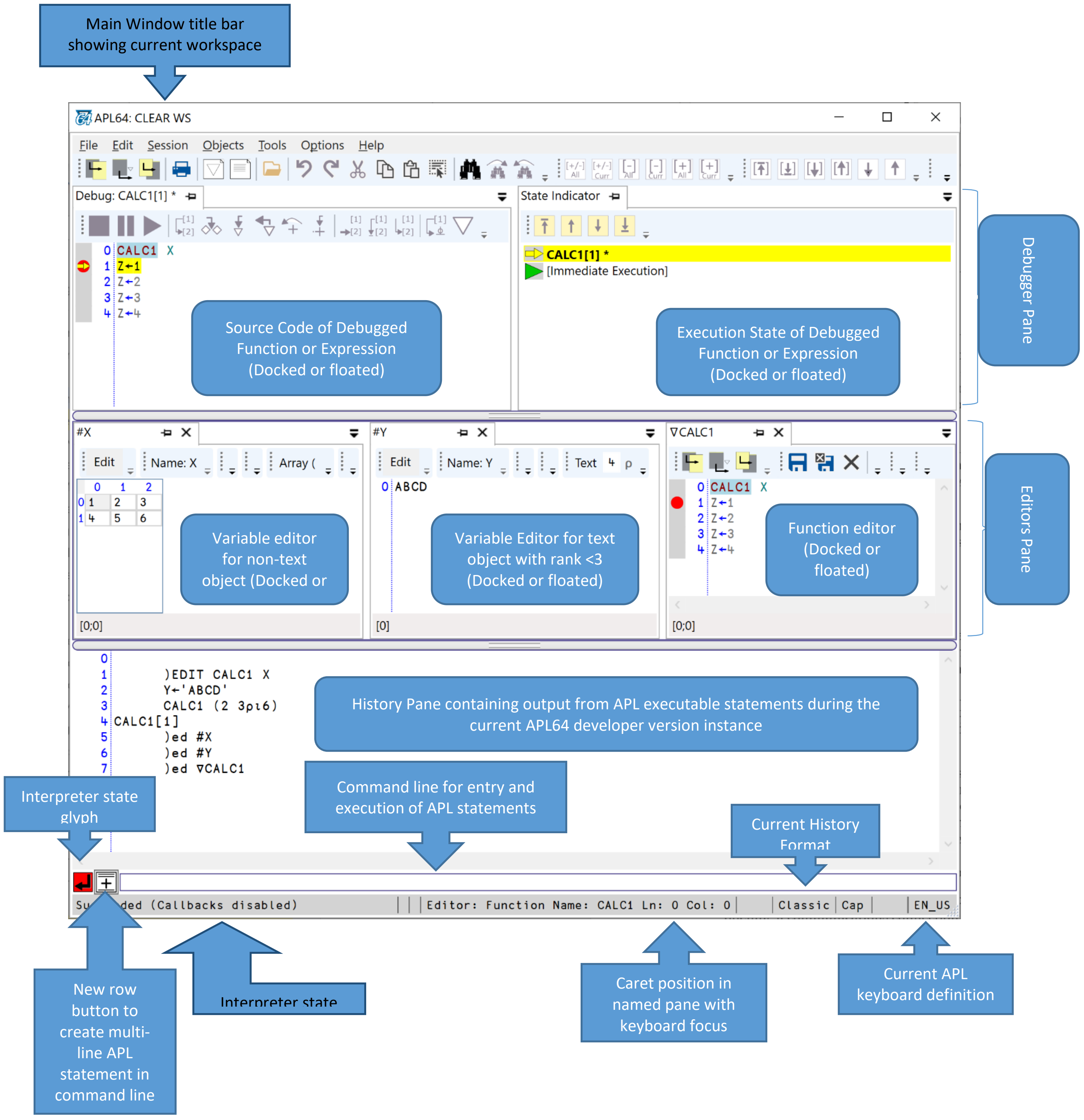
Some menu items also have shortcut keys, which are keystroke combinations you can press to select the item directly without pulling the menu down at all. Access keys and shortcut key combinations are identified on the menus. You can also select some menu items by pressing a button on the Toolbar.

When the system begins, you are in an APL64 session. You can type APL statements in the history pane or the APL command line, and the system executes them immediately. You can also open edit windows to edit or create APL functions and variables, and you can open the debugger to use the debugging facilities of APL64.

When the APL64 session begins, you are also typically in a workspace. A workspace is an APL construct; it is basically an arena in which you are operating. All the variables to which you assign values and all the functions you define exist in the workspace. You can save current work at any time by saving a workspace. You can perform new work in a clear workspace, or you can load a saved workspace into an empty workspace to resume the work you had saved. You can load an existing workspace to replace an active workspace, and you can copy an existing workspace or objects from an existing workspace into an active workspace. If you want to save programs or data independent of a workspace, you can use files.

In an APL64 session, you can manipulate some things that are outside the workspace. Writing to a file, opening a communications connection to a network, or creating a graphical user interface to Windows are all manipulations of external objects. You can write and save APL functions that create the external connections and re-create them by simply running the functions when you reopen the workspace.

APL64 Session Window Overview


Main Window title bar showing current workspace
APL64: CLEAR WS
File Edit Session Objects Tools Options Help
Debug: CALC1[1] *
State Indicator
Source Code of Debugged Function or Expression (Docked or floated)
Execution State of Debugged Function or Expression (Docked or floated)
Debugger Pane
#X
#Y
∇CALC1
Edit
Name: X
Array (
Name: Y
Text
Variable editor for non-text object (Docked or
Variable Editor for text object with rank <3 (Docked or floated)
Function editor (Docked or floated)
Editors Pane
History Pane containing output from APL executable statements during the current APL64 developer version instance
Interpreter state glyph
Command line for entry and execution of APL statements
Current History Format
New row button to create multi-line APL statement in command line
Interpreter state
Caret position in named pane with keyboard focus
Current APL keyboard definition

```
0 CALC1  X
1 Z←1
2 Z←2
3 Z←3
4 Z←4
```

```
CALC1[1] *
[Immediate Execution]
```

```
   0  1  2
0  1  2  3
1  4  5  6
[0;0]
```

```
0 ABCD
[0]
```

```
      )EDIT CALC1 X
      Y←'ABCD'
      CALC1 (2 3⍴⍳6)
CALC1[1]
      )ed #X
      )ed #Y
      )ed ∇CALC1
```

```
Suspended (Callbacks disabled) | | Editor: Function Name: CALC1 Ln: 0 Col: 0 | Classic | Cap | EN_US
```

The History Pane

The History pane is the location where you can type APL statements in immediate execution mode, and at the same time it is a record of what you have done in the session.

You start with the selection cursor at a six-space indent; if what you type results in output, the output displays starting with no indentation. (Some APL constructs, nested arrays for example, have initial spaces.) You can scroll through the History pane to make use of previous work.

If you place the selection cursor on an earlier line and press Enter, the system copies the line to the bottom of the session and executes it. The system displays any output following the copied line. You can also edit a line of the session. When you press Enter, the system copies the changed line to the session command line, executes it, and restores the earlier line to its original state. You can also select text to Copy and Paste to the session command line.

Many of the menu choices have equivalent commands you can type to perform the action. See the descriptions of commands in the System Commands chapter in this manual.

Timestamps that display in the APL64 session are now provided in a format consistent with your settings in the Time and Date tabs of the Regional Settings dialog in the Windows Control Panel. This means you will get either the default time format for your location in the world or the customized format that you have selected. For purposes of APL, you can choose between the short date format and the long date format with the **File | Emit Timestamps in Long Date Format** menu.

You can increase the size of the font in your APL64 session by moving to the right of the selection on the **Session | View Scale APL Text** menu or the Ctrl and mouse wheel forward**.** You can also reduce the size of the font by moving to the left of the selection on the **Session | View Scale APL Text** menu or the Ctrl and mouse wheel backward.

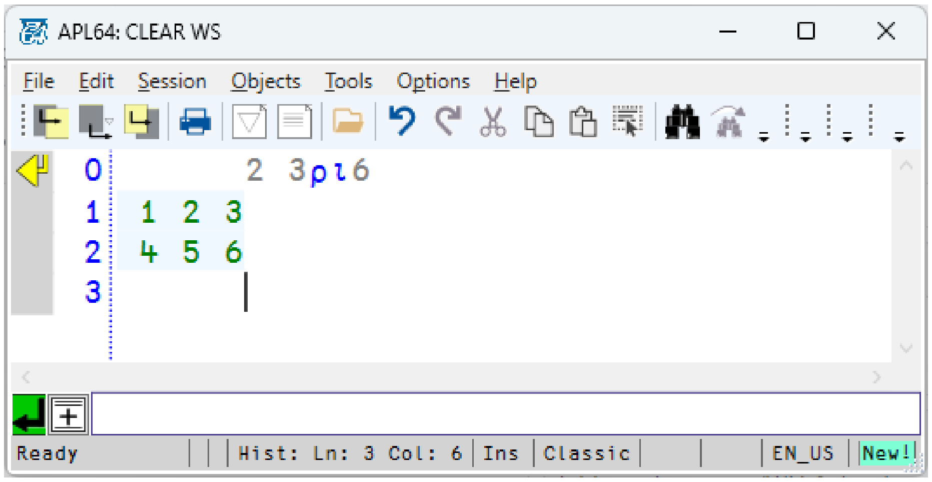
The APL64 session recognizes mouse-wheel events. This means you can go to the Control Panel/Settings/Mouse and: turn on the wheel; choose the direction; specify the number of lines per click; or scroll a screen at a time. You can also turn on the wheel button and get various behaviors that depend on your mouse driver, as you could previously. In addition, some wheel mouse drivers have a Universal Scrolling option that converts mouse events to keyboard scroll events. These may also work in the session, and they may have previously. APL64 does not support the panning facility, although you may get the effect of this working because of Universal Scrolling.

You can choose a caption for your APL64 session. The system function object has a caption property that you can set. The caption appears in the title bar to the left of the workspace path. This is particularly useful when you have multiple sessions running.

Syntax coloring and left margin glyphs are optionally available in the classic history pane. The new left margin delta glyph (Δ) indicates if a line in the classic history has been user-edited but not executed. The enhancement is configured with the menu **Session | Classic History Format | Show Syntax Coloring When Editable**.
Note:

- The delta glyph will be displayed in the editable classic history for a blank line inserted with the Ctrl+Enter shortcut.
- The delta glyph will not be displayed in the editable classic history on the last line of the history if that line contains none or only space characters.
- The delta glyph will not be displayed in the editable classic history if a history line is user-edited and immediately executed and the line was not previously a delta line.
- Example: User editing of the output of an APL executable expression

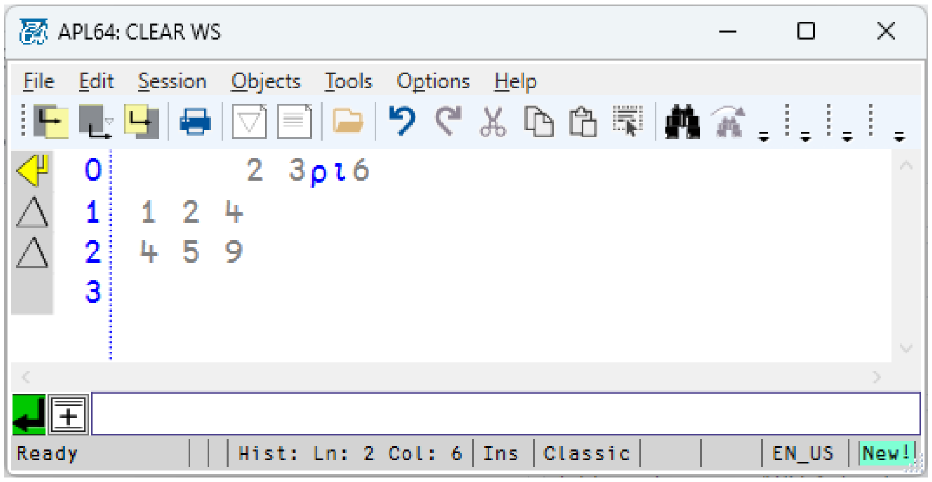
User executes: 2 3⍴⍳6 [Enter] in the classic history pane:



```
0        2 3⍴⍳6
1   1 2 3
2   4 5 6
3
```

```
Ready | | Hist: Ln: 3 Col: 6 | Ins | Classic | | EN_US | New!
```

User modifies rows 1 (replace 3 with 4) and 2 (replace 6 with 9). The left margin delta glyph marks rows 1 and 2 as user-edited rows:

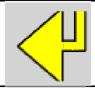
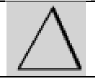
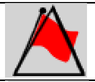
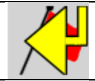
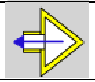


```
0        2 3⍴⍳6
1   1 2 4
2   4 5 9
3
```

```
Ready | | Hist: Ln: 2 Col: 6 | Ins | Classic | | EN_US | New!
```

History Pane Glyphs in APL64

In the APL64 history pane the glyphs on the left of each line of text indicate the executed status of that line:

Glyph	Description
	Statement executed by the interpreter
	Programmer-entered text which was not executed
None	Output which is the result of interpreter execution
	Not executed, with syntax error
	Executed, with syntax error
	Interpreter call-back statement which was executed

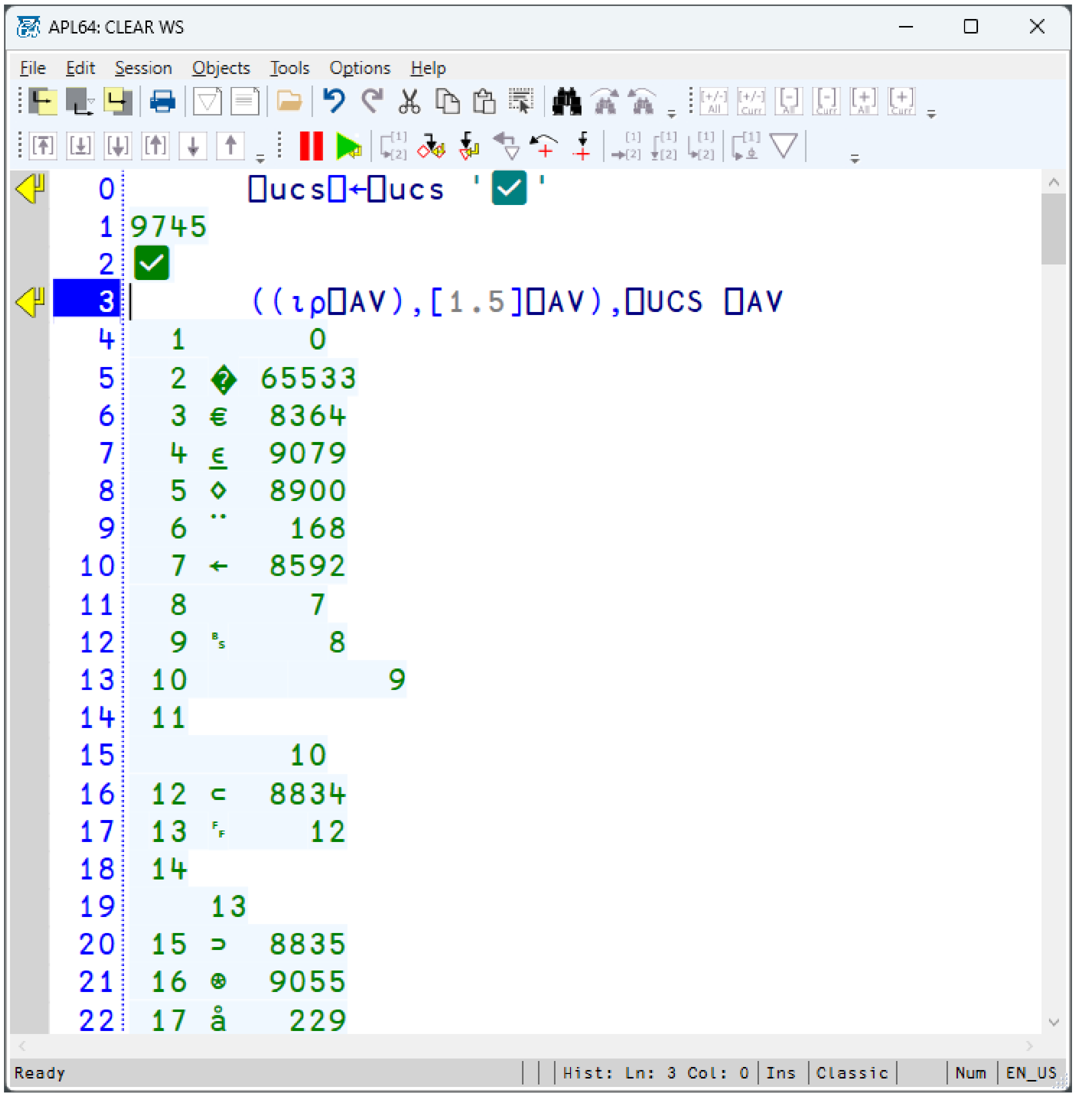
│	Indicates line included in a multi-line APL64 executable statement
↵	Indicates last line in a multi-line APL64 executable statement

Unicode Aware

In contrast to APL+Win, any Unicode code point may be displayed in APL64. This Unicode 'aware' enhancement in APL64 refers to the ability to render Unicode codepoints to the display, printer, etc., by including such Unicode code points in displayable APL64 variables. In APL64 the rendering of Unicode code points relies on the APL385 Unicode monospaced font to properly render Unicode code points.

The Unicode 'aware' enhancement in APL64 does not imply that ⎕AV contains all Unicode code points. ⎕AV is a well-defined list of Unicode code points which does not contain all Unicode code points.

For Unicode code points that are in ⎕AV, they may be used by indexing into ⎕AV. This is a 'convenience' property of ⎕AV. For those Unicode code points which are not in ⎕AV, the code point must be provided by the APL programmer in the form of an APL variable which can then be rendered by APL64. For example:



```
      ⎕ucs⎕←⎕ucs '☑'
9745
☑
      ((ιρ⎕AV),[1.5]⎕AV),⎕UCS ⎕AV
  1       0
  2 �  65533
  3 €   8364
  4 ⍷   9079
  5 ⋄   8900
  6 ¨    168
  7 ←   8592
  8        7
  9 ␈       8
 10            9
 11
          10
 12 ⊂   8834
 13 ␌     12
 14
      13
 15 ⊃   8835
 16 ⍟   9055
 17 å    229
```

```
Ready | | Hist: Ln: 3 Col: 0 | Ins | Classic | | Num | EN_US
```

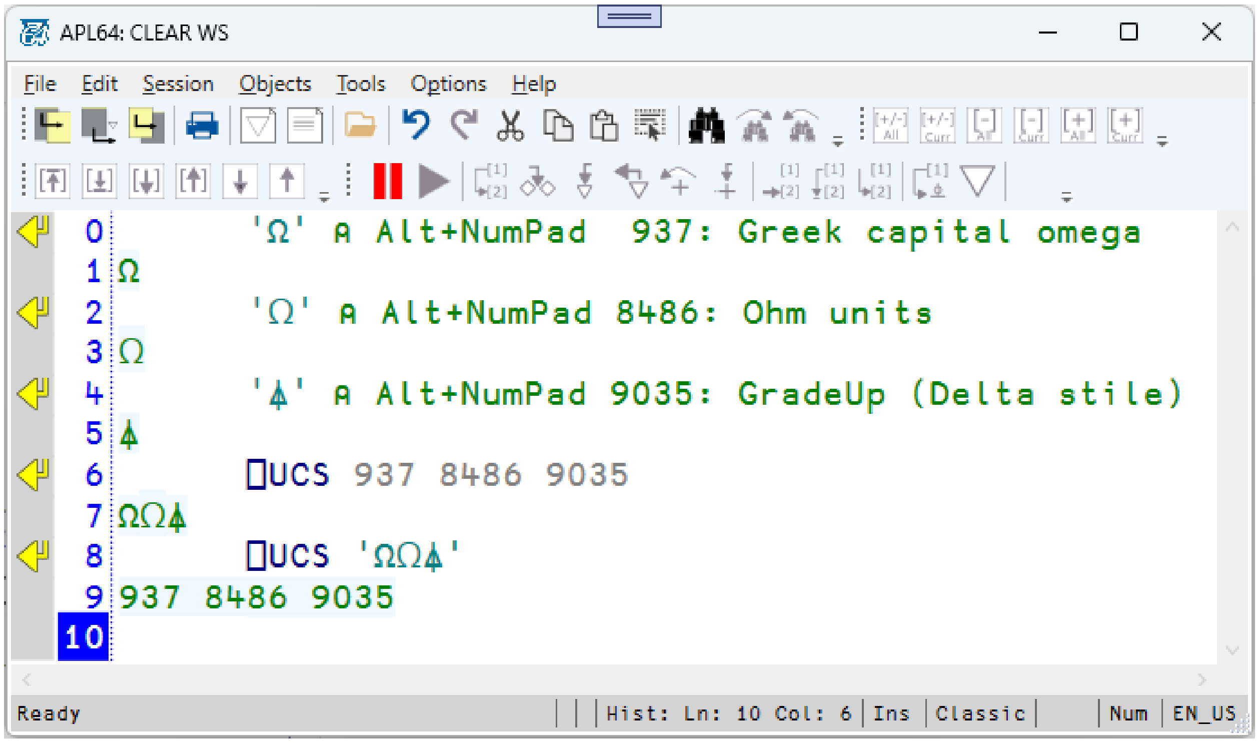
Alt+NumCodes Support in APL64

The APL64 history formats, function editor and variable editor support the entry of ALT+#### codes to render Unicode characters which are not directly accessible by the keyboard. **Note**: The virtual On-Screen Keyboard is not supported.

To use this feature:
(a) Place the keyboard focus on the desired APL64 pane
(b) Depress and hold the Alt keyboard key
(c) Use the numeric keypad to enter digits
(d) Release the Alt key

The programmer-entered digits are interpreted by APL64 as the decimal Unicode code point for the desired glyph.

For example:



```
      'Ω' ⍝ Alt+NumPad  937: Greek capital omega
Ω
      'Ω' ⍝ Alt+NumPad 8486: Ohm units
Ω
      '⍋' ⍝ Alt+NumPad 9035: GradeUp (Delta stile)
⍋
      ⎕UCS 937 8486 9035
ΩΩ⍋
      ⎕UCS 'ΩΩ⍋'
937 8486 9035
```

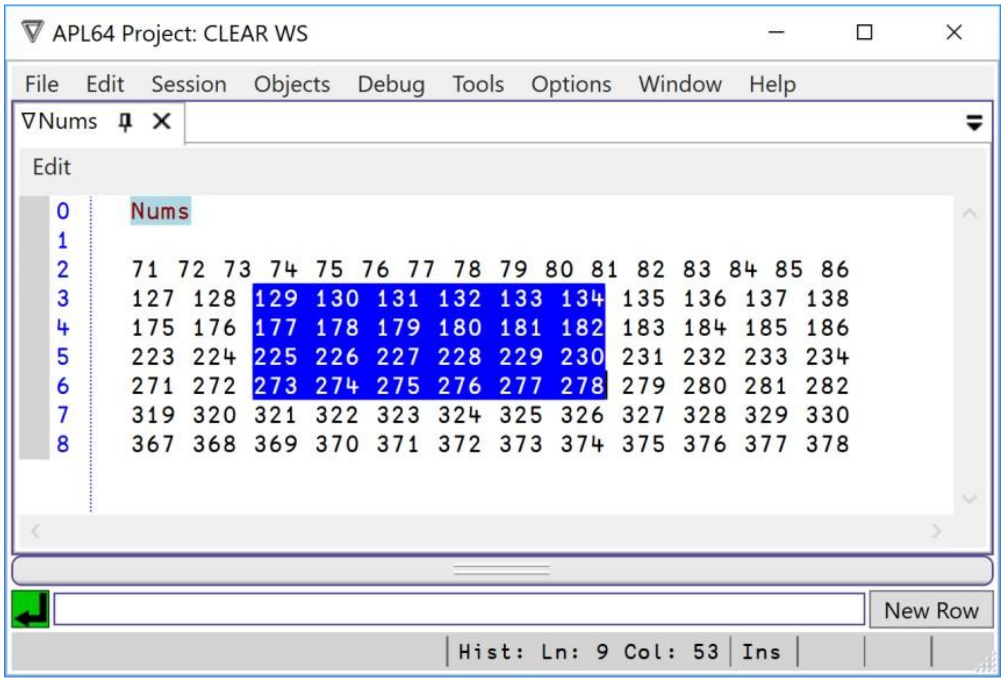
```
Ready | | Hist: Ln: 10 Col: 6 | Ins | Classic | | Num | EN_US
```

For more information about Unicode code points, see here: https://www.codetable.net/

Selection of Linear or Rectilinear Text Blocks in Function Source Code

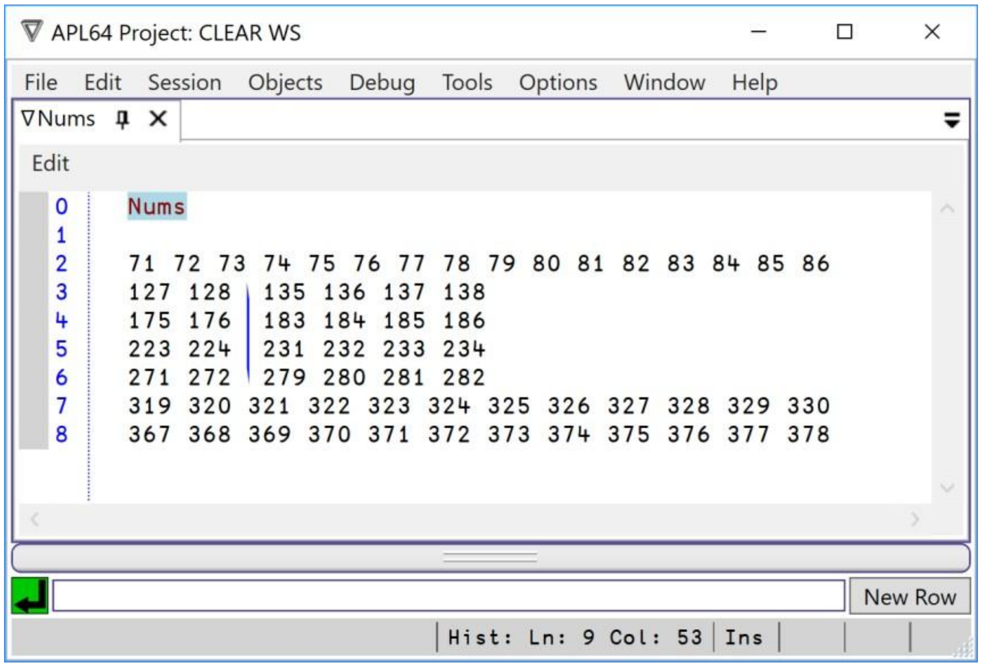
The APL64 session (editor and History) also supports linear or rectilinear text block selection. The Shift + Cursor keystrokes select a contiguous, linear block of text in the source code of a function being edited. The Alt + Shift + Cursor keystrokes select a contiguous, rectilinear block of text in the source code of a function. Selected text may be deleted, pasted-in or copied. Replacement by paste or drag & drop is supported within a rank 0, 1 or 2 text array.

Select a rectilinear block of function source code text:



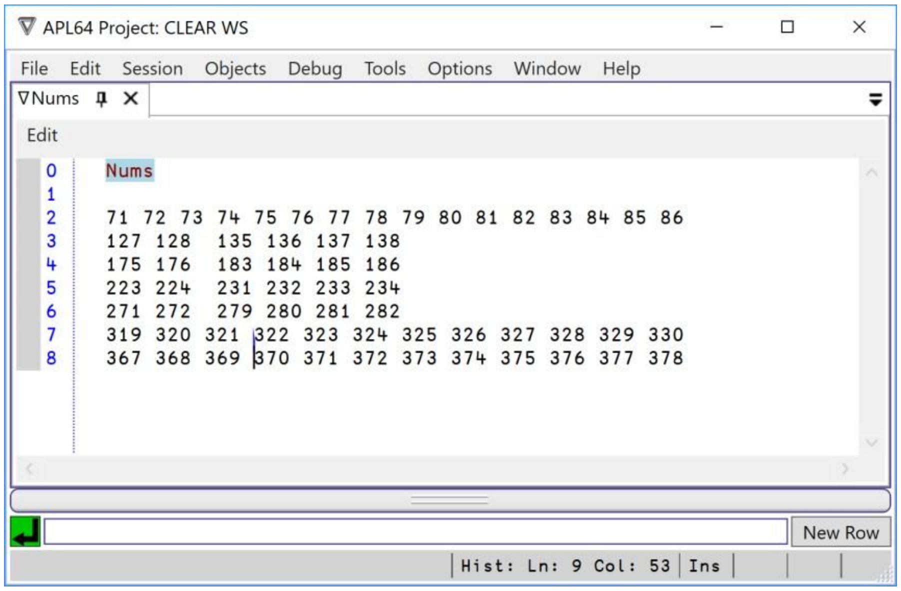
```
Nums

71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86
127 128 129 130 131 132 133 134 135 136 137 138
175 176 177 178 179 180 181 182 183 184 185 186
223 224 225 226 227 228 229 230 231 232 233 234
271 272 273 274 275 276 277 278 279 280 281 282
319 320 321 322 323 324 325 326 327 328 329 330
367 368 369 370 371 372 373 374 375 376 377 378
```

Delete the selected block of text from the function source code:



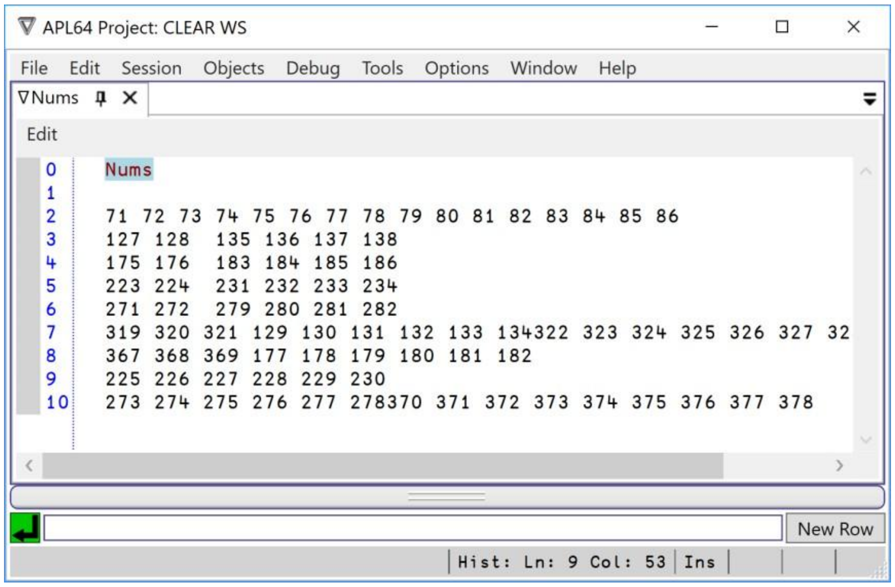
```
Nums

71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86
127 128 135 136 137 138
175 176 183 184 185 186
223 224 231 232 233 234
271 272 279 280 281 282
319 320 321 322 323 324 325 326 327 328 329 330
367 368 369 370 371 372 373 374 375 376 377 378
```

Select a new location in the function source for the text in the Windows Clipboard:



```
Nums

71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86
127 128  135 136 137 138
175 176  183 184 185 186
223 224  231 232 233 234
271 272  279 280 281 282
319 320 321 322 323 324 325 326 327 328 329 330
367 368 369 370 371 372 373 374 375 376 377 378
```

Paste the block of text in the Windows Clipboard in the function source code at the selected location:

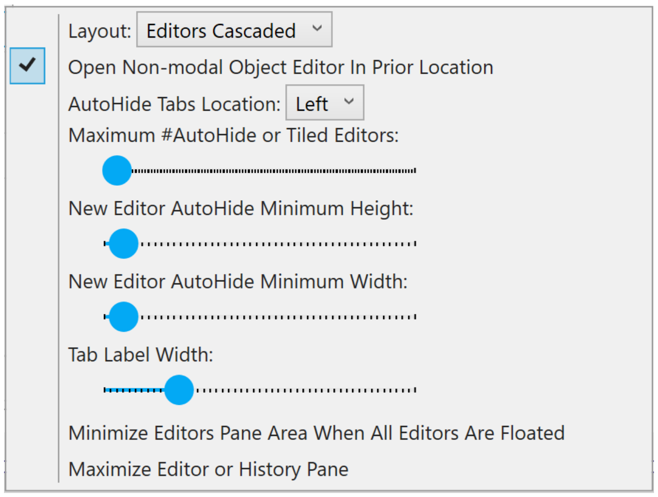


```
Nums

71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86
127 128  135 136 137 138
175 176  183 184 185 186
223 224  231 232 233 234
271 272  279 280 281 282
319 320 321 129 130 131 132 133 134322 323 324 325 326 327 32
367 368 369 177 178 179 180 181 182
225 226 227 228 229 230
273 274 275 276 277 278370 371 372 373 374 375 376 377 378
```

Editors Pane

The Editors pane is a session display facility that displays the edit windows for existing and new functions and variables. The Editors pane section of the main window and the History pane section of the main window are shared depending on the position of the GridSplitter Bar between these sections. Double-clicking the left mouse button on this GridSplitter Bar will alternately minimize/maximize the Editors pane section of the main Window. The user may manually drag this GridSplitter Bar to any intermediate position to share the space proportionally between the Editors pane and History pane sections of the main window.

The layout of the Editors pane is controlled by the **Objects | Editors Pane Format** menu:



Layout: Editors Cascaded

[✓] Open Non-modal Object Editor In Prior Location

AutoHide Tabs Location: Left

Maximum #AutoHide or Tiled Editors:

New Editor AutoHide Minimum Height:

New Editor AutoHide Minimum Width:

Tab Label Width:

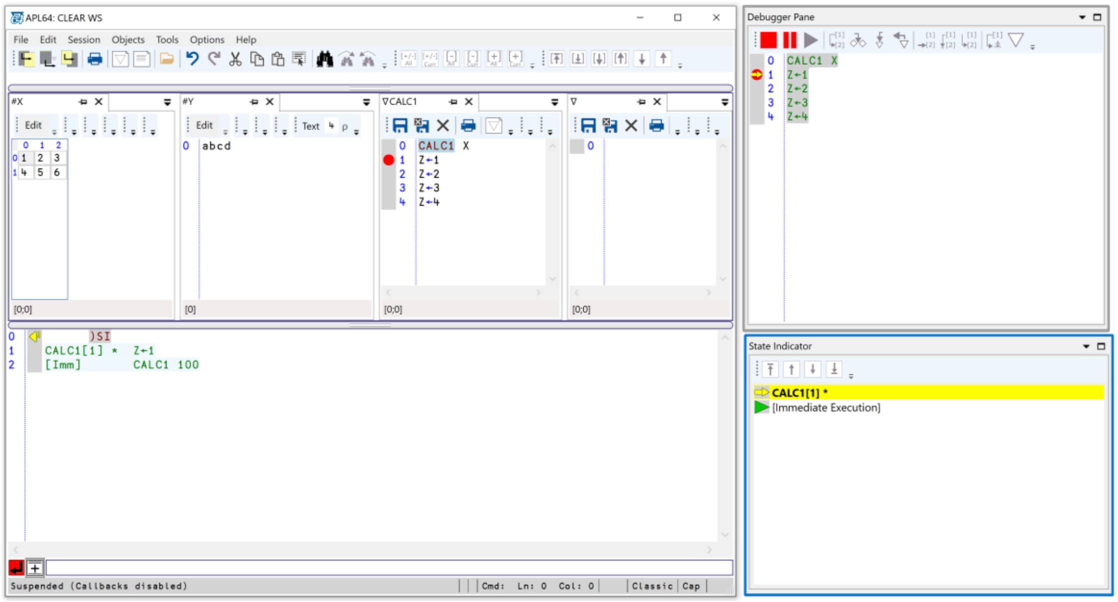
Minimize Editors Pane Area When All Editors Are Floated

Maximize Editor or History Pane

The **Layout** provides the following options: Auto Hidden, Cascaded (default), Tiled Horizontally, Tiled Vertically and Floated.

Floating a Pane

Any pane can be floated by dragging the pane's tab to the desired location: e.g., the Debug and State Indicator panes



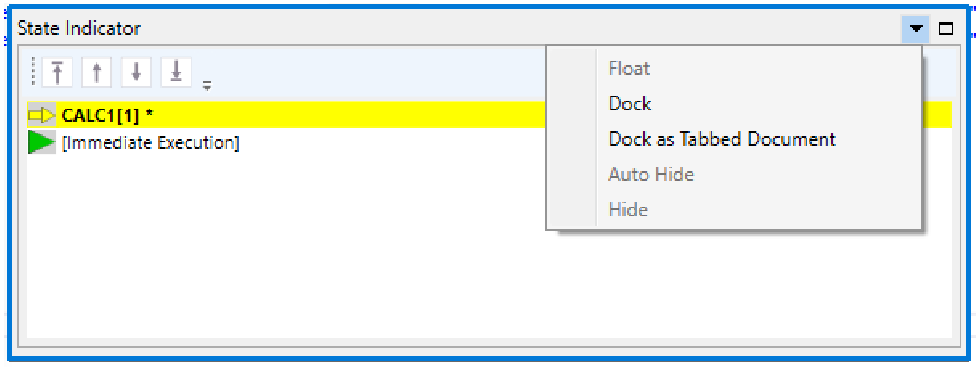
```
      )SI
CALC1[1] *   Z←1
[Imm]        CALC1 100
```

```
0  CALC1  X
1  Z←1
2  Z←2
3  Z←3
4  Z←4
```

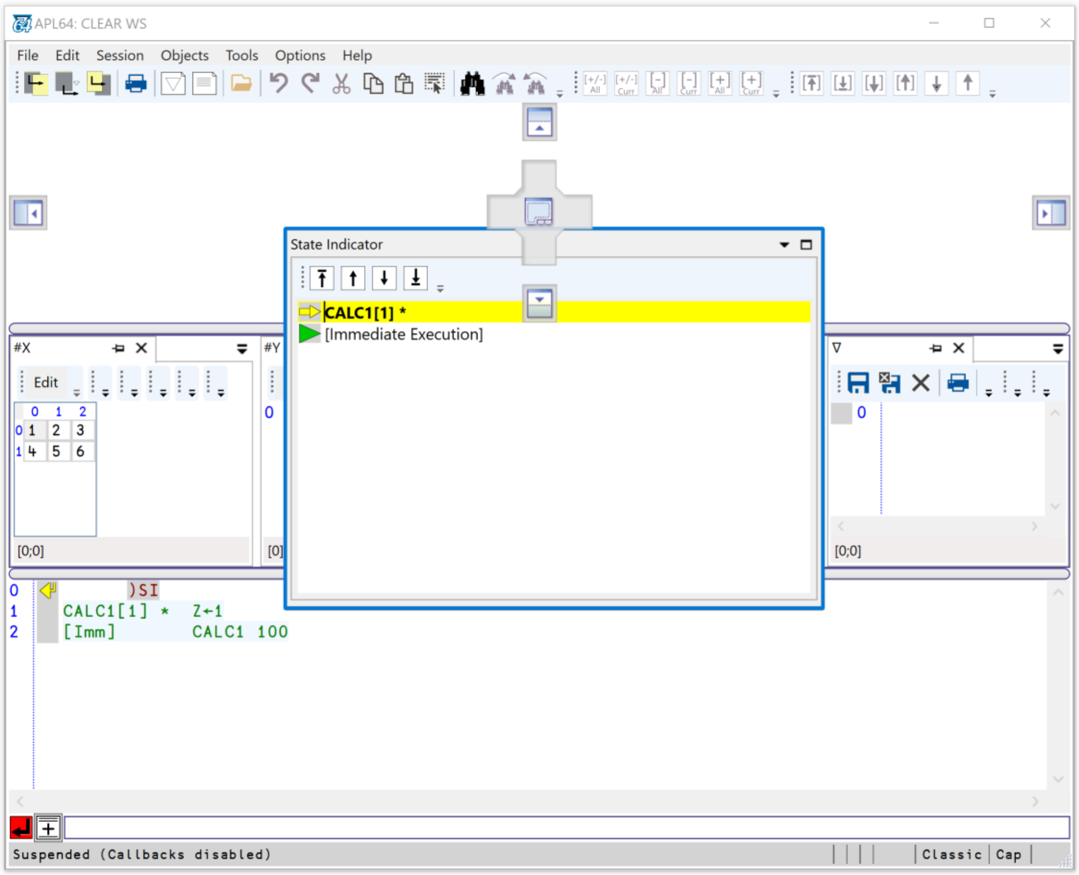
CALC1[1] *
[Immediate Execution]

Suspended (Callbacks disabled)   Cmd: Ln: 0 Col: 0   Classic Cap

A floating pane may be docked by using the pane's title bar options list to select a docking option:



A floating pane may also be docked by dragging the pane's tab back to the editors pane section of the main window and dropping the pane onto a selected docking location:



```
0      )SI
1   CALC1[1] *    Z←1
2   [Imm]         CALC1 100
```

Editing a Function

The system identifies a function edit window with a leading del (∇) in the edit window's tab. If there is text on the header line, the system derives the function's name from the header and the name also appears in the edit window's tab. The system places you on line [0], the header line, for a new function.

You can type any available character in a function window. If you press Enter when the cursor is at the end of a line, the system places you on a new line and displays the line number at the left of the window. If you press Enter in the middle of an existing line, the system begins a new line with the text at the insertion point and renumbers every line below the spot where you pressed Enter.

You can use the text editing functions on the Edit menu to help you write more efficiently. See the description of the Edit menu later in this chapter.

Editing a Variable

The system identifies a variable edit window with a leading pound sign (#) in the edit window's tab. If the variable already has a name, the name also appears in the edit window's tab.

You can use the variable edit window to modify existing text strings. For example, if you want to add a new code in the middle of a string of two-letter state codes, you can open the variable in the vector Edit window, type the new characters in the string, and save the variable.

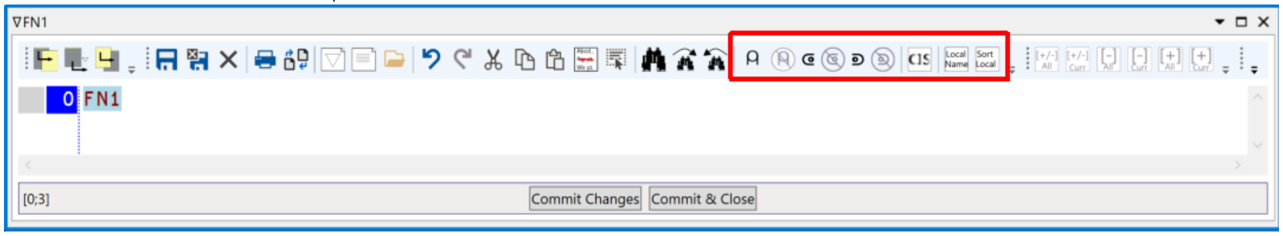
You can create and save a new text string as a variable. For example, you could type the text for a footnote, save it as a variable, and append it to the end of a report.

If you press Enter, the system places you on a new line, but the variable is still a one-dimensional vector. The newline character (⎕tcnl) is one element of the vector.

Whatever you type in the variable edit window is character data. If you type numerals, each digit is an individual character.

You can use the APL64 system function ⎕fi to convert a character representation of a numeric vector, with spaces between the numbers, to a numeric vector. You can use the APL primitive function Format (⍕) to convert a numeric vector to a character vector with spaces.

The Comment and Localize Options in Toolbar



Comment (**Ctrl+/**) and Uncomment (**Ctrl+Shift+/**) actions are available in the toolbar when you are in a function Edit window. The Comment toolbar button (⍝) adds a comment to the beginning of the current line or selected lines. The Uncomment toolbar button (⍝) removes the comment from the beginning of the current line or selected lines.

Insert a starting inline comment (**Alt+Shift+Z**) and Delete a starting inline comment (**Ctrl+Alt+Shift+Z**) actions are available in the toolbar when you are in a function Edit window. The Insert a starting inline comment toolbar button ( ) adds an inline comment to the beginning of the current line or selected lines. The Delete a starting inline comment toolbar button ( ) removes the starting inline comment from the beginning of the current line or selected lines.

Insert an ending inline comment (**Alt+Shift+X**) and Delete an ending inline comment (**Ctrl+Alt+Shift+X**) actions are also available in the toolbar when you are in a function Edit window. The Insert an ending inline comment toolbar button ( ) adds an ending inline comment to the end of the current line or selected lines. The Delete an ending inline comment toolbar button ( ) removes the ending inline comment from the end of the current line or selected lines.

The Insert CPC IntelliSense Scaffold (**Ctrl+Alt+I**) action is available in the toolbar when you are editing a CPC public function in a function Edit window. The CIS (CPC IntelliSense Scaffold) toolbar button (CIS) inserts the IntelliSense scaffold into the first three lines of the function Edit window.

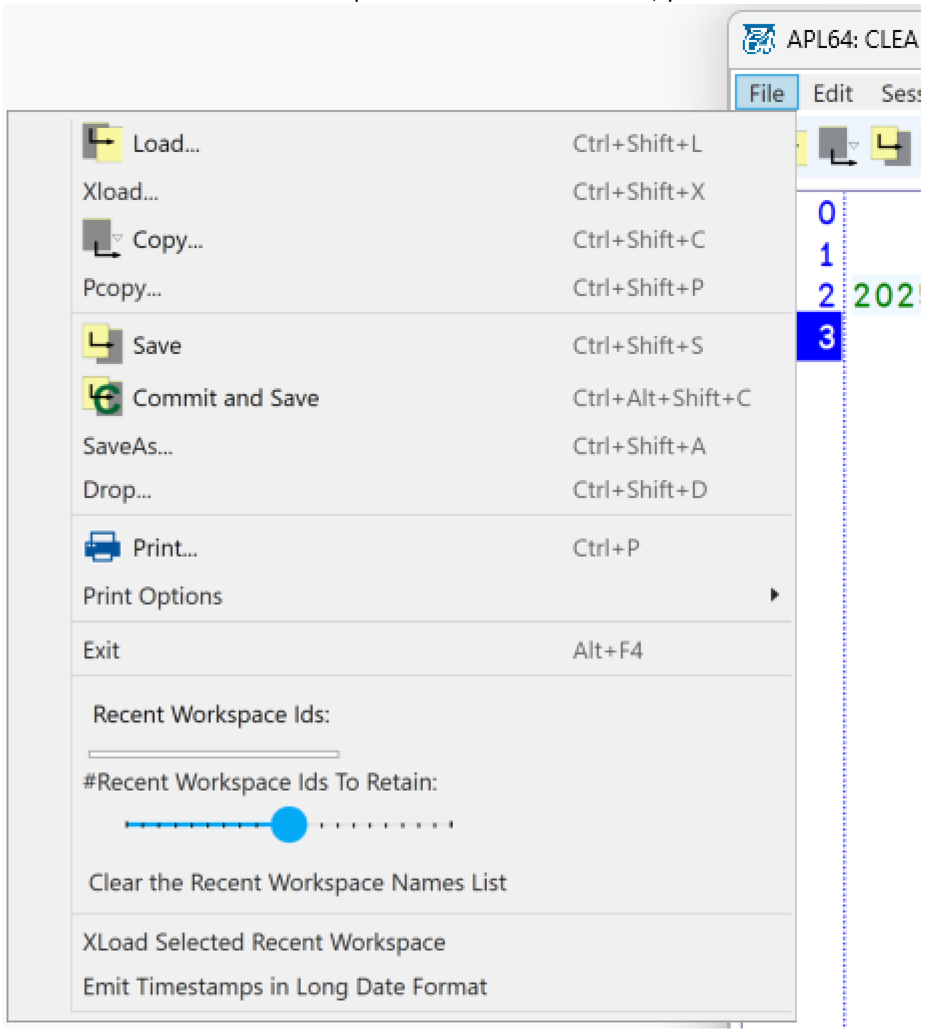
Localize (**Ctrl+L**) and Sort Locals (**Ctrl+Alt+L**) actions are also available in the toolbar when in a function Edit window. The Localize toolbar button (Local Name) allows you to toggle between making the name at the selection cursor local or global. This feature applies to system variables, names quoted in strings, and names in comments. If you attempt to localize names that you should not, the system generates an error message; this applies to system functions, read-only system variables, undefined quad-names, labels, and strings that do not count as names such as numbers or symbols.

- To localize a function or variable, place the selection cursor on the name and select Localize from the Options menu. The system adds the function or variable name to the end of the function header.
- To make a function or variable global, place the selection cursor on the name and select Localize from the Options menu. The system removes the function or variable name from the function header.
- Two styles of localization lists are supported. You can separate each name by a semicolon as in APL+Win or you can use just one semicolon at start of list and remaining local names can be separated by whitespace. The latter option is enabled by the system menu option **Objects | Function Editor | White Space Local Variables**.

To sort the localized names, click the Sort Locals toolbar button (Sort Local) or the menu option **Objects | Function Editor | Sort Local Variables**.

The File Menu

The File Menu is the tool for controlling major steps in your APL64 session. You can perform the following functions, some of which have options. From the File Menu, you can:



Load a Workspace

When you select Load from the File menu or press **Ctrl+Shift+L**, you load an existing workspace. The system opens a window where you can type or select the name of the workspace you want to load. You can use this window to find any workspace that is accessible on your disk or network.

The system opens the workspace as it was last saved in a previous session. This includes all the functions and variables defined in that session including any functions that were interrupted during execution. The title bar of the History pane displays the name of the workspace.

Xload a Workspace

By default, the system displays the name of the workspace, the library designation, if any, and the time it was saved. If a latent expression (`⎕lx`) is defined, the system executes that expression. If you want to load a workspace without running the latent expression, select Xload (**Ctrl+Shift+X**).

Copy a Workspace

When you select Copy from the File menu or press **Ctrl+Shift+C**, you copy the functions and variables of an existing workspace into your active workspace. The system opens a window where you can type or select the name of the workspace you want to copy. The system copies the workspace you select into the current APL64 session. In doing so, the system replaces any functions or variables that have the same name in both workspaces.

Pcopy a Workspace

If you want to copy a workspace without replacing any functions or variables that exist in the current workspace, select Pcopy (Protected Copy) or press **Ctrl+Shift+P**. The system copies the workspace you select into the current APL64 session except for any functions or variables that have the same name in both workspaces. The system lists any objects it did not copy.

If you want to copy selected objects from a workspace, use either of the system commands)copy or)pcopy with appropriate arguments.

If you change directories using the Directories: box on Load, Xload, Copy, or Pcopy dialog boxes, the system updates the current folder setting to reflect your selection. The next time you display the Load, Xload, Copy, or Pcopy dialog box, the Directories: box displays the folder specified by ⎕chdir.

Save a Workspace

When you select Save from the File menu or press **Ctrl+Shift+S**, you save the current workspace. If you already named the workspace or if you loaded a workspace from storage, the system saves the workspace under the same name.

If you did not name the active workspace, the system places you in the Save As window where you see the names of existing workspaces and can type the name under which you want to save and store the current workspace. If you select a name that is already a workspace name, the system prompts you to confirm that you want to replace the existing file.

If you want to save a named workspace under a different name, select Save As from the File menu or press **Ctrl+Shift+A**. You specify a new name for the current workspace. The workspace under the existing name remains as it was when you loaded it or when you last saved it during this session. The workspace with the new name you specify includes everything that exists in the workspace as of the time you select Save As.

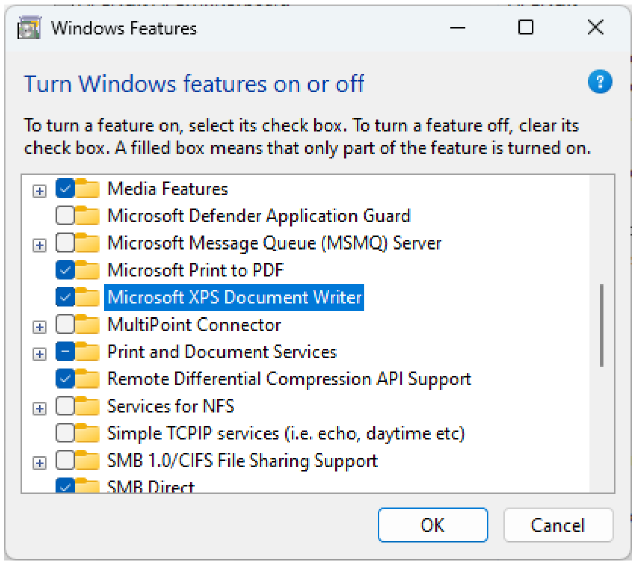
Drop (Delete) a Workspace

When you select Drop from the File menu or press **Ctrl+Shift+D**, you delete an existing workspace file that is stored on disk. The system places you in a window where you can type or select the name of the workspace you want to drop. You can use this window to find any workspace that is accessible on your disk or network. When you specify the workspace, the system prompts you to confirm that you want to delete the file; once dropped, a workspace cannot be restored. (It does not go to the Recycle bin.) If you specify the name of the workspace that is currently active in your APL64 session, the system deletes the stored workspace file from your disk, but it does not change the contents of the current workspace.

Print Contents of a Session

When you select Print from the File menu or press **Ctrl+P**, you can print the entire contents or a highlighted portion of the contents of the APL64 session or the current editing window.

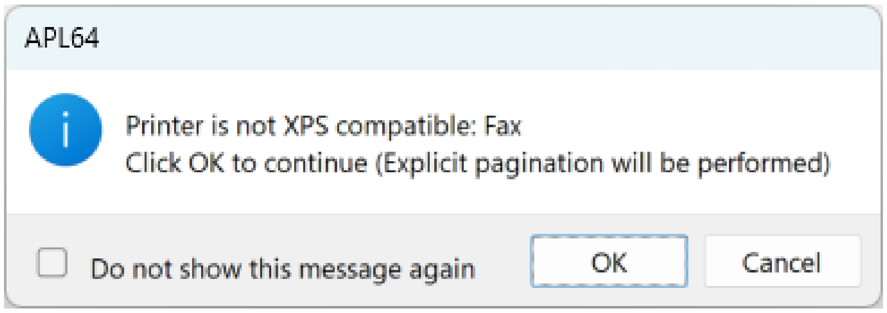
Note: Some printers may require adding the Microsoft XPS Document Writer printer in Windows if it isn't already present. This step can be performed in the Windows features setting as demonstrated below:



To turn a feature on, select its check box. To turn a feature off, clear its check box. A filled box means that only part of the feature is turned on.

- [✓] Media Features
- [ ] Microsoft Defender Application Guard
- [ ] Microsoft Message Queue (MSMQ) Server
- [✓] Microsoft Print to PDF
- [✓] Microsoft XPS Document Writer
- [ ] MultiPoint Connector
- [■] Print and Document Services
- [✓] Remote Differential Compression API Support
- [ ] Services for NFS
- [ ] Simple TCPIP services (i.e. echo, daytime etc)
- [ ] SMB 1.0/CIFS File Sharing Support
- [✓] SMB Direct

When printing, APL64 will detect whether the user-selected printer is not XPS compatible and provide the user the option to continue or cancel printing:



Printer is not XPS compatible: Fax
Click OK to continue (Explicit pagination will be performed)

If the user clicks OK to continue printing, be aware that APL64 will explicitly paginate the document to be printed and transmit it to the user selected printer with one or more of the following conditions:
(a) printing performance for large documents may be reduced
(b) glyphs which are not supported by the printer-embedded fonts may not be rendered correctly
(c) the temporary file containing the paginated document cannot be automatically deleted by the system

Print Options

Printing is supported with or without syntax colors from the menu **File | Print Options**. You can also specify the margins to be used on the printed page from the menu **File | Print Options | Margins**. You can also specify whether to display the message when the printer is not XPS compatible from the menu **File | Print Options | Show Message If Printer is Not XPS Compatible**.

Exit

When you select Exit from the File menu or press **Alt+F4**, you close the active APL64 session and return to another application or to Windows. If you have unsaved changes in an Edit window, the system prompts you to confirm the exit. Optionally, you can also have the system prompt you on every exit with a message that this will end your APL64 session. See the **Session | Confirm Exit** menu.

Recent Workspace Ids

Displays a list of the most recently used workspaces. The system remembers this list between APL64 sessions. To load one of these workspaces, select the workspace name from the bottom of the menu. The system loads the workspace and executes ϒlx.

Clear the Recent Workspace Names List

When clicked, the workspace names in Recent Workspace Ids List will be cleared.

XLoad Selected Recent Workspace

When checked, workspaces selected in the Recent Workspace Ids list will be XLoad.

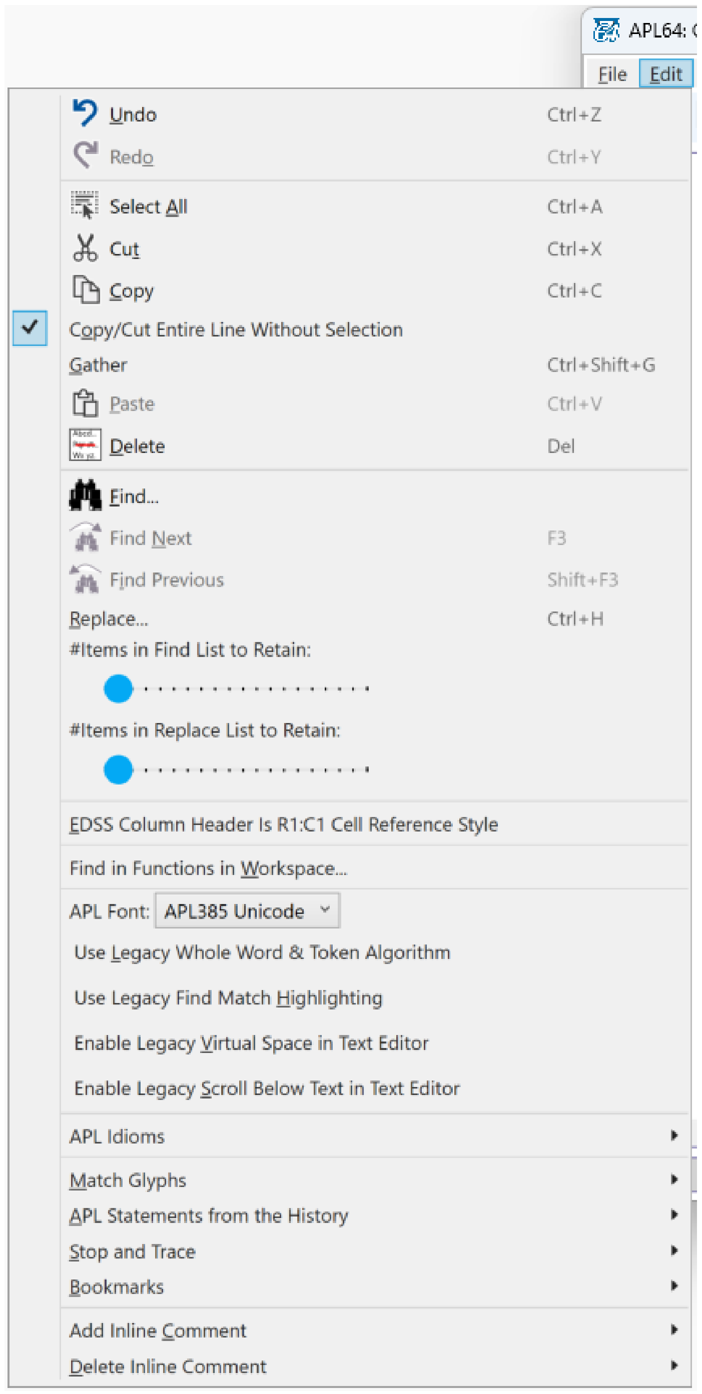
Emit Timestamps in Long Date Format

When checked, the timestamps associated with loading, saving, copying, dropping workspaces and session log display timestamps in the long date format.

The Edit Menu

The Edit Menu is the tool for making keyboard entry more efficient. From the Edit menu you can:

- Undo an action
- Redo an action
- Perform text processing actions
- Locate and replace existing material
- Specify the APL font for editing and printing
- Insert and manage APL idioms
- Match left and right parentheses in expressions
- Recall lines you recently executed
- Insert or delete stops and traces
- Insert, delete or navigate to bookmarks
- Add or delete inline comments


APL64:
File
Edit
Undo Ctrl+Z
Redo Ctrl+Y
Select All Ctrl+A
Cut Ctrl+X
Copy Ctrl+C
Copy/Cut Entire Line Without Selection
Gather Ctrl+Shift+G
Paste Ctrl+V
Delete Del
Find...
Find Next F3
Find Previous Shift+F3
Replace... Ctrl+H
#Items in Find List to Retain:
#Items in Replace List to Retain:
EDSS Column Header Is R1:C1 Cell Reference Style
Find in Functions in Workspace...
APL Font: APL385 Unicode
Use Legacy Whole Word & Token Algorithm
Use Legacy Find Match Highlighting
Enable Legacy Virtual Space in Text Editor
Enable Legacy Scroll Below Text in Text Editor
APL Idioms
Match Glyphs
APL Statements from the History
Stop and Trace
Bookmarks
Add Inline Comment
Delete Inline Comment

Undo an Action

When you select Undo from the Edit menu or press **Ctrl+Z**, the system restores the object on which you are working to the state that preceded the last action. When you are editing a character object, if you last performed a Cut, you can select Undo to replace the material you highlighted and removed. If you last performed a Paste, the system removes the inserted material. If you last typed one or more characters, the system removes the characters you typed.

If you are working in the numeric Edit window, Undo restores the last number you changed by typing, but it does not remove numbers you inserted by using Paste. It does replace the last row or column you deleted.

You can select Undo again to restore the last action.

Redo an Action

When you select Redo from the Edit menu or press **Ctrl+Y**, the system reverses the undo to restore the object on which you are working to the state that preceded the last undo action.

Select All

When you click on Select All from the Edit menu or press **Ctrl+A**, the system highlights the entire contents of your session or Edit window. You can then perform another action on that block, for example, delete it.

Cut

When you highlight a block of text and select Cut from the Edit menu or press **Ctrl+X**, the system removes the text. The system saves the text to the Windows clipboard so you can Paste it in another location. You cannot use Cut in the numeric Edit window.

Copy

When you highlight a block of text and select Copy from the Edit menu or press **Ctrl+C**, the system saves the text to the Windows clipboard. You can Paste it in another location. You can Copy a block of numbers in the numeric Edit window.

Copy/Cut Entire Line Without Selection

This option allows you to choose the behavior for Copy (**Ctrl+C**) and Cut (**Ctrl+X**) when no text is selected on a line. When checked, if you choose Copy or Cut on a line, the system retains the behavior that whatever is on the current line is copied or cut to the clipboard when you press **Ctrl+C** or **Ctrl+X**, respectively.

Gather

When you highlight a block of text and select Gather from the Edit menu or press **Ctrl+Shift+G**, the system copies to the Windows clipboard all the lines within that highlighted block that were executed in the session. You can then Paste these lines in another location. For examples, you could paste them at the bottom of the session to facilitate examination or re-execution; to a function edit window; or to another application for documentation.

Paste

When you select Paste from the Edit menu or press **Ctrl+V**, the system inserts the last block of text you saved to the clipboard by a Cut, Gather, or Copy action. You must move the selection cursor to the location where you want to insert text before you select Paste. You can Paste a block of numbers in the numeric Edit window that you saved to the clipboard with Copy.

Delete

When you highlight a block of text and select Delete from the Edit menu, the system removes the text without saving it to the clipboard.

Find

When you select Find from the Edit menu or press **Ctrl+F**, the system opens the Find dialog that allows you to identify the text you want to locate. The Find dialog uses an Edit box to define the search target text. The number of entries retained are controlled from the menu **Edit | #Items in Find List to Retain**.

You can optionally make the search case sensitive. The Match Case (match case or case-insensitive) selection is independent of the search type, except that when you do a token search, strings such as system functions, system variables, and control structure keywords are always case-insensitive.

You also have the option to specify APL Tokens. This option is like the Complete Word option in Find actions in other applications. If the system locates the text you specify as a unit, it stops. If the text exists as part of a longer string, the system does not stop. **Note**: See the menu option **Use Legacy Whole Word & Token Algorithm** for additional information.

There are two non-token search types: whole word and match substrings. One important difference between a token and a whole word is that a leading colon is part of the text for a token, but it is a separate entity when matching a whole word. Thus, searching for the token "end" will find "end" but not ":end"; searching for the whole word "end" will find it in both places. **Note**: See the menu option **Use Legacy Whole Word & Token Algorithm** for additional information.

If you perform a Find action on the previous paragraph for the letters "stop" with the Tokens option not selected, the system locates two instances: one as part of the word "stops" and the other at the end of the paragraph. If you perform the same search with the Token option selected, the system locates only the instance at the end of the paragraph.

A token is an APL syntactic element. It can be a primitive symbol, a system function or variable, a control keyword (such as `:if`), a user-specified name, or a character string. You can also search for a string of tokens or for multiple tokens with embedded spaces. Tokens follow APL syntax rules for case-sensitivity. System functions are not case-sensitive, but user names are; therefore, `⎕WA` and `⎕wa` are the same token, but `WA` and `wa` are different tokens. You can search for the backslash character in the text lines below, for example. The system locates each slash in the workspace designations.

```
              )LOAD C:\APL\TEST
        "C:\APL\TEST.ws64" LAST SAVED 7/23/2022 14:12:48
```

You can also search for the token string `\APL`, and the system finds both instances. In this case, the text matches the token string because the leading backslash is itself a token and the backslash following delimits the string of letters ("`APL`"). If you search for the token `\AP`, the system provides a message that it did not find the text string.

Characters that can be part of an individual string include the uppercase and lowercase letters of the English alphabet, numerals, and the characters that can be part of the names of APL objects, which include the delta (∆), the underscore (_), and delta-underscore (⍙). All other printable characters, including accented letters, punctuation marks, arithmetic symbols, and symbols particular to APL, are each a separate token. Spaces are a special case.

Spaces delimit tokens, so you can find the word "`SAVED`" as a token in the example above. You can also have spaces within a token string. For instance, you can find the string "`SAVED 11`" in the text. The system ignores multiple consecutive spaces within a token string. If the system finds "`SAVED     11`" in the text, it treats it as a match with the token "`SAVED 11`".

If you are in the numeric Edit window, you can search for a number in the matrix. The options for case-sensitivity or tokens are not available.

Find Next (Search Down)

When you select Find Next from the Edit menu or press **F3**, the system repeats the Find action using the same search text and options you last specified, except that it assumes the search direction is down. This is the same action you perform when you push the Find Next button on the Find window, but you can perform this action without reopening the Find window.

You can also search for the token the cursor is on by pressing **Ctrl+F3**. There is no corresponding menu item. **Note**: The Find dialog must be opened then closed to cancel the find highlights. This is analogous to Visual Studio.

Find Previous (Search Up)

When you select Find Previous from the Edit menu or press **Shift+F3**, the system repeats the Find action as though you had selected Up for the search direction. If Up is selected, this choice still searches up. You can search up for the token the cursor is on by pressing **Ctrl+Shift+F3**.

Replace

When you select Replace from the Edit menu or press **Ctrl+H**, the system performs a combination Find and Replace action. The Replace dialog uses a drop-down Combo box to define the search target and replace text. The number of entries retained are controlled from the menu **Edit | #Items in Replace List to Retain**. The system opens a window where you identify the text you want to locate and the text you want to replace it with. You have the same case-sensitivity and Token options in this window that you do in the Find window. You can choose to Replace every instance of located text with one action or to replace the next instance of text that matches the target string.

EDSS Column Header Is R1:C1 Cell Reference Style

When you select EDSS Column Header Is R1:C1 Cell Reference Style, the column headers in an EDSS worksheet will display with numbers (1 …). When unchecked, the A1 cell reference style is applied to an EDSS worksheet and will display column headers with letters (A …), which is the default behavior.

Find In Functions in Workspace

When you select Find in Functions in Workspace, the Find Text in Workspace Function window appears. See Using Find in Functions in Workspace in **Help | Developer Version GUI | Using Find in Functions in Workspaces** menu for details on how you use this window.

APL Font

This option lists the current font for APL editing and printing.

Use Legacy Whole Word & Token Algorithm

When you select Use Legacy Whole Word & Token Algorithm, the APL+Win algorithm will be used in the Find dialog for the Whole Word and APL Token matching options and APL64 will behave like APL+Win. When unchecked, the APL64 algorithm will be used and APL64 may behave differently from APL+Win.

Use Legacy Find Match Highlighting

When you select Use Legacy Find Match Highlighting, only the current match will be highlighted like in APL+Win. This option provides fast performance when searching extremely large source documents. When unchecked, all matches found will be highlighted, which is the default behavior.

Enable Legacy Virtual Space in Text Editor

When you select Enable Legacy Virtual Space in Text Editor, the caret may be placed to the right of existing text in 'virtual space'. Entering text in virtual space will convert the virtual space on the associated line to existing text. This option only applies to the classic history pane, the command line, the function editor and pure text elements in the variable editor. The default is not checked.

Enable Legacy Scroll Below Text in Text Editor

When you select Enable Legacy Scroll Below Text in Text Editor, it is possible to vertically scroll the existing text in the pane to display empty, non-editable, white space below the existing text. If not checked, the visibility of the existing text will be maximized. This option applies to the classic history pane, function editor and pure text elements in the variable editor. The default is not checked.

APL Idioms

The Idioms Manager is a programmer productivity tool to enable the deployment of APL idioms into APL64-based application systems. An APL idiom is text which represents frequently used APL statements. The APL idiom may contain complete APL executable statements or may require additional APL programmer input to become APL executable statements. An option is provided to specify the placement of the caret when the idiom text is inserted into a function editor, variable editor, command line or editable classic history. See Using Find in Functions in Workspace in **Help | Developer Version GUI | Using the Idioms Manager** menu for details on how you use this window.

Match Glyphs

The Find Matching Glyphs and Match and Find and Tag Matching Glyphs options enable you to find a matching parenthesis, bracket, or quote in an APL expression. For example, you can use the Find Matching Glyphs option to ensure that you have a matching close parenthesis for every open parenthesis in an expression and vice versa. In addition to matching parentheses, brackets, or quotes, the Find and Tag Matching Glyphs option highlights the expression between the two delimiters.

Warning: You can find matching nested delimiters except when the outer delimiters are quote marks. Since any expression may belong within a quoted string, including those with unmatched parentheses, this facility does not work within quotes. You can find matching quotes nested inside parentheses.

To find a matching parenthesis, bracket**,** quote mark, or double quote mark:

1. Position the selection cursor immediately to the left of the delimiter you want to match. The selection cursor can be at either the opening or the closing glyph. For example, you can position the selection cursor to the left of either an open parenthesis or a close parenthesis.
2. Select the Find Matching Glyphs option from the Edit menu or press **Ctrl+M, Ctrl+J**. If you want the system to locate the matching glyph and highlight the text enclosed in those symbols, select the Find and Tag Matching Glyphs option instead or press **Ctrl+Shift+M, Ctrl+Shift+J**.

If the system finds a matching glyph, it moves the selection cursor to the left of that glyph. If the system cannot locate a matching glyph, it beeps and does not move the selection cursor.

Note: You can also locate unmatched parentheses, quote marks, and brackets using syntax coloring. For more information, see the description of the Colors option on the Options Menu.

APLS Statements from the History

There are several options on the Edit menu that allow you to recall one of the last `20` lines (and as many as `1,000)` you executed and display it in the History pane. When you recall a line, the system displays it on the line containing the selection cursor. The recalled line replaces any existing text on that line. When you find a line, the system moves the cursor to the location in the session where the line was executed.

- The Recall previous executed line option displays the lines you executed in reverse order. Each time you select this option or press **F9**, the system displays the line it executed just prior to the line being displayed. If you recall too many lines, you can use the Recall next option to move forward in the list.
- The Recall next executed line option displays the lines you executed in the order you executed them. Each time you select this option or press **Shift+F9**, the system displays the line it executed just after the line being displayed.
- The Execution History option (**Ctrl+F9**) displays the Executed Statements History window where you can select the line you want to recall. The Executed Statements History window uses toggle buttons to select the target and statement type. A toggle button has a current state which is user-selected from the available states by single clicking the toggle button multiple times until the button content displays the desired state. The target states that are available: Clipboard, History, Command Line, and UTF8 Text File. The statement types that are available: All Stmts (All Statements), Result Stmts (Result Statements), and Exec Stmts (Executed Statements). The 'Select Last Statement' checkbox has been implemented in the Executed Statements History dialog so that APL64 programmers have a choice of first or last statement in the list to be selected.
- The Find previous executed line option moves the cursor to the previously executed line so you can see it in context. Each time you select this option or press **Alt+F9**, the system places you on the line it executed just prior to the line where the cursor is located. This option is not limited to 20 lines. **Note**: Not supported when the Classic History format is editable.
- The Find next executed line option moves the cursor to the next executed line so you can see it in context. Each time you select this option or press **Alt+ Shift+F9**, the system places you on the line it executed just after the line where the cursor is located. **Note**: Not supported when the Classic History format is editable.

Stop and Trace

There are several options on the Edit menu that allow you to set stops and traces for debugging functions. Two of the options allow you to toggle one or more settings. If you select a contiguous group of lines, the action toggles the first selected line and matches all the others to it. The other options allow you to clear all the settings. Note that you can set an individual stop on a line simply by clicking in the margin.

- The Insert/Remove STOP on selected line(s) option (**Ctrl+period**) toggles a stop on the selected line or lines.
- The Remove STOP from all lines option (**Ctrl+Shift+period**) removes all stops from the function.
- The Insert/Remove TRACE on selected line(s) option (**Ctrl+comma**) toggles a trace setting on the selected line or lines.
- The Remove TRACE from all lines option (**Ctrl+Shift+comma**) removes all trace settings from the function.

Bookmarks

There are several options on the Edit menu that allow you to navigate through an edit window or the session using bookmarks. These settings tag a line with a temporary margin marker that allows you to return quickly to the line. Unlike stops and traces, which you can save with a function, bookmarks persist only while a window is open.

- The Insert/Remove bookmark on selected line(s) option (**Ctrl+F2**) toggles a bookmark on the selected line or lines.
- The Remove bookmark from all lines option (**Ctrl+Shift+F2**) removes all bookmarks from the window.
- The Find Next Bookmark option (**F2**) moves the cursor to the next line that is tagged with a bookmark and makes the line visible in the window. This is equivalent to searching down.
- The Find Previous Bookmark option (**Shift+F2**) moves the cursor to the previous line that is tagged with a bookmark and makes the line visible in the window. This is equivalent to searching up.

Add Inline Comment

These options allow you to add inline comment glyphs that would behave like to C, C++ or C# inline comment characters /* and */ to comment out multi-line blocks or insert annotations in the middle of executable code.

- The Start Inline Comment Glyph option (**Alt+Shift+Z**) displays an opening inline comment glyph (⟃) on the line.
- The End Inline Comment Glyph option (**Alt+Shift+X**) displays am ending inline comment glyph (⟄) on the line.

Inline comments begin and end with glyphs that look somewhat like sideways lamps: ⟃ ⟄

They referred to as opening and closing inline comments, respectively.

Unlike traditional comments, inline comments can optionally be terminated before the end of line using a closing comment “⟄” glyph.

If a single opening comment glyph is used without a closing comment glyph, it behaves like a traditional comment and runs to the end of line:

```
      10 20 ⟃ 30 40 50 60
10 20
```

But if you terminate the comment it stops before the end of line:

```
   10 20 ⟃ 30 40 ⟄ 50 60
10 20 50 60
```

If you use two or more opening inline comment glyphs, the comment block can span multiple lines (e.g., the end of a function).

Note: The number of contiguous opening inline comment glyphs determines the number of contiguous closing inline comment glyphs you need to end the comment block; when they don't match, the comment block runs to the end of the function.

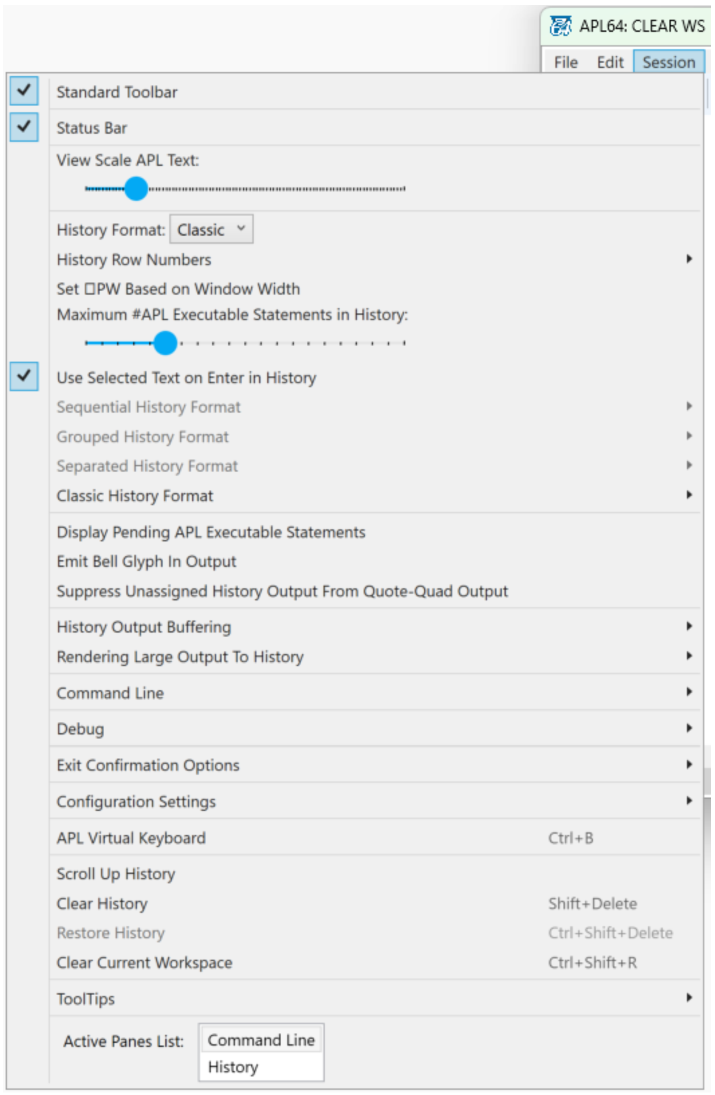
Delete Inline Comment

These options allow you to delete inline comment glyphs.

- The Start Inline Comment Glyph option (**Ctrl+Alt+Shift+Z**) removes an opening inline comment glyph (⟃) on the line.
- The End Inline Comment Glyph option (**Ctrl+Alt+Shift+X**) removes an ending inline comment glyph (⟄) on the line.

The Session Menu

The Session menu allows you to customize the APL64 session.


APL64: CLEAR WS
File
Edit
Session
✓ Standard Toolbar
✓ Status Bar
View Scale APL Text:
History Format: Classic
History Row Numbers
Set ⎕PW Based on Window Width
Maximum #APL Executable Statements in History:
✓ Use Selected Text on Enter in History
Sequential History Format
Grouped History Format
Separated History Format
Classic History Format
Display Pending APL Executable Statements
Emit Bell Glyph In Output
Suppress Unassigned History Output From Quote-Quad Output
History Output Buffering
Rendering Large Output To History
Command Line
Debug
Exit Confirmation Options
Configuration Settings
APL Virtual Keyboard Ctrl+B
Scroll Up History
Clear History Shift+Delete
Restore History Ctrl+Shift+Delete
Clear Current Workspace Ctrl+Shift+R
ToolTips
Active Panes List: Command Line
History

Standard Toolbar

Option to display the standard toolbar.

Status Bar

Option to display the status bar.

View Scale APL Text

Option to set the scaling factor which will apply to the function and variable editors, the APL command line, the history and the Debug and State Indicator panes.

History Format

Option to set the History format in the APL64 history. Classic is the default.

History Row Numbers

This has options for displaying and highlighting row numbers in the APL64 history.

Set ⎕PW Based on Window Width

Option to set ⎕PW based on the APL64 history width.

Maximum #APL Executable Statements in History

Option to set the maximum number of APL executable statements to retain in the APL64 history.

User Selected Text on Enter in History

Option to specify the result when Enter or Return is pressed in the history pane and there is a selection. When checked, the selected text will be used for the Enter/Return action. And when unchecked, the unselected text will be used for the Enter/Return action.

Sequential History Format

This has options for the Sequential History.

Grouped History Format

This has options for the Grouped History.

Separated History Format

This has options for the Separated History.

Classic History Format

This has options for the Classic History.

Display Pending APL Executable Statements

This option specifies when to display a pending APL executable statement in the pending statements list. Note: This option does not apply to the editable Classic history format.

Emit Bell Glyph in Output

This option displays a bell glyph in the History pane when ⎕tcbel is executed.

Suppress Unassigned History Output From Quote-Quad Output

This option specifies when to display the quote-quad output value when assigned to the history.

History Output Buffering

This has options for displaying interpreter output to the APL64 history pane.

Rendering Large Output To History

This has options for enabling and specifying the size of interpreter text output displayed in the APL64 history.

Command Line

This has options for the APL command line.

Debug

This has options for displaying debugger panes.

Exit Confirmation Options

This has options to confirm on exit and confirm on exit with the Exit MessageBox displayed in the default desktop. If the Confirm Exit choice has a check mark beside it, the system displays a warning message when you select Exit from the File menu or enter the)off system command. You must click OK to exit the system. You can turn this option on and off by clicking on the menu item. If you have active edit sessions open and you did not save your work, the system warns you whether this option is checked or not.

If the Confirm Exit MessageBox Location is Default Desktop is checked, the Exit APL64 MessageBox will be displayed on the main display of the workstation. But if not checked, the Exit MessageBox will be displayed on the main window of the current APL64 instance.

Configuration Settings

This has options for the APL64 configuration settings.

Save Settings on Exit

If the Save Settings on Exit choice on the Options menu has a check mark beside it, the system automatically accepts choices and settings you make with this menu as default values. The system carries over any changes to subsequent APL64 sessions. If you want to be able to change settings and options within a session and retain your previous defaults, be sure this menu choice is not checked. You can turn this option on and off by clicking on the menu item.

Save Settings Now

When you select Save Settings Now from the Options menu, the system saves all current APL64 session choices and settings as your defaults. The system saves the settings, but there is no visible indication that it has done so.

APL Virtual Keyboard

Option to display the APL virtual keyboard.

Scroll Up History

The result is analogous to the execution of the ⎕tcff system constant in the session in APL+Win. Also available in the context menu.

Clear History

This option will delete the APL executed and results statements in the History.

Restore History

This option will restore the History that was previously cleared with Clear History provided nothing new has been entered in the History. **Note**: The workspace and History log are not affected by this action.

Clear Current Workspace

This option will erase the variables and functions from the active workspace without saving them.

ToolTips

This has options for displaying Tooltips for most session menu and toolbar items.

Active Panes List

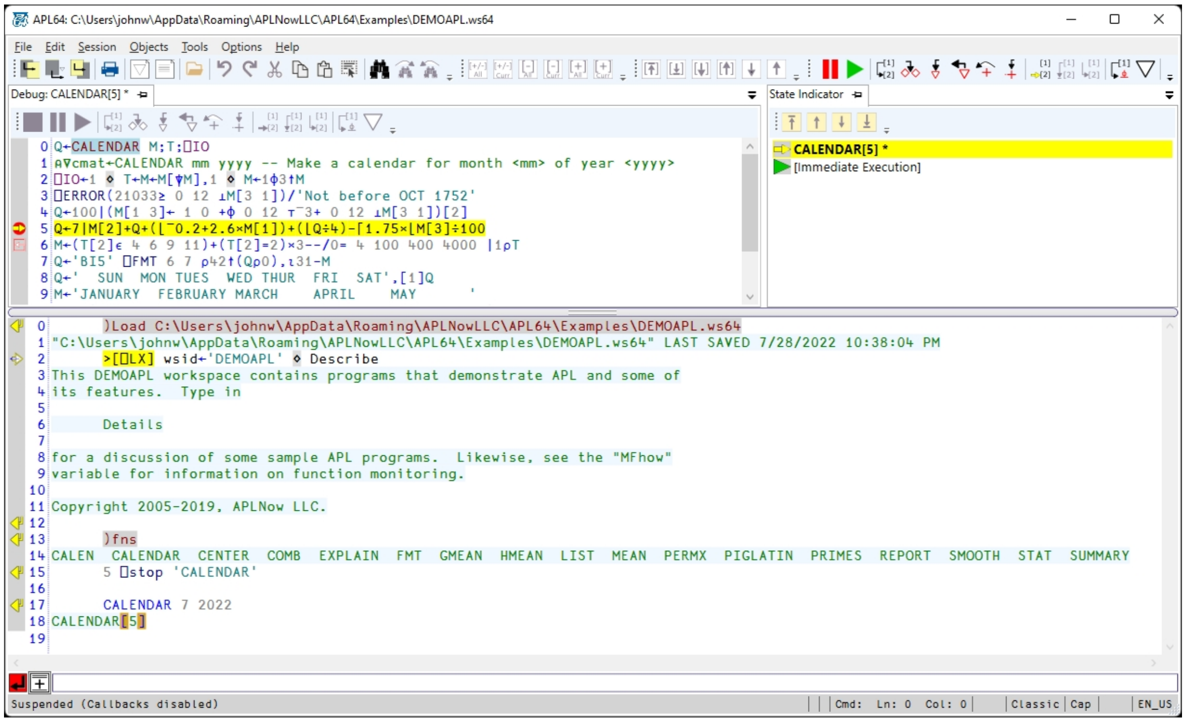
This option lists the current active panes for the session.

Working in the Debugger

Debugger is a session display facility that allows you to test, analyze, and debug functions. Debugger works with the traditional APL debugging aids invoked by the system functions `⎕stop` and `⎕trace`. Debugger makes flagging the lines of a function substantially easier and more responsive. It also provides a variety of options for restarting execution and moving through your code by various increments.

Debugger allows you to have three panes in view simultaneously: the History pane, the Debug pane that shows a function, and the State Indicator pane. The Debugger/State Indicator (SI) pane section of the main window and the remaining area of the main window are shared depending on the position of the GridSplitter Bar between these sections. Double-clicking the left mouse button on this GridSplitter Bar will alternately minimize/maximize the Debugger/SI pane section of the main Window. The user may manually drag this GridSplitter Bar to any intermediate position to share the space proportionally between the Debugger/SI panes and remaining area of the main window. The displayed function can be the one that is currently suspended or a pendent function, that is, one that called another function and is waiting for it to complete. While a function is suspended, you can move among the three panes in immediate execution mode to evaluate and explore values and expressions.

The figure below shows the History pane with both debugger panes in their default positions.


APL64: C:\Users\johnw\AppData\Roaming\APLNowLLC\APL64\Examples\DEMOAPL.ws64
File Edit Session Objects Tools Options Help
Debug: CALENDAR[5] *
State Indicator
CALENDAR[5] *
[Immediate Execution]

```
0 Q←CALENDAR M;T;⎕IO
1 ⍝∇cmat←CALENDAR mm yyyy -- Make a calendar for month <mm> of year <yyyy>
2 ⎕IO←1 ⋄ T←M←M[⍒M],1 ⋄ M←1⌽3↑M
3 ⎕ERROR(21033≥ 0 12 ⊥M[3 1])/'Not before OCT 1752'
4 Q←100|(M[1 3]← 1 0 +⌽ 0 12 ⊤¯3+ 0 12 ⊥M[3 1])[2]
5 Q←7|M[2]+Q+(⌊¯0.2+2.6×M[1])+(⌊Q÷4)-⌈1.75×⌊M[3]÷100
6 M←(T[2]∊ 4 6 9 11)+(T[2]=2)×3--/0= 4 100 400 4000 |1⍴T
7 Q←'BI5' ⎕FMT 6 7 ⍴42↑(Q⍴0),⍳31-M
8 Q←'  SUN  MON TUES  WED THUR  FRI  SAT',[1]Q
9 M←'JANUARY  FEBRUARY MARCH    APRIL    MAY       '
```

```
      )Load C:\Users\johnw\AppData\Roaming\APLNowLLC\APL64\Examples\DEMOAPL.ws64
"C:\Users\johnw\AppData\Roaming\APLNowLLC\APL64\Examples\DEMOAPL.ws64" LAST SAVED 7/28/2022 10:38:04 PM
      >[⎕LX] wsid←'DEMOAPL' ⋄ Describe
This DEMOAPL workspace contains programs that demonstrate APL and some of
its features.  Type in

      Details

for a discussion of some sample APL programs.  Likewise, see the "MFhow"
variable for information on function monitoring.

Copyright 2005-2019, APLNow LLC.

      )fns
CALEN  CALENDAR  CENTER  COMB  EXPLAIN  FMT  GMEAN  HMEAN  LIST  MEAN  PERMX  PIGLATIN  PRIMES  REPORT  SMOOTH  STAT  SUMMARY
      5 ⎕stop 'CALENDAR'

      CALENDAR 7 2022
CALENDAR[5]

```

```
Suspended (Callbacks disabled)          Cmd:  Ln: 0  Col: 0     Classic Cap    EN_US
```

In the figure above, in the History pane (bottom pane), the left pointing return arrows (which are yellow) show lines of code that were executed. The round-cornered square box (light blue) is a bookmark; it allows you to return to that line easily from anywhere in the History pane. You can set bookmarks in any of the three windows.

In the top right pane in the State Indicator pane, the right pointing arrow (yellow) shows that the system is suspended on line 5 of the CALENDAR function. The right pointing acute triangles (green) indicate other functions on the stack (also Immediate Execution).

In the Debug pane, the top left pane, lines 5 and 6 show a Stop setting (red octagon) and a Trace setting (red outline) respectively. If you move the cursor to a different line in the SI window, the function displayed in the code window would change accordingly. You can scroll either window without changing the display in the others.

You do not see any content in the Debugger windows until a function suspends. This can occur in several ways:

- If your program gives an error when it runs, you can simply run the program. When it halts, Debugger displays the faulty function in the code pane.
- Assign one or more stop points somewhere in your program and then run it. When the program reaches a line you have marked, the suspended function is displayed.
- Type a statement that runs your program, then select Step INTO Function twice instead of pressing Enter. Your program halts on line [1] and the function listing appears in the code pane.

In general, there are three ways to perform most of the actions in Debugger; you can use the menu, the toolbar, or shortcut keystrokes. Thus, you can invoke the "Step INTO Function" action by selecting it from the Walk Menu, pressing the command button that shows a thin blue arrow pointing into the del symbol, or by pressing **F11**.

Arrows in the Debugger Panes

Various arrows are displayed in the left margins of the code pane and the state indicator pane:

➪ A yellow right-facing arrow indicates the line you are suspended on.

➪ A blue left arrow is superimposed when you have partially executed a line. This happens when you've executed one or more diamond-delimited segments on a line, or when you exit from a subroutine and stop in the calling routine, at which point you have only partially executed the calling statement. When you are in this condition, only the part of the statement that remains to be executed is highlighted in yellow.

A yellow left-facing arrow is displayed on line [0] as you are exiting a function. Note that at this point, you can examine the result of the function by referencing the result variable as declared on the header line.

A green triangle indicates the current line in one of the pendent functions leading up to the suspended function. In the State Indicator pane, these indicate each pendent function and immediate execution at the bottom of the stack. This last item in the state indicator pane represents the line you typed in immediate execution mode to start the program running. This line does not appear in the output from APL's)si command. It is provided in Debugger to allow you to step through the statements in the line or subroutines called in the line. It appears in the code pane when you type a statement in the session and press `F11`, or when you move to the bottom of the State Indicator pane.

Using the Debug Pane

Although the Debug pane looks like an edit session, any printing keystroke you type while in the code pane results in the keystroke and the focus being transferred to the History pane or APL Command Line. If you want to edit the function, you must open an edit session (see below). You can, however, set stops, traces, and bookmarks in any function displayed in the code pane. These take effect immediately, regardless of whether the function is pendent or suspended. Stops and traces are saved; bookmarks are ephemeral. These markers are displayed in the left margin of the code pane.

- You can toggle the stop setting on any line by clicking in the margin or by pressing **Ctrl+.** (period) for the line the cursor is on or for all lines that are selected. When you have multiple lines selected, the system reads the current setting for the first one, toggles it and matches all the other lines. Typing **Ctrl+Shift+.** (period) clears all stops in the function that is displayed in the window.
- You can toggle the trace setting on any line by pressing **Ctrl+,** (comma) for the line the cursor is on or for all lines that are selected. When you have multiple lines selected, the system reads the current setting for the first one, toggles it and matches all the other lines. Typing **Ctrl+Shift+,** (comma) clears all traces in the function that is displayed in the window.
- You can toggle the bookmark setting on any line by pressing **Ctrl+F2** for the line the cursor is on or for all lines that are selected. When you have multiple lines selected, the system reads the current setting for the first one, toggles it and matches all the other lines. Typing **Ctrl+Shift+F2** clears all bookmarks in the function that is displayed in the window (or in your session, if you type it there).

Moving through the State Indicator Pane

As mentioned above, the function that is displayed in the code pane is the one that corresponds to the line that is highlighted in the state indicator pane. You can see where you are in a pendent function without changing the state of the suspended function by moving the highlight in the State Indicator pane. You can click on a line in the state indicator window, or you can move through the stack by using the Ctrl key in conjunction with the mathematical operation keys on the numeric keypad (plus, minus, multiply [asterisk] and divide [slash]). When you are at the top of the stack, as you are when you first suspend, you can move down one level by pressing **Ctrl+(Numeric+)** or to the bottom level of the stack with **Ctrl+(Numeric/)**. After you have moved down the stack, you can move up one level by pressing **Ctrl+(Numeric-)** or to the top level of the stack with **Ctrl+(Numeric*)**.

Using Immediate Execution in the History Pane

While your function is suspended in Debugger, you can work in immediate execution mode in the History pane; you can execute virtually anything that does not require Windows callbacks that you could if there were no suspension. You can even invoke the function again; it will appear in the State Indicator stack a second time. More appropriately, you can check the values of variables, perform test calculations, or take other actions that will help you understand, fix, or improve your program.

While you are in the session you can use traditional APL debugging techniques, such as executing)sinl, which shows what is localized at each level on the stack, or ⎕error ' ', which removes the top function from the stack. You can resume execution by branching to a specific line or by executing →⎕lc; however, Debugger offers numerous options for continuing execution with more control.

Debugging with Watchpoints

A watchpoint is a variable in a function that triggers a suspension when its value changes, or when its value could change, and when a condition you specify is satisfied. Watchpoints cause suspension in the debugger, if you have turned it on, or in the session.

You specify watchpoints as a paired entity consisting of the name of an APL variable and an expression. Whenever an APL primitive is executed that might cause a change in value to the named variable, the system evaluates the expression. If the result of the evaluation is a non-zero singleton, execution suspends in a phantom function named <WATCHPOINT>. You can resume execution by typing a branch statement in immediate execution mode, or by using the debug options.

This facility allows you to interrupt your program when a given condition occurs for a specific variable, which differs from the breakpoint facility that allows stopping on a specific line. For example, your watchpoint specification might look like:

```
            1 2 ρ 'total' 'total > limit'
```

Whenever your program reaches a statement that assigns a value to the variable total, and that value is greater than the value of the variable threshold, the function suspends with a message in the session.

```
   foo
WATCHPOINT TRIGGERED
total > threshold
^
```

You can examine the state of your application in immediate execution mode or in debug mode. The stack might look like:

```
   )si
⎕WATCHPOINT[1]
foo[5]
```

and you could resume with either of the following two branch statements:

```
→ ⎕lc+1
→ 0
```

Note that you ran the function foo and the system created the function ⎕watchpoint with the watched variable names(s) prefixed the "Var:" or "Vars:". You must branch out of the system-created function or past the line that caused the suspension to continue your function.

The system variable ⎕watchpoints

Watchpoints are maintained in a system variable ⎕watchpoints or abbreviated ⎕wp. Note that the system variable is plural, while the function the system creates is singular. You can use the system variable in a statement in a function or in the session as you can any system variable. If you attempt to run the ⎕watchpoint function, the system signals a syntax error.

The system variable ⎕watchpoints is an N by 2 nested array. Each element of the array must be a character vector. Elements in the first column contain variable names. Elements in the second column contain corresponding APL expressions to be evaluated. Unlike APL+Win, multiple APL expressions can be separated by the diamond statement separator. You can invoke an input form by clicking on the Edit Watchpoints item on the **Session | Debug** menu, or pressing **Ctrl+W**, or you can set ⎕watchpoints directly. Here is an example:

```
    ⎕watchpoints ← 3 2 ρ 'sum' 'sum>max' (,'Z') 'alpha=0' 'oz' 'goo'
    ⎕watchpoints
 sum sum>max
 Z   alpha=0
 oz  goo
```

Note that the expression does not have to include the watchpoint variable. In the second case, the function suspends whenever the variable Z is assigned a value and the variable alpha equals zero. Also note that you must make single-character variable names into vectors; the system variable does not accept scalars. A watchpoint expression triggers a suspension if the result is a singleton with a non-zero value or if there is no result.

Warning: The ⎕watchpoint function is not benign. If your watchpoint expression is the name of a function, such as in the third line of the example above, and that function returns a zero, there still may be effects. Every time the function that you originally invoked assigns a value to the variable oz, the function goo runs. Even though your original function does not suspend, whatever actions are in the function goo have occurred. You could use this feature constructively in debugging, for example, to record the value of oz each time it is updated. You could also use this feature dangerously.

Watchpoint Triggers

The following expressions trigger execution of a watchpoint expression for the variable a:

- Simple assignment: a←3
- Strand assignment: (a b c)←1 2 3
- Indexed assignment: a[2]←3
- Selective Specification: (1⌷a)←5
- Control structure specification: :for a :in ι10 (each time through the loop)

Assignment to a system variable triggers execution of a watchpoint expression for that variable:

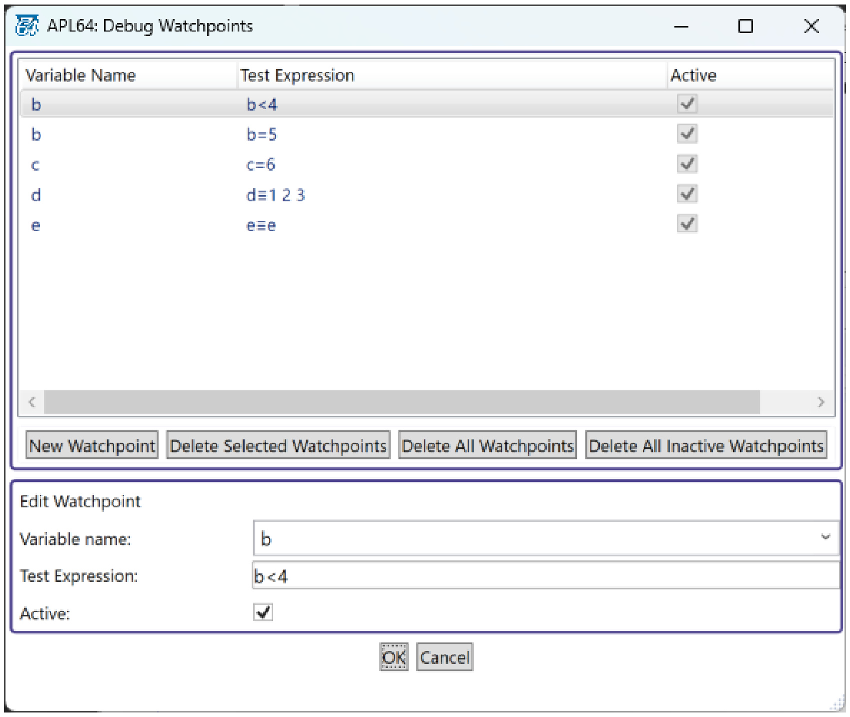
- Quad-Names: ⎕io←0

Watchpoints in the Debug Pane

To illustrate a watchpoint suspension, consider the following function and watchpoint specification:

```
   ∇ decrement a
[1]    :while a>0
[2]       (b c d) ← a
[3]       a←a-1
[4]    :endwhile
[5]    'done'
   ∇

    ⎕watchpoints
 b b<4
 b b=5
 c c=6
 d d≡1 2 3
 e e≡e
```

If the Watchpoint dialog is visible, the information displayed look approximately like this:



Variable Name	Test Expression	Active
b	b<4	✓
b	b=5	✓
c	c=6	✓
d	d≡1 2 3	✓
e	e≡e	✓

Edit Watchpoint

Variable name: b

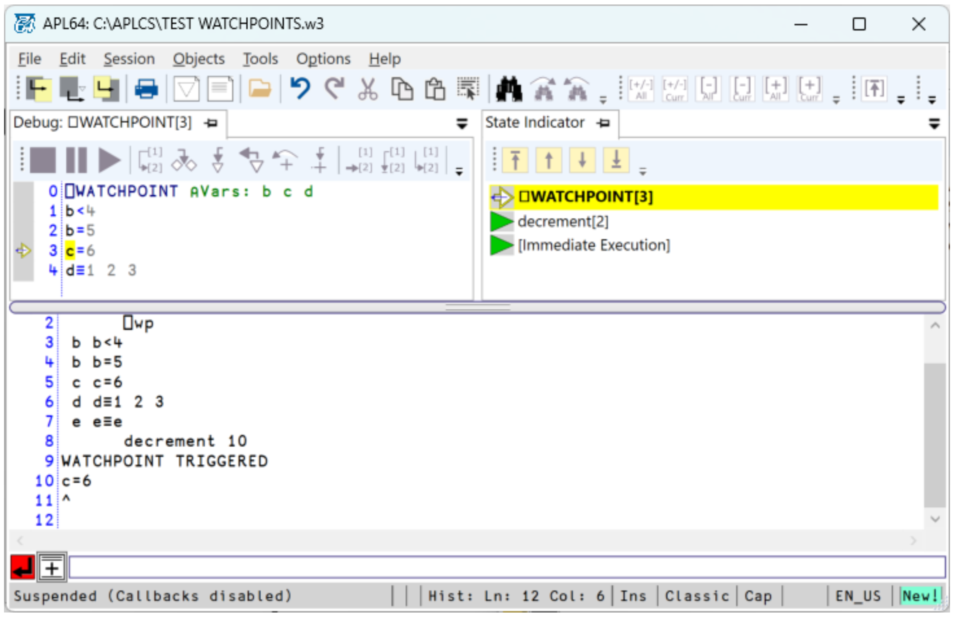
Test Expression: b<4

Active: ✓

With ⎕watchpoints defined as above, run the expression decrement 10. The function suspends when the value of c becomes 6:

```
    decrement 10
WATCHPOINT TRIGGERED
c=6
^
```

If the debug pane is visible, the function and stack displays look approximately like this:


APL64: C:\APLCS\TEST WATCHPOINTS.w3
File Edit Session Objects Tools Options Help
Debug: ⎕WATCHPOINT[3]
State Indicator

```
0 ⎕WATCHPOINT ∆Vars: b c d
1 b<4
2 b=5
3 c=6
4 d≡1  2  3
```

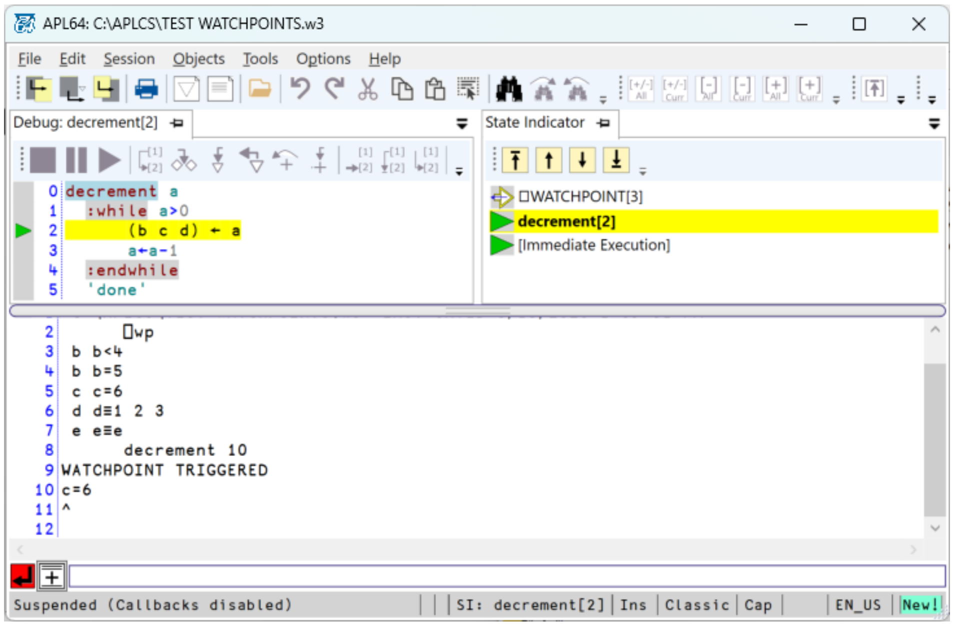
```
⎕WATCHPOINT[3]
decrement[2]
[Immediate Execution]
```

```
        ⎕wp
 b  b<4
 b  b=5
 c  c=6
 d  d≡1  2  3
 e  e≡e
        decrement  10
WATCHPOINT  TRIGGERED
c=6
^
```

Suspended (Callbacks disabled) | Hist: Ln: 12 Col: 6 | Ins | Classic | Cap | EN_US | New!

There may be multiple watchpoint entries with the same variable name, each with a different watchpoint expression; each expression is evaluated when the variable is updated. Multiple watchpoint expressions on different variables may be executed for strand assignments. Note that only the watchpoints that are relevant to the current line appear in the debug pane. The watchpoint variable e would appear only if that variable were subject to change. The expression assigned for ⍰ would always return 1, so any function will suspend when it attempts to assign a value to e.

You can click on the > decrement[2] line in the stack display to show the function suspended on the strand assignment line. You can then use the stepping options to move through your function:


APL64: C:\APLCS\TEST WATCHPOINTS.w3
File Edit Session Objects Tools Options Help
Debug: decrement[2]
State Indicator

```
0 decrement a
1   :while a>0
2       (b c d) ← a
3       a←a-1
4   :endwhile
5   'done'
```

```
⎕WATCHPOINT[3]
decrement[2]
[Immediate Execution]
```

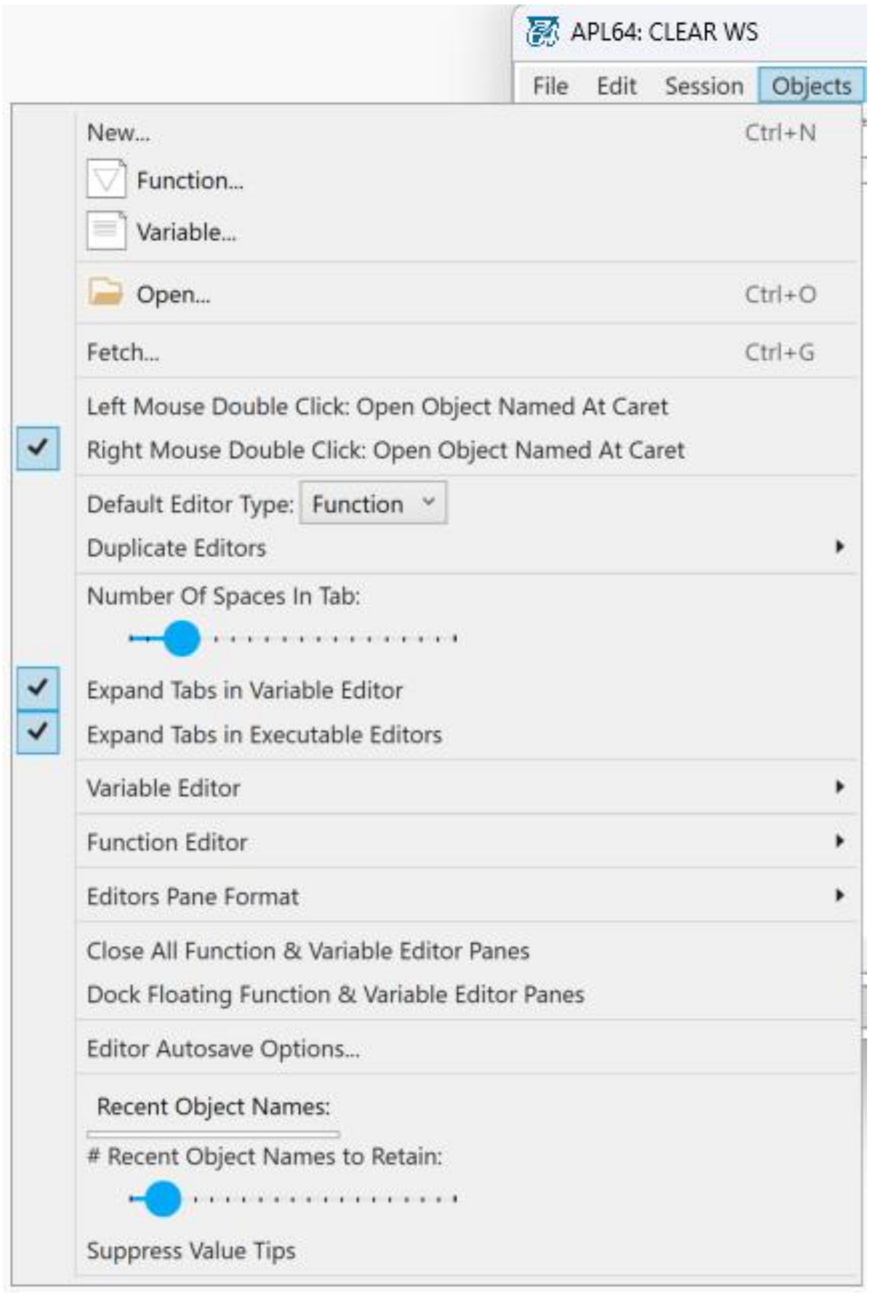
```
       ⎕wp
 b b<4
 b b=5
 c c=6
 d d≡1 2 3
 e e≡e
       decrement 10
WATCHPOINT TRIGGERED
c=6
^
```

```
Suspended (Callbacks disabled) | | SI: decrement[2] | Ins | Classic | Cap | EN_US | New!
```

The Objects Menu

The Objects menu allows you to create or edit any of five types of APL objects:

- a function
- a character vector
- a character matrix
- a numeric variable
- a nested array


APL64: CLEAR WS
File
Edit
Session
Objects
New... Ctrl+N
Function...
Variable...
Open... Ctrl+O
Fetch... Ctrl+G
Left Mouse Double Click: Open Object Named At Caret
✓ Right Mouse Double Click: Open Object Named At Caret
Default Editor Type: Function
Duplicate Editors
Number Of Spaces In Tab:
✓ Expand Tabs in Variable Editor
✓ Expand Tabs in Executable Editors
Variable Editor
Function Editor
Editors Pane Format
Close All Function & Variable Editor Panes
Dock Floating Function & Variable Editor Panes
Editor Autosave Options...
Recent Object Names:
# Recent Object Names to Retain:
Suppress Value Tips

New

When you select New from the Objects menu or press **Ctrl+N**, the system will create a new object based on the setting for Default Editor Type in the Objects menu. You can also use)edit *newobjectname* from the History or the APL Command Line.

Function

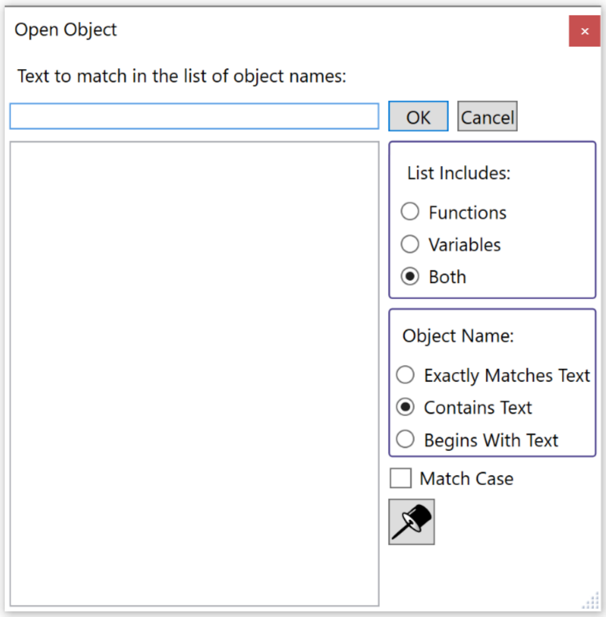
When you select Function from the Objects menu, the system will create a new function edit session.

Variable

When you select Variable from the Objects menu, the system will create a new variable edit session.

Open (an Object)

When you select Open from the Objects menu or press **Ctrl+O**, you are opening an edit session on an existing object. The system opens a window that contains a list of objects that exist in the current workspace. You can choose to see functions only, variables only, or both functions and variables. When you select an object in the Open Objects window, the system opens an editing window that displays the contents of the function or variable you select. The name of the object appears in the title bar.



Text to match in the list of object names:

List Includes: Functions Variables [✓ Both]

Object Name: Exactly Matches Text [✓ Contains Text] Begins With Text

Match Case

The Open Objects window has a filter at the bottom left corner of the window. The filter is used to restrict the names of the objects appearing in the list. It has a very simple syntax. The filtering is case insensitive. Asterisks may be used to denote zero or more characters. The filter drop-down stores new filters when objects are opened from the dialog. Up to eight filters are stored in the drop-down. The drop-down contents are not saved across sessions.

The Open Objects window has a push pin in the upper left corner. If you click on the push pin, the Open Objects window with its list of objects stays visible on the screen after you select an object.

When the Open Objects window is pinned, you can select another object to edit without returning to the Objects menu to select Open. You can use this facility when you plan to open several objects in succession.

If you click on the push pin a second time, the system unpins the window; it closes the window when you select an object. If you press the Cancel button, the system unpins and closes the window.

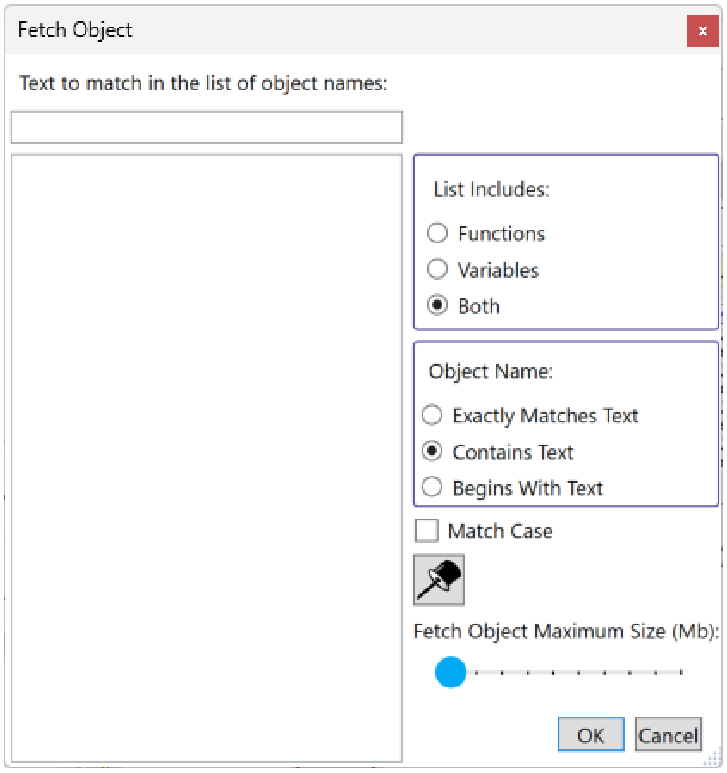
You can also open an Edit window for an object by double-clicking with the right mouse button on the name of the object or typing)edit *objectname* anywhere in the session.

Fetch (an Object)

The Fetch (Get) menu (**Ctrl+G**) allows you to insert the contents of a variable or function into the current character or function Edit or History pane. You can fetch the contents of a character variable, a function, or the character representation of a numeric variable. The Fetch option is unavailable in numeric editing sessions.

The Fetch Objects window has a filter like the Open Objects window and serves the same purpose.

The Fetch Object Maximum Size option allows you to specify the maximum size for an object that is fetched.



List Includes: Functions ( ) Variables ( ) [✓ Both]
Object Name: Exactly Matches Text ( ) [✓ Contains Text] Begins With Text ( )
Match Case ☐

Left Mouse Double Click: Open Object Named At Caret

The menu determines if a left mouse double click will open an object editor for the object name below the mouse (traditional APL behavior) or a select the text of the name of the object below the mouse (Windows standard behavior).

Right Mouse Double Click: Open Object Named At Caret

The menu determines if a right mouse double click will open an object editor for the object name below the mouse (traditional APL behavior) or display a context menu for the GUI control below the mouse (Windows standard behavior). If the Windows standard behavior is selected, the context menu will provide the Open Object Named at Caret option if applicable to the object below the mouse.

Default Editor Type

The menu allows you to specify the editor type to open by default.

Duplicate Editors

The menu allows you to specify the action to take when a duplicate editor is opened.

- **Prevent Duplicate Editors Option**
  When this menu item is checked, a duplicate editor will not be created and the existing editor will be selected. When this menu item is not checked, multiple editors of the same type and name can be created in an APL64 session.

- **Prompt Before Creating Duplicate Editor Session**
  This menu item is active only when 'Prevent Duplicate Editors' is not checked. If this menu item is checked, APL64 will prompt the user when an attempt is made to create a duplicate editor with the same name as an existing editor.

- **Duplicate Function Editors Are Read Only**
  This menu item is active only when 'Prevent Duplicate Editors' is not checked. If this

menu item is checked, duplicate function editors will be read only.

Number of Spaces in Tab

The menu sets the number of spaces used to replace a tab character forwards and backwards.

Expand Tabs in Variable Editors

When checked (default), and the tab key is pressed in the variable editor, the current text from the caret position will be adjusted to reflect the selected tab spacing. When unchecked, the tab key will insert a tab character at the caret position.

Expand Tabs in Executable Editors

When checked (default), and the tab key is clicked in an executable editor, the current text from the caret position will be adjusted to reflect the selected tab spacing. When unchecked, the tab key will insert a tab character at the caret position. Executable editors are the function editor, the editable classic history and the APL Command Line.

Variable Editor

The menu sets various settings for the variable editor.

Function Editor

The menu sets various settings for the function editor.

Editors Pane Format

The menu sets various layout settings for the Editors pane.

### Recent Object Names

The menu displays a list of the recent object names and object types that were edited. The system remembers this list between APL64 sessions. To edit one of these objects, select the object name. The system opens an edit window for the object you select.

#Recent Object Names to Retain

The menu sets the number of recent objects names to display in the Recent Object Names

Suppress Value Tips

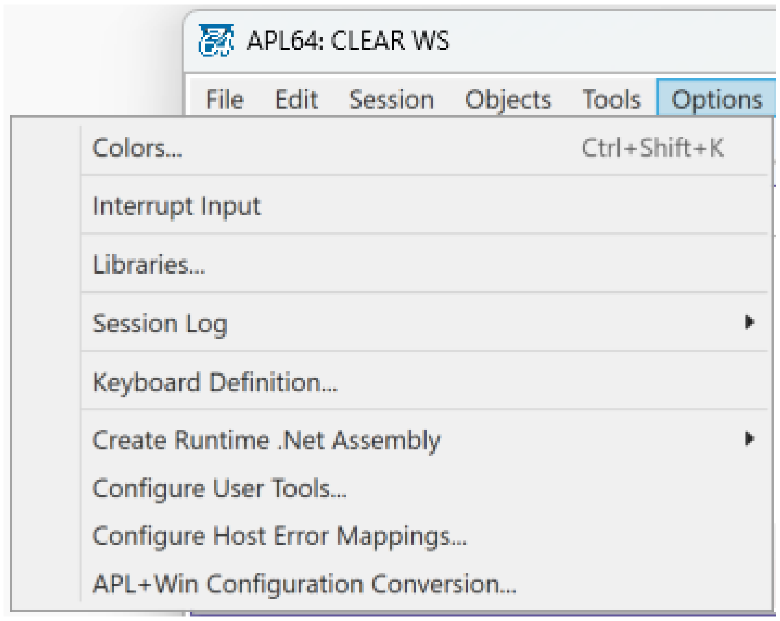
If you select this option, the system suppresses "value tips" in the history, edit and debugger. Value tips correspond to tooltips in the graphical user interface. They are small popup windows that indicate the header of a function or the value of a variable when you hover the cursor over the name. For variables, value tips give you a one-line display of the current value of a name preceded by shape. For functions, value tips show the function name and arguments.

The Tools Menu

Initially, the Tools menu has no choices. You can create your own selection of tools to perform the actions on your system that you find most useful. A Tool is any APL statement or expression you can write on one line. A Tool is often a user command, either user-defined or system-defined, but it can be a system command, a user-defined function, or any APL statement. You associate this command, or statement, with a menu choice on the Tools menu. When you select that choice, the system executes the command. The system does not display the APL statement that it executes, but it does display any output from the Tool in the APL64 History pane. See Using the History Log in **Help | Developer Version GUI | User Tools** menu for details.

The Options Menu

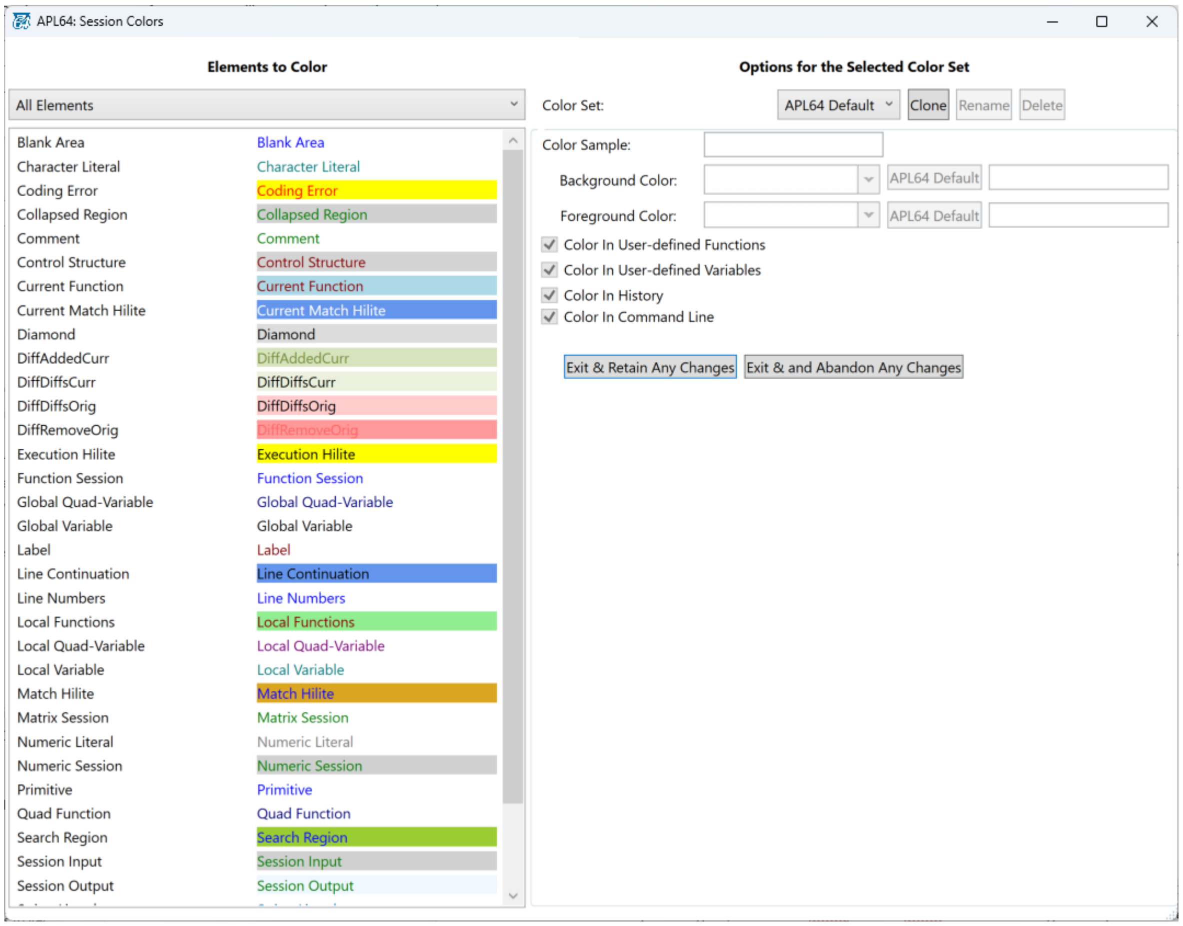
The Options Menu allows you to tailor your system for your own preferences.



Colors

When you select Colors from the Options menu, the system opens the APL64 Session Colors window where you can set text and background colors for various elements in the APL64 session. Multiple color sets are supported.

Note 1: The APL64 Default color set cannot be modified. You must first make a clone of it and then apply your changes to it.


APL64: Session Colors
Elements to Color
Options for the Selected Color Set
Clone
Rename
Delete
Exit & Retain Any Changes
Exit & and Abandon Any Changes

All Elements

Blank Area	Blank Area
Character Literal	Character Literal
Coding Error	Coding Error
Collapsed Region	Collapsed Region
Comment	Comment
Control Structure	Control Structure
Current Function	Current Function
Current Match Hilite	Current Match Hilite
Diamond	Diamond
DiffAddedCurr	DiffAddedCurr
DiffDiffsCurr	DiffDiffsCurr
DiffDiffsOrig	DiffDiffsOrig
DiffRemoveOrig	DiffRemoveOrig
Execution Hilite	Execution Hilite
Function Session	Function Session
Global Quad-Variable	Global Quad-Variable
Global Variable	Global Variable
Label	Label
Line Continuation	Line Continuation
Line Numbers	Line Numbers
Local Functions	Local Functions
Local Quad-Variable	Local Quad-Variable
Local Variable	Local Variable
Match Hilite	Match Hilite
Matrix Session	Matrix Session
Numeric Literal	Numeric Literal
Numeric Session	Numeric Session
Primitive	Primitive
Quad Function	Quad Function
Search Region	Search Region
Session Input	Session Input
Session Output	Session Output

Color Set: APL64 Default

Color Sample:

Background Color: APL64 Default

Foreground Color: APL64 Default

✓ Color In User-defined Functions
✓ Color In User-defined Variables
✓ Color In History
✓ Color In Command Line

The drop-down list and the list box below it is related to each other but not part of the same control. The drop-down list gives you three choices: General Elements, Syntactic Elements, or All Elements. In the picture, All Elements is chosen, and the list contains, as expected, all the elements for which you can specify colors. If you choose General Elements, you get approximately one-third of the list (all the elements with the word "Session" plus Line Numbers, Collapsed Region, Execution Hilite, Match Hilite and Search Region. If you choose Syntactic Elements, you get the rest of the list (including the element User Function, which refers to the name of a user-defined function when it appears in the session; when you are defining a function, that is the general element "Function Session").

When you highlight one of the selections in the list, the current setting shows in the sample edit box, and you may choose your color preferences. For the general elements, you may always select either or both background and text colors by clicking on the appropriate button. The background color is essential for the entire frame of the general element you have highlighted.

The Blank Area syntactic element is only background, although the Text button is not disabled for it. This choice colors the area outside of any typing; it is particularly useful for making obvious any trailing blanks.

The Coding Error syntactic element changes dynamically as you type; this includes not only mismatched delimiters, but other types of errors as well. The DiffAddedCurr, DiffDiffsCurr, DiffDiffsOrig, and DiffRemoveOrig elements apply to the syntax color choices in the in the compare format of the APL64 function editor.

The Execution Hilite general element applies only with debugger; it applies to the selected line that in the past has been bright yellow.

The Undefined Name syntactic element is a newly recognized category; it represents a name that currently has no value and is not localized in the current context, useful to identify possible value errors.

The Match Hilite general element applies to the syntax color choices for the highlighting of matching pairs of elements, control structure names and user defined names.

The new color selection scheme is also recorded in the XML configuration file. If new entries are not set, the system reads old-style entries and converts them to the new entries. However, it leaves the old entries intact, so you can still use the .INI file with older systems.

The following constitute error conditions that may be colored in a function header: more than two parameters; left arrow in the wrong position; missing or multiple consecutive semicolons; extraneous characters; localization of invalid or non-localizable system names.

The following constitute error conditions that may be colored in function text: invalid control structure names or invalid system names; system or user commands; mismatched delimiters.

The following constitute error conditions that may be colored in the APL64 session: invalid control structure names or invalid system names; mismatched delimiters.

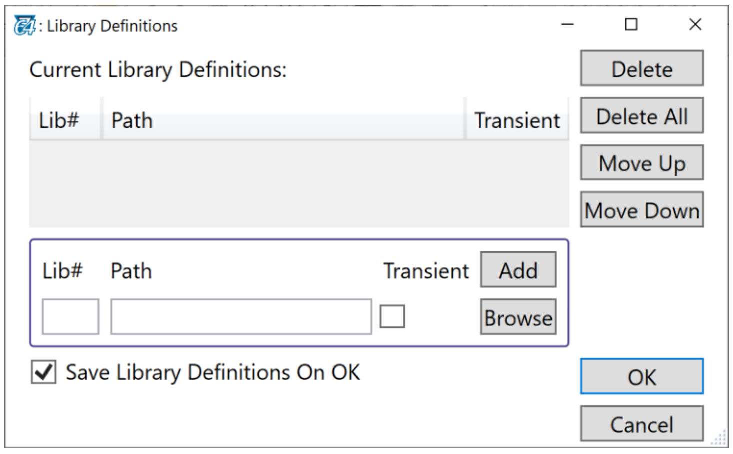
In the APL64 session, only input lines and matching pairs of punctuation elements get syntax coloring. If you move the cursor to a line in the session using a mouse click, that line gets syntax coloring temporarily. If you move the cursor to a line in the session using the keyboard and then alter the line, the line gets colored. System and user commands are colored as primitive functions.

Interrupt Input

The Interrupt Input choice on the Options menu enables you to interrupt a program that is requesting Quad (⎕) or Quote-quad (⍞) input; the system cannot respond to Ctrl+Break at the point of this input because the program has returned temporarily to immediate execution. This choice is available only when the system stops for the input; it is usually greyed out. The system suspends execution of the program with an interrupt. To resume execution, type `→⎕lc`.

Libraries

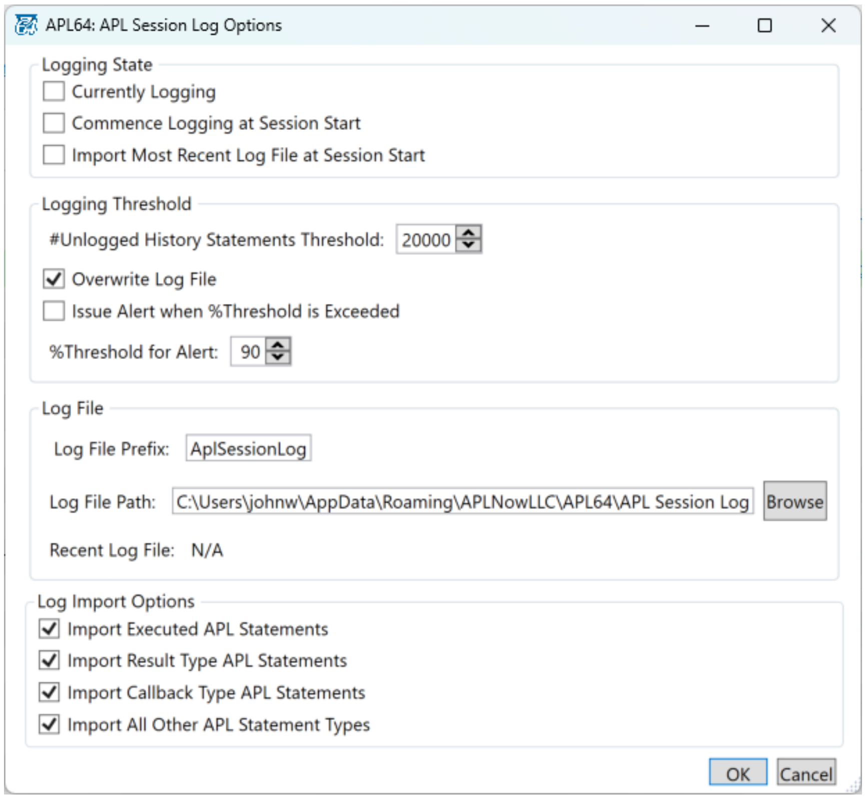
When you select Libraries from the Options menu, the system opens the Libraries Definitions window where you can edit the contents of `⎕libs` and, optionally, record the changes in your XML configuration file. See the description of `⎕libs` in the *Systems Functions Manual* for more information on using libraries. However, note that libraries are largely obsolete in APL64. See Using the History Log in **Help | APL Language | APL Libraries** menu for details on how you use this window.



☑ Save Library Definitions On OK

Session log

You can save a record of your session. Click on the Options menu and then on Session Log. If you click the check box button under Logging State to turn on the Session log facility, each line in the history, both what you type and what is output, is recorded in the log file whose name you specify. Lines that you paste into the session are not included in the log unless you subsequently execute them. See Using the Session Log in **Help | Developer Version GUI | Using the Session Log** menu for details on how you use this window.


APL64: APL Session Log Options

Logging State
- [ ] Currently Logging
- [ ] Commence Logging at Session Start
- [ ] Import Most Recent Log File at Session Start

Logging Threshold
#Unlogged History Statements Threshold: 20000
- [✓] Overwrite Log File
- [ ] Issue Alert when %Threshold is Exceeded

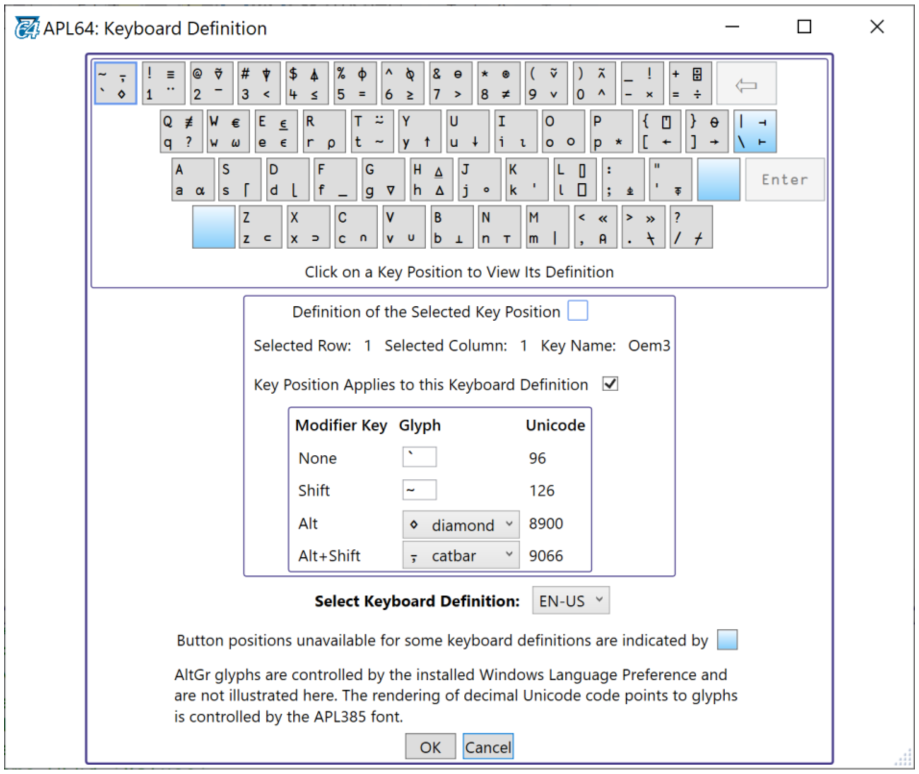
%Threshold for Alert: 90

Log File
Log File Prefix: AplSessionLog
Log File Path: C:\Users\johnw\AppData\Roaming\APLNowLLC\APL64\APL Session Log
Recent Log File: N/A

Log Import Options
- [✓] Import Executed APL Statements
- [✓] Import Result Type APL Statements
- [✓] Import Callback Type APL Statements
- [✓] Import All Other APL Statement Types

Keyboard Definition

When you select Keyboard Definition from the Options menu, the system opens the Keyboard Definition window to select or create a custom APL keyboard definition to be used in new APL64 session instances from a user-selected 'Keyboard Definition Basis'. See Using the History Log in **Help | Developer Version GUI | Keyboard Definition** menu for details.



Click on a Key Position to View Its Definition

Definition of the Selected Key Position

Selected Row: 1 Selected Column: 1 Key Name: Oem3

Key Position Applies to this Keyboard Definition ☑

Modifier Key	Glyph	Unicode
None	`	96
Shift	~	126
Alt	⋄ diamond	8900
Alt+Shift	⍪ catbar	9066

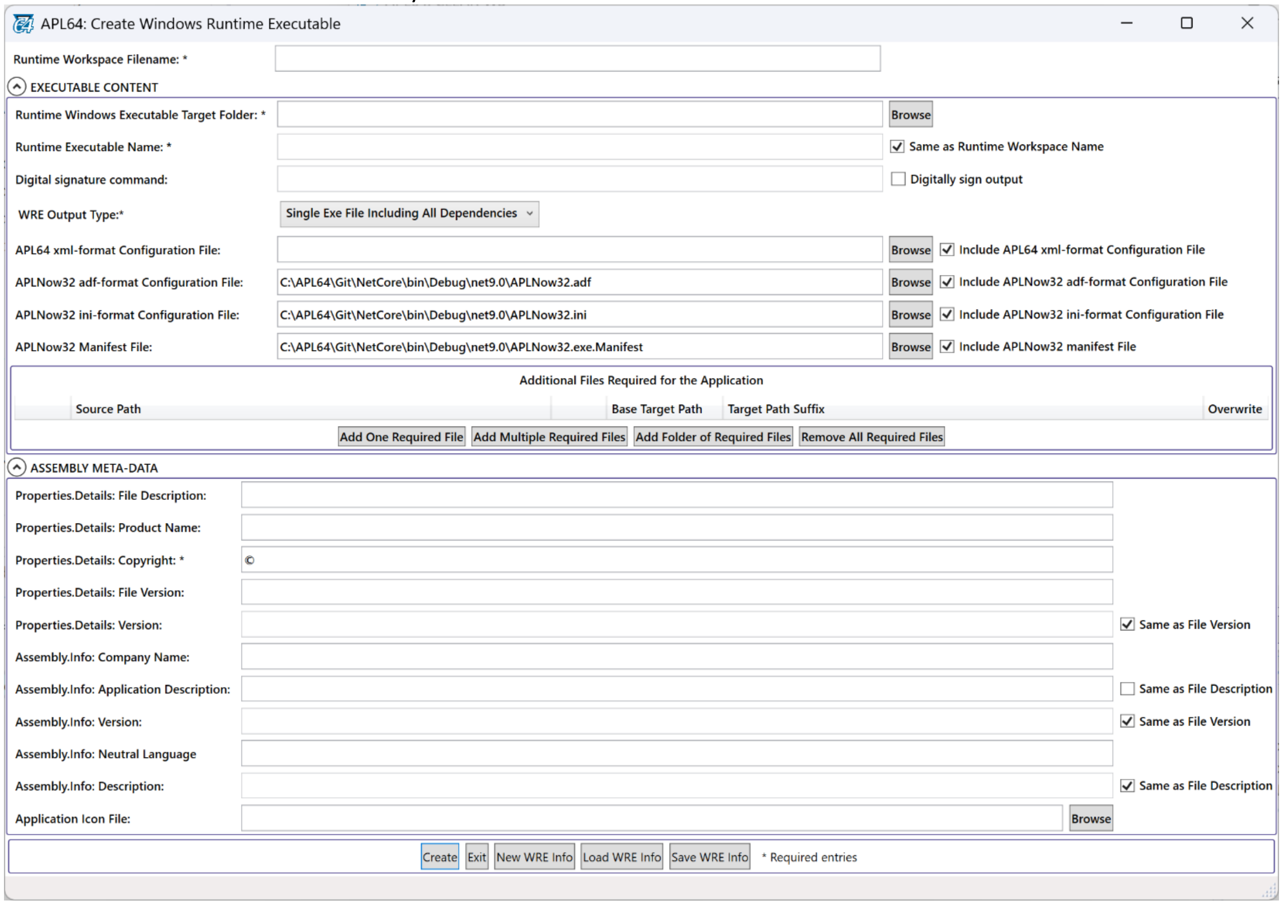
Select Keyboard Definition: EN-US

Button positions unavailable for some keyboard definitions are indicated by 🟦

AltGr glyphs are controlled by the installed Windows Language Preference and are not illustrated here. The rendering of decimal Unicode code points to glyphs is controlled by the APL385 font.

Create Runtime .Net Assembly

When you select **Create Runtime .Net Assembly | Create Windows Runtime Executable** from the Options menu, the system opens the Create Windows Runtime Executable window where you can create or modify an existing Windows runtime executable project. See the "Defining a Tool" section earlier in this chapter for details on how you use this window. See Create Windows Runtime Executable in **Help | Developer Version GUI | Create Windows Runtime Executable** menu for details on how you use this window.



APL64: Create Windows Runtime Executable

Runtime Workspace Filename: *

EXECUTABLE CONTENT

Runtime Windows Executable Target Folder: *		Browse
Runtime Executable Name: *		✓ Same as Runtime Workspace Name
Digital signature command:		☐ Digitally sign output
WRE Output Type:*	Single Exe File Including All Dependencies	
APL64 xml-format Configuration File:		Browse ✓ Include APL64 xml-format Configuration File
APLNow32 adf-format Configuration File:	C:\APL64\Git\NetCore\bin\Debug\net9.0\APLNow32.adf	Browse ✓ Include APLNow32 adf-format Configuration File
APLNow32 ini-format Configuration File:	C:\APL64\Git\NetCore\bin\Debug\net9.0\APLNow32.ini	Browse ✓ Include APLNow32 ini-format Configuration File
APLNow32 Manifest File:	C:\APL64\Git\NetCore\bin\Debug\net9.0\APLNow32.exe.Manifest	Browse ✓ Include APLNow32 manifest File

Additional Files Required for the Application

Source Path	Base Target Path	Target Path Suffix	Overwrite

Add One Required File | Add Multiple Required Files | Add Folder of Required Files | Remove All Required Files

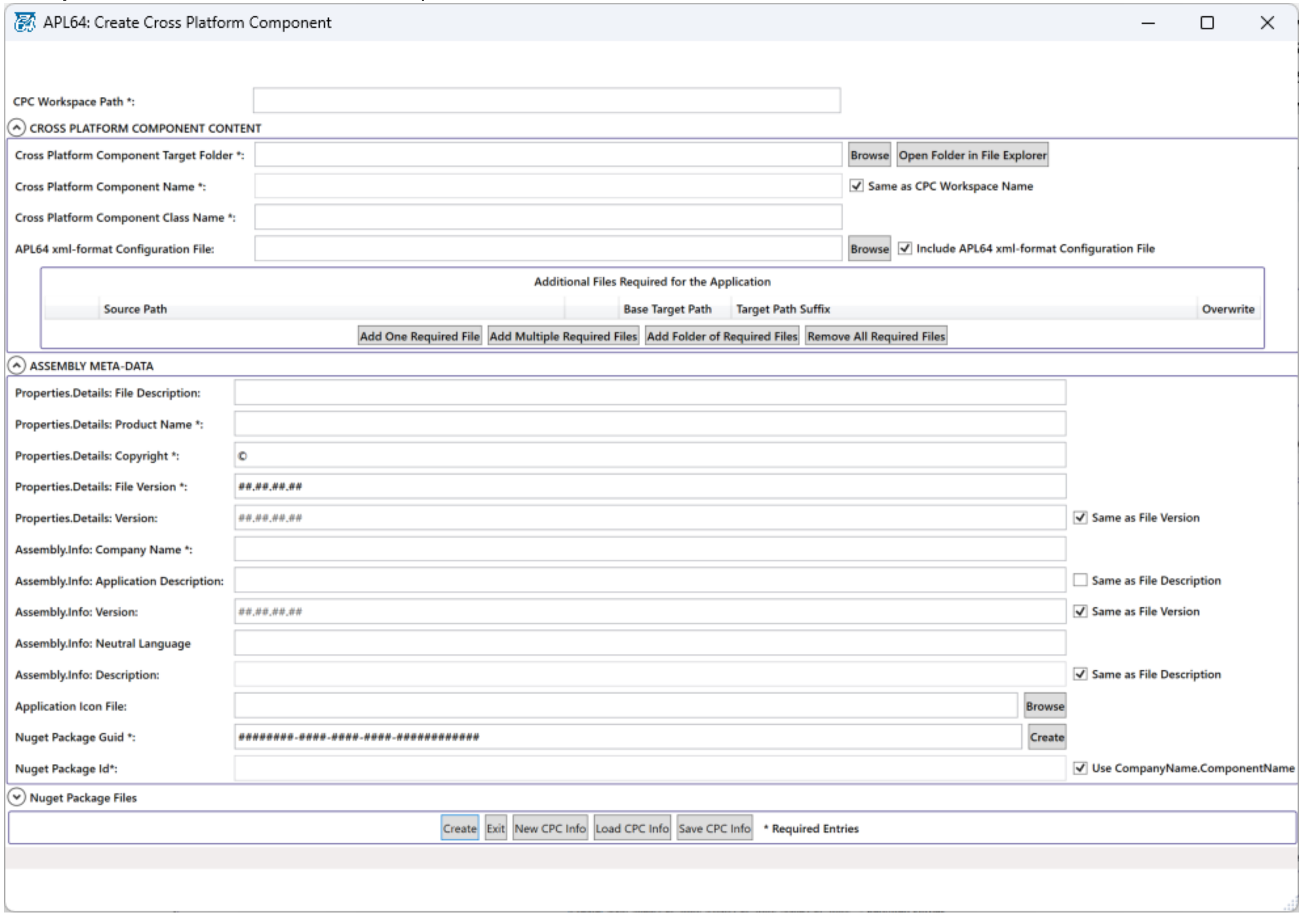
ASSEMBLY META-DATA

Properties.Details: File Description:		
Properties.Details: Product Name:		
Properties.Details: Copyright: *	©	
Properties.Details: File Version:		
Properties.Details: Version:		✓ Same as File Version
Assembly.Info: Company Name:		
Assembly.Info: Application Description:		☐ Same as File Description
Assembly.Info: Version:		✓ Same as File Version
Assembly.Info: Neutral Language		
Assembly.Info: Description:		✓ Same as File Description
Application Icon File:		Browse

Create | Exit | New WRE Info | Load WRE Info | Save WRE Info | * Required entries

Create Cross Platform Component

When you select **Create Runtime .Net Assembly | Create Cross Platform Component** from the Options menu, the system opens the Create Cross Platform Component window where you can create or modify an existing cross platform component project. See the "Defining a Tool" section earlier in this chapter for details on how you use this window. See Create Cross Platform Component in **Help | Developer Version GUI | Create Cross Platform Component** menu for details on how you use this window.


APL64: Create Cross Platform Component

CPC Workspace Path *:

CROSS PLATFORM COMPONENT CONTENT

Cross Platform Component Target Folder *:		Browse · Open Folder in File Explorer
Cross Platform Component Name *:		✓ Same as CPC Workspace Name
Cross Platform Component Class Name *:		
APL64 xml-format Configuration File:		Browse · ✓ Include APL64 xml-format Configuration File

Additional Files Required for the Application

Source Path	Base Target Path	Target Path Suffix	Overwrite

Add One Required File · Add Multiple Required Files · Add Folder of Required Files · Remove All Required Files

ASSEMBLY META-DATA

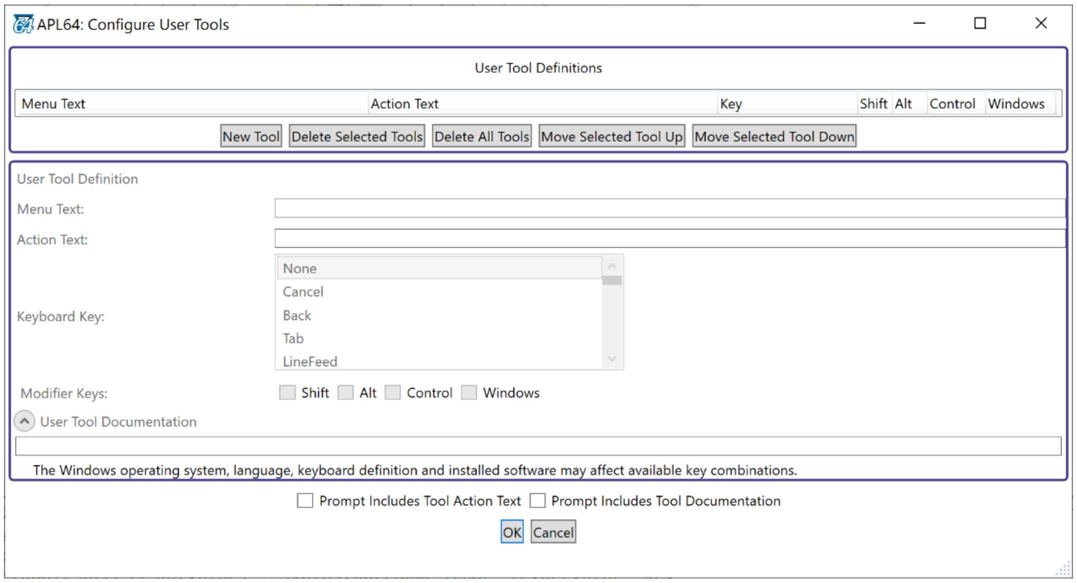
Properties.Details: File Description:		
Properties.Details: Product Name *:		
Properties.Details: Copyright *:	©	
Properties.Details: File Version *:	##.##.##.##	
Properties.Details: Version:	##.##.##.##	✓ Same as File Version
Assembly.Info: Company Name *:		
Assembly.Info: Application Description:		☐ Same as File Description
Assembly.Info: Version:	##.##.##.##	✓ Same as File Version
Assembly.Info: Neutral Language		
Assembly.Info: Description:		✓ Same as File Description
Application Icon File:		Browse
Nuget Package Guid *:	########-####-####-####-############	Create
Nuget Package Id*:		✓ Use CompanyName.ComponentName

Nuget Package Files

Create · Exit · New CPC Info · Load CPC Info · Save CPC Info · * Required Entries

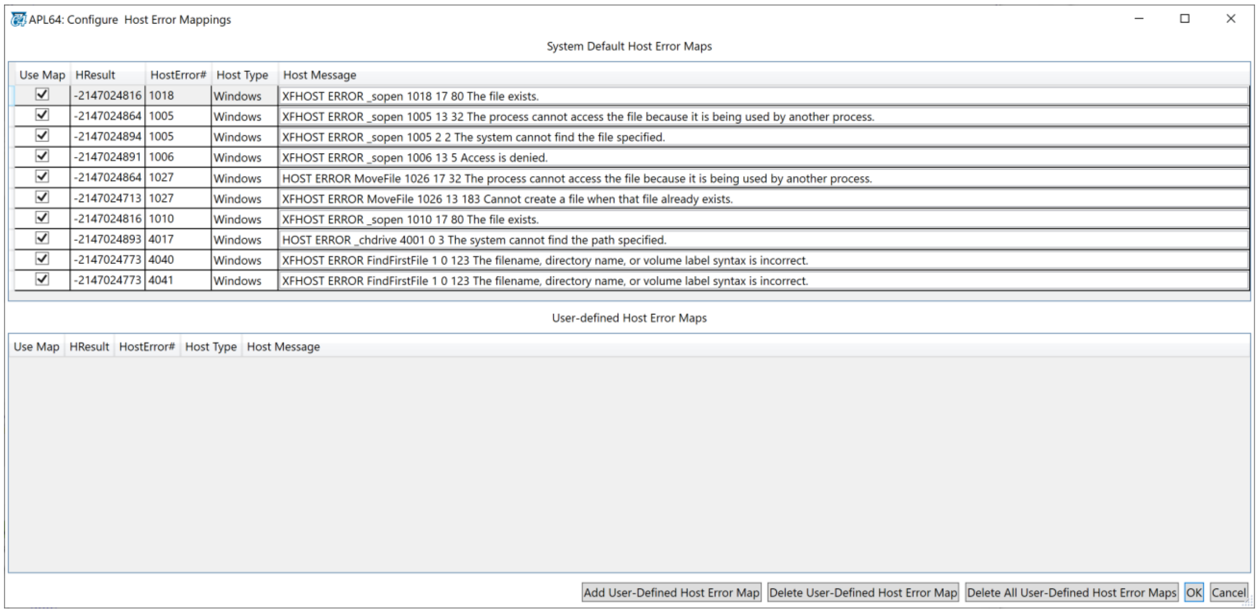
Configure User Tools

When you select Tools from the Options menu, the system opens the Configure User Tools window where you can add, delete, or modify a tool that you define. See the "Defining a Tool" section earlier in this chapter for details on how you use this window. See Using the History Log in **Help | Developer Version GUI | User Tools** menu for details on how you use this window.



Configure Host Error Mappings

When you select Configure Host Error Mappings from the Options menu, the system opens the Configure Host Error Mappings window where you can coordinate exception handling in an APL+Win-based application with an APL64-based application system.

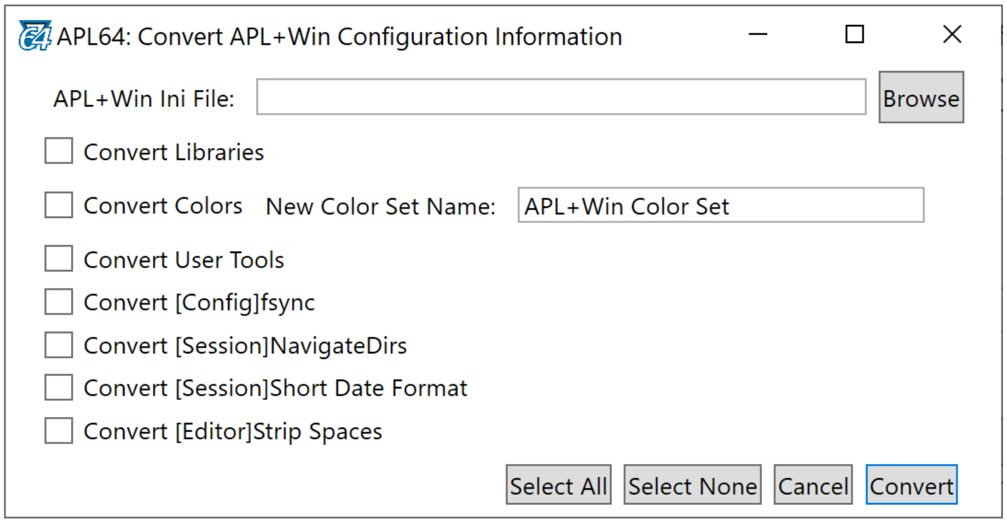

APL64: Configure Host Error Mappings
System Default Host Error Maps

Use Map	HResult	HostError#	Host Type	Host Message
✓	-2147024816	1018	Windows	XFHOST ERROR _sopen 1018 17 80 The file exists.
✓	-2147024864	1005	Windows	XFHOST ERROR _sopen 1005 13 32 The process cannot access the file because it is being used by another process.
✓	-2147024894	1005	Windows	XFHOST ERROR _sopen 1005 2 2 The system cannot find the file specified.
✓	-2147024891	1006	Windows	XFHOST ERROR _sopen 1006 13 5 Access is denied.
✓	-2147024864	1027	Windows	HOST ERROR MoveFile 1026 17 32 The process cannot access the file because it is being used by another process.
✓	-2147024713	1027	Windows	XFHOST ERROR MoveFile 1026 13 183 Cannot create a file when that file already exists.
✓	-2147024816	1010	Windows	XFHOST ERROR _sopen 1010 17 80 The file exists.
✓	-2147024893	4017	Windows	HOST ERROR _chdrive 4001 0 3 The system cannot find the path specified.
✓	-2147024773	4040	Windows	XFHOST ERROR FindFirstFile 1 0 123 The filename, directory name, or volume label syntax is incorrect.
✓	-2147024773	4041	Windows	XFHOST ERROR FindFirstFile 1 0 123 The filename, directory name, or volume label syntax is incorrect.

User-defined Host Error Maps
Use Map
HResult
HostError#
Host Type
Host Message
Add User-Defined Host Error Map
Delete User-Defined Host Error Map
Delete All User-Defined Host Error Maps
OK
Cancel

APL+Win Configuration Conversion

When you select APL+Win Configuration Conversion from the Options menu, the system opens the Convert APL+Win Configuration Information window where you can specify the APL+Win configuration file and the information to import to APL64 from APL+Winings in future sessions.



APL64: Convert APL+Win Configuration Information

APL+Win Ini File:
Browse

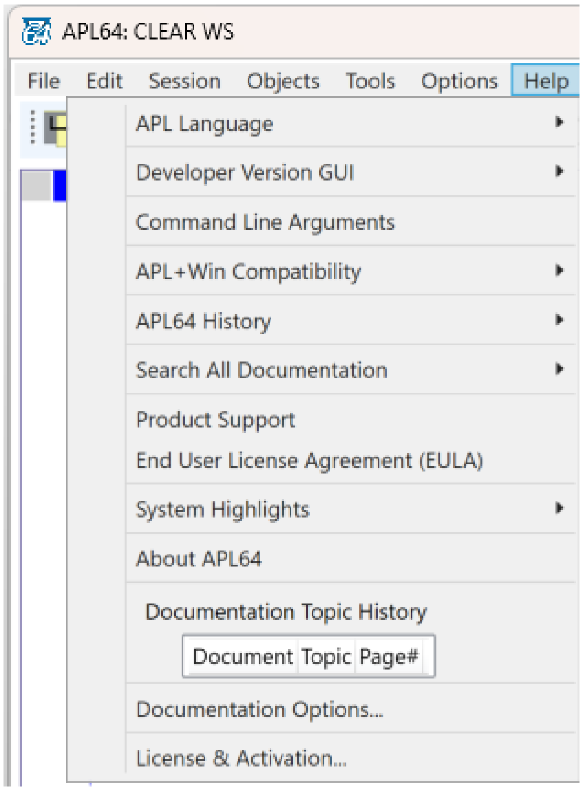
- [ ] Convert Libraries
- [ ] Convert Colors   New Color Set Name: APL+Win Color Set
- [ ] Convert User Tools
- [ ] Convert [Config]fsync
- [ ] Convert [Session]NavigateDirs
- [ ] Convert [Session]Short Date Format
- [ ] Convert [Editor]Strip Spaces

Select All | Select None | Cancel | Convert

The Help Menu

Documentation is directly available from the APL64 Developer GUI. Use the Help menu items to select comprehensive documentation of APL64 features. For context-sensitive documentation of a particular APL64 element, move the keyboard focus to the History, APL Command Line or an editor in the APL64 session, then place the caret immediately before a feature of interest or select a portion of the pane's content containing a feature of interest and click the F1 key.

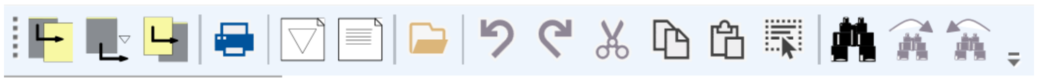
The APL64 documentation window displays the user-selected documentation in the default Windows browser and a history of documentation topics previously viewed during the current APL64 Developer version instance is listed in the **Help | Documentation History Topic**.



- APL Language provides reference information on APL primitive functions, system functions, system variables, system commands, control structure, user commands, user-defined functions, user-defined variables, network interface (⎕ni), C# script engine, libraries, ⎕nfe, ⎕xl, ⎕xml, ⎕path, SQL system functions, ⎕askx, ⎕skd, ⎕editevents and ⎕editinfo, and ⎕edss. You select the topic of choice from the submenu.
- Developer Version GUI provides reference information on using the facilities of APL64 and navigating an APL64 session. You select the topic of choice from the submenu.
- Command Line Arguments provides reference information on how to utilize optional command line arguments to configure an instance of APL64.
- APL+Win Compatibility provides key reference information for those migrating to APL64 from APL+Win, and points out some key differences between the two. This also has reference information on creating and using the facilities that allow you to build a graphical user interface and interact with other applications using Windows standards.
- APL64 History provides reference information on the progress of the APL64 system from inception.
- Search All Documentation:
    - Search All Documentation with Adobe provides an overview on searching all documentation for a keyword with a locally installed Adobe Acrobat program
    - Search All Documentation with Google searches all documentation for a keyword with the Google search engine
- Product Support tells you where to find more information and how to contact APL2000 for product information or assistance.
- End User License Agreement (EULA) provides the license agreement for the APL64 system.
- System highlights provide options to be alerted in the status bar when new System Highlights topics and updates are available.
- About APL64 displays the official product window. This provides key information, including copyright information and the version numbers of the APL64 system and related components.

- Documentation Topic History provides a list of the recent documentation topics opened.
- Document Options launches the Document Options window to allow the online documentation to be stored and accessed from the machine locally.
- License & Activation launches the tells you how to activate and deactivate the APL64 system.

The Standard Toolbar



The icons on the standard toolbar represent the following actions or menu choices:

File Menu (Actions on a Workspace)

- Load
- Copy
- Save

Print

Objects Menu (Actions to Edit a New Object)

- New Function
- New Variable

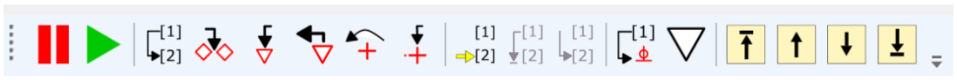
Objects Menu (Actions to Edit an Existing Object)

- Open

Edit Menu (Actions while in an Edit Window)

- Undo
- Redo
- Cut
- Copy
- Paste
- Select All
- Find
- Find Next
- Find Previous

The Debugger Toolbar



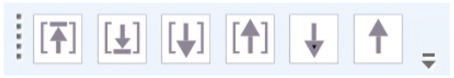
The icons on the Debugger Pane toolbar represent the following actions or keyboard shortcut:

(Actions while in the Debug Pane)

- Pause (Pause)
- Run Resume (F5)
- Step to the next LINE (Shift+F10)
- Step to the next STATEMENT (F10)
- Step INTO function (F11)
- Step OUT of function (Shift+F11)

- Step by primitive OVER function (Ctrl+F10)
- Step by primitive INTO function (Ctrl+F11)
- Set next line (Ctrl+F5)
- Run to caret or below (Ctr+F7)
- Run to caret (Ctrl+Shift+F7)
- Copy pending code to APL (F12)
- Edit Pending Function (Ctrl+F12)
- Top SI Level (Ctrl+ Num *)
- Up one SI Level (Ctrl+Num +)
- Down one SI Level (Ctrl+Num -)
- Bottom SI Level (Ctrl+Num /)

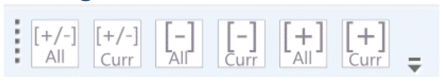
The Control Structure Keyword Toolbar



There options allow you to navigate in control structure blocks in the function editor. Note that single line control statements such as :Return, :Leave, and :Continue, do not participate in these movements for either Up/Down, Top/Bottom, or Next/Previous actions. Only compound control structures that have a start and end are involved with these actions. Also note that these options are also not on the Edit menu because when a function editor is undocked or floated, it would not have direct access to the Edit menu in the APL64 session to invoke the Control Structure Keyword options. So, for convenience's sake, the decision was to make them available only on the toolbar.

- The Jump to Top Keyword in Current Control Structure Block (**Ctrl+M, Ctrl+T**) moves the input caret to the top keyword in the control structure block.
- The Jump to Bottom Keyword option in Current Control Structure Block (**Ctrl+M, Ctrl+B**) moves the input caret to the bottom keyword in the control structure block.
- The Jump to Next Keyword in Current Control Structure Block (**Ctrl+M, Ctrl+N**) moves the input caret to the next keyword in the control structure block.
- The Jump to Previous Keyword in Current Control Structure Block (**Ctrl+M, Ctrl+P**) moves the input caret to the previous keyword in the control structure block.
- The Jump Down to Start of Next Control Structure Block (**Ctrl+M, Ctrl+D**) moves the input caret to the top keyword in the next different control structure block.
- The Jump Up to Start of Previous Control Structure Block (**Ctrl+M, Ctrl+U**) moves the input caret to the top keyword in the previous different control structure block.

The Outlining Toolbar



These options allow you to expand and collapse regions in the function editor. Note that unlike APL+Win, they're not also on the Edit menu because when a function editor is undocked or floated, it would not have direct access to the Edit menu in the APL64 session to invoke the Outlining options. So, for convenience's sake, the decision was to make them available only on the toolbar.

- The Toggle All Regions option (**Ctrl+M, Ctrl+L**) toggles all outlining. This combines expand and collapse into a toggle action.
- The Toggle Current Region option (**Ctrl+M, Ctrl+M**) toggle outlining expansion. This operates the same as clicking the region icons displayed in the left margin of the function editor with the mouse. If the input caret is not on a region icon line, the toggle occurs on the nearest region icon located above the input caret line.
- The Collapse All Regions option (**Ctrl+M, Ctrl+A**) collapses all regions.
- The Collapse Current Region option (**Ctrl+M, Ctrl+C**) collapses the current region. If the input caret is not on a region icon line, the collapse occurs on the nearest region icon above the input caret line.
- The Expand all Regions option (**Ctrl+M, Ctrl+X**) expands all regions.
- The Expand Current Region option (**Ctrl+M, Ctrl+E**) expands the current region.

Chapter 3: Introduction to the Language

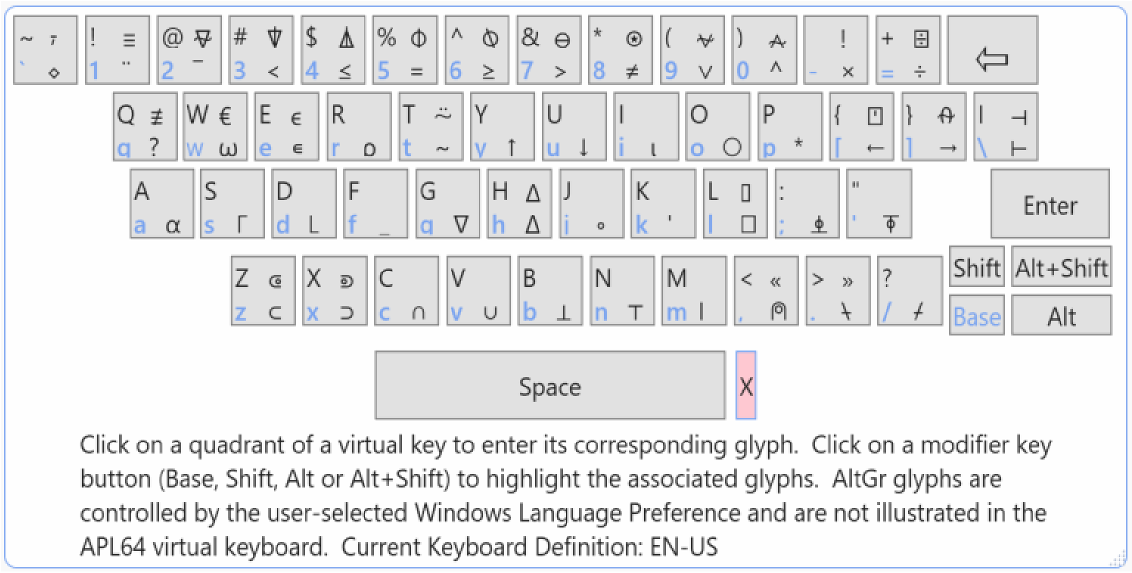
This chapter is an introduction to a language. Just as it could take years to become fluent in a spoken language, and a lifetime to learn eloquence, APL has riches that are beyond the scope of this brief treatment. Just as you use words to learn a spoken language, this chapter uses some of the constructs and syntax of APL before they have been introduced. You can infer the meaning of the symbols in the examples, or you can look them up in the Primitive Functions and Operators chapter in this manual. The examples in this chapter demonstrate the point being discussed; it is hoped that the elegance of the language is also apparent.

Getting Started

When you initiate an APL64 session, the system opens a clear workspace. You can type any valid APL statement in this session, and the interpreter evaluates it immediately. This is called immediate execution. As you are reading this chapter, you can type any of the statements, or you can experiment with variations. If you define more variables than you are comfortable with, you can begin again by selecting Clear on the File menu. If you want to end the session, select Exit on the File menu. You can save a workspace before you exit by using the File menu.

APL64 uses many special symbols and its own font. To see the location of the special symbols on the keyboard, type **Ctrl+B** in your session. The figure below shows the standard APL Text keyboard layout.

Standard APL64 Keyboard



Click on a quadrant of a virtual key to enter its corresponding glyph. Click on a modifier key button (Base, Shift, Alt or Alt+Shift) to highlight the associated glyphs. AltGr glyphs are controlled by the user-selected Windows Language Preference and are not illustrated in the APL64 virtual keyboard. Current Keyboard Definition: EN-US

The system displays a keyboard with the normal lower- and uppercase letters or symbols on the left side of the keys in black and the special symbols to the right in red. You can type these special characters by pressing the Alt key plus the appropriate key for the red symbol at the lower right; if there is a second symbol in red at the top right of the key, press Shift+Alt plus the key.

You can also generate the symbol in the edit field of the display window by clicking with the mouse on the appropriate quadrant of the key. You must click on or very close to the symbol you want. When you create the symbol or expression you want, click on the Enter button in the display window. The system copies the text to the clipboard, which you can then paste in your session or an Edit window. If you use the edit field in this display window extensively, you can also use the Clear, Cut, Copy, and Paste buttons to the left of the field and the Backspace button to the right of the field to create expressions more efficiently.

Understanding Arrays

APL is an extremely powerful programming language. One of its greatest strengths is that it handles entire arrays of data as single objects.

Working with Numeric Arrays

As you might expect, APL performs basic arithmetical functions on individual data elements. You can type a number as a constant in an APL expression, or you can assign the value to a variable and use the variable name in the expression.

```
      2+5
7

      A←2
      B←5
      A+B
7
```

Note that in an APL64 session, the cursor starts at a six-space indent. When the system returns output, it starts with no indentation.

APL also performs arithmetical functions on vectors (strings or lists of numbers). This use of the term "vector" is different from the more traditional meaning of a quantity having magnitude and direction. A vector can have any number of independent elements along one dimension.

```
      1 2 3 + 4 5 6
5 7 9

      VA←1 2 3
      VB←4 5 6
      VA+VB
5 7 9
```

It does not matter how many numbers are in the vectors. APL can just as easily add `10,000` pairs of numbers with one instruction. The theoretical limit in APL64 is greater than `200` million; the practical limit is dictated by the amount of available memory.

APL64 also performs arithmetical functions on matrices (tables of numbers). A matrix is a two-dimensional array of numbers. The rows and columns can be of any length within practical limits. You can form a matrix in APL64 by using the Reshape function (dyadic ⍴).

```
      MA←2 3⍴⍳6 ⋄ MB←2 3⍴7 8 9 10 11 12
      MA
1 2 3
4 5 6

      MB
 7  8  9
10 11 12

      MA+MB
 8 10 12
14 16 18
```

APL64 also performs arithmetical functions on multidimensional arrays. APL64 displays a multidimensional array as a series of matrices.

```
      NA←2 2 3 4⍴⍳48
      NA
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24


25 26 27 28
29 30 31 32
33 34 35 36

37 38 39 40
41 42 43 44
45 46 47 48
```

Note that the blank line between the first and second matrices indicates a third dimension. Two blank lines between the second and third matrices indicates that NA is a four-dimensional array. If the assign statement had been NB←4 3 4⍴⍳48, then APL64 would show only one blank line after each matrix. The numbers would display in the same order, but NB would be a three-dimensional array.

All of the above examples are arrays. A single value is a scalar. A string of numbers, no matter how long, is a vector. A two-dimensional array, with numbers in rows and columns, is a matrix. Arrays with more than two dimensions are multidimensional arrays.

The rank of an array is the number of dimensions it has. The shape of an array is a vector that contains the lengths of its dimensions. You can determine the shape of an array by using the Shape (monadic ⍴) function followed by the name of the array or its designation. You can determine the rank of an array by determining the shape of its shape (⍴⍴).

Classifying Arrays

Example Above	**Shape ⍴**	**Rank ⍴⍴**	**Name**	**Description**
A		0	Scalar	Single item
VA	3	1	Vector	Linear string of items
MA	2 3	2	Matrix	Two-dimensional table of items
NA	2 2 3 4	4	Multidimensional array	Any set of items structured in three or more dimensions

Scalars, vectors, and matrices are all arrays. You could just as accurately call a vector a one-dimensional array. This manual uses the commonly accepted names to discuss frequently used arrays of small ranks. Generally, the same functionality applies to arrays of any rank. Restrictions are noted in the descriptions of the primitive functions.

Recognizing Arrays where the Rank is not Obvious

Every array, even an empty one, has a shape and a rank. A matrix defined by MV← 0 6 ⍴⍳0 has no elements. It does, however, have a shape and a rank.

```
      MV
      ⍴MV
0 6
      ⍴⍴MV
2
```

Empty arrays are useful in APL64. For example, you can use one as the starting value of a variable that grows in successive executions of a program or that grows in successive iterations of a loop within a program. In other programming languages, you must use special tests to detect empty arrays and to avoid potential errors. Typical APL statements work regardless of whether or not an array is empty.

Similarly, a single data element is not necessarily a scalar. It can be a one-element vector, a one-element matrix, or a one-element array of any rank. If you select one element from a vector using the Take (↑) function, the result is a one-element vector, not a scalar.

```
      V ← 1 2 3 4 5 6
      VA ← 1↑V
      VA
1
      SA←1
      SA
1
```

These two variables are equal in value, but they do not match, because their shapes are different. (The Equal function returns `1` if the values it compares are the same; the Match function returns `1` only if the rank, shape, and values of the two arguments are the same.)

```
      VA=SA
1
      VA≡SA
0
```

Under many circumstances involving primitive functions, you can use either shape array without being aware of the difference. The results of a calculation display the same numerical values. However, the shapes and ranks of the results can differ, which could affect you if you store the result to use at a later time.

Working with Nested Arrays

Thus far, this chapter described simple arrays that have one value (either one number or one character) in each position, no matter how many dimensions. You can also create arrays with more complex structure.

An array can contain another array as one element of its structure. This is called nesting. The number of levels of nesting is called depth. You can create a nested array by enclosing values within parentheses or by assigning an array to another array. See also the description in the Primitive Functions and Operators chapter of the Enclose function (⊂), which creates a nested scalar.

```
      NestA ← 1 2 3 (4 5 6) 7
      NestA
 1 2 3  4 5 6  7
```

When APL64 displays the result, the nesting is subtly indicated by the extra spaces before and after the nested element. You can see the structure more clearly using the `]display` user command.

The array named `NestA` is a vector, rank `1`, with five elements, one of which is another vector. Thus, its depth is `2`. Monadic ≡ is the Depth function.

```
      ⍴NestA
5
      ⍴⍴NestA
1
      ≡NestA
2
```

At the level of `NestA`, the fourth element is a scalar. When you descend a level and talk about the array in the fourth position of `NestA`, that array is called an item. The vector `(4 5 6)` is an item of `NestA`.

```
      NestA[4]
 4 5 6
      ⍴NestA[4]

      ⍴⍴NestA[4]
0
```

The Disclose function (⊃) retrieves a level of nesting from an array.

```
      ⊃NestA[4]
4 5 6
      ⍴⊃NestA[4]
3
      ⍴⍴⊃NestA[4]
1
```

You can perform scalar arithmetic functions on a nested vector. Note that the fourth element of the right argument is applied with all three elements of the nested vector that is the fourth item of `NestA`.

```
      NestA + ⍳5
 2 4 6  8 9 10  12
```

You must be cautious when using some functions with nested arrays. If you search for the value `5` in the example nested array, APL64 tells you that it is not an element of `NestA`.

```
      5 ∊ ⍳7
1
      5 ∊ NestA
0
```

The value `5` is an element of an item of `NestA`. (**Note**: When you use the epsilon symbol twice in succession, the rightmost one is the monadic function Enlist. This function reduces the structure of its argument from a nested array to an ordinary vector. The second use of epsilon is the dyadic Boolean function Member of; it uses the result of Enlist as its right argument and returns a `1` if the left argument is an item of the right argument.)

```
      5 ∊ ∊NestA
1
```

There are a number of structural functions that allow you to work with nested items in arrays and to change the levels of nesting in an array. You can get the same structure as `NestA` by creating a vector and assigning that vector by name to another variable:

```
      SUB4 ← 4 5 6
      NestB ← 1 2 3 SUB4 7
      NestB ≡ NestA
1
```

However, not all structures are identical:

```
      NestC ← (⍳3),SUB4,7
      NestC
1 2 3 4 5 6 7
      NestD ← (⍳3) SUB4 7
      NestD
 1 2 3  4 5 6  7
```

Neither `NestC` nor `NestD` is the same as `NestB`. `NestC` is a simple seven-element vector that results from using the Catenate (**,**) function, whereas `NestD` is a three-element vector containing two nested vectors of three elements each and one scalar.

Note that if you put parentheses around a scalar value, it remains a scalar.

```
      NestE ← (1) 2 (3) 4 (5) 6 (7)
1 2 3 4 5 6 7
      ⍴NestE
7
```

Working with Heterogeneous Arrays

A simple array, as mentioned above, is not nested. It contains either one number or one character in each position. If a simple array contains only numbers or only characters, it is homogeneous. If a simple array contains both numbers and characters, it is heterogeneous. Here are three examples of simple arrays:

Simple Arrays

NA **(Numeric)**	CA **(Character)**	HA **(Heterogeneous)**
1 2 3	ABC	1 B 3
4 5 6	DEF	D 5 F
7 8 9	GHI	7 H 9

You can perform arithmetic calculations on the numeric elements of a heterogeneous array. For example, using the heterogeneous array from the table above:

```
      HA[1 3;1 3]×2
  2  6
 14 18
```

A simple character array can contain numerals. You cannot perform arithmetic calculations on numerals.

```
      CHA←'1' 'B' '3' 'D' '5' 'F'
      2 3⍴CHA
1B3
D5F
```

Here are samples of nested vectors. The table shows the vectors reshaped into nested arrays.

```
      NNA← 1 (2 3) 4 (5 6) 7 (8 9)
      NCA← 'A' 'BC' 'D' 'EF' 'G' 'HI'
      NMA← 1 'BC' 4 'E' (6 7) 'H'
```

Nested Arrays

2 3⍴NNA **Numeric**	2 3⍴NCA **Character**	2 3⍴NMA **Mixed**
`   1   2 3     4`	` A BC  D`	`1   BC 4`
` 5 6     7   8 9`	`EF  G HI`	`D  6 7 H`

Internally, nested arrays contain a pointer for each nested item. The pointer identifies where the system stores the nested values. If a nested array contains some simple items and some nested items, it is heterogeneous, even if all the items have the same datatype. If an array contains all nested items, it consists entirely of pointers and is homogeneous.

Understanding Data in APL64

APL64 has a small number of data types. Basically, all data are either character or numeric.

Working with Numeric Data

APL64 uses three internal storage representations of numeric data. Boolean data consist of only 1s and 0s. Each simple element of a Boolean array requires one bit. Integers require 4 bytes per element. Floating point numbers require 8 bytes per element. In general, the system manages representation of the data with no prompting from a user. (You may need to concern yourself with this topic if you store very large databases or communicate with non-APL64 programs.)

APL64 typically displays numbers with up to 10 significant digits. It calculates and stores numbers with 15 significant digits. The system, by default, displays numbers greater than 10^{10} in exponential notation. You can also specify smaller numbers in exponential notation. In traditional mathematical notation, the last number in the example below would be 1.8446744×10^{19}.

```
      1÷2
0.5
      .095÷12
0.007916666667
```

```
      2⋆32
4294967296
      2⋆64
1.844674407E19
```

APL designates negative numbers with a special symbol called a high minus (¯). This symbol is different from the symbol that denotes subtraction (−).

```
      5.4 - 7.4
¯2
      5.4 - ¯2
7.4
```

Working with Hexadecimal and Floating Point Data

A hexadecimal integer is prefixed by 0x or 0X followed by 1-16 hex digits (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F). For example, 0x00100000 is equal to 1048576. Hexadecimal numbers with 1-8 digits are always integers in APL64. They do not overflow into floating point numbers when the high order bit is set. For example, the hexadecimal number 0xFFFFFFFF is ¯1 rather than 4294967295. A hexadecimal float can be prefixed by 0x or 0X followed by 9-16 hex digits or by a 0r or 0R prefix followed by 1-16 hex digits. Hexadecimal integers can also be specified in binary notation with a 0b or 0B prefix followed by 1-64 binary digits (0 or 1). For example, 0b1111 is 15. In the case of a 0r or 0R prefix, the following 1-16 hex digits are the hexadecimal representation of the bit pattern for an IEEE 64-bit float. For example, 0r3FF0000000000000 = 1.0. If less than 16 digits are specified, the value is padded with trailing zeros to 16 hex digits. Thus 0r3FF is the same as 0r3FF0000000000000.
A relatively easy to read summary of IEEE-754 floating point specification can be found here: http://en.wikipedia.org/wiki/IEEE_754-1985. The formal IEEE-specification can be found here: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4610935&filter%3DAND%28p_Publication_Number%3A4610933%29
Here are some other examples:

Hexadecimal Integer	Value
0x00	0
0x01	1
0xFF	255
0xFFFFFFFF	¯1
0x0FFFFFFFF	4294967295
0b1100	12
0b11111111	255
0b10101010	170
0r3FF0000000000000	1.0
0r4000000000000000	2.0
0rC000000000000000	¯2.0
0r4010000000000000	4.0
0r3FE0000000000000	0.5
0r4	2.0 (Note that 0r4 = 0r4000000000000000)
0rC	¯2.0 (Note that 0rC = 0rC000000000000000)

Be careful to avoid specifying bit patterns using "R" notation that represent infinities or NaNs. APL64 does not expect to encounter these and can exhibit unexpected behaviors. For example, you can cause APL64 to hang by trying to display the value of `0r7FF` (positive infinity). This is the reason that `0R` notation is not enabled by default. So be especially careful to avoid generating them.

Example:

```
      (↑645 ⎕dr 82 ⎕dr 110619 ¯2147483648)
```

Could be more directly entered in hex like this (noting the 0x prefix):

```
      (↑645 ⎕dr 0x0001B01B 0x80000000)
```

Or even more directly as a floating point 64-bit hex like this (noting the `0r` prefix):

```
      0r800000000001B01B

      ⎕VI '0xFF'
1
      ⎕FI '0xFF'
255
      ⎕DR '0xA'
82
      ⎕DR 0xA
323
```

Performance Improvement for Floating Point Arithmetic Operations: + - × ÷

SSE2 Architecture

SSE2 is a computer processor architecture and associated instruction set which provides for a single instruction to process multiple data items. Its application to APL64 array operations has beneficial effects on performance.

Demo Workspace in APL64

The DEMO_SSE2.ws64 workspace, in the EXAMPLES sub-folder, is included in this release to test this feature.

Performance Benefits of SSE2 Architecture in APL64

Performance benefits of SSE2 in APL64 depend on the operations being performed, the operand data types, the number of data elements in the operation and the workstation environment. Experimentation using an actual APL64 application system is recommended to determine the magnitude of performance improvements with SSE2 in APL64. Here is a typical result set:

% Improved All Array Sizes by Operand Data Type

Operands	+	-	×	÷
Float,float	34.72	41.46	26.56	42.65
fScal,float	47.44	55.53	37.28	44.99
Float,fScal	46.06	56.34	39.08	45.73
Float,int	32.18	40.85	42.48	19.02
fScal,int	30.65	39.62	46.58	24.10
Float,iScal	46.68	35.26	52.96	15.84
Int,float	37.06	27.07	36.83	13.72
iScal,float	42.04	37.15	49.25	24.57
Int,fScal	32.96	42.07	36.62	22.30

Overall %Improved All Operations & All Array Sizes: 33.93%

Notes:

1. float, int indicate floating point array and integer array respectively
2. Array sizes were varied in the range [10, 100000]
3. fScal, iScal indicate floating point scalar and integer scalar respectively
4. Workstation: Windows 10 Pro, dual core 2.30Mhz

SSE2 Architecture Disabled by Default

SSE2 floating point arithmetic operations, + - × ÷, can yield results that are very slightly different than traditional APL64 result values. In most cases results will be identical. But in some cases very slight rounding differences can cause the low-order bits of the result value to be different. These possible result differences are limited to the last digit of precision in APL64 and may not be apparent unless ⎕ct is set sufficiently small or ⎕pp is set sufficient high. However, you should be aware that result values may not be exactly the same with and without using the SSE2 optimization, even though you are unlikely to notice this difference with default ⎕ct and ⎕pp settings.

Working with Character Data

APL also processes character data in arrays; character data require 1 byte per element. APL64 recognizes 256 characters. This set is known as the atomic vector and is stored in the system constant ⎕av. You specify character data by delimiting the desired string with the single quote (') or the double quote ("). Each character within the delimited string is a separate element.

```
    ρ'ABCDEF'
6
```

If you have numerals in a character string, each one is a separate element.

```
    ρ'123def'
6
```

Numerals are symbols and not values. You cannot perform arithmetic calculations on symbols. You cannot add '9'.

```
    '9'
9
    5+'9'
DOMAIN ERROR
    5+'9'
     ^
```

If you want to include the single quote as a character in a string, enter two single quotes in succession or enclose the string containing a single quote in double quotes.

```
    ρ'JOE''S'
5
    CHAR" 'JOE''S'
    CHAR
JOE'S
    "JOE'S"
JOE'S
```

You can use many of the functions on character arrays that you use on numeric arrays.

```
    2 3ρ'ABCDEF'
```

ABC

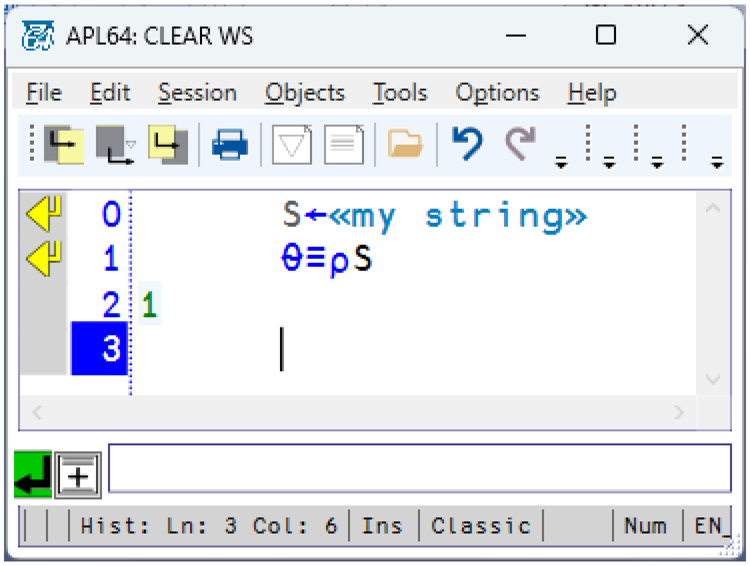
DEF

Note that a numeric array with three columns displays with spaces between the numbers, but character data do not display with spaces unless the space is a character in the array. If you have a character array that looks like
 A B C
 D E F

then its shape is probably 2 6. (It could be larger if there are trailing blanks along either dimension.) A quoted string with a single element is a scalar; a longer string is a vector.

Working with String Data

An APL string is an ordered group of characters treated as a scalar, with the chevron string delimiters, «…». The chevron string delimiters are available on physical and virtual keyboards as Shift + Alt + < for « and Shift + Alt + > for ».

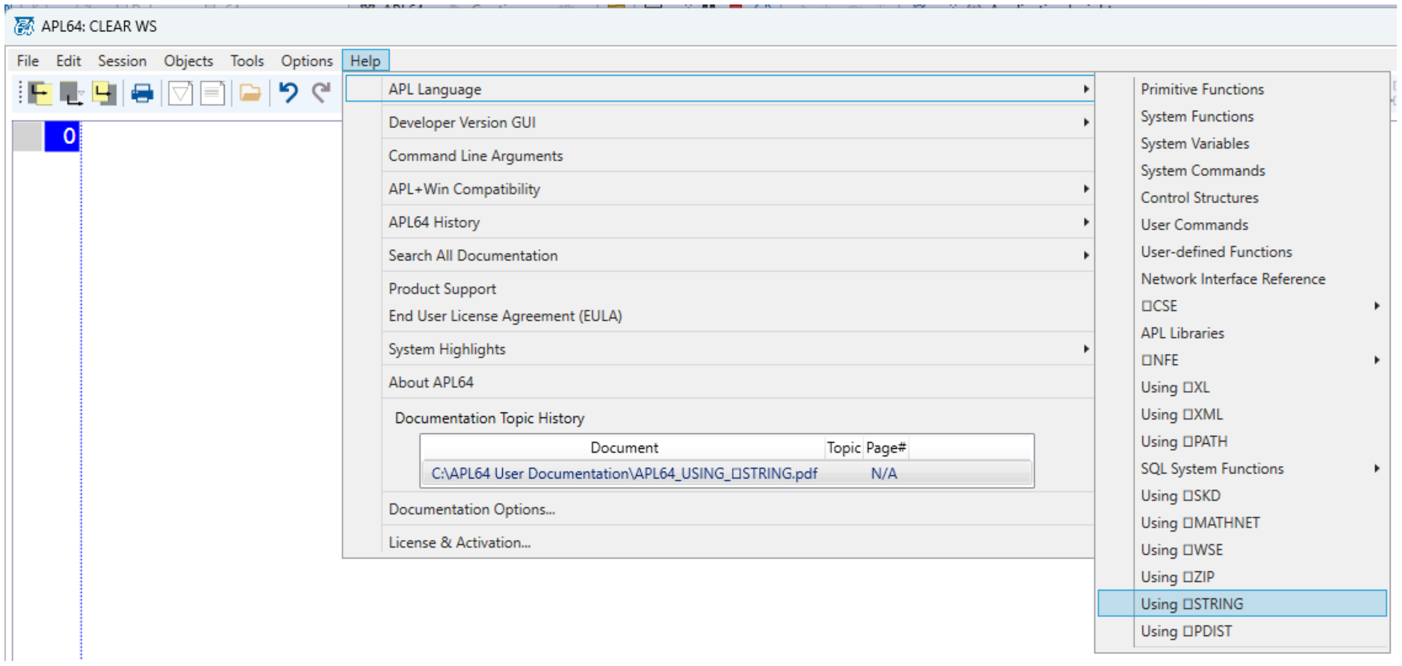

APL64: CLEAR WS
File Edit Session Objects Tools Options Help

```
      S←«my string»
      θ≡⍴S
1
```

Hist: Ln: 3 Col: 6 Ins Classic Num EN

⎕string Actions

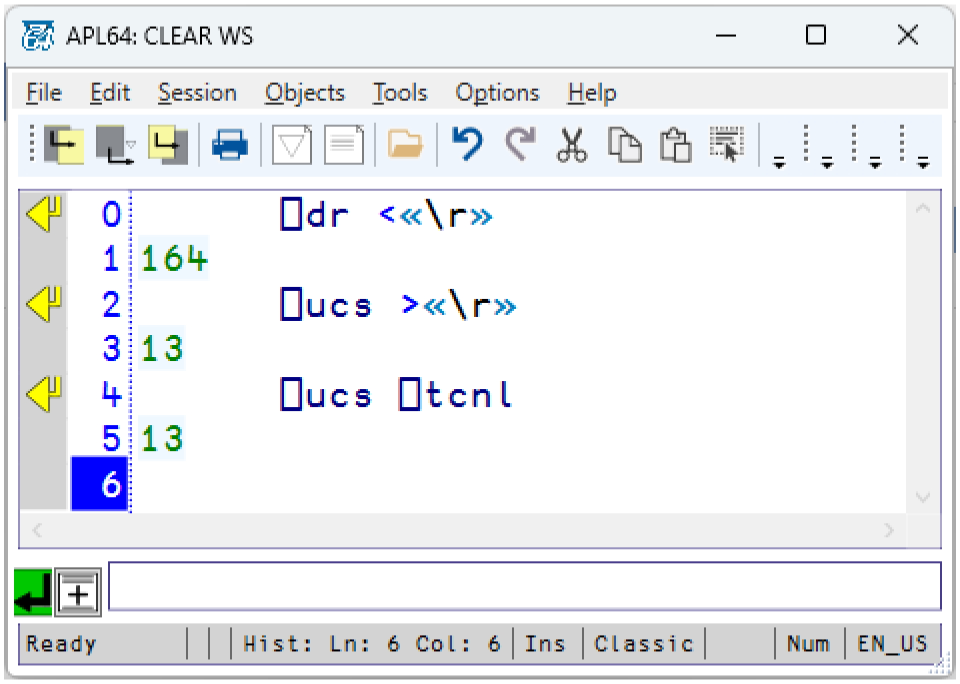
The APL64 ⎕string system function supports many convenient actions for string data, including comparison, concatenation, indexing, joining, padding, removing, replacing, splitting and trimming. To learn more, go here:



Document	Topic	Page#
C:\APL64 User Documentation\APL64_USING_⎕STRING.pdf		N/A

Multi-Line Strings

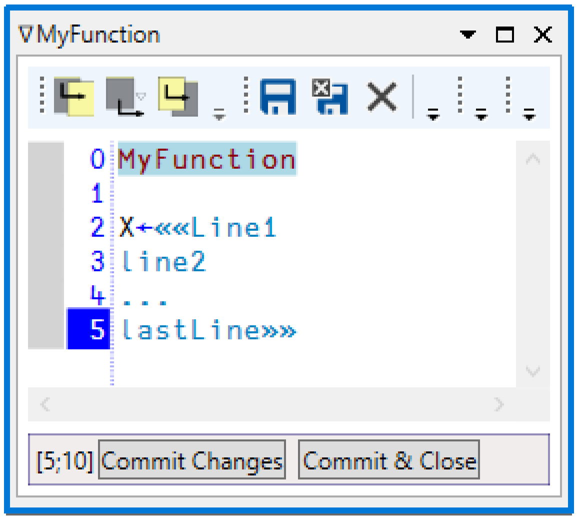
APL64 supports multi-line strings. When a multi-line string is created in APL64 each subsequent line in the string is separated from the previous line in the string by an APL new line character (⎕tcnl)



```
      ⎕dr <«\r»
164
      ⎕ucs >«\r»
13
      ⎕ucs ⎕tcnl
13
```

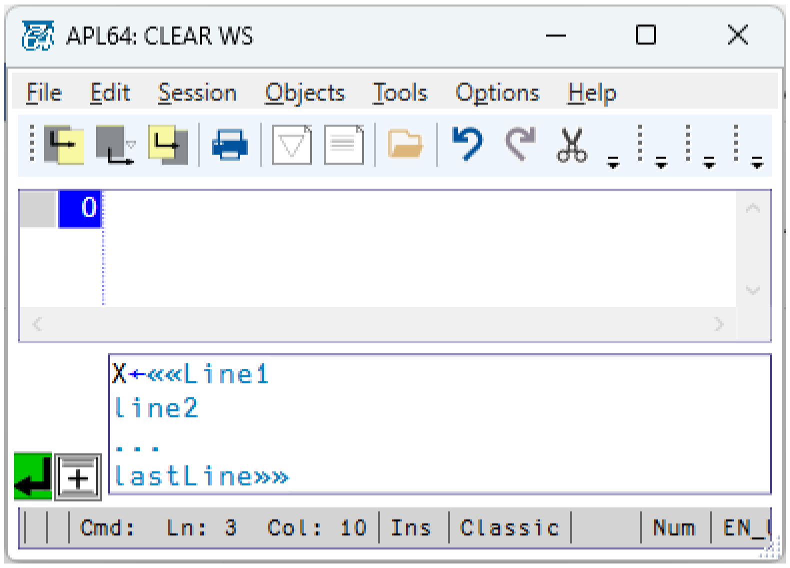
To enter a single line string in the function editor, editable classic history pane, or command line, use syntax like X←«abcd...» or X*←««bcd...»».

To enter a multi-line string in the function editor, use syntax like:



```
MyFunction

X←««Line1
line2
...
lastLine»»
```

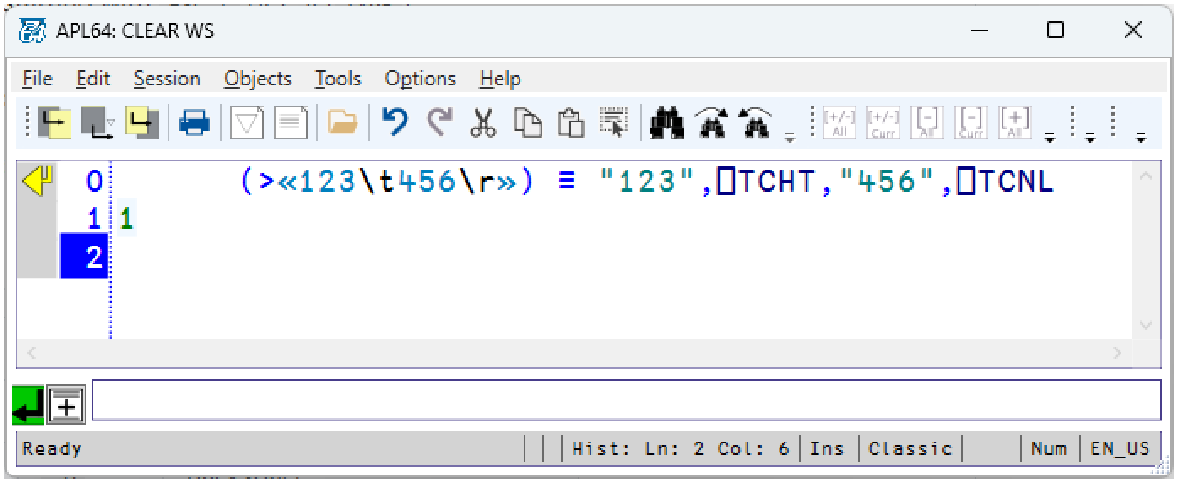
To enter a multi-line string in the command line use syntax like:


APL64: CLEAR WS
File Edit Session Objects Tools Options Help
Cmd: Ln: 3 Col: 10 Ins Classic Num EN_

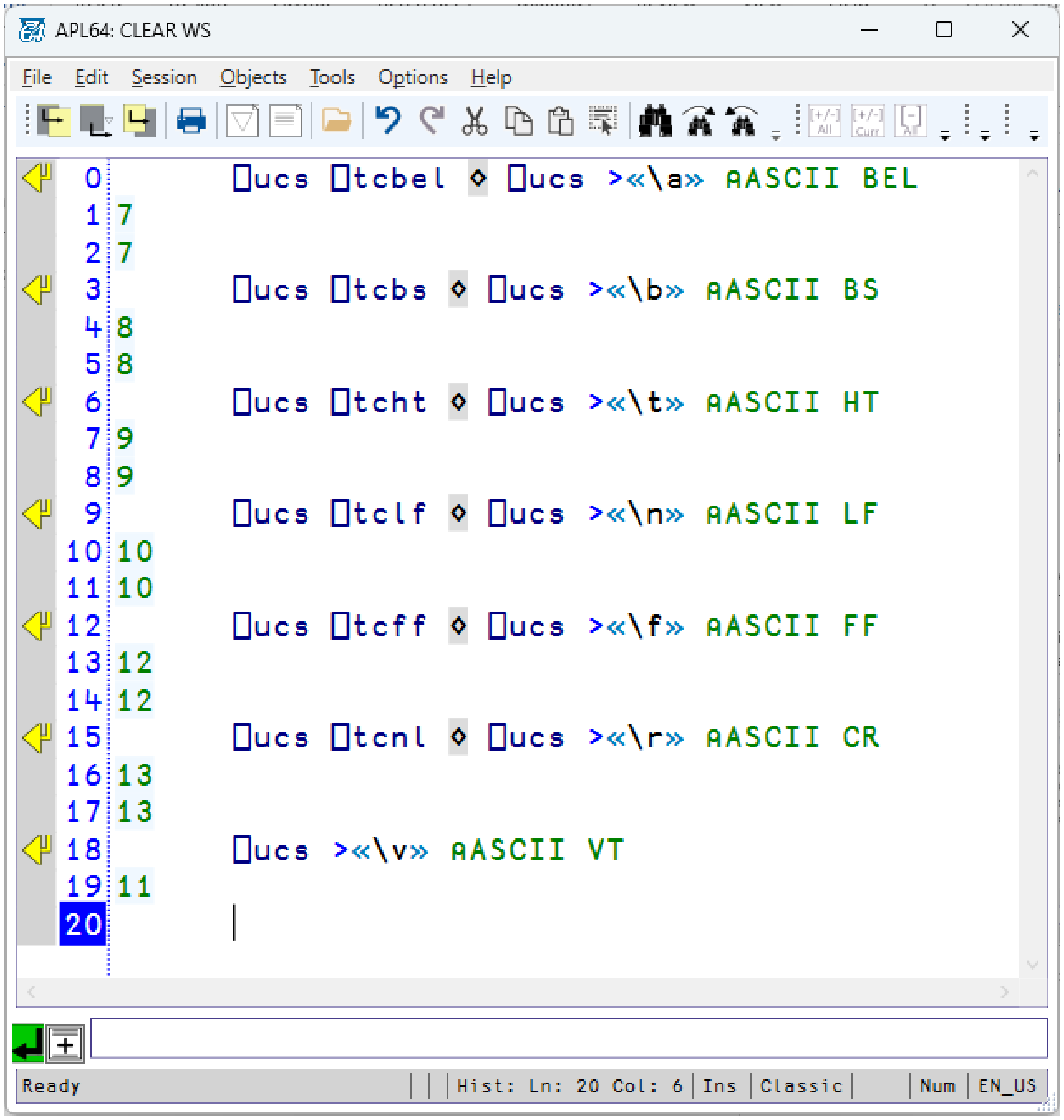
```
X←««Line1
line2
...
lastLine»»
```

Handling Special Characters in a String Variable

- ASCII control characters in strings can be ‘escaped’. Escaped ASCII control characters in a string can be more convenient than concatenating them in a character vector.

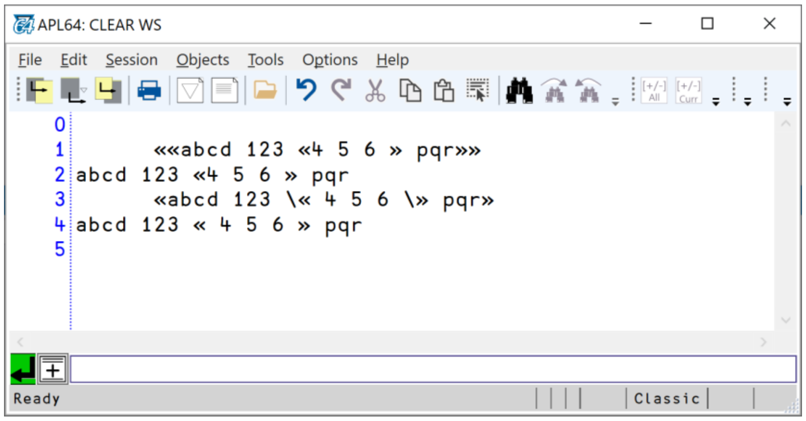


```
      (>«123\t456\r») ≡ "123",⎕TCHT,"456",⎕TCNL
1
```



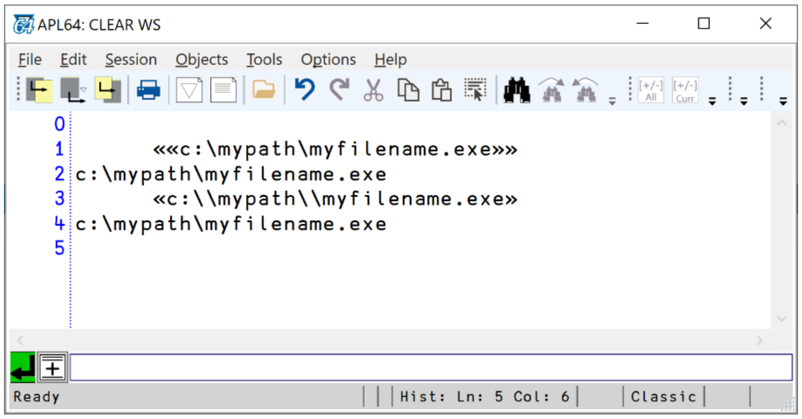
```
      ⎕ucs ⎕tcbel ⋄ ⎕ucs >«\a» ⍝ASCII BEL
7
7
      ⎕ucs ⎕tcbs ⋄ ⎕ucs >«\b» ⍝ASCII BS
8
8
      ⎕ucs ⎕tcht ⋄ ⎕ucs >«\t» ⍝ASCII HT
9
9
      ⎕ucs ⎕tclf ⋄ ⎕ucs >«\n» ⍝ASCII LF
10
10
      ⎕ucs ⎕tcff ⋄ ⎕ucs >«\f» ⍝ASCII FF
12
12
      ⎕ucs ⎕tcnl ⋄ ⎕ucs >«\r» ⍝ASCII CR
13
13
      ⎕ucs >«\v» ⍝ASCII VT
11
```

- Chevrons may be included in a string by using either double enclosing chevrons or escaping the included chevrons.



```
      ««abcd 123 «4 5 6 » pqr»»
abcd 123 «4 5 6 » pqr
      «abcd 123 \« 4 5 6 \» pqr»
abcd 123 « 4 5 6 » pqr
```

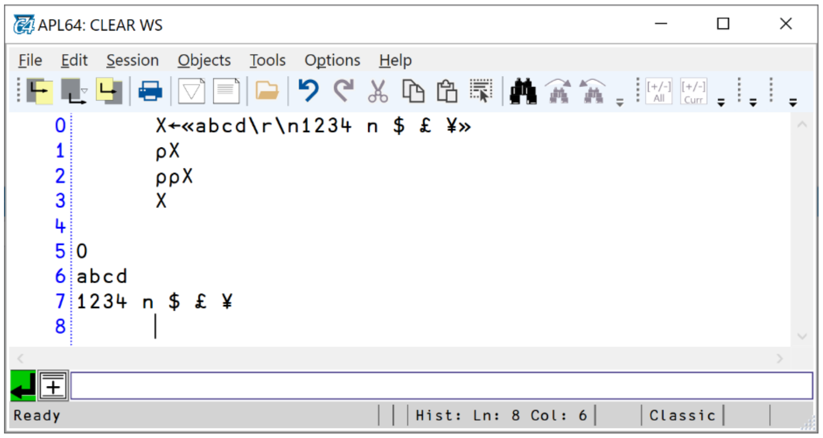
- Paths which contain backslash characters can be done without escaping the backslash if double enclosing chevrons are used.



```
      ««c:\mypath\myfilename.exe»»
c:\mypath\myfilename.exe
      «c:\\mypath\\myfilename.exe»
c:\mypath\myfilename.exe
```

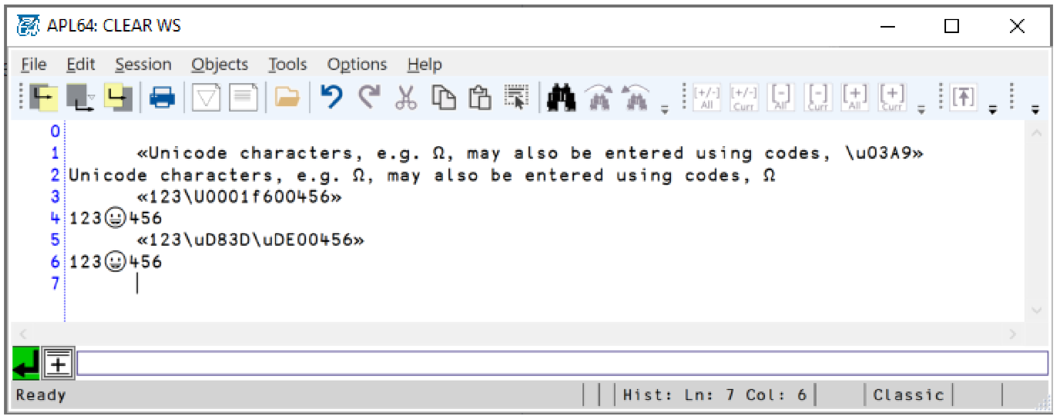
String Variables support Unicode Character Elements

- Unicode characters may be entered and are visible in the APL64 session, functions and arrays.



```
      X←«abcd\r\n1234 n $ £ ¥»
      ⍴X
      ⍴⍴X
      X

0
abcd
1234 n $ £ ¥
```

- Unicode characters may be entered using their hex codes via the **\u**HHHH for UTF-16 escape sequence, **\U**00HHHHHH for UTF-32 escape sequence or **\x**H[H][H][H] for a variable length UTF-16 escape sequence or 2-hex digit codes.

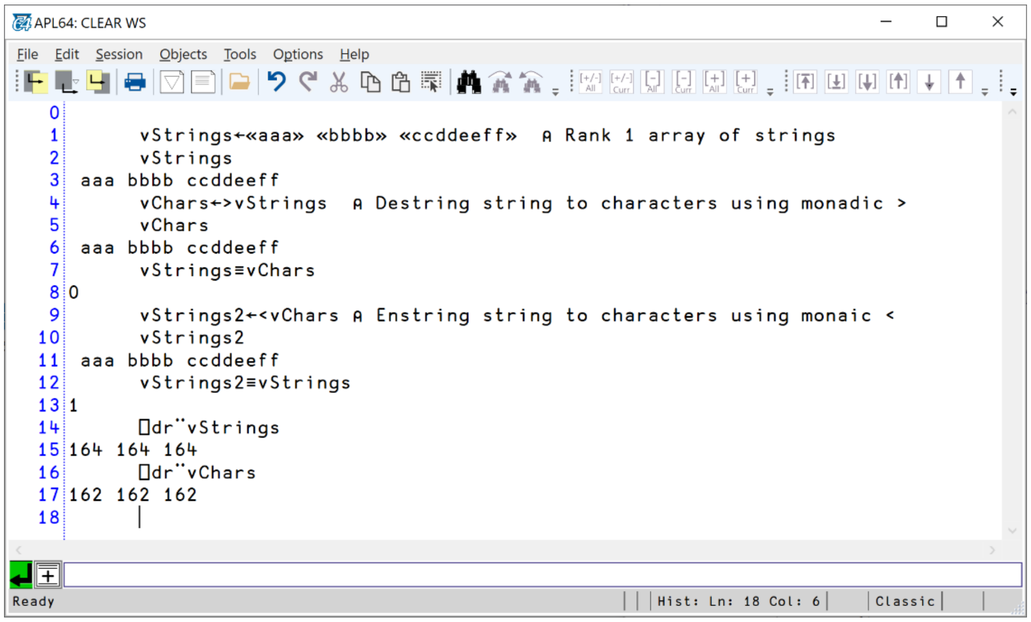


```
      «Unicode characters, e.g. Ω, may also be entered using codes, \u03A9»
Unicode characters, e.g. Ω, may also be entered using codes, Ω
      «123\U0001f600456»
123😀456
      «123\uD83D\uDE00456»
123😀456
```

Note[1]: When using the \x escape sequence and specifying less than 4 digits, if the characters that immediately follow the escape sequence are valid hext digits (i.e. 0-9, A-F , and a-f), they will be interpreted as being part of the escape sequence. For example, \xA1 produces "¡", which is code point U+00A1. However, if the next character is "A" or "a", then the escape sequence will instead be interpreted as being \xA1A and produce "ਚ", which is code point U+0A1A. In such cases, specifying all 4 hext digits (e.g. \x00A1) will prevent any possible misinterpretation.

Conversion between String and Character Variables

Conversion between string and character objects via the DeString, monadic >, and EnString, monadic <, operators:



```
      vStrings←«aaa» «bbbb» «ccddeeff»  ⍝ Rank 1 array of strings
      vStrings
 aaa bbbb ccddeeff
      vChars←>vStrings  ⍝ Destring string to characters using monadic >
      vChars
 aaa bbbb ccddeeff
      vStrings≡vChars
0
      vStrings2←<vChars ⍝ Enstring string to characters using monaic <
      vStrings2
 aaa bbbb ccddeeff
      vStrings2≡vStrings
1
      ⎕dr¨vStrings
164 164 164
      ⎕dr¨vChars
162 162 162
```

[1] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/

Assigning Variables

As shown above, you can specify values for APL64 to process, or you can assign values to a variable. APL64 processes the variable exactly the same as if you had specified the values. Variable names can consist of any combination of uppercase and lowercase letters, the numerals 0 through 9, and three special characters, delta (∆), delta underscore (⍙), and underscore (_), except that the first character cannot be a numeral.

APL64 uses the left arrow (←) to assign values to a variable, where other languages use an equal sign. The left arrow is unambiguous and eliminates the need for possibly confusing expressions such as Y=Y+1. You automatically create a variable the first time you assign values to it.

```
      VarA ← 11-⍳11
      VarA
10 9 8 7 6 5 4 3 2 1 0
```

When you use a variable in a calculation, its value does not change unless you assign a new value to it. You can change the value of a variable by assigning new values to it, by assigning new values to particular elements specified by their positional index, or by selective assignment.

When you assign new values to an existing variable, the system erases the old values, and the variable contains the new values. You can also base the assignment on the old values, such as by performing a calculation involving them.

```
      VarA ← VarA+1
```

You can also assign new values by catenating (appending) values to the existing ones.

```
      VarB ← ⍳6
      VarB ← ¯1 0,VarB,7 8
      VarB
¯1 0 1 2 3 4 5 6 7 8
```

The comma symbol (,) is an APL64 primitive function, which is named Catenate when used dyadically; it is not used as punctuation.

Using Indexing and Indexed Assignment

In addition to specifying an array by a variable name, you can specify one or more elements of an array by specifying their positions in the array. You do this by a process known as indexing. An index to a vector is the integer that specifies the position of the element in the string. You use square brackets to identify an index.

```
      VarB ← ⍳6
      VarB[3]
3
```

You can specify more than one position within the brackets. You can also use expressions to calculate the values of the index.

```
      VarB[3 5]
3 5
      VarB[⍳3]
1 2 3
      VarB[6 (3+1)]
6 4
```

When you index into a matrix, you must specify both row and column numbers. You separate the positions with a semicolon. The element in the first row and second column of a matrix is [1;2]. You can specify two elements in one row with [2;3 4] or two elements in one column with [2 3;2]. You can specify the four elements at the intersections of two rows and two columns with [2 3;1 4]. You do not have to specify elements in the same order as they appear in the array.

```
      VarM ← 3 4⍴⍳12
      VarM
1  2  3  4
5  6  7  8
9 10 11 12
```

```
      VarM[2;3]
7
      VarM[3 2;4]
12 8
      VarM[3 2;2 4]
10 12
 6  8
```

You can specify an entire row or column of a matrix by leaving the index for the other dimension blank.

```
      VarM[;3]
3 7 11
```

You can index into multidimensional arrays in an analogous fashion. You must specify each dimension of the array, separated by semicolons. If you do not specify a value for a given dimension (but you must put in the semicolon), you get all the values for that dimension.

You can replace one or more elements of an array by indexing the array and specifying the values to replace the indexed elements. You can replace each of the specified elements with the same value, or you can replace each of the specified elements with a different value. If you want to replace elements with different values, you must specify the same number of values as you specify indexed positions.

```
      VarB ← ⍳6
      VarB[3 4] ← 7 8
      VarB
1 2 7 8 5 6
```

The index expression `[4 2 6;5 3]` results in a matrix of six elements. If you want to replace the values represented by that expression, you must specify a `3` by `2` six-element matrix.

```
      MatA ← 6 5⍴1
      MatA[4 2 6;5 3] ← 3 2⍴120 + ⍳6

      MatA
1 1   1 1   1
1 1 124 1 123
1 1   1 1   1
1 1 122 1 121
1 1   1 1   1
1 1 126 1 125
```

Using Selective Assignment

You can also replace elements of an array by forming an expression that calculates all or a portion of the array. Enclose the expression in parentheses and assign the new values to the elements you calculate.

```
      VarB ← ⍳6
      (2↑VarB) ← 7 8
      VarB
7 8 3 4 5 6
```

The descriptions of primitive functions identify which ones you can use in selective assignment.

Making Multiple Assignments

You can assign multiple variables in one statement by forming an expression of variable names. As with indexed assignment, you can assign the same value to each of the variables or you can assign different values to each variable.

```
      (C D E) ← 4
      C + D + E
12
      (C D E) ←(⍳3) (4×5) 6
      C + D
21 22 23
      C + (D × E)
```

```
121 122 123
```

Using this technique, you can even rearrange data with one statement rather than moving something to temporary storage and then moving it again.

```
      SR←'A' ⋄ JR←'B' ⋄ SOPH←'C' ⋄ FR←'D'
      FR SOPH JR SR
DCBA
      (SR JR SOPH FR)←FR SOPH JR SR
      FR SOPH JR SR
ABCD
```

Using Functions and Operators

Functions in APL64 perform actions according to specific rules, usually acting on arrays. You can classify functions in many ways. One useful distinction is the number of arguments a function requires. If the function requires one argument, it is monadic; if it requires two arguments, it is dyadic. Addition is dyadic; you must have two numbers to perform addition. When a function requires two arguments, you place one argument to the left of the function symbol and the other to the right. Usually, it makes a difference where you place the arguments. Although the expressions A+B and B+A are equivalent, with most functions the order makes a difference; for example, if you use the Power function.

```
      2⋆3
8
      3⋆2
9
```

Monadic functions place the argument to the right of the function designator. Although you would write 5! in traditional mathematical notation, in APL64 you write:

```
      !5
120
```

Functions are said to be ambivalent if you can use the same symbol either monadically or dyadically. However, it is the symbol that is ambivalent rather than the function. In reality, there are two different functions. Sometimes these pairs of functions are very similar in action and result. The dyadic Divide and the monadic Reciprocal both use the same symbol (÷).

```
      35 3÷7 2
5 1.5

      ÷.5 2
2 0.5
```

Monadic Shape and dyadic Reshape use the same symbol (⍴). Shape returns the size of a data array, which is the length, or number of elements, along each of its dimensions. Reshape constructs a new array with the dimensions you specify as the left argument; the elements of the array are copies of the elements of the right argument, repeated cyclically as necessary.

```
      4⍴1 2 3
1 2 3 1
```

Recognizing Scalar Functions

APL64 also classifies functions as scalar or nonscalar (mixed). A scalar function manipulates one data element at a time, or one datum from the left argument with the corresponding datum from the right argument. It returns a result that has the same shape as at least one of the arguments.

When you manipulate arrays that have more than one element with dyadic scalar functions, the two arguments must have the same shape. You can multiply two vectors, for example, only if they are the same length. APL64 multiplies the elements in corresponding positions of the two vectors and places the product in that position in the result vector.

```
      6 5 4 3 × ⍳4
6 10 12 12
```

Similarly, you can multiply two matrices or two arrays of higher dimension, if both arguments have the same shape. Each element of the result is the product of the two values from corresponding points in the multiplicand and multiplier arrays. You can also use a scalar value (one element) as one of the arguments to a scalar function. In this case, the single value of the scalar is used repeatedly as one argument that corresponds to each of the values in the other array. The result has the shape of the nonscalar array.

```
      6 5 4 3 × 4
24 20 16 12
```

Recognizing Nonscalar Functions

Nonscalar functions can also manipulate arrays, but they have a much wider range of behavior. Some nonscalar functions select or rearrange data without recalculating any values or combine two data arrays into a third array. Others calculate values based on the arrangement of the data in the argument. Some nonscalar functions return a property that describes the data array, rather than the values of the elements. See the description of any function in the Primitive Functions and Operators chapter for more information.

Other Function Categories

In the Primitive Functions and Operators chapter, primitive functions are in these categories:

Arithmetic
Functions that require numeric arguments and calculate numeric results.

Boolean
Boolean data consist of only `0`s and `1`s. Boolean functions manipulate arrays and return a Boolean result; for example, Equal (=). Some of these functions require Boolean arguments and return a Boolean result; for example, AND (∧). These Boolean functions are known as logical functions. Some Boolean functions can be used both ways; if you use them with Boolean arguments, you can use them as logical functions. For example, the Not equal (≠) function used with two Boolean arrays as arguments is the equivalent of the logical Exclusive or (XOR) function.

Structural
Structural functions select and rearrange some or all of the data in an array. These functions do not calculate new values, but some of them create additional instances of values.

General
These functions do not fall in any of the above categories. They are in three groups:

- functions that return a property of the argument
- functions that return an index to the elements of the argument
- functions that either return only a character result or require a character argument.

Note: There are also categories of functions other than primitive. System functions are built-in functions designated by the quad symbol (⎕) and a name. Some manipulate system entities such as files or workspaces; others are utilities that extend the language. You can also write your own functions (user-defined functions), which are called programs in other languages.

In addition to functions, there are sets of instructions, called commands, you can use. System commands are instructions to the APL64 System rather than the facilities of the APL language interpreter. System commands begin with a right parenthesis "`)`". User commands also instruct the system; these begin with a right bracket "`]`". This system provides some user commands that are always available from the user command processor. You can also write your own user commands.

Each of these categories of instructions is explained later in this manual or in the *System Functions Manual*.

Operators are another category of primitives. An operator takes one or two functions or arrays as its operands and derives a new function that acts on its argument or arguments. In order to explain the use of operators, it is necessary to understand nested arrays and how multiple functions can act in the same expression.

Writing Expressions

You can combine elements of APL into more complex expressions to perform certain actions. The position of the elements determines the order in which the expression is evaluated. APL uses parentheses, brackets, and other symbols to define or modify the relationships of the elements of an expression.

One of the most important concepts of the syntax of APL is that APL expressions evaluate from right to left. In certain circumstances, this may make no difference.

```
      9 + 5 − 3
11
```

The expression above evaluates to the same value whether you subtract `3` from `5` first and add the result to `9`, or if you add `9` and `5` and subtract `3` from the result. However, this is the exception rather than the rule. The expression

```
      9 − 5 + 3
1
```

can surprise newcomers to APL. If you want to write this expression so that it evaluates to `7` in APL, you must change the order of execution:

```
      (9 − 5) + 3
7
```

Spaces make no difference with APL numeric data and functions, so that `(9−5)+3` is the same expression as in the example above. Similarly, `⍳4` is the same as `⍳ 4`. It may be tempting to think of `⍳4` as an entity, but in an expression, these are two items, a function and its argument, not one.

Given the right to left evaluation, there are implicit parentheses around the rightmost function and its arguments. The expression `⍳4+1` is the same as writing `⍳(4+1)`.

```
      ⍳4 + 1
1 2 3 4 5
```

However, changing the order of the functions changes the result.

```
      1+⍳4
2 3 4 5
```

You must also remember that a vector can be the left argument to a function. In the expression `2 3⍴10 20−1`, the implicit parentheses include the `10` as part of the left argument to the Minus function. Thus, `(10 20−1)` evaluates to a vector of two elements `(9 19)`, which becomes the right argument to the Reshape function.

```
      2 3⍴10 20−1
 9 19  9
19  9 19
```

It is perfectly acceptable to use parentheses in your APL code, even when the APL64 system does not need them. Additional parentheses do not slow down the code. Do not hesitate to add parentheses if they make it easier for you to see how the system evaluates a statement; for example:

```
      (2 3)⍴(10 20)−1
 9 19  9
19  9 19
```

Using Operators

Operators are a special category of APL primitives that derive functions. In APL expressions, functions act on arrays and create arrays as a result. Operators, in contrast, act on functions or arrays and create functions as a result. These created ("derived") functions then act on arrays and create arrays as a result.

Some operators are monadic and some are dyadic, but no operator symbol is ambivalent. A monadic operator can take a function, and sometimes an array, as an operand, and the result is a derived function that acts on one or two arrays. Operators can change the syntax for a function. For some operators, the operand function is dyadic, but the derived function requires only one argument. This is not a contradiction in the use of the term dyadic; the derived function uses different elements of the single argument as its left and right arguments in repeated applications.

In contrast to a monadic function, whose single argument is to the right of the function, a monadic operator has its operand to the left. When a monadic operator has an operand array to the left and an argument array (for the derived function) to the right, it is difficult to distinguish the difference between the monadic operator deriving a monadic function and a dyadic function. The difference is apparent only when these constructions are used in conjunction with another operator.

Dyadic operators have operand functions both on the left and right. If the derived function takes only one argument, it is to the right of the right operand. If the derived function takes two arguments, the left argument precedes the left operand. Although it is theoretically possible for a dyadic operator to have arrays as its operands, there are no such operators in APL64.

Operators greatly expand the power of APL and allow complex manipulation of data with very concise expressions. A brief description of each of the operators follows. See the Primitive Functions and Operators chapter for more information.

Working with Reduction and Scan

Although the operator symbols are not ambivalent, some do have multiple uses. One symbol can derive different functions depending on whether its operand is a function or an array. The Reduction and Scan operators use the slash and backslash symbols, respectively, with a function as the operand. The operand function is dyadic, but the derived function is monadic. The syntax for the Reduction and Scan operators is *fn opr arg*. There are two versions of each operator, with and without a bar across the slash, with the default applying along the first or last dimension. You can use an optional index to specify the dimension with either symbol.

Reduction, as its name implies, reduces the rank of the argument by one. The operator (conceptually) places the function between each pair of elements in the argument along the specified dimension. To find the sum of a vector:

```
      V←⍳5
      +/V
15
```

The effect of the reduction operator (`/`) is that `+/V` evaluates as `(1+2+3+4+5)` reducing the vector argument to a scalar. Note that the order of evaluation is still right to left.

If you do a minus reduction, the result is not so obvious.

```
      -/V
3
```

If you could interrupt the calculation `(1-2-3-4-5)` halfway through, you would see the following intermediate result:

```
      IR ← 3 - (4-5)
      IR
4
```

Completing the calculation is equivalent to:

```
      1 - (2-IR)
3
```

When you apply a function that the Reduction operator derives to an array of higher rank, the default dimension comes into play. In the first example below, the first element of the result (the reduced array) is the sum of four elements `(1+2+3+4)`. These four elements are the first plane, first row, each column, since columns are the last (default) dimension.

When the Reduction operator reduces along the last dimension in this example, it reduces a three-dimensional array that has a shape of `(2 3 4)` to a two-dimensional array that has a shape of `(2 3)`. The second element in the first row of the result is the sum of the second row of the first plane of the original array `(5+6+7+8)`, and so on until the sum of the second plane, third row of the argument `(21+22+23+24)` becomes the element in the second row, third column of the result.

```
      M←2 3 4⍴⍳24
      M
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24

      +/M
10 26 42
58 74 90
```

If you reduce along the first dimension (planes), the result is a `3`-by-`4` matrix that represents the sums of `(1+13)` through `(12+24)`.

```
      +⌿M
14 16 18 20
22 24 26 28
30 32 34 36
```

If you reduce along the second dimension (rows), as the bracketed integer specifies, the result is a `2`-by-`4` matrix that represents the sums of trios of values, `(1+5+9)` through `(16+20+24)`.

```
      +/[2]M
15 18 21 24
51 54 57 60
```

The Scan operator (\) operates similarly, but the derived function does not reduce the rank of the result. It provides an array of the same size as the argument, showing results for successively larger arrays. The progression of included elements is from left to right, but the calculations for each value are still evaluated right to left.

```
      -\V
1 ¯1 2 ¯2 3
```

The first value is the first element of the argument (by definition). The second value represents the result of applying the function to the first two elements of the argument (`1-2`), and the third value represents the result of applying the function to the first three elements of the argument `(1-(2-3))` and so on.

The example above shows a vector as the argument. The same principle applies to arrays of greater rank, with the derived function applied across the default or designated dimension of the argument. Note that reduction is an integral part of both the Scan and Inner Product operators.

Working with Replicate and Expand

The Replicate and Expand operators also use the slash and backslash symbols, respectively, with an array as the operand. The derived function is monadic and structural; that is, it selects and rearranges some or all of the data in one argument array and may create additional instances of values. The syntax for the Replicate and Expand operators is *array opr arg*. There are two versions of each operator, with and without a bar across the slash, with the default applying along the first or last dimension. You can also use an index to specify the dimension with either symbol.

Expand, as its name implies, increases the size of an array along one dimension. The operand array must be a Boolean vector; that is, it is limited to ones and zeroes, and there must be as many ones as there are elements along the key dimension. The result array consists of items of the argument array corresponding to ones in the operand, and fill elements corresponding to zeroes in the operand array. Fill elements are blanks for a character array or zeroes for a numeric array.

```
      0 1 0 1 0 1 0 1 \ 6 7 8 9
0 6 0 7 0 8 0 9
      1 0 0 0 1 0 0 1 0 1 ⍀ 'abcd'
a   b  c d
```

As with the previously described operators, you can specify a dimension explicitly. If you want to add a row or a column to a matrix, you can do the following:

```
       1 0 1 \[1] 2 3⍴'abcdef'
```

```
abc

def
      1 0 1 \[2] 3 2 ⍴⍳6
 1 0 2
 3 0 4
 5 0 6
```

You can use the Replicate operator with a Boolean argument for the inverse purpose; thus, it is sometimes called Compress. With Replicate, the length of the operand vector must equal the shape of the argument array along the chosen dimension. The result array contains only those items from the argument array that correspond to ones, but excludes the items the correspond to zeroes. Thus, the result is usually smaller.

```
      5 ? 100
87 25 54 46 75
      87 25 54 46 75 > 50
1 0 1 0 1
      1 0 1 0 1 / 87 25 54 46 75
87 54 75

      a← 5?100
      (a > 50) / a
96 99 60
```

If you use values greater than one in the operand, the operator, as its name implies, generates multiple copies of the appropriate elements in the result array.

```
             1 2 3 / 96 99 60
96 99 99 60 60 60
```

And if you use negative numbers in the operand vector, the operator combines the functionality of Expand and Replicate. Note that the high minus that designates a negative number is not the same as a minus sign.

```
      ¯2 1 0 ¯1 2 ⌿ 3 3 ⍴⍳9
 0 0 0
 0 0 0
 1 2 3
 0 0 0
 7 8 9
 7 8 9
```

Working with Inner and Outer Product

The Inner Product operator, whose symbol is the period (**.**) requires two dyadic functions as operands and derives a dyadic function. The length of the last dimension of the left argument must equal the length of the first dimension of the right argument. The result has a shape that combines the shapes of the arguments without the common dimension. If Inner Product processes arrays with shapes (`4 3 2`) and (`2 5 6`), the result is a `4`-dimensional array of shape (`4 3 5 6`).

The syntax for Inner Product is *larg lfn* **.** *rfn rarg*. Inner Product processes the elements of the left argument against the elements of the right argument using the right function operand, and then reduces (the Reduction operator) the result along the common dimension using the left function operand.
VectorA ×**.**+ VectorB means add the two vectors and multiply all the elements of the result to give a scalar answer. The following example is equivalent to (`2×4×6`).

```
      1 2 3×.+1 2 3
48
```

Inner Product is a generalized matrix multiplication. The specific form `MA+.×MB`, where `MA` and `MB` are matrices, is matrix multiplication as it is defined in Linear Algebra.

The Outer Product operator uses two symbols (`∘.`). You can think of the left symbol, the jot, as an operand, a function that defines the product operator as an outer product. (Or you can think of it as a placeholder for the left operand.) This symbol has no other defined purpose in APL64. The right operand must be a dyadic function; outer product derives a dyadic function that uses each of the elements in the left argument with each of the elements in the right argument, thereby increasing the rank of the result. The syntax is *larg* `∘.` *fn rarg*. Two vectors of length *m* and *n* used as arguments create a matrix of shape (*m n*).

```
      10 20∘.×1 2 3
10 20 30
20 40 60
```

Applying Outer Product to two matrices (two dimensions each) results in a four-dimensional array.

Working with Each

The Each operator, denoted by the dieresis (¨) takes either a monadic function or a dyadic function as an operand and derives a function of the same type. It applies the function to each of the elements of the argument in succession, creating a nested vector (see below) as the result. Each is most useful with nonscalar functions. The syntax for Each with a monadic operand function is *fn* ¨ *arg*; when the operand function is dyadic, it requires two arguments: *larg fn* ¨ *rarg*. You can best understand Each by examining the effect of using a function with and without the operator.

```
       2 3 4 , 5 6 7
2 3 4 5 6 7
       2 3 4 ,¨ 5 6 7
 2 5  3 6  4 7

       2 3 4 ⍴ 8 9 10
  8  9 10  8
  9 10  8  9
 10  8  9 10

  8  9 10  8
  9 10  8  9
 10  8  9 10
       2 3 4 ⍴¨ 8 9 10
 8 8  9 9 9  10 10 10 10
```

The Each operator is useful in performing repeating actions without writing loops. For example, you can use the function that reads one element from a file combined with the Each operator to derive a function that reads an entire file with one instruction.

Reading Statements that Contain Operators.

There is a difference in reading adjacent symbols when one is an operator. You can frequently put two functions together in APL. One example this chapter uses is applying Reshape to an array the Index Generator creates (`m⍴⍳n`). Recognizing the right to left execution of statements, the system evaluates the expression (`⍳n`) and then the Reshape function uses the result as its right argument. In execution, the statement looks like (`m⍴(⍳n)`).

However, when there is an operator in the statement, the operator takes precedence. The statement (`m∘.⍴n`) derives a dyadic function and applies it to the two arguments. Conceptually, it looks like (`m(∘.⍴)n`).

Similarly, with a monadic operator, there is a difference in precedence. This example also shows the difference between an operator and a function. The system function `⎕repl` is a dyadic function that replicates its right argument in accordance with its left argument. There is no difference in the result from using the Replicate operator with a vector operand; it is a simple `6`-character vector.

```
      0 1 2 3 ⎕repl 'abcd'
bccddd
      0 1 2 3 / 'abcd'
bccddd
```

However, there is a difference between using the dyadic system function with the Each operator and using the monadic, derived, function in the second expression with the Each operator. The result here has four separate elements of, respectively, zero, one, two, and three characters. The elements of the left argument are applied to the corresponding elements of the right argument as they were without Each. The difference in the result is that the function builds a simple vector, while the operator/function combination builds a vector for each element.

```
      0 1 2 3 ⎕repl¨ 'abcd'
  b cc ddd
      ]display  0 1 2 3 (⎕repl¨) 'abcd'
.→--------------.
|.⊖..→..→-..→--.|
||  ||b||cc||ddd||
|'-''-''--''---'|
'∊--------------'
```

The syntax of the above statement is: `0 1 2 3 (⎕repl¨) 'abcd'` In fact, you could put the parentheses in the expression and it would execute the same way. In contrast, the syntax of using the Replicate operator to derive a monadic function and using that function with Each is: `(0 1 2 3 /)¨ 'abcd'`
In this case, the same derived function is applied to each of the elements of the one argument. This example also demonstrates scalar extension. The function requiring a four-element argument is applied to each scalar; the scalar is extended so it is as if there were four a's, four b's, four c's, and four d's in the argument.

```
      0 1 2 3 /¨ 'abcd'
 aaaaaa bbbbbb cccccc dddddd
      result← (0 1 2 3 /)¨ 'abcd'
      ]display result
.→------------------------------.
|.→-----..→-----..→-----..→-----.|
||aaaaaa||bbbbbb||cccccc||dddddd||
|'------''------''------''------'|
'∊------------------------------'
```

If you tried to execute either expression with the inappropriate parentheses, `(0 1 2 3 ⎕repl)¨ 'abcd'` or `0 1 2 3 (/¨) 'abcd'`, you would get an error.

Using Boolean Functions

Boolean data consist of `1`'s and `0`'s. You can think of these two data as representing Yes/No, True/False, or Exists/Does not Exist. Boolean dyadic functions make a comparison between elements of their arguments and return a Boolean result. If the comparison is true, they return a `1`, otherwise `0`.

```
      1 2 3=3 2 1
0 1 0

      'RASH'='ITCH'
0 0 0 1
```

Boolean functions are powerful tools in selecting and manipulating data. For example, if you have a vector of unknown length representing the amounts of various loans, and you want to analyze all the loans for more than half a million dollars, you can identify those that qualify with one statement:

```
      BoolBig ← LoanAmt > 500000
```

The result is a vector the same length as `LoanAmt` that has `1`'s in each position that corresponds to a loan of sufficient size to qualify. You can count how many loans qualify with `+/BoolBig`. You can also create a vector containing the values for only those that qualify with the statement `BoolBig / LoanAmt`. This expression selects elements from the right argument when the corresponding element in the left argument is `1`. Note that the slash in the first statement is the monadic Reduction operator , while the slash in the second statement is the dyadic construct Replicate (compress). You can get a total of these loans using:

```
      +/ BoolBig / LoanAmt
```

If `LoanAmt` were one column of a matrix, where other columns contained interest rate, loan number and borrower's name, you could use the same technique to extract a matrix of records for only these loans.

Recognizing Logical Functions

Logical functions use Boolean arrays as arguments and return a Boolean result. You can use logical functions to do complex conditional tests on data. For example, extending the example above, if you want to analyze loans over a certain amount where the borrower is divorced, you can use the AND logical function to combine two comparisons.

```
      DivHiRlr ←(LoanAmt>500000) ∧ (MarStat ='D')
```

The result would be a Boolean vector with `1`'s representing only those positions where both values are true: the loan qualifies and the character code of the marital status variable equals the desired value.

Three other dyadic logical functions are OR, NAND and NOR. There is one monadic logical function, NOT (~), that replaces the values in a Boolean array with their complements, changing zeroes to ones and ones to zeroes.

You can also use functions that return a Boolean result as logical functions if you use Boolean arrays as arguments. The six relational functions, Equal (=), Not equal (≠), Greater than (>), Less than (<), Greater than or equal(≥), and Less than or equal (≤) represent meaningful combinations of two conditions.

Using Structural Functions

Structural functions reorganize the data in existing arrays or select subsets of data from arrays, or both. These functions can change data by replacing items in an array or by restructuring their relationships, such as by creating a nested array, but they do not recalculate an individual datum. Some structural functions create new data by replicating instances of existing data or by inserting fill items in an array.

You can append data to an array with Catenate. When you append data to an array, you not only have additional data items, you reorganize the data, because the shape, and perhaps the rank, of the result is different from the array you started with.

You have already seen numerous examples of the Reshape function (dyadic ⍴) that reorganizes data.

```
      2 3⍴ 1 2 3 4
1 2 3
4 1 2
```

Note that Reshape creates additional instances of the data in the vector that forms its argument to complete the array that is its result.

Other functions that create additional instances of data are Replicate (/) and Expand (\); the number of data items depends on the argument you supply. If you replicate a datum zero times, it disappears from the resulting array. Other functions that reorganize data are Ravel (,), Reverse (⊖ and ⌽), Rotate (dyadic ⊖ and ⌽), and Transpose (⍉). The Primitive Functions and Operators chapter describes each of the structural functions.

Some structural functions reorganize data by creating a level of nesting in an array or removing a level of nesting from an array. These functions may also create instances of new data by padding the result with a fill item, which is a blank for character data or a zero for numeric data.

```
      A← (1 2 3) (4 5) (6)
      A
 1 2 3  4 5  6
      ⍴A
3
      ⍴⍴A
1
```

The example above shows a nested vector, named `A`, with three items; its rank is `1`. You can add a three-item vector to a three-item vector.

```
      1 2 3 + A
 2 3 4  6 7  9
```

The Disclose function (⊃) retrieves a level of nesting from a nested array, raising its rank. The result is a three-row by three-column matrix; its rank is `2`.

```
      ⊃A
 1 2 3
```

```
  4 5 0
  6 0 0

      ⍴⊃A
3 3
      ⍴⍴⊃A
2
```

You cannot add a three-element vector to a matrix.

```
      1 2 3 + ⊃A
RANK ERROR
      1 2 3 + ⊃A
            ^
```

The Enclose function (⊂) creates a nested scalar from an array that is not a simple scalar. The result is a one-item nested array; its rank is zero.

```
      ⊂A
  1 2 3  4 5  6
      ⍴⊂A
      ⍴⍴⊂A
0
```

You can add a three-element vector to a scalar.

```
      1 2 3 + ⊂A
  2 3 4  5 6  7   3 4 5  6 7  8   4 5 6  7 8  9
```

Other structural functions select subsets of arrays. You select data by position, such as by taking or dropping a given number of elements from the beginning or end of a vector. You can also select rows or columns from a matrix or matrices from an array of higher dimension. When you select a subset of data, you also change the shape, and perhaps the rank, of the array. The data you do not select do not exist in the result. They still exist in the array from which you selected unless you reassign the result to that variable.

```
      V←⍳12
      3 ↑ V
1 2 3

      SUB←2×3↑V
      SUB
2 4 6

      V
1 2 3 4 5 6 7 8 9 10 11 12
      V←2×3↑V
      V
2 4 6
```

To select a subset of data by their values, you can use a Boolean function that tests for the desired values, and then use a structural function with the result of the Boolean function as one argument and the data array as the other argument. An example of this type of selection appears in the section on Boolean functions in this chapter. You can also select a subset of data by the absence of their values by using the Without (~) function. To get everything except the odd, single-digit numbers, you can use the following expression:

```
      1 2 3 4 5 6 7 8 9 10 11 ~ 1 3 5 7 9
2 4 6 8 10 11
```

Interestingly, you can select a subset of data by their values using the Without function twice. To get all the odd, single digit numbers:

```
      A←1 2 3 4 5 6 7 8 9 10 11 1 2 3
      B←1 3 5 7 9
      A ~ B
2 4 6 8 10 11 2
      A ~ A ~ B
      1 3 5 7 9 1 3
```

Using General Functions

APL64 includes some primitive functions that do not fall into any of the above categories. These general functions, which the Primitive Functions and Operators chapter describes, include:

- those that return a property of the argument, such as Shape (monadic ⍴)
- those that return an index to elements of the argument, which you can use, for example, to sort the data in the argument
- character functions, including functions that convert numeric data to characters and the Execute function, which converts character numerals to numeric data as its simplest application, and, in more general usage, executes any character string the system can execute.

Chapter 4: Writing APL Functions

The APL64 language allows you to create your own functions to perform a task repeatedly, potentially with different input data, without having to retype the expressions that execute your task. Depending on the complexity of the function you write, this can be equivalent in another programming language to writing a standalone subroutine or a program.

You can use your function as you use primitive functions or system functions. You can evaluate it in immediate execution mode, combine it with other APL functions in an expression, or call it from another function. When you define the function to have an explicit result, you can display the result, use the result in an APL expression, or assign it to a variable.

As an example, for a generic APL function, this chapter uses `FOO`. The function is monadic and returns an explicit result; the following statements are valid APL statements:

```
      FOO 6
8
      25 + FOO 7
38
      X" FOO 8
      X
21
      FOO FOO 6
21
```

Basic Concepts

When you create a function, you give it a name, specify whether it returns a result, and define how many arguments it has. You accomplish this in the header statement, which is line number `0` in the function editor. In the session manager, from the Objects menu, select New and Function. The system places you on line zero.

The header is a statement that the system does not execute but uses to get information about the function. The minimum content of the header line is one name. If you have only that one name in the header, you define a niladic function that does not return a result.

If you put two names in the header, the left one is the function name and the right one is its single argument. If you put three names in the header, the center one is the function name and the left and right variables are the arguments.

If you want the function to return a result, place a variable name and the left arrow to the left of the function name and its arguments. Note that a function can have an effect without returning a result. The table below shows how types of functions relate to the content of their header lines. The content of each cell is what appears on line `[0]` (the header line) of the function.

Types of Functions

	Niladic	Monadic	Dyadic
With result	`Z←DOO`	`Z←FOO T`	`Z← U GOO V`
Without result	`HOO`	`KOO W`	`X LOO Y`

Recognizing Functions as Black Boxes

An important concept in APL programming is that the calculations inside a function occur in a protected environment. The system uses any variable that appears in the header of a function only within that function. When it finishes evaluating the function, it erases the variable without affecting anything else in the workspace. If you have a variable with the same name already defined in the workspace, the calculations within the function do not affect it.

The system accomplishes this with a process called localization. If you have variables named Z and T in your workspace, and you call the function that has the header Z←FOO T, the system stores the workspace values of Z and T temporarily. It evaluates the function, using the input value that you specify as the local value of T and stores the result in the local value of Z. As long as you are inside the function, you can perform any valid actions with the local values of Z and T. When you exit the function, the system erases the local variables and restores the workspace values of Z and T.

The function FOO, as defined above, calculates a nonlinear numeric series and returns an integer result based on the value of the argument. Suppose you define the variables Z and T in your workspace and that the first and last lines of FOO are as shown below:

```
      Z←3.14159
      T← 7
```

In the Edit window, the function displays as:

```
[0]  Z←FOO T
...
[n]  END:  Z ← Q1+Q2
```

Then suppose you type the following statement:

```
      FOO 6
8
```

While the system evaluates this function, T has the value 6, and Z ends up with the value 8, within the function. The system displays the last value of Z as the explicit result of the statement. However, when the system exits the function, these local values disappear. In the workspace, Z and T still have the global values.

```
      T
7
      Z
3.14159
```

Note that you can use the variable T, coincidentally, as the argument to FOO.

```
      FOO T
13
      T
7
      T←8
      FOO T
21
      T
8
```

If you assign values to other variables within your function for intermediate results, you can keep them local by specifying them in the header line following the right argument; you separate variables in the header with semicolons. If the header in the example includes Q1, you can use Q1 in the function without affecting the workspace.

```
[0]  Z←FOO T;Q1
...
[m]  (Q1 Q2 I) ← Q2 (Q1+Q2) (I+1)
...
[n]  END:  Z ← Q1+Q2
```

However, when you use a variable in a function that you do not name in the header line, the variable retains its value from the function when the system exits the function. The variable then exists in the workspace with the last value it held within the function. If you define Q1 and Q2 in your workspace prior to calling the function, the prior value of Q2 is lost, just as if you assign a new value to it in immediate execution mode.

```
      Q1←0.06375
      Q2←0.06875
      FOO 9
34
```

```
      Q1
0.06375
      Q2
21
```

It is useful to place the name of any variable whose value you do not explicitly want to save in the workspace in the header line. Even if you do not assign a value to it before you run your function, if you do not localize it in the header, it exists in the workspace afterwards. For example, there is no reason to store the value of `I` that you use as a counter in the function. If you place `I` in the header line, it has no value in the workspace afterwards and thus you reduce clutter. You can display the visual representation of a function using the system command `⎕vr`.

```
      ⎕vr 'FOO'
   ∇ Z←FOO T;Q1;Q2;I
[1]  (Q1 Q2 I) ← 0 1 2
[2]  START: → (T≤I)/END
[3]  (Q1 Q2 I) ← Q2 (Q1+Q2) (I+1)
[4]  → START
[5]  END:  Z ← Q1+Q2
   ∇

      FOO 10
55
      I
VALUE ERROR
      I
      ^
```

Writing Assignment and Implicit Output Statements

Many of the statements in a function you write are simply APL expressions, which use numbers (constants), variables, APL primitive symbols, and functions to manipulate numeric and character data in arrays. When the system executes the function, it executes these statements in sequence, much as it would if you typed them sequentially in immediate execution.

Ordinarily, you assign the result of each statement using the assignment arrow. When you do this, there is no immediate indication that the system has performed the action that the statement represents. If you do not assign the results of a statement and the statement returns a result, the system displays that result in the History log while it is still executing the function.

When you assign the result of a statement to a variable, that value is available as an intermediate result for use in a subsequent statement. If you want to have the value available and also see the intermediate result in your session, you can use the Quad function in a statement like the following simplistic example.

```
[0]  Length← x Pythag y
[1]  Side1←x⋆2
[2]  Side2←y⋆2
[3]  ⎕←Sum←Side1+Side2
[4]  Length←Sum⋆0.5
     3 Pythag 4
25
5
```

Of course, you could write all four steps in one simple statement:

```
[1]  Length←((x⋆2)+y⋆2)⋆0.5
```

Note that the system does return the final result of a function you assign in the header. If you want to capture that result, for example to use as a value in another function, you can assign the result in the statement that calls `Pythag`.

```
      Hyp← 3 Pythag 4
```

Rather than assigning the result to `Hyp`, you could make the statement above more complex by performing a calculation using the result in the same statement that calls `Pythag`. You can make single APL statements very complex, and thereby make your functions very compact. There is a trade-off between compactness and readability.

You can write almost any statement in a function that is a valid statement in immediate execution. Two exceptions are that you cannot directly use either user commands (those beginning with a right bracket) or system commands (those beginning with a right parenthesis). You can, however, invoke user commands; the system command `⎕ucmd` takes any user command as an argument, and the system executes the user command. Many of the system commands (the ones you are likely to need under program control) have system function counterparts; for example, `⎕copy` performs the same action in a function that `)copy` performs in immediate execution.

There are also facilities you can use under program control that you cannot use in immediate execution; for example, branching and control structures. These two facilities are explained later in this chapter.

Using Statement Separators

You can place multiple statements on a single line of a function by separating the statements with a diamond (⋄). The system executes the statements from left to right starting to the left of the first diamond, although it retains the right-to-left execution order within each statement. This behavior allows you to to reduce the number of separate lines easily by inserting diamonds in place of new-line characters. The three lines below are equivalent to the single line following them.

```
[1]  Side1←x⋆2
[2]  Side2←y⋆2
[3]  Sum←Side1+Side2

[1]  Side1←x⋆2 ⋄ Side2←y⋆2 ⋄ Sum←Side1+Side2
```

Creating Output

When you want to create output, you can use implicit output (no assignment) statements in a function. For example, you could display the results of a simple function that calculates sales tax.

```
[0]  Z←P STAX T
[1]  Z←(⌈(100×P)×T)÷100
[2]  'PRICE' P
[3]  ' TAX ' Z
[4]  V←P+Z
[5]  'TOTAL' V

      TAX ← 2.13 STAX .05
PRICE 2.13
 TAX  0.11
TOTAL 2.24
```

Note that you have the amount of the tax in the variable TAX to use in another function. See the Formatting Output chapter in the *System Functions Manual* for more information on creating output.

Inserting Comment Statements

You can enter comments in a function by using the Lamp (⍝) symbol. The system does not execute anything on a line after the lamp. You can make an entire line a comment, or you can place a comment after an executable statement.

```
[0]  Z←P STAX T
[1]  ⍝ Sales tax; left arg price; right arg
[2]  ⍝ tax rate; rounds up to nearest penny
[3]  Z←(⌈(100×P)×T)÷100  ⍝ End of function
```

If you place a comment on a line that has multiple statements separated by diamonds, place the comment in the rightmost statement. If you follow a comment by a diamond, the diamond is part of the comment. If you include a comment in a character string enclosed in single quotes, the lamp is a character in the string. The system treats the lamp as a comment designator only if it executes the character string.

```
      a←'3+5 ⍝ I want to comment'
      a
3+5 ⍝ I want to comment
      ⍎a
8
```

Inline Comments

Inline comments are denoted by the glyphs `⪽` and `⪾` surrounding a block of text. Note the similarity of their appearance to the standard comment `⍝` glyph. These glyphs are defined on the APL64 keyboard as keystrokes Alt+Shift+Z and Alt+Shift+X, respectively. These are easy to remember because Alt+Z and Alt+X give the similar looking `⊂` and `⊃` glyphs respectively.

In its simplest form a single opening comment glyph `⪽` behaves like the standard comment glyph `⍝`. It begins a comment that by default runs to the end of the current line. However, unlike a standard comment an inline comment can be terminated before the end of the line by using the closing comment glyph `⪾`. For example, the following expression:

```
X ← a ⪽+ b⪾ × c
```

has "commented out" the `+ b` characters so that the above expression is equivalent at execution time to the following:

```
X ← a × c
```

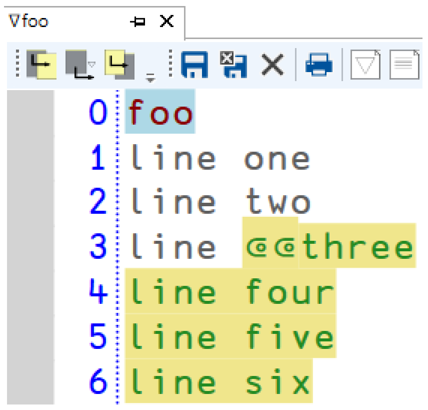
If you don't terminate a single opening comment glyph `⪽` with a closing comment glyph `⪾` on the same line, then the comment runs to the end of the line like this:

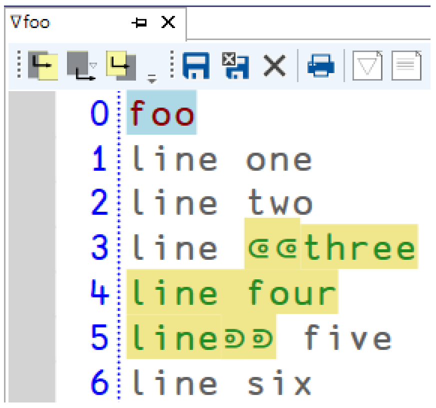
```
X ← a ⪽+ b × c
```

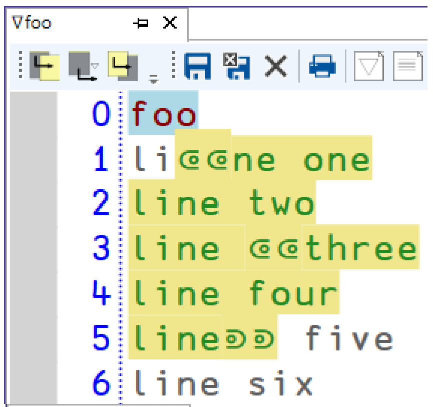
which is functionally identical to a standard comment like this:

```
X ← a ⍝+ b × c
```

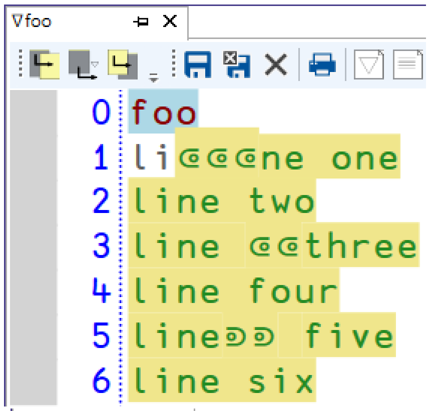
You can use two OR MORE contiguous opening comment glyphs such as `⪽⪽` or `⪽⪽⪽⪽⪽` to begin a comment that runs to the end of the function, unless terminated by a matching number of contiguous closing comment glyphs such as `⪾⪾` or `⪾⪾⪾⪾⪾`. Here are some examples:


```
∇foo
foo
line one
line two
line ⪽⪽three
line four
line five
line six
```

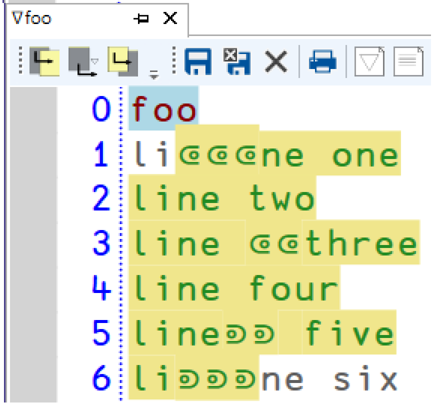

```
∇foo
foo
line one
line two
line ⪽⪽three
line four
line⪾⪾ five
line six
```



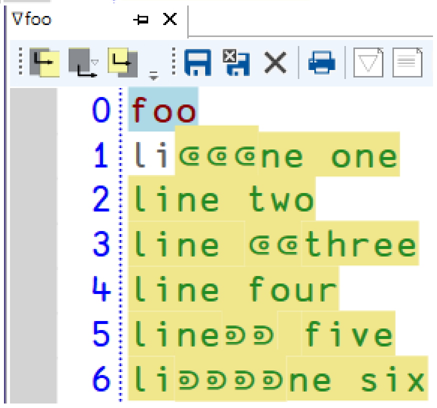
```
0 foo
1 liരരne one
2 line two
3 line രരthree
4 line four
5 lineᘐᘐ five
6 line six
```



```
0 foo
1 liരരരne one
2 line two
3 line രരthree
4 line four
5 lineᘐᘐ five
6 line six
```

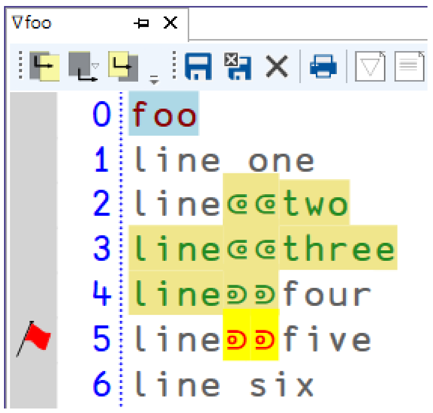


```
0 foo
1 liരരരne one
2 line two
3 line രരthree
4 line four
5 lineᘐᘐ five
6 liᘐᘐᘐne six
```



```
0 foo
1 liരരരne one
2 line two
3 line രരthree
4 line four
5 lineᘐᘐ five
6 liᘐᘐᘐᘐne six
```

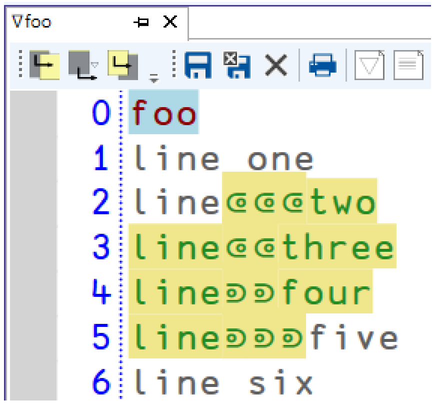
Note that N opening comment glyphs will only match N closing comment glyphs, no more and no less. So, while `ᘐᘐᘐ` will close a comment opened with `രരര` it will not close a comment opened by `രര` or `രരരര`. Also note that comments do NOT nest. For example:



```
0 foo
1 line one
2 lineരരtwo
3 lineരരthree
4 lineᘐᘐfour
5 lineᘐᘐfive
6 line six
```

The comment block stops at the first matching set of closing comment glyphs with the correct number of contiguous `ᘐ` matching the number of `ര` that opened the comment. So, the pair of `ᘐᘐ` on line 4 matching the pair of `രര` on line 2 and does not associate with the pair of `രര` on line 3 that happen to be inside the comment started on line 2.

In most cases you won't have reason to use more than a two opening `രര` and closing `ᘐᘐ` comment glyphs. The primary reason to use a larger number of contiguous comment glyphs is to enclose other comment blocks that are defined using smaller counts of contiguous comment glyphs such as the following:



```
0 foo
1 line one
2 line⍝⍝⍝two
3 line⍝⍝three
4 line⍝⍝four
5 line⍝⍝⍝five
6 line six
```

Using the Branch Statement

You can use a branch statement to alter the execution sequence of your function. The branch statement consists of a right arrow and an expression. You can branch to a line number or you can branch to a label. Using labels is safer; if you edit the function and change the line numbers, the branch to a label still goes where you intend it to. The system treats a label as a named constant. The label assumes the value of the line number it is on while the system executes the function. You cannot assign values to labels.

In the function, you place the label (followed by a colon) on the line where you want to branch. When you branch to that line, the program continues evaluating the function starting with that line number. In the following example, when the system reaches line `[6]`, it branches to line `[8]` and continues execution from there.

```
      . . .
[6]   →TARGET
[7]   . . .
[8]   TARGET:
```

If the expression following the branch arrow is an empty vector, for example (`0⍴1`), the system does not branch and continues with the next line of the function.

You can use this property in conjunction with the Replicate function to make a branch/not branch condition. In the following example, if the value of the second row, second column of the output matrix is not a positive number, the system does not branch. If that element is a positive number, the system branches to line `[18]`.

```
[10]  . . .
[11]  →(OUTPUT[2;2]>0)/ TAXCUM
[12]  ⍝ Lines that finish the function
[13]  ⍝ when there is no tax.
      . . .
[18]  TAXCUM:
[19]  ⍝ Lines that cumulate taxes by state
```

You do not have to use a label in the expression to the right of the branch arrow. The expression can be a constant, a variable, or an expression that performs a calculation. If the expression to the right of the branch arrow evaluates to a number that is a valid line number in the function, the system branches to that line and continues executing the function. If the number is zero or a number that is not a line number in the function, the system exits the function, but continues execution. If another function called the function that uses branch to zero, the calling function continues.

If the expression to the right of the branch arrow results in a vector, the system uses only the first element of the vector. However, you can use a vector in the expression and calculate an index to that vector. This allows you to branch to different line labels as easily as to different line numbers. For example:

```
  →(LABEL1 LABEL2 LABEL3 LABEL4) [I]
```

In this example, the system branches to one of four labeled lines depending on whether `I` has the value of `1`, `2`, `3`, or `4`.

If the branch has nothing after it (a naked branch), the system stops all execution and returns to immediate execution mode. If another function calls the function that uses the naked branch, you cannot restart the calling function at the point of interruption.

Writing Control Structures

APL64 provides another method for controlling the execution within user-defined functions. Control structures identify blocks of statements that the system executes conditionally or repetitively, based on values in the control expressions.

Control structures comprise three parts: keywords, control expressions, and blocks of ordinary APL statements. Keywords identify the type of control structure and mark the boundaries of the blocks of statements they control. The control expressions define the conditions under which the system executes a block of statements. These statements perform the calculations or other actions you want to perform when the control expression is valid.

Throughout this section, control structures are always shown in lowercase for readability. They are case-insensitive in the system, so you can use `:IF`, `:if`, or `:If` interchangeably.

Writing Conditional Structures

The simplest conditional control structure has a single condition statement, for example:

```
[30]  . . .
[31]  :if X=10
[32]          ⍝ Begin block of statements
[33]     Z←Y1 + Y2
. . .
[36]          ⍝ End block of statements
[37]  :endif
[38]  . . .
```

In the above example, the control structure spans seven lines. The keywords, which always start with a colon (`:`), mark the beginning and end of the structure. The control expression specifies that if the variable `X` has any value other than `10` when the system reaches line `[31]`, the system does not execute the block of statements within the control structure. If the value of `X` is `10`, the system executes the statements starting with line `[32]`.

A control expression must evaluate to a Boolean scalar or one-element vector. If the value is `1`, the condition is true. If the value is zero, the condition is false. If the expression returns a vector longer than one element, the system displays `LENGTH ERROR` and suspends execution.

The pattern of the keywords, those words beginning with a colon, defines the outer syntax of the control structure. The outer syntax has specific requirements, but you can build very complex structures.

Using Conditional Structures with Alternatives

You can include alternate blocks of statements for the system to execute.

```
[40]  . . .
[41]  :if X=10
[42]          ⍝ Block 1 of statements
[43]     Z←Y1 + Y2
. . .
[46]  :else
[47]          ⍝ Block 2 of statements
[48]     Z←Y3 + Y4
. . .
[51]  :endif
```

In this example, the system executes the first block of statements (from line `[43]` to the next keyword) if the value of `X` is `10`. If the value of `X` is anything other than `10`, it executes the second block of statements (from line `[48]` to the final statement in the structure).

You can use as many alternatives as you have meaningful conditions. The system evaluates the control expressions from the start of the structure and executes the block of statements associated with the first condition that evaluates to true. Note that the system does not evaluate subsequent conditions once it finds a condition that is true. The system executes only the block of statements following the first true condition.

```
[53]   . . .
[54]   :if X1=10
[55]            ⍝ Block 1 of statements
[56]      Z←Y1 + Y2
. . .
[59]   :elseif (X2-5)≥7.3
[60]            ⍝ Block 2 of statements
[61]      Z←Y3 + Y4
. . .
[64]   :else
[65]            ⍝ Block 3 of statements
[66]      Z←Y5 + Y6
. . .
[69]   :endif
```

You do not have to have an `:else` keyword. If you do have one, it must follow all the `:elseif` keywords in its outer structure. In a structure without an `:else` keyword, the system can execute one block of statements or none. If none of the control expressions associated with the `:if` statement or the `:elseif` statements is true, the system continues with the statement following the final statement of the structure.

Using Condition Extension Keywords

You can also make a block of conditions by compounding the control expressions. You can make multiple conditions with the `:andif` keyword or alternate conditions with the `:orif` keyword.

```
[70]    . . .
[71]   :if X1=0
[72]   :andif X2≠19
[73]            ⍝ Block 1 of statements
[74]      Z←Y1 + Y2
. . .
[77]   :elseif X3≤5
[78]   :orif X4≥6
[79]            ⍝ Block 2 of statements
[80]      Z←Y3 + Y4
. . .
[84]   :endif
```

When you use the `:andif` condition extension statement, all the control expressions in the block of conditions must be true for the system to execute the block of statements following the block of conditions. If any condition is not true, the system does not execute the block of statements.

The system evaluates the condition statements sequentially. If it finds a control expression that is not true, it does not evaluate the other statements in that block of conditions. This makes it possible to use a combination of conditions such as the following:

```
      :if X1≠0
      :andif (X2÷X1)≥19
```

A statement using the APL primitive AND function, for example, `(X1≠0)∧(X2÷X1)≥19`, causes a `DOMAIN ERROR` if `X1` is zero.

When you use the `:orif` condition extension statement, the system evaluates each control expression in that block independently. If any control expression in the block is true, the system executes the block of statements immediately following the block of conditions; in the above example, starting with line `[79]`. If none of the statements is true, the system does not execute the block of statements.

The system evaluates the condition statements sequentially. If it finds a control expression that is true, it does not evaluate the other statements in that block of conditions. This makes it possible to use a combination of conditions such as the following:

```
        :if X1=0
        :orif (X2÷X1)<19
```

A statement using the APL primitive OR function, for example, `(X1=0)∨(X2÷X1)<19`, causes a `DOMAIN ERROR` if `X1` is zero.

You can string multiple `:andif` statements in one block, and you can string multiple `:orif` statements in one block, but you cannot mix the two types of condition extension statements in the same block of conditions.

Writing Repetitive Structures

The simplest repetitive control structure has a single condition statement, for example:

```
[85]   . . .
[86]   :while X>20
[87]           ⍝ Begin block of statements
[88]      Z←Y1 + Y2
. . .
[91]   :endwhile
```

In the above example, the control structure spans six lines. The keywords mark the beginning and end of the structure. The control expression specifies that if the variable `X` is less than or equal to `20` when the system reaches line `[86]`, the system does not execute the block of statements. If `X` is greater than `20`, the system executes the statements starting with line `[87]`.

When the system reaches line `[91]` it branches back to line `[86]` and evaluates the condition again. This typically implies that one of the statements in the block alters the value of `X`, a statement within the block branches outside the control structure, or something external to the function interrupts the function while the system executes the block. Otherwise, your program is in an infinite loop.

In this structure, the system evaluates the condition once more than it executes the block of statements. Since the test precedes the block of statements, it is known as a leading test.

Using a Trailing Test

You can also have one condition statement at the end of the control structure.

```
[93]   . . .
[94]   :repeat
[95]           ⍝ Begin block of statements
[96]      Z←Y1 + Y2
. . .
[99]   :until X>30
```

In the above example, the control structure spans six lines. The keywords mark the beginning and end of the structure. When the system reaches line `[94]`, it automatically executes the block of statements. The control expression specifies that if `X` is greater than `30` when the system reaches line `[99]`, it continues with the next line. If `X` is less than or equal to `30`, the system branches back to line `[94]` and executes the block again.

This typically implies that one of the statements in the block alters the value of `X`, a statement within the block branches outside the control structure, or something external to the function interrupts the function while the system executes the block. Otherwise, your program is in an infinite loop.

In this structure, the system executes the block of statements before it evaluates the condition the first time. Since the test follows the block of statements, it is known as a trailing test.

Using Leading and Trailing Tests

You can combine leading and trailing tests in one control structure. The control expressions can test the same variable, as in the example below, or different variables.

```
[101]   . . .
```

```
[102]  :while X>20
[103]          ⍝ Begin block of statements
[104]     Z←Y1 + Y2
. . .
[107]  :until X>30
```

In the above example, the control structure spans six lines. The keywords mark the beginning and end of the structure. The control expression specifies that if `X` is less than or equal to `20` when the system reaches line `[102]`, the system does not execute the block of statements. If `X` is greater than `20`, the system executes the block.

When the system reaches the end of the control structure, it evaluates the second control expression. This control expression specifies that if `X` is greater than `30`, it continues with the next line. If `X` is less than or equal to `30`, the system branches back to line `[102]` and evaluates the first control expression again.

In this structure, the trailing condition must be false and the leading condition true for the system to execute the block of statements more than once. If the trailing condition is false, it does not necessarily mean the system executes the block a second time. If the value of `X` is reduced to less than `20` during the first execution of the block of statements, the system continues with the statement following line `[107]` even though `X` is not greater than `30`.

Writing Repetitive Structures with No Control Expression Test

You can also have a repetitive control structure with no control expression.

```
[109]  . . .
[110]  :repeat
[111]          ⍝ Begin block of statements
[112]     Z←Y1 + Y2
. . .
[115]  :endrepeat
```

In the above example, the control structure spans six lines. The keywords mark the beginning and end of the structure. When the system reaches line `[110]`, it automatically begins to execute the block of statements. If the system reaches line `[115]`, it branches back to line `[110]` and begins to execute the block again. This implies that you have a statement within the block that branches outside the control structure or ends your program.

Using Condition Extension Keywords

You can also make a block of conditions in a repetitive control structure by compounding the control expressions. You can make multiple conditions with the `:andif` keyword or alternate conditions with the `:orif` keyword when you combine them with either the `:while` or `:until` keywords.

Valid outer syntax structures include:

```
:while              :while
:andif              :orif
block of statements block of statements
:endwhile           :endwhile


:repeat             :repeat
block of statements block of statements
:until              :until
:andif              :orif
```

You can also combine condition keywords in a structure that combines leading tests with trailing tests.

```
:while              :while
:andif              :orif
:andif              block of statements
block of statements :until
:until              :andif
```

```
:andif          :andif


:while          :while
:andif          :orif
block of statements     :orif
:until          block of statements
:orif           :until
:orif           :orif
```

Writing Iterative Structures

The iterative control structure has a two-part control statement. The first keyword specifies a controlled variable, and the second keyword defines an expression. The system assigns each of the values in the expression to the controlled variable and executes the block of statements once for each value. For example:

```
[117]   . . .
[118]   :for J :in ⍳10
[119]           ⍝ Begin block of statements
[120]      Z←Mat[J]
. . .
[123]   :endfor
```

In the above example, the control structure spans six lines. The keywords mark the beginning and end of the structure. When the system reaches line `[118]`, it automatically begins to execute the block of statements, with `J` having the first value in the expression, (`1` in this case).

When the system reaches line `[123]`, it branches back to line `[118]`, assigns the next value to `J`, and begins to execute the block again. The system executes the block of statement once for each value of the expression. When the system reaches a line containing a `:for` statement, it always executes the block of statements unless the expression following `:in` is empty.

The expression does not have to be an arithmetic sequence. You could craft a nested vector, for example:

```
:for J :in 2.2 (A-B) (20+2×⍳4) (⍟10)
```

The system would execute the block of statements four times, with `J` set to the value produced by each item of the vector in turn.

Note that localization of variables works only at the function level and not at the level of a control structure. When the system exits the structure, the controlled variable has the last value assigned it. If the variable had a value within the function before the system reached the beginning of an iterative control structure, that value is gone. However, you can reassign a value to the controlled variable within the block of statements; the execution of the iterative structure does not depend on the value of the controlled variable. When the system branches back to the `:for` statement, it simply assigns the next value to the controlled variable. Similarly, the values of the vector following the `:in` keyword are evaluated only once at the beginning of the structure. If you specify the vector with a named variable, you can change the variable within the structure without altering the execution of the function (unless you exit the structure and return to it).

Continuing a Repetitive or Iterative Structure

To simplify the coding of certain kinds of looping patterns within repetitive or iterative control structures, you can use the `:continue and :continueif` keywords. The `:continue` keyword takes no argument, but `:continueif` does. When the system executes these statements, it branches to the end of the block of statements but does not necessarily leave the control structure. In a repetitive structure that ends with an `:until` keyword, the system performs the trailing test. In a repetitive structure that has the `:while . . . :endwhile` pattern, the system performs the leading test. In an iterative structure, the system starts the next loop if there is another value in the expression following the `:for` statement. This keyword has the effect of curtailing only the current loop of execution within a control structure. In the example below, which simulates processing data for weekdays, when the system reaches line `[130]` and executes the `:continue` statement, it next executes the trailing test on line `[133]` to determine whether to return to line `[126]` or continue to line `[134]`.

```
[125]  . . .
[126]  :repeat
[127]        ⍝  Begin block of statements
[128]     :if DayName = Saturday
[129]     :orif DayName = Sunday
[130]        :continue
[131]     :endif
. . .
[133]  :until EndOfMonth = 1
[134]
```

Exiting a Repetitive or Iterative Structure

To exit one of these structures without meeting the completion condition or exhausting the list of arguments in a `:for` statement, you can use the `:leave` keyword. This keyword takes no argument. When the system executes this statement, it exits the block of statements defined by the beginning keyword and its corresponding `:end` keyword. The possibilities for the terminal identifier include `:end`, `:endwhile`, `:endrepeat`, `:until` and `:endfor`.

When the system encounters a `:leave` statement within one of these structures, it executes the statement following the terminal keyword. In the example below, if the system reached line `[140]` and executed the `:leave` statement, it would next execute the statement following line `[144]`.

```
[136]  . . .
[137]  :while transactions > 0
[138]        ⍝  Begin block of statements
[139]     :if Balance < ItemValue
[140]        :leave
[141]     :endif
[142]     Balance←Balance - ItemValue
. . .
[144]  :endwhile
```

Writing Selection Structures

Selection control structures use multiple control expressions. The system identifies the block of statements to execute based on two expressions matching. Note that match means in the sense of the APL primitive function Match (≡); the expressions must have the same rank, shape and value. It is not sufficient if they are equal. A scalar `5` equals a one-element vector whose value is `5`, but the two do not match.

The simplest selection control structure has one top-level control expression and a series of case expressions, each of which can potentially match the top-level expression; for example:

```
[146]  . . .
[147]  :select ×value
[148]  :case 1   ⍝ Positive case
[149]            ⍝ Block 1 of statements
[150]     Z←Y1 + Y2
. . .
[153]  :case ¯1   ⍝ Negative case
[154]             ⍝ Block 2 of statements
[155]     Z←Y3 + Y4
. . .
[158]  :endselect
```

In the above example, the control structure spans twelve lines. The keywords, which always start with a colon (`:`), mark not only the beginning and end of the structure, but also the segments of the structure.

The combined control expressions specify that if the signum of `value` is `1` (`value` is positive), when the system reaches line `[147]`, the system executes the first block of statements. If the signum of `value` is `¯1` (`value` is negative) when the system reaches line `[147]`, the system executes the second block of statements.

If the value of the `:select` expression does not match the value in any of the `:case` expressions, the system branches to the `:endselect` statement and begins executing the statement that follows.

Note that with the selection structure, the control expression in the `:select` statement does not have to evaluate to a Boolean `1`. When the system reaches the `:select` statement, it evaluates that expression to find its value. It then evaluates the control expression in the first `:case` statement. The system tests the two expressions to see if they match. If the result of the match is a Boolean `1`, the system executes the block of statements that follows the `:case` statement.

If the control expression for the first `:case` statement does not match the top-level expression, the system evaluates subsequent `:case` control expressions until it finds one that matches or exhausts the possibilities. You can use as many alternatives as you have meaningful conditions. Once the system finds a match, it does not evaluate any further expressions. The system executes at most one block of statements following a `:case` statement.

Writing Selection Structures with Alternate Matching Values

If there are several cases for which you want to execute the same block of statements, you can extend the possibilities by using a `:caselist` statement that specifies multiple alternate expressions. If any of the expressions in the list matches the `:select` statement expression, the system executes the block of statements associated with that `:caselist`. You must define the conditions so that any of the alternatives might match the top-level expression. You cannot have multiple expressions in the `:select` statement for a `:case` or `:caselist` expression to match.

```
[160]  . . .
[161]  :select state
[162]  :case 'DE'
[163]          ⍝ Block 1 of statements
[164]     Z←Y1 + Y2
. . .
[167]  :caselist 'GA' 'HI' 'NY'
[168]          ⍝ Block 2 of statements
[169]     Z←Y3 + Y4
. . .
[172]  :endselect
```

In this example, if `state` matches `'DE'`, the system executes the first block of statements. If `state` matches `'GA'`, `'HI'`, or `'NY'`, the system executes the second block of statements. You can have multiple `:case` and `:caselist` statements; they can be interspersed. The system executes at most one block of statements within this structure.

Writing Selection Structures with a Default Alternative

You can include a block of statements for the system to execute in the event that none of the `:case` or `:caselist` statement expressions matches the `:select` statement expression.

```
[174]  . . .
[175]  :select X1
[176]  :caselist J1 J2 J3
[177]          ⍝ Block 1 of statements
[178]     Z←Y1 + Y2
. . .
[181]  :case   K
[182]          ⍝ Block 2 of statements
[183]     Z←Y3 + Y4
. . .
[186]  :else
[187]          ⍝ Block 3 of statements
[188]     Z←Y0
. . .
[191]  :endselect
```

In this example, if `X1` matches any of the variables `J1`, `J2` or `J3`, the system executes the first block of statements. If `X1` matches the variable `K`, the system executes the second block of statements. If `X1` does not match any of the variables, the system executes the third block of statements.

If you have an `:else` keyword, it must follow all the `:case` and `:caselist` keywords in its outer structure. In a selection structure without an `:else` keyword, the system can execute one block of statements or none. In a selection structure with an `:else` keyword, the system executes one block of statements.

Using Control Expressions of Higher Rank

If you place multiple values or expressions on the `:select` line, you create an expression of higher rank (for example, a vector) which the `:case` statement expression must match. If your top-level expression is `:select (1 + 2 3)`, then a `:case` expression must evaluate to a two-element vector with the value (`3 4`) to match. If the `:case` expression evaluates to either the scalar `3` or the scalar `4`, it does not match. If you have the expression `:caselist (3 4)`, that also does not match, because the values are interpreted as two different list values. If you have either `:caselist (3 4) (5 6)` or `:caselist (3 4) ⍬`, there is a match, and the system executes the block of statements associated with the `:caselist`.

Writing Nested Control Structures

You can nest one control structure inside another. When you nest one structure within another, the entire body of the inner control structure must be within one block of statements that the outer control structure controls.

For readability, APL2000 recommends that you keep the colons of the keywords of any structure on the same vertical column in your function. You should indent blocks of statements several characters. If you have a nested structure, indent those keywords from the keywords of the structure that contains it, and indent the block of statements of the nested structure farther still.

```
:if condition
   block of statements
:elseif condition
   block of statements
   :if condition
      block of statements
   :else
      block of statements
   :endif
   block of statements
:else
   block of statements
:endif
```

You can nest different types of structures within each other; for example, you can nest a conditional structure within a repetitive structure. You can nest more than one structure within another, and you can nest structures more than two levels deep.

If you use indentation in your functions, make sure you select “Enable Automatic indentation in functions” on the Editor Options window, which you reach from the Options menu. Then after you press Tab at the beginning of a line in the full screen editor, APL inserts the specified number of spaces on each subsequent line until you press Backspace at the beginning of a line. This holds true for each level of indentation you add.

Other Syntax Rules

There are other rules that apply to all the control structures:

Errors

When you attempt to save a function in the function Edit window, the interpreter analyzes the outer syntax of a control expression and warns you of any `OUTER SYNTAX ERROR`. However, when you define (`⎕def`, `⎕defl`, or `⎕fx`) a function under program control that contains such an error, the system does not display an error message at that time. If a control structure is ill-formed, the system displays the error message when the function is called. You cannot execute any part of a function with an outer syntax error.

Other errors within control structures generate the same message they would elsewhere.

An attempt to execute a control statement in immediate execution mode results in a `SYNTAX ERROR`. You can use control structures only in user-defined functions.

An attempt to branch illegally into a control structure generates a `DESTINATION ERROR`. See the “Branching Restrictions in Control Structures” section later in this chapter.

Some modifications to suspended functions containing control structures may mark the suspension as having `SI DAMAGE`. When this happens, use the `)reset` or `)sic` commands or a naked branch statement to clear the function call from the stack; then invoke the function again..

The `:end` keyword

You can use `:end` in any control structure in place of the more explicit keywords shown above; for example, `:endwhile`. You cannot interchange the others, however.

Specifically, you cannot use the `:endif` keyword when you combine the condition extension keywords `:andif` or `:orif` with the keywords `:while` or `:until`. You must use `:endwhile` or `:endrepeat` in their structures, or you can use `:end` in either type of structure.

Interspersing Comment Lines and Blank Lines

You can place a blank line or a comment line between any two lines within a control structure without affecting the way the system executes your function.

You can use the diamond symbol (⋄) to combine lines in a control structure. Comments must be the rightmost portion of a line, if you include them. A label cannot follow a diamond on a line. The system executes the portion of the line to the left of the first diamond first, and treats the portion after each diamond on the line as though it were on a separate line; for example, the following line is a valid first line for a selection structure.

```
      :select x ⋄ :case 1 ⋄ Y←3
```

You can place a label on any line in a control structure that does not begin with a keyword. See the "Using the Branch Statement" section earlier in this chapter for information on labels.

Branching Restrictions in Control Structures

You cannot branch into an iterative control structure (a `:for` loop). An attempt to do so results in a `DESTINATION ERROR`.

Since you cannot place a label on a line that contains a keyword, you cannot branch to a keyword statement using labels. In some cases you can branch to a keyword statement using absolute line numbers, although this is not recommended. You can branch into some control structures using labels.

You cannot branch to a `:case` statement or to a `:caselist` statement in a selection control structure. You can branch to a line within the block of statements following a `:case` or a `:caselist`. In this situation, the system executes the remainder of the block of statements up to the line where it encounters the next keyword. It then branches to the `:endselect` statement and continues.

You can branch into a conditional or a repetitive control structure. In this situation, the system begins execution at the line to which you branch and continues execution of the control structure following the same rules as if you entered the control structure without branching.

You can branch out of any control structure. This terminates execution of the control structure, and the system continues execution at the line to which you branch.

Using Keywords to Branch within Functions

To change the sequence of execution within a user-defined function, you can use the `:goto` keyword. This keyword requires exactly one argument, which must be a label identifier; it cannot be a variable, number, or other expression. When the system encounters this keyword, it continues executing the function with the line on which the label occurs. The `:goto` keyword is an alternative to using a branch arrow with a label. The following two examples are equivalent. When the system reaches line `[193]` of the function, the system branches to line `[195]` and continues execution.

```
[193] → TARGET
[194]
[195]  TARGET:
```

```
[193] :goto TARGET
[194]
[195]  TARGET:
```

Exiting a Function

As an unequivocal and easy-to-read method for ending the execution of a function, you can use the `:return` and `:returnif` keywords. The `:return` keyword takes no argument except when a function declares a result variable in which case the argument is the value returned . The `:returnif` keyword takes an argument that is the conditional statement. When the system executes these statements, it exits the function but continues execution.

The following examples are equivalent. The system stops executing the current function and returns either to the function that called this one or to immediate execution mode.

```
[198] :return
```

```
[198] :return value
```

```
[198] → 0
```

Chapter 5: Primitive Functions and Operators

This chapter summarizes the APL primitive functions and operators. Each synopsis contains the syntax, a description, and one or more examples. In some examples, a variable or result is shown in display form to illustrate the data structures. The `DISPLAY` function from the `UTILITY` workspace and the `]display` user command in APL64 produce the display form. There are also descriptions of other symbols that APL uses that are neither functions nor operators. This chapter uses the following conventions:

• *res*	the explicit result
• *arg*	the argument of a monadic function
• *larg*	the left argument of a dyadic function
• *rarg*	the right argument of a dyadic function
• *conforming*	the shapes of the left and right arguments must agree in some manner; frequently, they must have the same shape or extend to the same shape
• *i*	a non-negative scalar that designates an axis or a dimension of an array
• *idx*	an index or a variable with valid indices
• *f, g*	any functions, primitive, system, or user-defined, that act as operands to an operator
• *oparray*	an array that acts as an operand to an operator
• *ext*	an external factor that affects the result of an operation (for example, `⎕ct`, `⎕rl`, `⎕io`)
• ↔	equivalence; expressions on both sides of the double arrow return the same result
• scalar	a scalar function calculates the result using one data element at a time, or one datum from the left argument with the corresponding datum from the right argument. Each element of the result is computed independently and depends solely on the corresponding left and right items. The entire result has the same shape as at least one of the arguments.

Note: As with a language dictionary that uses words to provide examples of the usage of other words, this manual uses some primitive functions in the examples of how other primitives work. If you are unfamiliar with APL, you should take note of the dyadic function Reshape (`⍴`) and the monadic function Index generator (`⍳`).

Contents

The primitives described in this chapter are arranged as follows:

Numeric Functions

Dyadic	+	Plus	Monadic	⌊	Floor
Monadic	+	Conjugate	Dyadic	∣	Residue
Dyadic	−	Minus	Monadic	∣	Magnitude
Monadic	−	Negate	Dyadic	!	Binomial
Dyadic	×	Times	Monadic	!	Factorial
Dyadic	÷	Divide	Dyadic	?	Deal
Monadic	÷	Reciprocal	Monadic	?	Roll
Dyadic	*	Power	Dyadic	○	Trigonometric functions
Monadic	*	Exponential	Monadic	○	Pi times
Dyadic	⍟	Logarithm	Dyadic	⊥	Base value
Monadic	⍟	Natural logarithm	Dyadic	⊥	Representation
Monadic	⍳	Index generator	Dyadic	⌹	Matrix divide
Dyadic	⌈	Maximum	Monadic	⌹	Matrix inverse
Monadic	⌈	Ceiling	Dyadic	⍨	Commute
Dyadic	⌊	Minimum			

Boolean Functions

Scalar Functions that Return a Boolean Array

Dyadic	=	Equal
Dyadic	≠	Not equal
Dyadic	>	Greater than
Dyadic	≥	Greater than or equal
Dyadic	<	Less than
Dyadic	≤	Less than or equal

Nonscalar Functions that Return a Boolean Array

Dyadic	≡	Match
Dyadic	≢	Mismatch
Dyadic	∊	Member of
Dyadic	⍷	Find

Scalar Logical Functions (Use and Return Boolean Arrays)

Dyadic	∧	AND
Dyadic	∨	OR
Dyadic	⍲	NAND
Dyadic	⍱	NOR
Monadic	~	NOT

Structural Functions

Dyadic	, ⍪	Catenate	Monadic	,	Ravel
Monadic	⊃	Disclose	Dyadic	⍴	Reshape
Dyadic	↓	Drop	Monadic	⌽ ⊖	Reverse
Monadic	⊂	Enclose	Dyadic	⌽ ⊖	Rotate
Monadic	∊	Enlist	Dyadic	↑	Take
Monadic	↑	First	Monadic	⍉	Transpose
Dyadic	[]	Indexing	Dyadic	⍉	Transpose
Dyadic	⊂	Partition	Monadic	∪	Unique
Dyadic	~	Without	Dyadic	⊃	Pick

General Functions

Functions that Return a Property

Monadic	×	Signum
Monadic	⍴	Shape
Monadic	≡	Depth

Functions that Return an Index

Dyadic	⍳	Index of
Monadic	⍋	Numeric Grade up
Dyadic	⍋	Character Grade up
Monadic	⍒	Numeric Grade down
Dyadic	⍒	Character Grade down

Miscellaneous Functions

Monadic	⍎	Execute
Monadic	⍕	Format
Dyadic	⍕	Pattern Format

Operators

Expand	*oparray* \ and *oparray* ⍀
Replicate	*oparray* /and *oparray* ⌿
Reduction	*f* / and *f* ⌿
Scan	*f* \and *f* ⍀
Inner Product	*f* .*g*
Outer Product	∘.*g*
Each	*f*¨

Other Primitive Symbols

Symbols Associated with Values

Left arrow	←	Assignment; also used as Sink
Brackets	[ ]	Index into
Parentheses	( )	Punctuation
High minus	¯	Negative numbers
Zilde	⍬	Empty value
Quad	⎕	Numeric input/output
Quote-Quad	⍞	Character input and output

Symbols Associated with Functions

Lamp	⍝	Comments
Left block	⟃	Inline comment
Right block	⟄	Inline comment
Diamond	⋄	Statement separator
Right arrow	→	Branch
Colon	:	Keyword (:*keyword*); Label (*label*:); also used as a delimiter

Symbols that Denote an Action Word

Right paren	)	System command, for example,)clear
Right bracket	]	User command, for example,]display
Quad	⎕	System function, variable or constant, for example, ⎕io

Other Syntax Symbols

Ampersand	&	Menu access key designator; Continuation character
Del	∇	Tool argument; function designator
Omega	⍵	Tool argument
Delta	∆	Character in object names
Delta underscore	⍙	Character in object names
Jot	∘	Placeholder for Outer Product operand
Underscore	_	Character in object names
Single quote	'	Character string delimiter (also called Apostrophe)
Double quote	"	Character string delimiter
Semicolon	;	Argument delimiter
Number sign	#	Syntactic designator

Numeric Functions

Plus +

Syntax: *res* ← *larg* + *rarg*

res +← *rarg* ↔ *res* ← *res* + *rarg*

Description:
Scalar. Add two numbers.
larg, *rarg:* any numeric arrays (conforming).
res: each item of *larg* added to corresponding item of *rarg*.

Example:
```
      ¯2 2 2 + 3.5 1 ¯2
1.5 3 0
```

Conjugate +

Syntax: *res* ← + *arg*

Description:
Scalar. Return the value of a number.
arg: any numeric array.
res: same as `0+`*arg*.

Example:
```
      +¯27.34 18 6
¯27.34 18 6
```

Minus −

Syntax: *res* ← *larg* − *rarg*

res −← *rarg* ↔ *res* ← *res* − *rarg*

Description:
Scalar. Subtract two numbers.
larg, rarg: any numeric arrays (conforming).
res: each item of *rarg* subtracted from the corresponding item of *larg*.

The Minus function is not the same as the High minus (¯) symbol, which designates a negative number.

Example:
```
      ¯2 2 2 - 3.5 1 ¯2
¯5.5 1 4
```

Negate −

Syntax: *res* ← − *arg*

Description:
Scalar. Change the sign of a number.
arg: any numeric array.
res: each item of *arg* subtracted from `0`.

Example:
```
      - 2 ¯2 1.5
¯2 2 ¯1.5
```

Times ×

Syntax: *res* ← *larg* × *rarg*
res ×← *rarg* ↔ *res* ← *res* × *rarg*

Description:
Scalar. Multiply two numbers.
larg, *rarg:* any numeric arrays (conforming).
res: each item of *larg* multiplied by the corresponding item of *rarg*.

Example:
```
      ¯2 2 2 × 3.5 0 2
¯7 0 4
```

Note: For monadic ×, see Signum in the "General Functions" section of this chapter.

Divide ÷

Syntax: *res* ← *larg* ÷ *rarg*
res ÷← *rarg* ↔ *res* ← *res* ÷ *rarg*

Description:
Scalar. Divide two numbers.
larg: any numeric array.
rarg: any numeric array (conforming).
res: each item of *larg* divided by the corresponding item of *rarg*. Division by zero causes a `DOMAIN ERROR` except in the case of `0÷0`, which returns `1`.

Example:
```
      2 ¯3 0 ÷ 1 3 0
2 ¯1 1
      0÷0
1
```

Reciprocal ÷

Syntax: *res* ← ÷ *arg*

Description:
Scalar. Find the reciprocal of a number.
arg: any nonzero numeric array.
res: `1` divided by each item of *arg*.

Example:
```
      ÷ 2 ¯1 ¯0.5
0.5 ¯1 ¯2
```

Power ⋆

Syntax: *res* ← *larg* ⋆ *rarg*
res ⋆← *rarg* ↔ *res* ← *res* ⋆ *rarg*

Description:
Scalar. Raise a number to a specific power.
larg, *rarg:* any numeric arrays (conforming), except that if *rarg* is fractional, *larg* must be non-negative.
res: *larg* raised to the corresponding *rarg* power.

Example:
```
      2 49 4 0 ⋆ 3 0.5 ¯1 40
8 7 0.25 0
```

Exponential ⋆

Syntax: *res* ← ⋆ *arg*

Description:
Scalar. Raise *e* to a power.
arg: any numeric array.
res: *e* (2.71828...) raised to the power specified by each item of *arg*.

Example:
```
      ⋆ 1 ¯1 0
2.718281828 0.3678794412 1
```

Logarithm ⍟

Syntax: *res* ← *larg* ⍟ *rarg*
res ⍟← *rarg* ↔ *res* ← *res* ⍟ *rarg*

Description:
Scalar. Compute the logarithm of a number.
larg, *rarg:* any positive numeric arrays (conforming).
res: the logarithm of each element of *rarg* to the corresponding base in *larg*.

Example:
```
      2 49 4 ⍟ 8 7 0.25
3 0.5 ¯1
```

Natural logarithm ⍟

Syntax: *res* ← ⍟ *arg*

Description:
Scalar. Compute the natural logarithm of a number.
arg: any positive numeric array.
res: logarithm (base *e*) applied to each item of *arg*.

Example:
```
      ⍟ 1 10 2.7182818284
0 2.302585093 1
```

Index generator ⍳

Syntax: *res* ← ⍳ *arg*

Description:
Return a set of consecutive integers.
arg: positive integer scalar.
res: a vector of *arg* integers in the sequence ⎕io, ⎕io+1, ⎕io+2, ...
ext: ⎕io.

Example:
```
      ⍳5
1 2 3 4 5
```

Note: For dyadic ⍳, see Index of in the "General Functions" section of this chapter.

Maximum ⌈

Syntax: *res* ← *larg* ⌈ *rarg*
res ⌈← *rarg* ↔ *res* ← *res* ⌈ *rarg*

Description:
Scalar. Select the greater of two numbers.
larg, *rarg:* any numeric arrays (conforming).
res: the larger of each corresponding pair of numbers in *larg* and *rarg.*

Example:
```
      ¯3.2 ¯4.1 ⌈ 7 ¯4.2
7 ¯4.1
```

Ceiling ⌈

Syntax: *res* ← ⌈ *arg*

Description:
Scalar. Round up to the nearest integer.
arg: any numeric array.
res: smallest integer greater than or equal to *arg*.
ext: ⎕ct.

Example:
```
      ⌈ 3.1416 ¯1.5 6
4 ¯1 6
```

Minimum ⌊

Syntax: *res* ← *larg* ⌊ *rarg*

Description:
Scalar. Select the lesser of two numbers.
larg, *rarg:* any numeric arrays (conforming).
res: the lesser of each corresponding pair of numbers in *larg* and *rarg*.

Example:
```
      ¯3.2 ¯4.1 ⌊ 7 ¯4.2
¯3.2 ¯4.2
```

Floor ⌊

Syntax: *res* ← ⌊ *arg*
res ⌊← *rarg* ↔ *res* ← *res* ⌊ *rarg*

Description:
Scalar. Round down to the nearest integer.
arg: any numeric array.
res: largest integer less than or equal to *arg*.
ext: ⎕ct.

Example:
```
      ⌊ 3.1416 ¯1.5 6
3 ¯2 6
```

Residue |

Syntax: *res* ← *larg* | *rarg*
res |← *rarg* ↔ *res* ← *res* | *rarg*

Description:
Scalar. Find the remainder.
larg, *rarg:* any numeric arrays (conforming).
res: the remainder after dividing each item of *rarg* by the corresponding item of *larg;* that is:
rarg − (⌊*rarg*÷*larg*) ×*larg*.
ext: `⎕ct`.

Example:
```
      2 ¯2 1 | 3 3 3.14159
1 ¯1 0.14159
```

Magnitude |

Syntax: *res* ← | *arg*

Description:
Scalar. Compute the absolute value of a number.
arg: any numeric array.
res: absolute value of each element of *arg*.

Example:
```
      | 2 0 ¯1.6
2 0 1.6
```

Binomial !

Syntax: *res* ← *larg* ! *rarg*
res !← *rarg* ↔ *res* ← *res* ! *rarg*

Description:
Scalar. Find the number of possible combinations for a set of objects.
larg, *rarg:* any positive numeric arrays (conforming).
res: the number of possible combinations of *rarg* objects selected *larg* at a a time.

Example:
```
      1 2 5 ! 5 4 5
5 6 1
```

Factorial !

Syntax: *res* ← ! *arg*

Description:
Scalar. Compute the factorial of a number.
arg: any numeric array, excluding negative integers.
res: if *arg* is a positive integer, *res* is the product of all positive integers from `1` through *arg*. If *arg* is `0`, *res* is `1`. All fractional numbers are computed using the gamma function on *arg*+`1`; the function is undefined for negative integers.

Example:
```
      ! 0 4 2.5
1 24 3.32335097
```

Deal ?

Syntax: *res* ← *larg* ? *rarg*

Description:
Select a set of unique pseudorandom integers.
larg, *rarg*: non-negative integer scalars where *larg* ≤ *rarg*.
res: *larg* elements selected pseudorandomly without replacement from ⍳*rarg*.
ext: ⎕io, ⎕rl.

Example:
```
      8 ? 10
1 5 3 4 9 6 8 7
```

Note: For additional information, refer to *Chapter 9: Random Number Generator* in the System Functions Manual.

Roll ?

Syntax: *res* ← ? *arg*

Description:
Scalar. Select a pseudorandom integer.
arg: any positive integer array.
res: an integer picked at random from the set of numbers given by ⍳ *arg*[*n*]; *res* contains a pseudorandom number for each element of *arg* where ⎕io ≤ *res*[*n*] ≤ *arg*[*n*]+⎕io−1.
ext: ⎕io, ⎕rl.

Example:
```
      ? 2000 12 30
1969 2 23
```

Trigonometric functions ○

Syntax: *res* ← *larg* ○ *rarg*
res ○← *rarg* ↔ *res* ← *res* ○ *rarg*

Description:
Scalar. Compute a trigonometric function for a number.
larg: any array of integers in the range ¯7 to +7.
rarg: any valid numeric array.
res: the trigonometric function selected by *larg* applied to each corresponding item in *rarg*.

Note: All angular right arguments and results are measured in radians. You can think of this function as being 15 different functions, where the left argument is a code that specifies the function you want to use.

larg	function	*larg*	function
¯7	ARCTANH	7	TANH
¯6	ARCCOSH	6	COSH
¯5	ARCSINH	5	SINH
¯4	(¯1+ *rarg*⋆2)⋆.5	4	(1+ *rarg*⋆2)⋆.5
¯3	ARCTAN	3	TAN
¯2	ARCCOS	2	COS
¯1	ARCSIN	1	SIN
0	(1− rarg⋆2)⋆.5		

Examples:
```
      0 ○ .6
0.8

      2 ○ 3.14159
```

```
¯1
      ¯3 ○ 0 1 2
0 0.7853981634 1.107148718
      1 2 3 ○ (○÷4)
0.7071067812 0.7071067812 1
```

Pi times ○

Syntax: *res* ← ○ *arg*

Description:
Scalar. Multiply a number by *pi* (3.141592...).
arg: any numeric array.
res: *arg* multiplied by *pi*.

Example:

```
      ○ 1 2 0
3.141592654 6.283185307 0
```

Base value ⊥

Syntax: *res* ← *larg* ⊥ *rarg*

Description:
Find the base value of a number whose successive digits are the elements of *rarg*.
larg, *rarg:* any numeric arrays (conforming).
res: the expression of *rarg* in radix *larg*. The representation of *res* is decimal.
This function is also known as Decode.

Relevant Definitions:
Base: the number of units in a given digit's place that is required to give 1 in the next higher place.
Radix: the base of a place-value notation.

Example:

```
      10 ⊥ 1 7 7 6          ⍝ Decimal radix
1776
      2⊥1 0 1 0             ⍝ Binary radix
10
      7 24 60 ⊥ 3 12 50     ⍝ Days/Hours/Minutes mixed radix
5090
      10 3 2 10 ⊥ 1 7 7 6   ⍝ Arbitrary mixed radix
276
           2 10 ⊥     7 6   ⍝ Build the above example step by step
76
         3 2 10 ⊥   7 7 6
216
       0 3 2 10 ⊥ 1 7 7 6   ⍝ Note the first element of larg is only
276                           a placeholder to make the arguments conform
```

Note: *res* ↔ *rarg*[*n*] + (*rarg*[*n*-1] × *larg*[*n*]) + (*rarg*[*n*-2] × (*larg*[*n*-1] × *larg*[*n*])) + ...
+(*rarg*[*n*-*m*] × (×/ *larg*[(*n*-*m*) + ⍳(*m*)])) + ... + (*rarg*[1] × (×/ *larg*[1 + ⍳(n-1)]))

Representation ⊤

Syntax: *res* ← *larg* ⊤ *rarg*

Description:
Find the representation of a number or numbers (represented in decimal notation) in another radix.
larg, *rarg:* any numeric arrays.
res: the expression of each element of *rarg* represented in a number system described by *larg*.
This function is also known as Encode.

Example:
```
      10 10 10 10 ⊤ 1776     ⍝ Decimal radix
1 7 7 6
      2 2 2 ⊤ 5              ⍝ Binary radix
1 0 1
      7 24 60 ⊤ 5090         ⍝ Days/Hours/Minutes mixed radix
3 12 50
      0 24 60 ⊤ 5090 6666   ⍝ Independent values in rarg.
 3    4
12   15
50    6
```

Matrix divide ⌹

Syntax: *res* ← *larg* ⌹ *rarg*

Description:
Solve a set of simultaneous equations.
larg, *rarg:* numeric scalars, vectors, or matrices; the rank of *rarg* must equal or exceed the rank of *larg;* if *rarg* is a matrix, the last dimension must not exceed the first.
res: the solution (or a least squares approximation if *rarg* has more rows than columns) of the matrix equation: *larg* ↔ *rarg* +.× *res*.

Example:
```
      14 26⌹2 2⍴1 3 4 2
5 3
      14 26 7⌹3 2⍴1 3 4 2 1 1
4.981481481  2.944444444
```

Matrix inverse ⌹

Syntax: *res* ← ⌹ *arg*

Description:
Calculate the inverse of a matrix.
arg: numeric scalar, vector, or matrix.
res: inverse of *arg* if *arg* is nonsingular and square. If *arg* is nonsingular but not square (must have more rows than columns), the result is the least squares approximation to the inverse of *arg*.

Example:
```
      ⌹ 2 2 ⍴ 1 1 2 3
 3 ¯1
¯2  1
```

Commute ⍨

Syntax: *res* ← *larg* f⍨ *rarg*

Description:
f is a dyadic function.
larg, rarg: any appropriate arrays (conforming).
res: The result of the function with the arguments reversed. (*larg* f⍨ *rarg* ↔ *rarg* f *larg)*
Note: The use of the commute operator with the / and \ primitive functions will result in a
NONCE ERROR. This limitation might be addressed in a future release. In the interim, this suggested workaround is to use the system functions, ⎕repl and ⎕expand, in place of / and \, respectively.

Example:
```
      10 - 7
3
      7 -⍨10
3
      10 minus 7      ⍝ minus is a user-defined function
3
      7 minus⍨ 10
3
      10 20 30 ∘.+ 1 2 3
 11 12 13
 21 22 23
 31 32 33
      10 20 30 ∘.+⍨ 1 2 3
 11 21 31
 12 22 32
 13 23 33
```

Boolean Functions

Boolean arrays consist of only two values, 0 and 1. You can think of these values as representing the absence or presence of a characteristic, no/yes, or false/true. A function that returns a Boolean array describes a relationship between the data arrays that are its arguments. The argument arrays can be of other data types.

Functions that use only Boolean arrays as arguments and also return a Boolean array are called Logical functions. You can also use Boolean arrays as arguments to non-Boolean functions. The results may not be Boolean. For example, you can sum a Boolean vector to find the number of occurrences.

Scalar Functions that Return a Boolean Array

Equal =

Syntax: *res* ← *larg* = *rarg*
res =← *rarg* ↔ *res* ← *res* = *rarg*

Description:
Scalar. Compare two arrays for equality.
larg, rarg: any arrays (conforming).
res: 1 for each pair of corresponding values where *larg* and *rarg* are equal; otherwise, 0.
ext: ⎕ct.

Example:
```
      'O'='COGNOS CORPORATION'
0 1 0 0 1 0 0 0 1 0 0 1 0 0 0 0 1 0
```

Not equal ≠

Syntax: *res* ← *larg* ≠ *rarg*

res ≠← *rarg* ↔ *res* ← *res* ≠ *rarg*

Description:
Scalar. Compare arrays for inequality.
larg, *rarg:* any arrays (conforming).
res: `1` for each pair of corresponding values where *larg* and *rarg* are not equal; otherwise, `0`.
ext: `⎕ct`.

Example:

```
          1 2 3 ≠ 2 1 3
1 1 0
```

Greater than >

Syntax: *res* ← *larg* > *rarg*

res >← *rarg* ↔ *res* ← *res* > *rarg*

Description:
Scalar. Compare two numeric arrays.
larg, *rarg:* any numeric arrays (conforming).
res: `1` for each pair of corresponding values where *larg* is greater than *rarg*; otherwise, `0`.
ext: `⎕ct`.

Example:

```
          1 2 3 > 2 1 3
0 1 0
```

Greater than or equal ≥

Syntax: *res* ← *larg* ≥ *rarg*

res ≥← *rarg* ↔ *res* ← *res* ≥ *rarg*

Description:
Scalar. Compare two numeric arrays.
larg, rarg: any numeric arrays (conforming).
res: `1` for each pair of corresponding values where *larg* is greater than or equal to *rarg*; otherwise, `0`.
ext: `⎕ct`.

Example:

```
          1 2 3 ≥ 2 1 3
0 1 1
```

Less than <

Syntax: *res* ← *larg* < *rarg*

res <← *rarg* ↔ *res* ← *res* < *rarg*

Description:
Scalar. Compare two numeric arrays.
larg, *rarg:* any numeric arrays (conforming).
res: `1` for each pair of corresponding values where *larg* is less than *rarg*; otherwise, `0`.
ext: `⎕ct`.

Example:

```
          1 2 3 < 2 1 3
1 0 0
```

Less than or equal ≤

Syntax: *res* ← *larg* ≤ *rarg*
res ≤← *rarg* ↔ *res* ← *res* ≤ *rarg*

Description:
Scalar. Compare two numeric arrays.
larg, *rarg:* any numeric arrays (conforming).
res: 1 for each pair of corresponding values where *larg* is less than or equal to *rarg*; otherwise, 0.
ext: ⎕ct.

Example:
```
      1 2 3 ≤ 2 1 3
1 0 1
```

Nonscalar Functions that Return a Boolean

Match ≡

Syntax: *res* ← *larg* ≡ *rarg*

Description:
Determine the equivalence of two arrays.
larg, *rarg:* any arrays.
res: 1 if both *larg* and *rarg* have the same rank, shape, and values; otherwise, 0.
ext: ⎕ct.

Example:
```
      'XYZZY' ≡ 1 5⍴'XYZZY'
0
      0 ≡ ,0
0
      A←2 3⍴⍳4
      A ≡ A
1
      A ≡ 'A'
0
```

Note: For monadic ≡, see Depth in the "General Functions" section of this chapter.

Mismatch ≢

Syntax: *res* ← *larg* ≢ *rarg*

Description:
Determine the non-equivalence of two arrays. (*larg* ≢ *rarg* ↔ ~ *rarg* ≡ *larg)*
larg, *rarg:* any arrays.
res: 1 if both *larg* and *rarg* does not have the same rank, shape, and values; otherwise, 0.
This function is also known as “not match” and “inequivalent”.
ext: ⎕ct.

Example:
```
     'bex' ≢ 'b','e','x'
0
      ⍬≢⍳0
0
      ''≢⍳0
1
```

Member of ∊

Syntax: *res* ← *larg* ∊ *rarg*

Description:
Compare the contents of two arrays.
larg , *rarg:* any arrays.
res: same size as *larg* and contains a `1` if *larg* item is found anywhere in *rarg*; otherwise, `0`.
ext: `⎕ct`.

Example:
```
      2 5 ∊ 1 2 3 4
1 0
```

Note: For monadic ∊, see Enlist in the "Structural Functions" section of this chapter.

Find ⍷

Syntax: *res* ← *larg* ⍷ *rarg*

Description:
Search for an array in another array.
larg, rarg: any arrays, nested or heterogeneous.
res: Boolean array of same shape as *rarg* with `1`s at locations of first element of copies of *larg*.
ext: `⎕ct`.

Example:
```
      'ISSI' ⍷ 'MISSISSIPPI'
0 1 0 0 1 0 0 0 0 0 0

      2 3 ⍷ 3 4⍴1 2 3
0 1 0 0
1 0 0 0
0 0 1 0

      (2 2⍴2 3 3 1)⍷3 4⍴1 2 3
0 1 0 0
1 0 0 0
0 0 0 0

      '' ⍷ 'ABCDE'
1 1 1 1 1

      (⊂1,⊂2 3)⍷¨'XYZ',(⊂1,⊂2 3),4,⊂2 3⍴1,⊂2 3
0 0 0  1 0  0   1 0 0
                0 1 0
```

Logical Functions

The functions described below not only return a Boolean array, they require Boolean arguments. In addition to these functions, you can also use the six scalar functions that return a Boolean array as logical functions if you use Boolean arrays as arguments.

AND ∧

Syntax: *res* ← *larg* ∧ *rarg*
res ∧← *rarg* ↔ *res* ← *res* ∧ *rarg*

Description:
Scalar. Logical AND of two Boolean arrays.
larg, *rarg:* any Boolean arrays (conforming).
res: `1` if both *larg* and *rarg* are `1`; otherwise, `0`.

Example:
```
      0 0 1 1 ∧ 0 1 0 1
0 0 0 1
```

OR ∨

Syntax: *res* ← *larg* ∨ *rarg*
res ∨← *rarg* ↔ *res* ← *res* ∨ *rarg*

Description:
Scalar. Logical OR of two Boolean arrays.
larg, *rarg:* any Boolean arrays (conforming).
res: `1` if either *larg* or *rarg* is `1`; otherwise, `0`.

Example:
```
      0 0 1 1 ∨ 0 1 0 1
0 1 1 1
```

NAND ⍲

Syntax: *res* ← *larg* ⍲ *rarg*
res ⍲← *rarg* ↔ *res* ← *res* ⍲ *rarg*

Description:
Scalar. Logical NAND of two Boolean arrays.
larg, *rarg:* any Boolean arrays (conforming).
res: `0` if both *larg* and *rarg* are `1`; otherwise, `1` (equivalent to ~ (*larg* ∧ *rarg*)).

Example:
```
      0 0 1 1 ⍲ 0 1 0 1
1 1 1 0
```

NOR ⍱

Syntax: *res* ← *larg* ⍱ *rarg*
res ⍱← *rarg* ↔ *res* ← *res* ⍱ *rarg*

Description:
Scalar. Logical NOR of two Boolean arrays.
larg, *rarg:* any Boolean arrays (conforming).
res: `1` if both *larg* and *rarg* are `0`; otherwise, `0`; (equivalent to ~ (*larg* ∨ *rarg*)).

Example:
```
      0 0 1 1 ⍱ 0 1 0 1
1 0 0 0
```

NOT ~

Syntax: *res* ← ~ *arg*

Description:
Scalar. Return the complement of a Boolean array.
arg: any Boolean array.
res: `1` for each item of *arg* that is `0`; `0` for each item that is `1`.

Example:
```
      ~ 0 1
1 0
```

Note: For dyadic ~, see Without in the "Structural Functions" section of this chapter.

Dyadic Scalar Functions with Axis

A scalar function calculates its result using one data element at a time, or one datum from the left argument with the corresponding datum from the right argument. Each element of the result is computed independently and depends solely on the corresponding left and right items. The entire result has the same shape as at least one of the arguments. In the basic form shown in the preceding sections, the two arguments to a dyadic scalar function must have the same shape, or one of the arguments must be a scalar value (scalar or one-element array). When you specify a scalar value, the function uses the scalar argument with each datum of the other argument.

Extending this concept, you can specify dyadic scalar functions with arguments of different ranks when the shape of the argument of lesser rank is a sub-array of the argument of greater rank. The function uses the smaller array with each sub-array of the other argument. The sub-arrays are conceptual; there is no additional nesting. This is known as dyadic scalar functions with axis. You specify the axes along which the arrays conform. The general specification for a dyadic scalar function with axis is:

Syntax: *res* ← *larg* f[*i*] *rarg*

Description:
`f` is a primitive dyadic scalar function:

+ − × ÷ ★ | ⍟ ○ ! ⌊ ⌈ < ≤ = ≥ > ≠ ∨ ∧ ⍱ ⍲

larg, rarg: any appropriate arrays (conforming), where the dimensions of the array of lesser rank match dimensions of the array of greater rank in the same order.
i is a scalar or vector, specifying without repetitions the axes of the argument of greater rank whose lengths match the axes of the argument of lesser rank. The number of elements of *i* must equal the rank of the argument of lesser rank.
res: An array whose shape matches the argument of greater rank.

Examples:
To add a vector to a matrix, the length of the vector must match one of the dimensions of the matrix. You specify that axis.

```
      v4←⍳4
      m24←2 4⍴10×⍳8
      m24
 10 20 30 40
 50 60 70 80

     v4+[2]m24
 11 22 33 44
 51 62 73 84
```

You specify the axis after the function. It applies to the argument of greater rank, regardless of which argument is on the left.

```
      m24-[2]v4
  9 18 27 36
 49 58 67 76
```

```
      m43←4 3⍴5×⍳12
      m43
 5 10 15
20 25 30
35 40 45
50 55 60
      m43×[1]v4
  5  10  15
 40  50  60
105 120 135
200 220 240
```

You can apply a dyadic scalar function to a vector and an array of any rank along any axis whose length matches the length of the vector. You can also apply dyadic scalar functions to arrays of higher ranks. The dimensions of the array of lesser rank must match the dimensions of some axes of the array of greater rank. The dimensions must match in the same order; thus, you can divide a three-dimensional array of shape `2 3 4` by a matrix of shape `3 4`, but you cannot divide it by a matrix of shape `4 3`.

However, you can specify the matching dimensions in any order. If you have an array `T234` of shape `2 3 4` and a matrix `M34` of shape `3 4`, the following statements are equivalent:

```
T234 ÷[2 3] M34
T234 ÷[3 2] M34
```

If you have an array `F3234` of shape `3 2 3 4` and a matrix `M34` of shape `3 4`, you could apply a scalar dyadic function in either of the following ways, since the shape of the matrix conforms to the shape of the array along two different pairs of axes:

```
F3234 ∧[1 4] M34
F3234 ∧[3 4] M34
```

Structural Functions

Structural functions reorganize data in arrays; select subsets of data in arrays; or select and reorganize data in arrays. These functions can change data by replacing items or by restructuring their relationships, such as by creating a nested array, but these functions do not recalculate an individual datum. Some functions create new data, for example, inserting fill items. You can recalculate data at the same time you reorganize them by using numeric functions in the arguments to structural functions. The descriptions in this section are alphabetical.

Catenate ⍪ or ,

Syntax: *res* ← *larg* ⍪ *rarg*
res ← *larg* , *rarg*
res ← *larg* ⍪[*i*] *rarg* or *res* ← *larg* ,[*i*] *rarg*
res ,← *rarg* or *res* ⍪← *rarg* (see Note 3)

Description:
Join two arrays along an axis specified by the index to the shape vector. The default axis for catbar (⍪) is along the first dimension; the default axis for the unadorned comma (,) is along the last dimension.

larg, *rarg:* any arrays (conforming). Conforming arrays for this function do not necessarily have the same shape. The two arrays can have different lengths along the axis you specify for catenation. You can even catenate arrays whose ranks differ by one, if the other dimensions are equal. The result array extends the extra dimension of the higher (rank) array. You can also catenate a scalar to an array of any rank.

i: scalar that indicates the dimension desired.

res: If *i* is an integer, an array consisting of the two arrays joined along the specified axis. If *i* is fractional, a new dimension is added to the shape vectors of *rarg* and *larg* in the position between ⌊*i* and ⌈*i*, and the function acts along that axis. Catenate with a fractional axis is also known as Laminate.

ext: ⎕io (for axis or dimension).

Example:

```
      2 3 5⍪99
```

```
2 3 5 99
      2 3 5,99
2 3 5 99

      'THREE'⍪2 5⍴'BLINDMICE '
THREE
BLIND
MICE

      (2 3⍴⍳6),2 2⍴33 333 66 666
1   2   3  33 333
4   5   6  66 666

      B←'HOW' ,[.5] 'NOW'
      B
HOW
NOW

      'HOW' ,[1.5] 'NOW'
HN
OO
WW
      ⎕io←0
      B←'HOW' ,[¯0.5] 'NOW'
      B
HOW
NOW
```

Note 1: *larg* ⍪ *rarg* ↔ *larg* ,[⎕io] *rarg*
larg , *rarg* ↔ *larg* ⍪[*i*] *rarg* (where *i* is ⍴⍴*rarg* and ⎕io is 1)

Note 2: Ravel (monadic ,) is also in this section.

Note 3: *res* ,← *rarg* ↔ *res* ← *res* , *rarg*
res ⍪← *rarg* ↔ *res* ← *res* ⍪ *rarg*
The syntax is similar to the in-place operations in other programming languages like C++ and C#. However, in APL64, besides the shorter notation, this can also provide significant performance gains particularly in cases that involve repetitive catentations with large arrays such as inside an iterative control structure (a :for loop).

Disclose ⊃

Syntax: *res* ← ⊃ *arg*
res ← ⊃[*i*] *arg*

Description:
Reduce a level of nesting from an array, raising its rank. (Evolution Level 2:)
arg: any array.
res: if *arg* is a nested scalar, it is expanded back to its enclosed array. Each nonscalar item of *arg* must have the same rank; a scalar item becomes the first item of an array of the same rank as the nonscalar items. Disclose pads the nested arrays along each dimension to conform to the item being disclosed that is largest along that dimension.

Disclose returns an array whose rank is one more than the common nonscalar rank. When no axis is specified, the first dimension has a length equal to the number of items, and the dimensions of the items of *arg* become the remaining dimensions of *res* in the same order.

For example, Disclose turns a nested vector of vectors into a matrix; each item becomes one row in the matrix. If *arg* is a nested vector of matrices, *res* is a three- dimensional array with as many planes as there are items; each plane contains as many rows as the nested matrix with the most rows and as many columns as the nested matrix with the most columns.

Example:
```
      A←(1 2 3) (2 4 6) (3 6 9)
      ]DISPLAY A
+→-------------------+
|+→----++→----++→----+|
||1 2 3||2 4 6||3 6 9||
|+~----++~----++~----+|
+∊-------------------+

      ⊃A
 1 2 3
 2 4 6
 3 6 9

      ⊃(1)(1 2)(1 2 3)
1 0 0
1 2 0
1 2 3
```

Description:
When you use Disclose with axis, you specify the axis along which to build an array from the items of the argument.
arg: a nested array whose items are scalars or arrays of the same rank.
i: non-negative integer scalar or vector that specifies the position in (⍴*res*) where the enclosed dimensions of the items of *arg* are to be placed. The number of elements in *i* must match the ranks of all the nonscalar enclosed items in *arg*. The system pads individual items of *arg* to the same shape before disclosing them.

Example:
```
      ⊃[1]'MARY' (⍳4) 'STEVE'
M 1 S
A 2 T
R 3 E
Y 4 V
  0 E
      ⊃[1 3](2 5⍴⍳10)(3 4⍴10×⍳12)
  1   2   3   4  5
 10  20  30  40  0

  6   7   8   9 10
 50  60  70  80  0

  0   0   0   0  0
 90 100 110 120  0

      ⍴⊃[2 3]M←2 3⍴⊂4 5⍴⍳20
2 4 5 3
      ⍴⊃[1 4]M
4 2 3 5
      ⍴⊃[4 1]M
5 2 3 4
```

Note 1: At Evolution Level `1`, monadic Disclose specifies the First function. See the descriptions of the First function (↑) in this section and `⎕mix` in the *System Functions Manual*.

Note 2: Pick (dyadic ⊃) is also in this section.

Drop ↓

Syntax: *res* ← *larg* ↓ *rarg*
res ← *larg* ↓[*i*] *rarg*

Description:
Exclude a set of elements from an array.
larg: any integer scalar or vector with one element per dimension of *rarg*.
rarg: any array.
res: all the elements of *rarg* except the subset specified by *larg*. *larg* specifies the number of elements in each dimension to exclude (starting from the end if *larg* is negative). If an element of *larg* is larger in magnitude than the corresponding dimension of *rarg*, *res* is empty along that coordinate. (Can be used in selective assignment.)

Example:
```
      5 ↓ 1 3 2 7 4 8
8
      A←2 3⍴1 2 3 4 5 6
      0 ¯1 ↓ A
1 2
4 5
```

Description:
When you use Drop with axis, you can specify some or all of the axes of *rarg*; the number of elements of *larg* must equal the number of axes specified in *i*.
i: integer scalar or vector specifying the axes of *rarg* along which to drop elements; the values in *i* must be unique and less than or equal to the rank of *rarg*.
res: all the elements of *rarg* except the subset specified by *larg*. The shape along axes not specified in *i* remains unchanged.

Example:
```
      B←2 2 3⍴ 1 2 3 4 5 6 7 8 9 10 11 12
      2 1 ↓ [3 2]B
 6

12
```

Enclose ⊂

Syntax: *res* ← ⊂ *arg*
res ← ⊂[*i*] *arg*

Description:
Create a nested scalar from any array that is not a simple scalar.
arg: any array.

Example:
```
      C←'A' 'MM' 'SSS'
      ⍴C
3
    C[2]←⊂2 2 ⍴⍳4
      C
A    1 2  SSS
     3 4
      ]DISPLAY C
.→------------.
|   .→--. .→--.|
|A ↓1 2| |SSS||
|- |3 4| '---'|
|   '~--'      |
'∊------------'
```

Description:
When you use Enclose with axis, you can specify some or all of the axes of an array, yielding a result of lower rank than the right argument that contains the same data. Enclose with Axis is similar to `⎕split` but is more general in function.
arg: any array.
i: non-negative integer scalar or vector that indicates the axes of the arguments to be enclosed.

Example:
```
      A←3 4⍴⍳12
      ⊂[1]A
 1 5 9  2 6 10  3 7 11  4 8 12
      ⊂[2]A
 1 2 3 4  5 6 7 8  9 10 11 12
      (⊂[1 2]A)≡⊂A
1
      (⊂[2 1]A)≡⊂⍉A
1

      B←2 3 4⍴⍳24
      ⊂[1 3]B
 1  2  3  4     5  6  7  8     9 10 11 12
13 14 15 16    17 18 19 20    21 22 23 24

 C←⊂[1 2 3]B
      ⍴B
2 3 4
      ⍴C

      ≡B
1
      ≡C
2
```

Note 1: See the description of `⎕penclose` for the Evolution Level `1` dyadic ⊂, Partitioned enclose.

Note 2: Partition (dyadic ⊂) is also in this section.

Enlist ∊

Syntax: *res* ← ∊ *arg*

Description:
Convert an array into a simple vector. (Evolution Level `2`.)
arg: any array.
res: a simple vector comprising the simple items of *arg*, arranged in depth first, row-major order. (Can be used in selective assignment.)

Example:
```
      A←'MARY' (2 3⍴⍳6) 0 'JOE'
      ]DISPLAY A
.→-----------------------.
| .→---. .→-----.   .→--. |
| |MARY| ↓ 1 2 3| 0 |JOE| |
| '----' | 4 5 6|   '---' |
|        '~-----'         |
'∊------------------------'

      ]DISPLAY ∊A
.→--------------------.
|MARY 1 2 3 4 5 6 0 JOE|
'+--------------------'
```

Note 1: See the description of `⎕enlist` in the *System Functions Manual* for the Evolution Level `1` monadic `∊`, Enlist.

Note 2: For dyadic `∊`, see the description of the Member of function in the "Boolean Functions" section of this chapter.

First ↑

Syntax: *res*← ↑ *rarg*

Description:
Return the first item of an array. (Evolution Level `2`.)
rarg: any array.
res: if *rarg* is a nested vector, the first item is selected and expanded into an array. If *rarg* is empty, *res* is the prototype of *rarg*. (See the technical note under the description of Reshape, below.)

Example:

```
      C←'ONE' (2 3 4 5)
      ↑C
ONE
      ⍴↑C
3

 D←0 3⍴ ⊂ 2 3⍴⍳6
      ↑D
0 0 0
0 0 0
```

Note 1: At Evolution Level `1`, the monadic up arrow specifies the Mix function. See the description of `⎕first` in the *System Functions Manual*.

Note 2: Take (dyadic `↑`) is also in this section.

Indexing ⌷

Syntax: *res* ← *larg* ⌷ *rarg*
res ← *larg* ⌷[*i*] *rarg*

Description:
Selects a subset of *rarg* by using the index values specified in *larg* along each axis.
rarg: any array
larg: a nested vector or scalar of depth two, or a simple vector or scalar, containing indices to the dimensions of *rarg*. The length of *larg* must equal the rank of *rarg*, except that if *rarg* is a vector, *larg* can be either a one-element vector or a scalar. The values of *larg* must be nonnegative integers less than or equal to the rank of *rarg*.
res: the elements of *rarg* specified by their positions in *larg*. (Can be used in selective assignment.)
ext: `⎕io`.

Examples:

```
      v← 'a' 2 'c' 4 5
      3 ⌷ v
c
      (⊂2 4) ⌷ v  ⍝ Note that (⍴ larg) = ⍴⍴ rarg
2 4
      v[2 4]
2 4

      ''≡(⊂⍳0) ⌷ v
1
      ⍬≡(⊂⍳0) ⌷ (,9)
1
      (⍳0) ⌷ 9
```

```
9

           m
11 12 13 14
21 22 23 24
31 32 33 34

      (2 3) ⌷ m
23
      (2 3) (1 4) ⌷ m
21 24
31 34
      (3 2) 1 ⌷ m
31 21
      (3 2) (,1) ⌷ m
31
21
      ⍴(⍳0) (⍳0) ⌷ m
0 0
```

Description:
If you use the indexing function with axis, it is similar to bracket indexing when you omit the index for one or more axes. In this case, the length of the left argument must equal the number of axes you specify.
(Can be used in selective assignment.)

```
      three_D_array
 1  2  3  4
 5  6  7  8
 9 10 11 12

 13 14 15 16
 17 18 19 20
 21 22 23 24

      1 ⌷[2] three_D_array
 1  2  3  4
13 14 15 16

      three_D_array[;1;]
 1  2  3  4
13 14 15 16

      (1 2) (3 4) ⌷[2 3] three_D_array
 3  4
 7  8

15 16
19 20
```

Partition ⊂

Syntax: *res* ← *larg* ⊂ *rarg*
res ← *larg* ⊂[*i*] *rarg*

Description:
Create a nested array from portions of another array. (Evolution Level 2.)
larg: a scalar or vector of non-negative integers that specifies the partioning of *rarg* along the specified axis; the default axis is the last dimension. If *larg* is a vector, its length must equal the dimension of the selected axis of *rarg*. If an element of *larg* is zero, the corresponding element from *rarg* is not included; if a non-zero element of *larg* is greater than the element that precedes it, the function starts a new partition. If the element is less than or equal to its predecessor, the function nests the corresponding element with its predecessor or predecessors. A scalar *larg* is equivalent to a vector of the appropriate length with all elements having the same value.
rarg: any nonscalar array.
i: non-negative scalar that indicates the dimension desired.
res: a nested array of the elements of *rarg*, excluding those corresponding to zero elements of *larg*. The pattern of nesting is determined by elements of *larg* that are greater than the preceding element. The rank of *res* is the same as the rank of *rarg*, but it is nested one level deeper. The items of res are scalars or sub-vectors of the vectors of *rarg* along the selected axis.

Example:
```
      ]display (10⍴1 2 3)⊂⍳10
.→---------------------------.
|.→..→..→--..→..→--..→..→---.|
||1||2||3 4||5||6 7||8||9 10||
|'~''~''~--''~''~--''~''~---'|
'∊---------------------------'

      3 2⍴⍳6
 1 2
 3 4
 5 6

      2 2 3 ⊂[1] 3 2⍴⍳6
.→---------.
↓.→--..→--.|
||1 3||2 4||
|'~--''~--'|
|.→.  .→.  |
||5|  |6|  |
|'~'  '~'  |
'∊---------
```

Note 1: See the description of ⎕penclose in the *System Functions Manual* for the Evolution Level 1 dyadic ⊂, Partitioned enclose. This system function uses a different format in the left argument to produce a similar result.

Note 2: Enclose (monadic ⊂) is also in this section.

Pick ⊃

Syntax: *res* ← *larg* ⊃ *arg*

Description:
Select a portion of an array.
rarg: any array.
larg: positive integers that describe (left to right) how deep into *arg* to go to select an item.
res: a subset of *arg* specified by *larg*. (Can be used in selective assignment.)
ext: ⎕io.

Example:
```
      A←'ONE' (2 2⍴⍳4) 'SIX'
      ]DISPLAY A
.→---------------.
|.→--. .→--. .→--.|
||ONE| ↓1 2| |SIX||
|'---' |3 4| '---'|
|      '~--'      |
'∊----------------'
      ρA
3
      2⊃A
1 2
3 4
      3 2⊃A
I
      (2 (2 1))⊃A
3
```

Note: Disclose (monadic ⊃) is also in this section.

Ravel ,

Syntax: *res* ← , *arg*
res ← ,[*i*] *arg*

Description:
Change an array into a vector.
arg: any array.
res: all the elements of *arg* in the same order as *arg*, but as a vector. (Can be used in selective assignment.)

Example:

```
      ,99
99
      ρ ,99
1
      ,2 4ρ2 3 5
2 3 5 2 3 5 2 3
```

Description:
When you use Ravel with axis, you can change an array to greater or lesser rank while keeping the same elements in the same order.
i: fractional scalar or an integer vector of ascending contiguous axes of *arg*. If *i* is fractional, the rank of *res* is increased by one. A new unit dimension is added to the shape vector of *res* in the position between ⌊*i* and ⌈*i*. If *i* is a vector, the specified axes of *arg* are combined into a single axis of *res*. The shape of *res* is the shape of *arg* with the specified dimensions replaced by their product.
res: the set of *arg* elements with a shape determined by *i*. (Can be used in selective assignment.)
ext: ⎕io (for dimension).

Example:

```
      B←2 2 3ρ 1 2 3 4 5 6 7 8 9 10 11 12
      , [2 3]B
 1 2 3  4  5  6
 7 8 9 10 11 12
       ρ, [1.5]B
 2 1 2 3
```

Note: Catenate (dyadic ,) is also in this section.

Reshape ⍴

Syntax: *res* ← *larg* ⍴ *rarg*

Description:
Create an array of specific shape and content.
larg: non-negative integer scalar or vector.
rarg: any array.
res: the elements of *rarg* selected in ravel order and formed into the new shape specified by *larg*. Some *rarg* elements may be lost (*res* will have fewer elements than *rarg*) or duplicated (*res* will have more elements than *rarg*) as needed. (Can be used in selective assignment.)

Example:

```
      3 ⍴ 99
99 99 99
      2 4 ⍴ 2 3 5
2 3 5 2
3 5 2 3
      2 3⍴1 1 2⍴7 8
7 8 7
8 7 8
```

Note: For monadic ⍴, see Shape in the "General Functions" section of this chapter.

Technical Note: Empty Arrays, Type, Prototype, and Fill Items

If you use zero as the left argument to Reshape, the result is an empty array. An empty array has no elements, but it does have a structure whose shape may be complex and deeply nested. To understand the structure, you must know that non-empty arrays have a "type." The type of an array has the same shape and depth as the array with numbers replaced by zeroes and characters replaced by spaces. You can generate the type of an array using the system function `⎕type`; since it may contain spaces, it is useful to use the `]display` user command to understand the structure.

The prototype of an array is the type of its first item (where item is one-level less nested than element). The prototype is the value that is enclosed and stored, nested, in an empty array, and it is this prototype that gives the empty array its structure. (Note that empty arrays can require a great deal of memory.)

The same value is used as the fill item in certain functions, for example, when you Overtake; that is, when you use the Take primitive function with a left argument that is greater than the length of the right argument. However, note that the empty array you create with `0 ⍴` *vector* does not match the value of the last element of a vector created with overtake. However, if you apply the First primitive function to the empty array, that result matches the last item of the same vector. Thus, in a somewhat circular definition, fill item is defined in language terms as `↑0 ⍴` *arg* .

```
      ⍴vec
2
      over←3↑vec
      over[3]≡⎕type vec[1]
1
      (0⍴vec)≡⎕type vec[1]
0
      (↑0⍴vec)≡⎕type ↑vec
1
      (↑ ⎕type vec) ≡ ⎕type ↑vec
1
```

If you create empty arrays of greater rank, such as by `0 3 ⍴ vec`, there is only one prototype. The exact content of the prototype will be of importance only when you materialize it, such as by using Overtake. The shape of the empty array may be extremely important for it to be conforming.

Reverse ⊖ or ⌽

Syntax: *res* ← ⊖ *arg*
res ← ⌽ *arg*
res ← ⊖[*i*] *arg* or *res* ← ⌽[*i*] *arg*

Description:
Reverse the order of the elements of an array array along an axis specified by the index to the shape vector. The default dimension for ⊖ is the first dimension; the default dimension for ⌽ is the last dimension.
arg: any array.
i: non-negative, integer-valued scalar that indicates the dimension desired.
res: the elements in *arg* reversed along the *i*th dimension. (Can be used in selective assignment.)
ext: ⎕io.

Example:
```
      ⌽ 'RATS LIVE ON'
NO EVIL STAR
      ⊖ 'RATS LIVE ON'
NO EVIL STAR

      A←3 3⍴'ABCDEFGHI'
      ⊖A
GHI
DEF
ABC
      ⌽A
CBA
FED
IHG
      ⌽[1]A
GHI
DEF
ABC
```

Note: ⊖ *arg* ↔ ⌽[⎕io] *arg*
⌽ *arg* ↔ ⊖[*i*] *arg* (where *i* is ⍴⍴*rarg* and ⎕io is 1)

Rotate ⊖ or ⌽

Syntax: *res* ← *larg* ⊖ *rarg*
res ← *larg* ⌽ *rarg*
res ← *larg* ⊖[*i*] *rarg* or *res* ← *larg* ⌽[*i*] *rarg*

Description:
Shift the elements of *rarg* toward the beginning (or end) of one dimension maintaining the same relative order (as if on a chain). When an element is displaced from the beginning, it rotates to become the last element of the dimension; when an element is displaced from the end, it rotates to become the first element of the dimension. The index to the shape vector specifies the dimension along which the elements rotate. The default dimension for ⊖ is the first dimension; the default dimension for ⌽ is the last dimension. *larg* specifies the direction and the number of positions. If *larg* is positive, elements shift toward the beginning; if *larg* is negative, the elements shift toward the end. The magnitude of *larg* specifies the number of positions to shift.
larg: integer scalar or vector of length equal to the chosen dimension of *rarg*.
rarg: any array.
i: non-negative, integer-valued scalar that indicates the dimension desired.
res: the elements in *rarg* rotated *larg* places along the *i*th dimension. (Can be used in selective assignment.)
ext: ⎕io (for dimension).

Example:
```
      2 ⊖ 'TODAY'
DAYTO
```

```
      2 ⌽ 'TODAY'
DAYTO
      B←3 4⍴⍳12
      B
1  2  3  4
5  6  7  8
9 10 11 12

      1 ⊖ B
5  6  7  8
9 10 11 12
1  2  3  4

      1 ⌽ B
 2  3  4 1
 6  7  8 5
10 11 12 9
      1 2 ¯3 ⌽ [2]B
 2  3  4  1
 7  8  5  6
10 11 12  9
```

Note: *larg* ⊖ *rarg* ↔ *larg* ⌽[⎕io] *rarg*
larg ⌽ *rarg* ↔ *larg* ⊖[*i*] *rarg* (where *i* is ⍴⍴*rarg* and ⎕io is 1)

Take ↑

Syntax: *res* ← *larg* ↑ *rarg*
res ← *larg* ↑[*i*] *rarg*

Description:
Select a set of elements from an array.
larg: any integer scalar or vector with one element per dimension of *rarg*.
rarg: any array.
res: the subset of *rarg* elements. *larg* specifies the shape of *res*. If *larg* is negative, the selection starts from the end rather than the beginning; if *larg* specifies an array larger than *rarg, res* is padded with the fill item (⎕type ↑ *arg*; blank or 0 for simple arrays). (Can be used in selective assignment.)

Example:
```
      2 ↑ 3 6 2
3 6
      5 ↑ 3 6 2
3 6 2 0 0
      ¯3 2 ↑ 2 3⍴1 2 3 4 5 6
0 0
1 2
4 5
```

Description:
When you use Take with axis, you can specify some or all of the axes of *rarg*; the number of elements of *larg* must equal the number of axes specified in *i*.
i: integer scalar or vector specifying the axes of *rarg* along which to select items; the values in *i* must be unique and less than or equal to the rank of *rarg*.
res: the subset of *rarg* elements. The shape along axes not specified in *i* remains unchanged.

Example:
```
      B←2 2 3⍴ 1 2 3 4 5 6 7 8 9 10 11 12
      2 1 ↑ [3 2]B
 1 2

 7 8
```

Note: First (monadic ↑) is also in this section.

Transpose ⍉

Syntax: *res* ← ⍉ *arg*

Description:
Reverse the axes of an array.
arg: any array.
res: *arg* with dimensions interchanged. (Can be used in selective assignment.)

Example:
```
      B
1  2  3  4
5  6  7  8
9 10 11 12
      ⍉B
1 5  9
2 6 10
3 7 11
4 8 12
      ⍴⍉B
4 3
```

Transpose (Dyadic) ⍉

Syntax: *res* ← *larg* ⍉ *rarg*

Description:
Select all the data or a cross section of data from, and optionally reorder the axes of, an array.
larg: non-negative, dense,* integer vector of length equal to rank of *rarg*.
rarg: any array of dimension 2 or higher; (vectors or scalars do not cause errors, but the results are trivial).
res: *rarg* with the dimensions interchanged (and optionally collapsed) in the order specified by *larg*.
(Can be used in selective assignment.)
ext: ⎕io.

*In this context, *dense* means you cannot skip any numbers starting with ⎕io. If *rarg* is a 3-dimensional array and ⎕io is 1, then *larg* can be any of the six permutations of (1 2 3), any of the three permutations of (1 1 2), any of the three permutations of (1 2 2), or (1 1 1). It cannot be (1 1 3) or (2 2 2).

If *larg* selects all the axes of *rarg*, that is, if there are no duplicate values in *larg*, *res* contains all the elements of *rarg*. The data are arranged with the axes specified by *larg*. The position of each element of *larg* corresponds to the same position in the shape vector of *rarg*; the value of an element of *larg* specifies its position in the shape vector of *res*. Thus, the last element of *larg* corresponds to the number of columns of *rarg*; the value of the last element of *larg* specifies where that number is located in the shape vector of the result array. For an n-dimensional array, if the last element of *larg* has the value n-1, there will be as many rows in *res* as there are columns in *rarg*. If the last element of *larg* has the value n-2, there will be as many planes in *res* as there are columns in *rarg*. The entire shape vector of the result array can be represented as:
⍴ *res* ↔ (⍴ *rarg*)[⍋ *larg*]. In the examples below, note that the values of the elements of the array B are also the index to their respective positions in the original array.

Example:
```
C←(100×⍳2)∘.+(10×⍳3)∘.+⍳4
      ⍴ C                         ⍴ 1 3 2 ⍉ C                ⍴ 2 3 1 ⍉ C
2 3 4                       2 4 3                      4 2 3
      C                           1 3 2 ⍉ C                  2 3 1 ⍉ C
 111 112 113 114             111 121 131                111 121 131
 121 122 123 124             112 122 132                211 221 231
 131 132 133 134             113 123 133
                             114 124 134                112 122 132
 211 212 213 214                                        212 222 232
 221 222 223 224             211 221 231
 231 232 233 234             212 222 232             113 123 133
                             213 223 233             213 223 233
                             214 224 234
                                                     114 124 134
                                                     214 224 234
```

If *larg* has duplicate values, *res* contains a subset of the data of *rarg*, with two or more axes of *rarg* mapped onto a single axis of *res*; you can think of this as a diagonal cross section of *rarg*. The elements of data that are selected are those whose position in *rarg* has duplicate values in the corresponding positions of its index.

```
      ⍴ 2 1 2 ⍉ C
3 2
      2 1 2 ⍉ C
 111 212
 121 222
 131 232
```

The result array when *larg* is all ones is the major diagonal; that is, all the elements whose positions in *rarg* are represented by an index of equal values.

```
      1 1 1 ⍉ C
111 222
```

The shape vector of *res* when *larg* contains duplicates will include the smaller (or smallest) values that correspond to the duplicate values in *larg*.

```
      ⍴ 1 2 1 ⍉ C
2 3
      ⍴ 1 2 2 ⍉ C
2 3
      ⍴ 2 2 1 ⍉ C
4 2
```

Unique ∪

Syntax: *res* ← ¡ *arg*

Description:
Select the unique elements of a vector.
arg: any vector.
res: a compression of *arg* with all but the first instance of each distinct element removed.
ext: ⎕ct.

Example:

```
      ¡ 'abracadabra'
abrcd
      ¡ 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3
3 1 4 5 9 2 6 8 7
      ¡ 'George' 'John' 'Thomas' 'James' 'James' 'John' 'Andrew'
George John Thomas James Andrew
```

Without ~

Syntax: *res* ← *larg* ~ *rarg*

Description:
Set difference.
larg: any scalar or vector.
rarg: any array.
res: vector that contains those elements of *larg* that do not occur anywhere in *rarg*. *ext:* ⎕ct.

Example:

```
      1 1 2 2 3 3 4 4~4 2 4
1 1 3 3
      R←'AEIOU'
      L←'ANY ARRAY'
      L~R
NY RRY
      L~L~R
AAA
```

Note: For monadic ~, see Not in the "Boolean Functions" section of this chapter.

Notes on Obsolete Functions

APL64 operates by default at Evolution Level 2. Certain functions and language features had different behaviors at Evolution Level 1. If you are converting applications from older systems, you can generate the same behavior at Evolution Level 2 by using the replacement functions named or the syntax shown below to produce the Evolution Level 1 behavior. See the description of `)evlevel` in the System Commands chapter in this manual for more information.

Dyadic ⊂ at Evolution Level 2 is the Partition Function. For Evolution Level 1, use `⎕penclose`.
Monadic ↓ (Split) generates a `SYNTAX ERROR`. Use `⎕split`.
Monadic ∊ at Evolution Level 2 is the Enlist function. For Evolution Level 1, use `⎕enlist`.
Monadic ↑ at Evolution Level 2 is the First function. For Evolution Level 1, use `⎕first`.
Monadic ⊃ at Evolution Level 2 is the Disclose function. For Evolution Level 1, use `⎕mix`.

The expressions `A B C[2]` and `A B C←X Y Z` generate an `EVOLUTION ERROR`. Use parentheses around the left argument to use this syntax: `(A B C)[2]` and `(A B C)←X Y Z` are both valid expressions.

At Evolution Level 1, there was a difference in the result for appropriate arguments between the expressions *larg* `/¨` *rarg* and *larg* `(/)¨`*rarg*. At Evolution Level 2, the syntax `(/)¨` generates an `EVOLUTION ERROR`. Use `⎕repl¨` to emulate the parenthesized slash.

Similarly, the backslash in parentheses, `(\)¨`, generates an `EVOLUTION ERROR`. Use `⎕expand¨` to emulate the parenthesized backslash.

If you are not converting applications from earlier systems, these changes are irrelevant.

General Functions

Functions that Return a Property

Signum ×

Syntax: *res* ← × *arg*

Description:
Scalar. Determine the sign of a number.
arg: any numeric array.
res: ¯1 if *arg* is negative, 0 if *arg* is 0, and 1 if *arg* is positive.

Example:

```
      × 3 0 ¯0.5
1 0 ¯1
```

Shape ⍴

Syntax: *res* ← ⍴ *arg*

Description:
Return the shape of an array.
arg: any array.
res: a vector containing the length of each dimension of *arg*.

Example:

```
      ⍴ 2 3 5
3
      ⍴ 2 3 5 ⍴⍳30
2 3 5
      ⍴ 99
      ⍴⍴ 99
0
```

Depth ≡

Syntax: *res* ← ≡ *arg*

Description:
Levels of nesting in an array.
arg: any array.
res: the number of times you must use Pick (dyadic ⊃) to extract the most deeply nested simple scalar from *arg*.

Example:

```
      ≡3.3
0
      ≡1 2 3
1
      ≡''
1
      ≡ 2 3 4⍴⍳24
1
      ≡(1 2) (2 3) 'AB'
2
      ≡⊂⊂⊂⊂⊂12 12
6
```

Functions that Return an Index

Index of ⍳

Syntax: *res* ← *larg* ⍳ *rarg*

Description:
Find the location of items in an array.
larg: any vector.
rarg: any array.
res: for each item of *rarg*, the corresponding element of *res* is the index of the item's first occurrence in *larg*. If *larg* does not contain the item, the *res* element is ⎕io+⍴*larg*.
ext: ⎕io, ⎕ct.

Example:

```
      A←3 4 7 3 8
      A ⍳ 7 4 3 12
3 2 1 6

      'ABCDEF' ⍳ 2 5⍴'DEAD BEEF'
4 5 1 4 7
2 5 5 6 7
```

Numeric grade up ⍋

Syntax: *res* ← ⍋ *arg*

Description:
Return the ascending sort order of a numeric array.
arg: any numeric nonscalar array.
res: a numeric vector containing the indices that arrange the elements of *arg* in ascending numeric order. The length of *res* is the same as the first dimension of *arg*. If *arg* is a vector, you can use *res* as a subscript vector. Duplicate values retain their original relative positions.
ext: `⎕io`.

If *arg* is a matrix, the result is formed by considering one column at a time, working from left to right. Grade up sequences the leftmost column as a vector. If the vector has no duplicate values, its ordering becomes the result. If the vector has duplicate values, the elements from the next column to the right are used in an attempt to sequence the duplicates. This process continues until either all duplications are resolved or all columns are used.

Arguments of more than two dimensions are treated as matrices, retaining the original first dimension and combining all the other dimensions into a single second dimension. In effect, the argument is treated as being reshaped as follows:

```
      ((1↑⍴arg) , ×/1↓⍴arg) ⍴ arg
```

Example:

```
      A←5 2 8
      ⍋A
2 1 3
      A[⍋A]
2 5 8
      Y← 3 4⍴ 1 12 25 6 1 15 11 7 1 12 25 5
      Y
1 12 25 6
1 15 11 7
1 12 25 5
      ⍋Y
3 1 2
```

Character grade up ⍋

Syntax: *res* ← *larg* ⍋ *rarg*

Description:
Return the ascending sort order of a character array.
larg, *rarg:* any character nonscalar arrays; *larg* is `⎕av` by default.
res: a numeric vector containing the indices that arrange the elements of *rarg* in ascending order according to the collating sequence specified by *larg*. The length of *res* is the same as the first dimension of *rarg*. If *rarg* is a vector, you can use *res* as a subscript vector.
ext: `⎕io`.

Character grade up associates a numeric value with each character in the right argument based on the elements of *larg*. The rules of Numeric grade up are then applied to the associated numeric values to produce the result. If the left argument is a vector, then the associated numeric values are equivalent to those produced by dyadic iota. Specifically, `V⍋A` is equivalent to `⍋V⍳A`.

For left arguments of rank `2` or greater, each dimension is used independently, working from the last to the first. The numeric ordering value for a character of *rarg* is the lowest coordinate index along the specified dimension of an occurrence of the character in *larg*. If the character does not appear in *larg*, its ordering value is determined like that of dyadic iota. Thus, the initial ordering value of a character is the same as the first column of *larg* in which it appears. If this ordering contains no duplications, it is used. If duplicates exist, their ordering values are sequenced with respect to the next dimension. This process continues until all duplications are resolved or all dimensions of the left argument are exhausted.

If the following matrix is the left argument

```
       ABCDEFGHIJKLMNOPQRSTUVWXYZ
       abcdefghijklmnopqrstuvwxyz
```

the initial ordering using the last dimension results in A and a coming before B and b, and so on. If both A and a are in the right argument, they are duplicates since they have identical coordinate values along the last dimension. A second evaluation, using the first dimension, places A before a.

Example:

```
      'ABC' ⍋ 'CAB'
2 3 1
      A←3 4⍴'FOURFIVESIX '
      A
FOUR
FIVE
SIX
      ⍋A
2 1 3
      A[⍋A;]
FIVE
FOUR
SIX
```

The example below uses three collating sequences (each starting with a blank) to produce the results shown in the table below.

Collating Sequence 1:	` abcdefABCDEF`
Collating Sequence 2:	` aAbBcCdDeEF`

Collating Sequence 3:	` abcdefg`
	` ABCDEFG`

Sorting with Collating Sequences

Original Data	Sort with Collating Sequence 1	Sort with Collating Sequence 2	Sort with Collating Sequence 3
Aba	aar	aar	aar
aba	aba	aba	aba
Deaf	abba	abba	Aba
babe	babe	Aba	ABA
deed	bead	ABA	abba
CCC	bA	babe	bA
Deed	deaf	bA	babe
deaf	deed	bead	Bach
bA	Aba	Bach	bead
bead	ABA	CCC	CCC
abba	Bach	deaf	deaf
ABA	CCC	deed	Deaf
Bach	Deaf	Deaf	deed
aar	Deed	Deed	Deed

Numeric grade down ⍒

Syntax: *res* ← ⍒ *arg*

Description:
Return the descending sort order of a numeric array.
arg: any numeric nonscalar array.
res: a numeric vector containing the indices that arrange the elements of *arg* in descending numeric order. The length of *res* is the same as the first dimension of *arg*. If *arg* is a vector, you can use *res* as a subscript vector. Duplicate values retain their original relative positions. See the description of Numeric grade up in this chapter for more information.
ext: ⎕io.

Example:
```
      A←37 9 18
      ⍒A
1 3 2
      A[⍒A]
37 18 9
```

Character grade down ⍒

Syntax: *res* ← *larg* ⍒ *rarg*

Description:
Return the descending sort order of character array.
larg, *rarg:* any character nonscalar arrays; *larg* is ⎕av by default.
res: a numeric vector containing the indices that arrange the elements of *rarg* in descending order according to the collating sequence specified by *larg*. The length of *res* is the same as the first dimension of *rarg*. If *rarg* is a vector, you can use *res* as a subscript vector. See the description of Numeric grade down in this chapter for more information.
ext: ⎕io.

Example:
```
      ⍒'CAB'
1 3 2
```

Miscellaneous Functions

Execute ⍎

Syntax: ⍎ *arg*
res ← ⍎ *arg*

Description:
Execute a character string representation of an APL expression.
arg: character scalar or vector.
res: the result, if any, generated by executing the expression.

Example:
```
      ⍎ '2+3'
5
      N←7
      'V',(⍕N),'←10×⍳',⍕N
V7←10×⍳7   ⍝ (string is displayed)

      V7
VALUE ERROR
      V7
      ^
      ⍎'V',(⍕N),'←10×⍳',⍕N
 ⍝  (string is executed; no visible output)
```

```
      V7
10 20 30 40 50 60 70
```

Format ⍕

Syntax: *res* ← ⍕ *arg*

Description:
Represent numeric data in character form.
arg: any array.
res: *arg* represented in character form.
ext: ⎕pp.

Example:

```
      ⍕ 2 3⍴1 2 3 4 5 6
 1 2 3
 4 5 6
      ⍴ ⍕ 2 3⍴1 2 3 4 5 6
2 6
      'REDUNDANT'≡⍕'REDUNDANT'
1
```

Pattern format ⍕

Syntax: *res* ← *larg* ⍕ *rarg*

Description:
Represent numeric data in character form, formatting the result.
larg: integer scalar or vector of pairs; a single pair is replicated as with scalar extension. The first number of each pair specifies the field width for the column; 0 requests a field large enough to accommodate the largest number. The second number specifies the number of decimal places. If the second number is negative, the result is formatted in exponential notation. A pair of numbers for each column specifies different formatting for each column. If only one number is specified, it is assumed to be the number of decimal places.
rarg: any numeric array.
res: a character representation of *rarg* formatted as specified by *larg*.

Example:

```
      1 0 ⍕ 2 3 5
235
      1 ⍕ 2 3 5
 2.0 3.0 5.0
      1 0 4 1 6 2 ⍕ 2 3⍴⍳ 6
1 2.0  3.00
4 5.0  6.00
```

Operators

An operator takes one or two operands and derives a function. The derived function takes one or two arguments and produces a result. The results described below are those of the derived functions, not of the operators themselves.

Expand ⍀ or \

Syntax: *res* ← *oparray* ⍀ *arg*
res ← *oparray* \ *arg*
res ← *oparray* ⍀[*i*] *arg* or *res* ← *oparray* \[*i*] *arg*

Description:
Expand *arg* by inserting fill items in a pattern specified by *oparray*. The fill items become new elements of the dimension specified by the index to the shape vector. The default dimension for ⍀ is the first dimension; the default dimension for \ is the last dimension.
oparray: Boolean vector whose sum equals the length of the chosen dimension of *arg*.
arg: any array; if the chosen dimension has length 1, *arg* is extended along the expansion axis to the number of positive elements in the operand (+/*oparray*). Note that this allows you to generate only fill elements for an operand consisting of zeroes and an argument that is a scalar or has length 1 along the chosen dimension.
i: non-negative, integer-valued scalar that indicates the dimension desired.
res: the array *arg* expanded by adding an additional blank or zero for each corresponding 0 in *oparray*. (Can be used in selective assignment.)
ext: ⎕io (for dimension).

Examples:
```
      0 0 1 0 1 \ 7 8
0 0 7 0 8
      1 0 1 ⍀ 2 3⍴⍳6
1 2 3
0 0 0
4 5 6
      1 0 1 \ 4
4 0 4
      1 0 1 ⍀ 1 3⍴⍳3
1 2 3
0 0 0
1 2 3
      0 0 \ 5
0 0
      0 0 ⍀ 1 4⍴⍳3
 0 0 0 0
 0 0 0 0

      A←2 3⍴'ABCDEF'
      A
ABC
DEF
      1 0 1 ⍀ A
ABC

DEF
      1 0 0 1 0 1\A
A  B C
D  E F
```

Note 1: *oparray* ⍀ *arg* ↔ *oparray* \[⎕io] *arg*
oparray \ *arg* ↔ *oparray* ⍀[*i*] *arg* (where *i* is ⍴⍴*arg* and ⎕io is 1)

Note 2: See also Scan (\) in this section .
See also ⎕expand in the *System Functions Manual*.

Replicate (compress) ⌿ or /

Syntax: *res* ← *oparray* ⌿ *arg*
res ← *oparray* / *arg*
res ← *oparray* ⌿[*i*] *arg* or *res* ← *oparray* /[*i*] *arg*

Description:
Replicate the elements of an array using all the elements of the dimension specified by the index to the shape vector. You can replicate zero times, which eliminates an element from *res*, and you can insert fill elements. The default dimension for ⌿ is the first dimension; the default dimension for / is the last dimension.
oparray: integer scalar, or integer vector where the number of non-negative elements equals the length of the chosen dimension of *arg*.
arg: any array; if the chosen dimension has length 1, *arg* is extended along that axis to the number of non-negative elements in the operand (+/*oparray*≥0). Similar to Expand, you can generate only fill elements for a non-positive operand and an argument that is a scalar or has length 1 along the chosen dimension.
i: non-negative, integer-valued scalar that indicates the dimension desired.
res: each element of *arg* is replicated the number of times specified by the corresponding non-negative *oparray* value; for negative values, the array is extended along the chosen axis by inserting the number of fill elements equal to the absolute value of the negative element of *oparray*.
(Can be used in selective assignment.)
ext: ⎕io (for dimension).

Examples:
```
      0 1 2 ⌿ 'JMO'
MOO
      0 1 2 / 'JMO'
MOO
      0 1 ¯1 2 / 'JMO'
M OO
      1 ¯1  2 ⌿ 1 3⍴'JMO'
JMO

JMO
JMO
      A←2 3⍴'ABCDEF'
      1 2⌿A
ABC
DEF
DEF
      1 2 3/A
ABBCCC
DEEFFF
      0 1/[1]A
DEF

     0 ¯2 0 ⌿5
0 0
     0 ¯2 0 ⌿ 2 3⍴⍳3
 0 0 0
 0 0 0
```

Note 1: *oparray* ⌿ *arg* ↔ *oparray* /[⎕io] *arg*
oparray / *arg* ↔ *oparray* ⌿[*i*] *arg* (where *i* is ⍴⍴*arg* and ⎕io is 1)
Note 2: See also Reduction (/) in this section .
See also ⎕repl in the *System Functions Manual*.

Reduction f⌿ or f/

Syntax: *res* ← *f* ⌿ *arg*
res ← *f* / *arg*
res ← *f* ⌿[*i*] *arg* or *res* ← *f* /[*i*] *arg*

Description:
Apply a specified function across an array, eliminating the *i*th dimension in the process. The default dimension for ⌿ is the first dimension; the default dimension for / is the last dimension.
arg: any array valid for *f*. If *arg* is a scalar, function *f* is not applied and *res* is *arg*.
i: non-negative, integer-valued scalar that indicates the dimension desired.
res: the function *f* is applied progressively across the array, reducing the number of dimensions.
ext: ⎕io (for dimension).

For the ⌿ operator:

```
res[1]←arg[1;1] f arg[2;1]...f arg[n;1]
res[2]←arg[1;2] f arg[2;2]...f arg[n;2]
res[3]←arg[1;3] f arg[2;3]...f arg[n;3]
.
.
.
res[m]←arg[1;m] f arg[2;m]...f arg[n;m]
```

For the / operator:

```
res[1]←arg[1;1] f arg[1;2]...f arg[1;m]
res[2]←arg[2;1] f arg[2;2]...f arg[2;m]
res[3]←arg[3;1] f arg[3;2]...f arg[3;m]
.
.
.
res[n]←arg[n;1] f arg[n;2]. . .f arg[n;m]
```

Examples:

```
      +⌿ 2 3 5
10
      ,/'ABC' 'DEF' 'GHI'
 ABCDEFGHI

      A←2 3 ⍴ 1 2 3 4 5 6
      A
 1 2 3
 4 5 6
      ×⌿A
 4 10 18
      ×/A
6 120
      ×/[1]A
 4 10 18
      ×/[2]A
6 120
```

For functions other than scalar primitives, the general case of reduction is defined for vectors (recursively, for *j* in the range of ⍳⍴*arg*−1) as: *res*←⊂(⊃ *arg*[⍴arg-*j*])*f* ⊃ *f*/¯*j*↑*arg*

```
      lc←'abcdefghijklmnopqrstuvwxyz'
      ~/'aeiou'  lc   'which vowels does this vector use?'
 eiou
      ⎕repl/ (6 5 4 3 2 1) (⍳3) ('abc' 9 '^')
  abc abc abc abc abc abc 9 9 9 9 9 9 9 9 9 ^^^^^^
```

If the argument to Reduction is an empty array, the derived function is the identity function; that is, the function returns an array that is the mathematical right identity for the operand function. (Zero is the identity element for addition; `1` is the identity element for multiplication; etc.)

```
      ×/ 0 ⍴ vec
1
      ÷/ 3 0 ⍴ vec
1 1 1
```

Some functions do not have meaningful identities; for example `⍟/ ⍳0` generates a DOMAIN ERROR. Some possible identities are not presently implemented; for example, `,/ 0⍴⊂2 3⍴⍳6` generates a NONCE ERROR. Since Reduction is inherently part of Outer Product (see below), identity functions are also relevant to that operator.

Scan f⍀ or f\

Syntax: *res* ← *f* ⍀ *arg*
res ← *f* \ *arg*
res ← *f* ⍀[*i*] *arg* or *res* ← *f* \[*i*] *arg*

Description:
Apply successive reductions to an array. The default for ⍀ is along the first dimension; the default for \ is along the last dimension
arg: any array valid for *f*.
i: non-negative, integer-valued scalar indicating the dimension desired.
res: the cumulative effect of successive applications of reduction to the *i*th dimension of *arg*.
ext: `⎕io` (for dimension).

Examples:

```
      +⍀ 2 3 5
2 5 10
      +\ 2 3 5
2 5 10
      ,\1 2 3
 1  1 2  1 2 3
      A←2 3 ⍴⍳6
      A
1 2 3
4 5 6
      ×⍀A
 1  2  3
 4 10 18
      ×\A
1  2   6
4 20 120
```

Inner Product f.g

Syntax: *res* ← *larg f*.*g rarg*

Description:
Generalized matrix multiplication.
larg, *rarg:* conforming arrays valid for *f* and *g* where the last dimension of *larg* is equal to the first dimension of *rarg*.
res: applying function *g* between the elements of the last dimension of *larg* and the corresponding elements of the first dimension of *rarg* followed by reducing the result using function *f*. The shape of *res* is `((¯1↓⍴`*larg*`) , 1↓⍴`*rarg*`)`. If *larg* is *n* by *k*, and *rarg* is *k* by *m*, then *res* is *n* by *m*:

res`[1;1]←`*f*`/`*larg*`[1;]` *g* *rarg*`[;1]`
res`[1;2]←`*f*`/`*larg*`[1;]` *g* *rarg*`[;2]`
.
.
res`[1;`*m*`]←`*f*`/`*larg*`[1;]` *g* *rarg*`[;`*m*`]`

```
res[2;1]←f/larg[2;] g rarg[;1]
.
.
.
res[n;1]←f/larg[n;] g rarg[;1]
.
.
res[n;m]←f/larg[n;] g rarg[;m]
```

For vectors *larg* and *rarg*: *res ← f / larg g rarg*

Examples:

```
      2 3 5 +.× 2 3 5
38
      'SPORT'+.='SHOUT'
3
     (3 3⍴'ABCDEFGHI')∧.='DEF'
0 1 0

      M←2 3⍴⍳6 ⋄ N←3 4⍴⍳12
      M+.×N (matrix multiplication)
 38 44  50  56
 83 98 113 128
      N∧.=⍉N
1 0 0
0 1 0
0 0 1
      'WHEAT GROATS'+.∊'AEIOU'
4
```

Outer Product ∘.g

Syntax: *res ← larg ∘.g rarg*

Description:
Apply a function between every possible pairing of items taking one from each of two arrays.
larg, *rarg*: any arrays valid for *g*.
res: if *g* produces a result, *res* is an array of size ((⍴ *larg*) , ⍴ *rarg*) that consists of the result of applying *g* between each combination of *larg* and *rarg* items. If *g* does not produce a result, then ∘.*g* does not return a result.

Examples:

```
      2 3 5 ∘.* 0 1 2 3
1  2   4   8
1  3   9  27
1  5  25 125

      1 2 3 4 5 ∘.⌈ 1 2 3 4 5
1 2 3 4 5
2 2 3 4 5
3 3 3 4 5
4 4 4 4 5
5 5 5 5 5

      'ABC' ∘.= 'ABC'
1 0 0
0 1 0
0 0 1
      'ABC' ∘., '01'
 A0 A1
 B0 B1
 C0 C1
```

Each f¨

Syntax: *res* ← *f*¨ *arg*
res ← *larg* *f*¨ *rarg*

Description:
Apply a function to each item.
rarg: any array with items valid for *f*.
larg: optional; any array with items valid for *f* (conforming).
res: the collection of all results (each result is a single nested scalar) from applying *f* to each item of *arg* one at a time.

Examples:

```
      A←1 2 3 ⍴¨ 4 5 6
      A
 4  5 5  6 6 6
      ⍴A
3
```

This example reads the first five components of a file.

```
      ⎕←TN←99 ,¨ ⍳5
99 1  99 2  99 3  99 4  99 5
      ⍴TN
5

      ]display TN
.→-----------------------------.
|.→---..→---..→---..→---..→---.|
||99 1||99 2||99 3||99 4||99 5||
|'~---''~---''~---''~---''~---'|
'∊-----------------------------'
      FILE←⎕fread¨TN
```

Other Primitive Symbols

Symbols Associated with Values

Left arrow ←

Use: Assignment (simple, indexed, selective) or Sink

Syntax: *name* ← *arg*
name [*idx*$_1$; *idx*$_2$;...] ← *arg*
(*exp*) ← *arg*
← *arg*

Description:
Simple assignment: Store a value in a variable.
name: a variable name.
arg: any valid expression that returns a value.

Example:

```
      V ← ⍳5
      V
1 2 3 4 5
      NewName←V+2
      NewName
3 4 5 6 7
```

Description:
Indexed Assignment: Modify a subset of an array specified by an index enclosed in brackets. See the description of Brackets (Index into) in this chapter for information on specifying an index.
name: a variable name.
idx: integer values specifying the items of *arg*
arg: any valid expression that returns a value.
ext: ⎕io.

Example:
```
      V←2 3⍴⍳6
      V
1 2 3
4 5 6
      V[2;2 3]←7 8
      V
1 2 3
4 7 8
```

Description:
Selective Assignment: Replace a portion of an array selected by an expression that consists of selection primitives and the name of the array.
exp: an expression that defines the portion of the array being replaced.
arg: value to be inserted in the selected portion of the array.

Example:
```
      V←⍳6
      V
1 2 3 4 5 6
      (3⊃V)←999
      V
1 2 999 4 5 6
      L←'ABCDEFG'
      M←1 0 1
      (M/3↑L)←'XY'
XBYDEFG
```

To set every data item in a complex nested array to zero, use (∊A)←0; for example:
```
      A←'MARY' (2 3⍴⍳6) 0 'JOE'
      ]DISPLAY A
.→------------------------.
| .→---. .→-----.    .→--. |
| |MARY| ↓ 1 2 3| 0 |JOE| |
| '----' | 4 5 6|    '---' |
|        '~-----'          |
'∊------------------------'

      (∊A)←0
      ]DISPLAY A
.→------------------------.
| .→---. .→-----.    .→--. |
| |0000| ↓ 0 0 0| 0 |000| |
| '----' | 0 0 0|    '---' |
|        '~-----'          |
'∊------------------------'
```

Description:
Sink: Suppress the output from any line of execution in immediate execution (session) and user defined function when executed monadically.
arg: any valid expression that returns a value.

Example:

```
        ← 2 3⍴⍳6
```

Brackets [ ]

Use: Index into

Syntax: *res* ←*arg*[*idx*$_1$;*idx*$_2$; ...]
res ←*arg*[*indexarray*]

Description:
Select a subset of elements from an array.
arg: any nonscalar array.
index: the index can be either of two forms: a list of simple arrays (*idxn*) separated by semicolons, or a nested array of enclosed vectors (*indexarray*).

If the index is a list of simple arrays, it must have the same number of arrays as *arg* has dimensions; you separate the items with semicolons. Each *idx* value corresponds to one dimension of *arg*. An *idx* value can be an integer array of any shape; its elements specify the values of the dimension you want to select. The elements of *idx* cannot exceed the length of *arg* for that dimension. You can leave an index blank, using only the semicolon; in this case, you specify the entire dimension. When the index has the form *idx1*;...*idxn*, *res* is the portion of *arg* specified by the index arrays. The number of elements in the *idx* items determine the shape of *arg*. See the Introduction to the Language chapter in this manual for more examples of this form of indexing.

If the index is an array of enclosed vectors, *indexarray* can have any shape, but each item of *indexarray* must be a simple vector of length *n*, where *n* is ⍴⍴*arg*. Each item specifies one point in *arg*; each element of the simple vector specifies the value of the dimension that represents the point. (For example, if *arg* is a three-dimensional matrix, an item of *indexarray* would have three scalars, representing the plane, row and column of the point desired.)

When the index has the form *indexarray*, *res* is an array whose shape is identical to the shape of *indexarray*, and whose values represent the points specified.
ext: ⎕io.

Example:

```
      3 4 7 3 8 [3 2 1]
7 4 3
      'ABCD'[3 2]
CB
      (3 4⍴⍳12)[;3]
3 7 11
      'ABC'[4 4⍴⍳3]
ABCA
BCAB
CABC
ABCA
      (3 4⍴⍳12)[(1 1)(2 2)(3 3)]
1 6 11
```

Parentheses ( )

Use: Parentheses have multiple uses as punctuation in APL expressions. They can change the order of execution from the simple right-to-left order that the interpreter would use in their absence; they are required at Evolution Level 2 for certain indexing and multiple assignment statements; and you can use parentheses to create nested vectors (or nested elements of arrays of higher ranks).

Syntax: *expression* (*expression*) *expression*
(*variable_string*) ← *value_string*
variable ← *value* (*variable_string*) *value*

Description:
Determine order of execution. When you enclose part of a statement within parentheses, the interpreter evaluates the parenthesized expression and determines a value, which it then evaluates within the full statement.

Example:
```
       19 − 3 + 6 × 2
4
       19 − (3 + 6) × 2
1
       19 (−3 + 6) × 2
38 ¯18
```
Description:
When you want to assign values to multiple variables or index into a strand, you must enclose the left argument or strand in parentheses. The same statements without parentheses generate an `EVOLUTION ERROR` at Evolution Level `2`.

Example:
```
      (C D E) ←(⍳3) (4×5 6 7) (8⍴9)
      E
9 9 9 9 9 9 9 9
      (3↑2↓E)←C
      E
9 9 1 2 3 9 9 9
      3↑2↓E←C
3 0 0
      (C D E)[2]
 20 24 28
```
Description:
You can create a nested vector by assigning an enclosed vector to a single element or a vector or enclosing a vector of values within parentheses as part of the assignment.

Example:
```
      A←1 3 5
      A
1 3 5
      A[2]←(⊂2 3 4)
      A
 1  2 3 4  5
      B←1 (2 3 4) 5
      A≡B
1
```
High minus ¯

Use: Negative decoration

Syntax: ¯*n* (*no space*)

Description:
The high minus sign placed immediately to the left of a numeric constant makes the number a negative number. This symbol is different from the Minus function; its use eliminates the ambiguities that arise when the same symbol has both meanings.

Example:
```
      ¯2 2
¯2 2
      −2 2
¯2 ¯2
      −¯2 2
2 ¯2
      ¯ 2
SYNTAX ERROR
```
Note: The APL64 numeric editor converts minus signs to high minus signs.

Zilde ⍬

Use: Empty vector

Syntax: ⍬ or (⍳0)

Description:
A constant that is an empty numeric vector. This is useful in any situation where you want to initialize something or compare something to an empty vector.

Example:

```
      A←⍬
      ⍴⍴A
1
      ⍴A
0
      B←⍳0
      A≡B
1
```

Quad ⎕

Use: Evaluated input or output

Description:
A utility that allows a user-defined function to request input. The system reads the input as an APL expression and evaluates it. If the expression generates an error, the system reports the error and repeats the prompt. The system returns the input as an explicit result. You can also use ⎕ to display the contents of a variable or an expression.

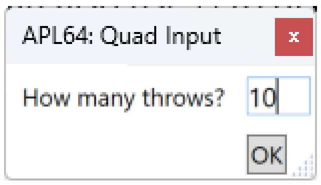
Note: Refer to the menu Help | Developer Version GUI | Quad and Quote-Quad for additional information on this function.

Example:

```
      ⎕←X←⍳5
1 2 3 4 5
      X
1 2 3 4 5
      ⎕←⍳5
1 2 3 4 5
      (⎕←3+4) × ⎕←5+6
11
7
77
```

The following example shows how you might use Quad in a simple demonstration program. The first line is a line of code; the value 10 represents user input.

```
⎕qqo←1⋄⍞←'How many throws?'⋄X←?⎕⍴100 ⋄ X
```


APL64: Quad Input

How many throws? 10

```
 How many throws? 10
75 35 80 20 61 55 26 21 58 52
```

The Quad symbol as the first character of a name designates a system function, system variable or system constant.

Quote-quad ⍞

Use: Character input or output

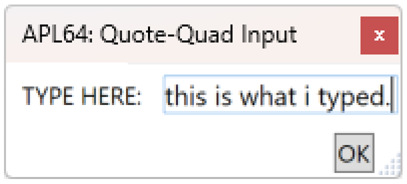
Description:
A utility that allows a user-defined function to display character output or request character input. If you use quote-quad by itself, the system returns the input as an explicit result. If you assign a value to quote-quad and then request input, the system displays the assigned value with no newline character. If you then use quote-quad to request input, the previous output acts as a prompt. The system returns the combined prompt plus user input as the result.

Note: Refer to the menu Help | Developer Version GUI | Quad and Quote-Quad for additional information on this function.

Example:

In this example below, the user types the following phrase: this is what i typed.

```
     ∇ CHARINPUT
[1]   ⍞←'TYPE HERE: '
[2]   ZINPUT ← ⍞
[3]   'This function is done.'
     ∇

CHARINPUT
```


APL64: Quote-Quad Input

TYPE HERE: this is what i typed.

```
This function is done.
      ZINPUT
this is what i typed.
```

Symbols Associated with Functions

Lamp ⍝

Use: Comment symbol

Description:
The system does not evaluate or execute anything on a line to the right of the lamp symbol.

Example:
```
      2 + 2 ⍝ 2 good to be true
4
      2 + 2 2 ⍝ good to be true
4 4
```

Diamond ⋄

Use: Statement separator

Description:
In user-defined functions, the diamond separates statements as though they were on separate lines. The system evaluates each APL statement from right to left, but the statements are executed in order from left to right, starting to the left of the first diamond. In the example below, the first line and its result are equivalent to the lines following them.

Example:
```
       B←A←7−3−2 ⋄ B←B+1 ⋄ A+B
```

```
13
      A←7-(3-2)
      B←A
      B←B+1
      A+B
13
```

Right arrow →

Use: Branch

Description:
In user-defined functions, the branch arrow used with a label or a line number alters the sequence of execution by specifying a different line for the system to execute. If you use the branch arrow with an empty right argument, the system executes the next statement; that is, it does not alter the sequence of execution. Branching to zero or a nonexistent line number exits the function. You can also use the naked branch arrow (with no right argument) to return to immediate execution mode. See the Writing APL Functions chapter in this manual for more information. In immediate execution mode, a naked branch removes one level of immediate execution from the stack.

Example:
```
[0]  BRANCH X
[1]  →(X>10) (X<0) /HI LO
[2]  Y3←Y1+Y2 ⍝ Fall through
[3]  → 0       ⍝ Exit
[4]  HI:
[5]  Y1←Y2+Y3 ⍝ Got first label
[6]  → 0
[7]  LO:
[8]  Y2←Y1+Y3 ⍝ Got second label
```

Colon :

Use: Syntactic identifier (and argument separator)

Syntax : *keyword*
label **:**
: *arg*

Description:
The colon has multiple uses as an identification symbol.

- When using control structures in user-defined functions, the colon is the first character of each keyword that is part of the control structure's outer syntax. See the Writing APL Functions chapter in this manual for more information.

  Example:
  ```
  :if X>10
     Y1←Y2+Y3
  :elseif X<0
     Y2←Y1+Y3
  :else
     Y3←Y1+Y2
  :end
  ```

- In user defined functions, a name with a colon as the last character that appears at the start of a line is a label. You can use a label with the branch arrow to alter the flow of execution. See the example under right arrow, above.
- In user defined functions, a colon prefix followed by a space (colon-space) at the beginning of a statement will suppress the output on any line of execution including the `:THEN` expression in Inline Control Structures.
- In contexts that are not specifically part of the language, you use the colon as syntactic punctuation to separate arguments, for example, in setting watchpoints for use in debugging.

Symbols that Denote an Action Word

Right parenthesis)

Description:
The right parenthesis as the first character of a word designates a system command instead of an APL expression. System commands are instructions to APL64 rather than facilities of the language interpreter. The System Commands chapter in this manual describes these commands.

Example:
You can use the system command `)si` to display the state indicator.

Right bracket]

Description:
The right bracket as the first character of an immediate execution input line designates a user command function instead of an APL expression. The User Command Processor and the User Command Descriptions chapters in this manual describe these commands.

Example:
You can use the system command `]display` to display an array on the screen in a manner that shows its structure as well as the values in the array.

Quad ⎕

Description:
The quad symbol as the first character of a word designates a system function, a system variable, or a system constant. You can use these functions and values in user-defined functions to perform many actions analogous to system commands. See the *System Functions Manual* for more information.

Example:
You can use the system variable `⎕io` to change the index origin from one to zero. Thereafter, the first two positions in an array are numbered `0` and `1` rather than `1` and `2`.

Note: The quad is also used by itself in functions. See the description in the "Symbols Associated with Values" section.

The Interrupt Input choice on the Options menu enables you to interrupt a program that is requesting Quad (⎕) input.

Other Syntax Symbols

Ampersand &

Description:
Menu access key designator. When defining a Tool for the APL64 window menu, you can place an ampersand in front of a letter of the tool name to make that letter function as a menu access key. The specified letter is underlined in the Tools menu.

Continuation character. In the user-defined function and developer command line, APL statements can be continued across multiple lines by coding an & as the last non-comment character of the line(s) to be continued such as shown below:

```
⎕vr'More'
  ∇ More;a
[1]  a ← 1 2 3 &
[2]     4 5 6 &   ⍝Comments are allowed
[3]     + &
[4]     600 500 400 300 200 100
[5]  a
  ∇
   More
601 502 403 304 205 106
```

Del ∇

Description:
The del has multiple uses as a syntactic designator.

- Tool argument: When defining a Tool for the APL64 window menu, you can use a del in the tool command. When the system executes the tool, it replaces the del with any text that is highlighted in the current window or with the object that is closest to the cursor in the current window if you did not highlight anything before invoking the tool.
- The del, following the lamp symbol (⍝∇), designates public comments.
- The del designates the beginning and end of a user-defined function. In older APL★PLUS systems, in immediate execution mode, the del invoked the function editor.
- In APL64, type `)edit ∇` to open a function Edit window.

Omega ⍵

Description:
Tool argument: When defining a Tool for the APL64 window menu, you can use an omega in the tool command. When the system executes the tool, it waits for user input to replace the omega.

Delta ∆

Description:
Valid character for object names. When you name a variable or a function, you can use the delta as you would an alphabetic character, in any position in the name.

Delta underscore ⍙

Description:
Valid character for object names. When you name a variable or a function, you can use the delta underscore as you would an alphabetic character, in any position in the name.

Underscore _

Description:
Valid character for object names. When you name a variable or a function, you can use the underscore as you would a numeric character, in any position except the first in the name.

Single quote '

Description:
Character string delimiter. When you specify character text, you use the single quote (also called apostrophe) to delimit the string. If you want an apostrophe in the string, use a pair together.

Example:
```
        T←'Joe''s'
        T
Joe's
```

Double quote "

Description:
Character string delimiter. When you specify character text, you can use the double quote just as you use a single quote to delimit the string. You can use pairs of double quotes inside double quotes. If you use double quotes outside, you do not have to double a single quote; similarly, you can use single quotes outside of double quotes. You can double either character to add a level of delimitation.

Example:
```
        T←"Joe's"
        T
Joe's
        'fmA.bnFoo' ⎕wi 'onClick' " 'fmA' ⎕wi 'caption' 'Joe''s quote "" ' "
```

Jot ∘

Description:
Placeholder used as the left operand in designating the Outer Product operator. Syntactically, it designates a function; however, the function has no other use in APL64.

Semicolon ;

Description:
The semicolon has multiple uses as a syntactic designator.

- As a language symbol, the semicolon separates indices (dimensions) in indexing or indexed assignment. For example, MA[2;3] returns the second row, third column of a matrix.
- In writing user-defined functions, you use semicolons in the header to separate variables that you want the system to localize.
- In formatting output, you use the semicolon to separate arguments to the ⎕fmt command. See the Formatting Data chapter in the *System Functions manual*.
- In contexts that are not specifically part of the language, you use the semicolon as syntactic punctuation to separate arguments, for example, in setting watchpoints for use in debugging.

Number sign

Description:
The number sign (octothorpe) has multiple uses as a syntactic designator.

- In APL64, type)edit # to open a numeric Edit window.
- In APL GUI, '#' represents the system object.

Control Structure Syntax

The following is a summary of the syntax of control structures. See the Writing APL Functions chapter in this manual for a complete description of control structures. Control structure keywords are case insensitive. Refer to **Help | APL Language | Control Structures** menu for the detailed information.

Conditional Structure `:if`

Description:

- Execute a block of statements if a test condition evaluates to `1`.

```
:if  condition
   block of statements
:endif
```

- Optionally, you can execute an alternate block of statements if the condition does not evaluate to `1`.

```
:if  condition
   block of statements
:else
   block of statements
:endif
```

- You can have multiple conditional blocks; the system executes only the block following the first condition that evaluates to `1`. The `:else` statement is optional.

```
:if  condition
   block of statements
:elseif  condition
   block of statements
:else
   block of statements
:endif
```

Repetitive Structures `:while; :until`

Description:

Execute a block of statements repeatedly.

- You can use a leading test, in which case the system executes the block as long as the test condition evaluates to `1`.

```
:while  condition
   block of statements
:endwhile
```

- You can use a trailing test in which case the system executes the block once and then repeatedly until the test condition evaluates to `1`.

```
:repeat
   block of statements
:until  condition
```

- You can use both a leading test and a trailing test, in which case the system executes the block until the trailing test is satisfied or the leading test is not satisisfied whichever occurs first.

```
:while  condition
   block of statements
:until  condition
```

- You can use no test, in which case the system executes the block until the program flow exits the block.

```
:repeat
   block of statements
:endrepeat
```

Conditional Extension Keywords `:andif; :orif`

Description:

Make multiple test conditions or alternate test conditions for conditional structures and repetitive structures.

- Both conditions must be satisfied.

```
:if  condition
:andif  condition
   block of statements
:endif
```

- The system executes the second block of statements when the `:if` condition fails and both the second and third conditions are satisfied.

```
:if  condition
   block of statements
:elseif  condition
:andif  condition
   block of statements
:endif
```

- The system executes the block only while both conditions are satisfied.

```
:while  condition
:andif  condition
   block of statements
:endwhile
```

- The system executes the block repeatedly until both conditions are satisfied at the same time of testing.

```
:repeat
   block of statements
:until  condition
:andif  condition
```

- The system executes the block if either condition is satisfied.

```
:if  condition
:orif  condition
   block of statements
:endif
```

- The system executes the second block of statements when the `:if` condition fails and either the second or third condition is satisfied.

```
:if  condition
   block of statements
:elseif  condition
:orif  condition
   block of statements
:endif
```

- The system executes the block while either condition is satisfied.

```
:while  condition
:orif  condition
   block of statements
:endwhile
```

- The system executes the block repeatedly until one condition is satisfied.

```
:repeat
   block of statements
:until  condition
:orif  condition
```

Note: You can string multiple `:andif` statements together, and you can string multiple `:orif` statements together. You can use both statements in the same structure, for example by using an `:andif` statement following an `:if` statement and an `:orif` statement following an `:elseif` statement. You cannot mix `:andif` statements with `:orif` statements contiguously.

Iterative Structure `:for; :foreach`

Description:
The `:for` statement executes a block of statements once for each value in an expression, assigning the value to a specified variable.

Syntax:

```
:for variable(s) :in expression(s)
        block of statements
:endfor
```

The `:for` statement also supports multiple variables strand notation like this:

```
    :for a b c :in (1 2 3) (4 5 6) (7 8 9) (10 11 12) (13 14 15)
```

In the first iteration `a b c` will be assigned `1 2 3`, in the second `4 5 6`, and so on.

Description:
The `:foreach` statement behaves similarly to the `:for` statement except items are iterated in parallel.

Syntax:

```
:foreach variable(s) :in expression(s)
        block of statements
:endforeach
```

The name list between the `:foreach` and `:in` keywords must contain two or more names. The argument value to the right of `:in` must be a vector containing the same number of items as the number of names in the name list between the `:foreach` and `:in` keywords. Each argument item must be the same shape or a singleton. For example,

```
    ∇ FOREACH;a;b;c
[1]
[2]   aa←1 2 3
[3]   bb←10
[4]   cc←100 200 300
[5]   :foreach a b c :in aa bb cc
[6]       ⎕←'a=',(⍕a), ' b=',⍕b,' c=',⍕c
[7]   :endforeach
    ∇

      FOREACH
a=1 b=10  c=100
a=2 b=10  c=200
a=3 b=10  c=300
```

Selection Structure `:select; :case; :caselist`

Description:

Execute one block of a series of blocks of statements. The system executes the block following the first `:case` or `:caselist` expression that matches (has the same rank, shape, and value as) the top-level control expression in the `:select` statement that precedes the series of blocks.

Syntax: `:select` *expression*
`:case` *expression*
block of statements
`:caselist` *expression* *expression* *expression ...*
block of statements
`:case` *expression*
block of statements
`:endselect`

You can also include a block of statements that the system executes when none of the `:case` or `:caselist` expressions matches the top-level expression. Following the last `:case`, use `:else` followed by the alternate block of statements. Note: The `:like` clause can also be intermixed with the `:case` or `:caselist` expression.

Alternate Selection Structure `:like`

Description:

The `:like` clause is an alternative to the `:case` or `:caselist` clause in the `:select` statement. Rather than matching the `:select` argument by value equivalence, it selects character scalar and vector cases matching its pattern similar to the `:catch` clause.

Syntax: `:select` *expression*
`:like` *expression*
`:endselect`

The `:like` clause supports wildcard pattern matching codes as an alternative form of `:case` or `:caselist` clause in a :select block. The `:like` control characters in patterns are shown below.

STAR	⋆	matches zero or more of any characters
QUERY	?	matches one of any character
RHO	ρ	is reserved for the future to denote a regular-expression
SHARP	#	is reserved for the future
SEMI	;	separates patterns
SLASH	\	makes the control character following it behave as a literal

For example,

```
    ∇ Like arg
[1]   :select arg
```

```
[2]   :like '???'
[3]       ⎕←'Any 3-letter character combination matched.'
[4]   :else
[5]       ⎕←'No match.'
[6]   :end
    ∇

      Like '1234'
No match.
      Like '123'
Any 3-letter character combination matched.
```

The `:end` Keyword

Description:
You can use `:end` as the last line in a control structure in place of any of the following more explicit keywords:

- `:endif`
- `:endwhile`
- `:endrepeat`
- `:endfor`
- `:endforeach`
- `:endselect`

The `:region` and `:endregion` Keywords

Description:
You can use `:region` and `:endregion` to define the bounds of collapsible regions. The `:region` and `:endregion` keywords can have an argument which is free text that is not executed. It is simply used as a label for the region both when the region is expanded and collapsed.

Other Program Flow Control Keywords

The keywords described in this section affect the flow of a function.

Keyword: `:continue :continueif`

Syntax: `:continue`
`:continueif` *condition expression*

Description:
In a repetitive (`:while` or `:repeat`) or an iterative (`:for`) control structure, transfer control to the final statement in the structure but continue execution within the structure.

Keyword: `:leave  :leaveif`

Syntax: `:leave`
`:leaveif` *condition expression*

Description:
In a repetitive (`:while` or `:repeat`) or an iterative (`:for`) control structure, transfer control past the final statement in the structure and continue execution with the statement following the structure.

Keyword: `:goto`

Syntax: `:goto` *label*

Description:
In a function, but not necessarily within a control structure, transfer control to the line on which the label occurs.

Keyword: `:return; :returnif`

Syntax:

```
:return
:return argument
:returnif conditional_argument
```

Description:
Anywhere in a function, exit the function.

Keyword: `:try; :catch; :catchall; :catchif; :endtry`

Syntax:

```
:try
   Statements to try
:catchif  first condition expression
    Statements to handle exceptions satisfying the first condition
:catchif  second condition expression
   Statements to handle exceptions satisfying the second condition
:catchall
   Statements to handle all other exceptions
:endtry
```

Description:
Try-catch control structure for static exception handling. For more information, see *Try-Catch Control Structure* in the System Functions Manual.

Keyword: `:finally`

Syntax:

```
:try
   Statements to try
:catchif  condition expression
    Statements to handle exceptions satisfying the condition
:finally
   Statements to execute when exiting :try clause
:endtry
```

Description:
It is executed whenever the context it is associated with (`:try` control structure or function) is exited for any reason.

Keyword: `:nextcase`

Syntax:

```
:select expression
:case expression
        block of statements
      :nextcase
:case expression
        block of statements
:endselect
```

Description:
The `:nextcase` statement flows from one case of a `:select` statement into the next without branching. This can be useful for parsing varying length arguments where each element gets special processed and then falls into the case for the next smaller length.

Keyword: `:debug`

Syntax: `:debug` *expression*

Description:
The `:debug` statement conditionally executes a single expression (whose body follows the `:debug` keyword) if and only if debug mode is ON. Implicit output is OFF during `:debug` statement execution. Unassigned result values in `:debug` expressions are not displayed in the APL64 session. Session output can only be accomplished explicitly via ⎕ assignment. Often however, the `⎕log` system function will be used to write something to a debug log file and/or the Windows Application Event Log instead.

Keyword: `:ex`

Syntax: `:ex` *expression*

Description:
The `:ex` statement is used in cascading decision statements (such as `:and`/`:or` extensions of `:if`, `:while`, etc., statements as a place to compute something before taking the next decision. Note: The `:ex` clause is not allowed in ICS expressions.

Keyword: `:ifdebug; endifdebug`

Syntax:
```
:ifdebug
      debug statement(s)
:else
      release statement(s)
:endifdebug (or :endif or :end)
```

Description:
In a repetitive (`:while` or `:repeat`) or an iterative (`:for`) control structure, transfer control past the final. The `:debug` statement argument is executed only if debug mode is ON.

Keyword: `:trace`

Syntax: `:trace`

Description:
The `:trace` statement implicitly calls the ⎕LOG function. The expression is conditionally executed if and only if running in debug mode and its result value (if any) is passed as an argument to the `⎕log` function for writing to the debug log file.

Keyword: `:assert`

Syntax: `:assert` *`condition expression`*

Description:
The `:assert` statement takes an conditional expression as its argument. This expression is only evaluated when debug mode is ON. It is not evaluated and therefore has zero execution overhead when debug mode is OFF. When the expression evaluates to singleton value 1, the assertion is considered successful and execution continues with the next logical statement. Similarly, an empty character vector is also considered to be a success. And likewise, if the expression does not return a value (such as a user defined function that doesn't set a result value) then the assertion is considered to succeed and execution continues at the next statement. All other values cause an `ASSERTION FAILURE` error to be signaled.

Keyword: `:verify`

Syntax: `:verify` *`condition expression`*

Description:
The `:verify` statement behaves the same as the :assert statement except that it is always executed, regardless of whether debug mode is ON or OFF. This provides a way to execute a condition and verify its result always, rather than only when debug mode is ON.

Keyword: `:test; :pass; :fail; :endtest`

Syntax:
```
:test
        init statements
:pass
        pass statements
:fail error
        fail statements
:endtest
```

Description:
The `:test` statement controls a block of statements that are not run during normal program execution. Any `:test` blocks in your code essentially disappear from normal program execution as if they were comments. They are not reachable and result in a `DESTINATION ERROR` if you try to branch into them under when running the function normally.

The `:pass` clause controls a block of statements that are expected to execute successfully without any errors. If any statements fails to do so, an error is logged.

The `:fail` clause controls a block of statements that are not expected to execute successfully. If any statements do not fail, then an error is logged. The `:fail` clause supports wildcard pattern matching codes for the *error* argument that is the error anticipated resulting from the *fail statements*. The `:fail` control characters patterns are shown below.

STAR	⋆	matches zero or more of any characters
QUERY	?	matches one of any character
RHO	⍴	is reserved for the future to denote a regular-expression
SHARP	#	is reserved for the future
SEMI	;	separates patterns
SLASH	\	makes the control character following it behave as a literal

Keyword: `:iftest; :test`

Syntax:
```
:iftest
        test statements
:else
        non-test statements
:endiftest
```

Description:

The `:iftest` statement controls a block of *test statements* that are conditionally executed in the test clause when a function is executed in test mode. If an `:else` statement is present, the *non-test statements* are executed when not run in test mode.

Inline Control Sequences

Inline control sequences (ICS) can articulate arbitrarily complex logical calculations with a concise and efficient notation, which uses colon-prefixed keywords `:and`, `:or`, `:then`, `:else`, and `:choose` embedded in expressions. It uses Progressive Partial Evaluation (PPE) similar to the `&&`, `||`, and `?` operators found in C, C++, C#, Java, JavaScript, etc. ICS expressions are an inversion of traditional control structures. They may contain cascading decisions and execution alternatives in a single statement, whereas traditional control structures involve multiple expressions spread across multiple statements typically on multiple lines. PPE is nothing new to APL. The `:IF` control statement uses PPE to jump into the True or False clause as soon as the overall outcome can be deduced from partial results. For example:

```
:IF Test 1
:ANDIF Test 2
:ANDIF Test 3
    True
:ELSE
    False
:ENDIF
```

If any of the test conditions are false the remaining tests are skipped and the False clause is executed immediately. If a test condition is true, the next test must be evaluated to determine the outcome of the calculation. The True clause is only executed if all tests are true. Step by step we have: if Test 1 is false, Test 2 and Test 3 are skipped and the False clause is executed. If Test 1 is true, Test 2 is evaluated. If it is false, Test 3 is skipped and the False clause is executed. If true, Test 3 is evaluated. The True clause is executed only if Test 3 is also true. Otherwise the False clause is executed.

Similarly, when the `:IF` statement is followed by a series of `:ORIF` tests, we skip the remaining tests and begin execution of the True clause as soon as *any* of the test conditions are True. We only continue evaluation of tests and eventually execute the False clause if all previous test conditions are false.

PPE is a powerful tool for simplifying program logic. For example, the first test in a series might determine if a function's left argument has a value (i.e., whether called monadically or dyadically) and a second test might check its rank or shape. If the argument doesn't have a value we discontinue the test sequence early since we cannot check its rank or shape without a value. By guarding later tests with previous tests that pre-screen certain conditions (such as existence of a value) we can efficiently and concisely test the conditions we want with a minimum of coding and a maximum of clarity.

The `:IF` … `:ANDIF` logical progression shown above using five statements can be stated more concisely using one ICS expression like this:

```
(Test 1 :AND Test 2 :AND Test 3 :THEN True :ELSE False)
```

The order of progressive evaluation above might not feel right to everyone at first glance. APL executes from right to left and this might feel like it's going the wrong way. While there isn't enough space in this paper to fully discuss the decisions that went into ICS design the following summary is offered:

- APL executes from right to left but is conceptualized, written, and read from left to right. We usually conceive of and type the left part before the right and always read it that way.
- Conditional expressions are usually typed with the condition to the left of the objective:

```
→(Condition)/L1                   ⍝ for branching
⍎(Condition)/'Expression'         ⍝ for execute
⎕ERROR(Condition)/'ERROR MESSAGE'  ⍝ for signaling errors
```

This is a good thing because we tend to think of the condition before we think of the objective, or at least we think about how to express the condition before the objective. Therefore, this ordering minimizes the number of superficial left and right key movements required to type the expression.

- Progressive logical decisions are generally conceptualized in their order of dependency. For instance, we need to know if a left argument exists before we test its rank or shape. So we tend to think of existence testing before rank or shape testing.
- The left to right order of ICS progressions is reflective of this natural thought order.
- If ICS expressions were evaluated right to left like this:

```
(True :ELSE False :WHEN Test 3 :AND Test 2 :AND Test 1)
```

we would have to type the tests in the reverse order from how we would naturally think of them.

- If we think of ICS expressions as being like diamonds with decisional powers then it is easier to understand why the left to right ordering makes sense. They follow the same flow pattern as diamond separated[2] statements and are consistent with a pattern of programming that was common in pre-control-structure days when hierarchical conditions would often be coded like this:

```
→(Condition1)/⎕LC+1 ⋄ →(Condition2)/⎕LC+1 ⋄ ObjectStatement
```

- The keywords in an ICS expression are not functions or operators. They are conjunctive in nature and exist in a different semantic space than the functional sub-expressions connected by them.
- It can be useful to think of ICS keywords as being like parentheses with decisional powers. ICS keywords morph normal order of execution in a similar way and exist in a similar semantic space.

The following list shows all recognized forms of ICS expression (for brevity the `:OR` form of each expression is not shown; wherever `:AND` appears it can be replaced by `:OR`):

```
( Cond  :THEN   Case1 :ELSE Case0 )
( Cond  :THEN   Case1 )
( Cond  :ELSE   Case0 )
( Cond1 :AND    Cond2 )
( Cond1 :AND    Cond2 ... :AND  CondN )
( Cond1 :AND    Cond2 ... :THEN Case1 :ELSE Case0 )
( Cond1 :AND    Cond2 ... :THEN Case1 )
( Cond1 :AND    Cond2 ... :ELSE Case0 )
( Index :CHOOSE Case1 : Case2 : ... : CaseN :ELSE Else )
( Index :CHOOSE Case1 : Case2 : ... : CaseN )
```

Parentheses around ICS expressions are only necessary to the extent needed to denote logical groupings, alter progression evaluation order, or disambiguate cases where the colon-prefix of the first keyword would otherwise be interpreted as declaring the name to its left as a statement label.

Any statement beginning with a colon-space (or any other character that is not part of an identifier immediately following the colon) tells the interpreter to not look for labels in that statement. A colon-space at the beginning of any statement in a user defined function also suppresses the output for the `:THEN` expression as well as for any line of execution. Either of the following statements may be used to prevent `Condition` from being interpreted as a label:

```
( Condition :THEN Action )
: Condition :THEN Action
```

[2] There probably still is not universal agreement that diamond separated statements are executed in the correct order. There are some who may still argue that they should not exist at all or that the rightmost statement should be executed before the left. However, the convention in all APL dialects known to the author executes them left to right.

The :AND and :OR keywords can be used in sequences of any length. However, they cannot be mixed in the same expression unless parentheses are used to group them like this:

```
( Cond1 :AND Cond2 ) :OR ( Cond3 :AND Cond4 :AND Cond5 )
```

The :THEN and :ELSE keywords offer a binary choice between two alternative expressions. Only one of the expressions is executed. These keyword can be used together or alone if only one choice is needed, with the implicit other choice being no execution and no result value. If both are coded :THEN must come before :ELSE. In contexts that require a result, both clauses are required.

The :CHOOSE keyword selects one expression from a list of choices separated by colons, which must be followed by a space or other character that doesn't begin a name. The value to the left of :CHOOSE must be a singleton integer. It is used as a ⎕IO sensitive index to select one of the cases to execute. An optional :ELSE clause can be executed if there isn't a case corresponding to the index value. An error is not signaled if the index does not select a case unless the expression is used in a context that requires a value, in which case an INDEX ERROR is signaled.

The following :CHOOSE expression is roughly equivalent to the :SELECT statement below it:

```
      ( Index :CHOOSE CaseA : CaseB : CaseC :ELSE ElseCase )
```

```
:SELECT Index
  :CASE ⎕IO
    CaseA
  :CASE ⎕IO+1
    CaseB
  :CASE ⎕IO+2
    CaseC
  :ELSE
    ElseCase
:ENDSELECT
```

Except :SELECT cannot be used to produce an argument value in the way :CHOOSE can:

```
 X Foo ( Index :CHOOSE CaseA : CaseB : CaseC :ELSE ElseCase )
```

Chapter 6: Programming Tools

This chapter describes workspaces that contain utility functions that can help you write efficient functions and develop applications.

- The ASMFNS workspace contains utility functions implemented in assembly language for faster execution.
- The COMPLEX workspace contains functions that perform numerical computations and structural manipulations of complex numbers.
- The DATES workspace contains functions that manipulate dates and time.
- The EIGENVAL workspace contains functions that find eigenvalues and eigenvectors of real matrices.
- The FFT workspace contains functions that compute forward and inverse discrete Fast Fourier Transforms.
- The UTILITY workspace contains general utility functions that can enhance your use of APL64 and help you develop effective APL applications.
- The WFCARE workspace contains functions that allow you to maintain and modify workspace files, which contain file images of the functions and variables in a workspace.
- The WSCOMP workspace contains utilities that compare the functions and variables in a workspace.

The *ASMFNS* Workspace

The assembly language functions that utilized the ⎕call system function in the APL+Win \Tools\Asmfns.w3 workspace have been deprecated and replaced in APL64 by system functions, e.g., ⎕WHERE, ⎕SSTOMAT, etc. The list of such replacement APL64 system functions is available from an APL64 Developer version instance: **Help | APL Language | System Functions**. Several functions are present in both APL and the new APL64 system function versions. In these cases, the APL versions have the character "a" at the end of the name.

You can directly replace the deprecated Win32 assembly language functions with the corresponding APL64 system function.

As an alternative you can use the cover functions for the replacement APL64 system functions. The updated cover functions are available in the ASMFNS.ws64 workspace in the folder C:\Users{Name}\AppData\Roaming\APLNowLLC\APL64\Tools. This path is returned in APL64 with the command: ⎕userpath. After loading your APL+Win workspace in APL64, replace the existing assembly cover functions in the workspace by copying them from the ASMFNS.ws64 workspace.

The functions handle exceptions by passing an error message to the level calling the function. If your application handles data validation, you can make the functions more efficient by removing error handling.

Many of the functions handle character data in "segmented strings." A segmented string (which construction preceded nested vectors) is a character vector that is parsed as a series of segments delimited by whatever character appears first in the vector. Thus, the string '/aaa/bbb/ccc/' is treated as three 3-element segments: 'aaa', 'bbb', and 'ccc'. The slashes are treated as delimiters and ignored as content. (You can use a character other than a slash as the delimiter.)

Function Reference

To see a complete list of functions along with the syntax for each, use Summary in the ASMFNS workspace. Some of the functions are described below. Some of the examples in this section use the variable CM2, which is a character matrix.

```
      CM2
   AA
  BBBB
 CCCCCC
DDDDDDDD

      ⍴CM2
 4 8
```

DEB Delete Extra Blanks

Purpose:
Removes all extra blanks (leading, trailing, and multiple) from the character vector *text*. If the argument is a matrix, removes extra columns if entirely blanks.

Syntax: *result* ← DEB '*text*'

Example:

```
      DEB '  The  car cost   $10,960 '
The car cost $10,960
```

MATtoSS Convert Matrix to Segmented String

Purpose: Converts a character matrix to a segmented string.

Syntax: *segstring* ← MATtoSS '*charmat*'
segstring ← '*delimiter*' MATtoSS '*charmat*'

The optional left argument, enclosed in quotes, specifies the delimiter for the segmented string. The default value is ⎕tcnl. The right argument is the character matrix you want to segment. MATtoSS removes trailing blanks as it converts each row to a segment.

Example:

```
      '★' MATtoSS CM2
★   AA★  BBBB★ CCCCCC★DDDDDDDD
```

OVER Vertically Catenate Arrays

Purpose:
Places the items specified in the left argument over the items specified in the right argument.

Syntax: *newchars* ← '*chars1*' OVER '*chars2*'

The arguments may be character or numeric scalars, vectors, or matrices. OVER pads the narrower array with blanks or zeroes. If you mix character arguments with numeric arguments, the spacing may be unsatisfactory. Mixing types of arguments may also cause integer values to be displayed as decimal numbers.

Example:

```
      '-' OVER '==' OVER CM2 OVER 2 3 ρ '★'
-
==
   AA
  BBBB
 CCCCCC
DDDDDDDD
★★★
★★★
```

ROWFIND Find Rows from One Matrix in Another Matrix

Purpose:
Returns the origin 1 indices of the first occurrence of the rows of one matrix (the right argument) in another matrix (the left argument).

Syntax: *indices* ← '*chars1*' ROWFIND '*chars2*'

The arguments must be character scalars, vectors, or matrices. ROWFIND ignores trailing blanks, if the data are characters. A 0 in the result indicates that a row is not present.

Example:

```
      CM2 ROWFIND '  BBBB'
2
      CM2 ROWFIND '  BBBB' OVER '???'
2 0
```

SSDEB Delete Extraneous Blanks from a Segmented String

Purpose:
Deletes leading, trailing, and multiple blanks or other characters from segments in a segmented string.

Syntax: *newsegstr* ← SSDEB '*segstr*'
newsegstr ← '*chars*' SSDEB '*segstr*'

The right argument must be a character vector. The first character is the segment delimiter. Note the differences in the examples below when a character other than the slash is the delimiter (2↓SS) and when a space itself is the delimiter (1↓SS). The optional left argument is a substitute list of extraneous characters (in addition to space).

Example:

```
      SS
/ A/B / C /44 / 5  55   555  55 5/6666/  /I
      SSDEB SS
/A/B/C/44/5 55 555 55 5/6666//I
      SSDEB 2↓SS
A/B / C /44 / 5 55 555 55 5/6666/ /I
      SSDEB 1↓SS
 A/B / C /44 / 5  55   555  55 5/6666/  /I
      '45' SSDEB SSDEB SS
/A/B/C// 5 5 5 /6666//I
```

SSINDEX Return Index of Substring in Segmented String

Purpose: Returns the segment substring in a segmented string that corresponds to specified indices.

Syntax: *segsubstr* ← '*segstr*' SSINDEX *ind*

The left argument must be a character vector. The first character is the segment delimiter. The right argument must be a numeric scalar or vector of index numbers. (Sensitive to ⎕io.)

Example:

```
      SS
/ A/B / C /44 / 5  55   555  55 5/6666/  /I
      SS SSINDEX 1 3 1 4
/ A/ C / A/44
```

SStoMAT Convert Segmented String to Matrix

Purpose: Converts a segmented string to a matrix.

Syntax: *charmat* ← SStoMAT '*segstring*'

The argument is a character vector. The first character is the segment delimiter.

Example:

```
      SStoMAT '/   A/  BBB// CCCCC/DDDDDD/'
   A
  BBB

 CCCCC
DDDDDD
```

TEXTREPL Replace Characters

Purpose: Replaces existing characters with new characters.

Syntax: *newvec* ← '*/old/new/old1/new1/...*' TEXTREPL '*vec*'

The left argument is an alternating series of existing and replacement character scalars or vectors. The first character is the delimiter. You can use up to 127 pairs of existing and replacement text. The right argument is the character vector that contains the text you want to replace.

Example:

```
      '/N/H/IS/WAS/TI/GA/!/?' TEXTREPL 'NOW IS THE TIME!'
HOW WAS THE GAME?
```

TRANSLATE Translate Character Array

Purpose: Translates characters of a character array to other arbitrary characters.

Syntax: *newchars* ← '*transtab*' TRANSLATE '*chars*'

The left argument specifies the translate table that you want to use. The *n*th character is the replacement for ⎕av[*n*]. The right argument specifies the text that you want to modify.

Example:
Replace ⎕tcbel and ⎕tcff characters with question marks.

```
      TT←⎕av
      TT[⎕io+7 12]←'?'
      NEW←TT TRANSLATE OLD
```

UPPERCASE Convert Lowercase to Uppercase

Purpose:
Converts the 26 lowercase English alphabetic characters in a character scalar, vector, or matrix to uppercase characters; also converts ä, ç, é ñ, ö and ü.

Syntax: *newchars* ← UPPERCASE '*chars*'

Example:

```
      UPPERCASE 'Convert to uppercase!!'
CONVERT TO UPPERCASE!!
```

WHERE Determine Element Indices

Purpose: Determines the indices of the non-zero elements of the argument. (Sensitive to ⎕io.)

Syntax: *indices* ← WHERE *numvec*

The argument is a numeric vector, usually Boolean. The result is a vector of indices equivalent to BV/⍳⍴BV for a Boolean vector, or (0≠V)/⍳⍴V for an arbitrary numeric vector.

Example:

```
      ⎕io←1 ⋄ WHERE 1 0 2 0 0 3 0 0 0 0
1 3 6
```

WORDREPL Replace Words in Character Vector

Purpose:
Replaces existing words in a character vector with new characters, as specified. WORDREPL does not affect partial words.

Syntax: *newvec* ← '*/old/new/old1/new1/...*' WORDREPL '*vec*'

The left argument is an alternating series of existing and replacement words. The first character is the delimiter. The right argument is a character vector that contains the words you want to replace.

Example:

```
      '/HE/HIS/NOW/WHAT/!/?' WORDREPL 'NOW IS THE TIME!'
WHAT IS THE TIME?
```

The *COMPLEX* Workspace

The COMPLEX workspace contains functions that perform computations with complex numbers and structural manipulations that invert matrices of complex numbers. The workspace also includes a function that derives the complex roots of an analytic function of a single complex variable. The following conventions are important in understanding and using the workspace.

- In the documentation and function descriptions, the notation *a*J*b* designates the complex number a+*i*b. Purely imaginary numbers are designated as 0J*b*. Purely real numbers, *a*J0, are abbreviated as *a*.
- An array of complex numbers is represented by a numeric array whose rank is one higher and whose first dimension is 2. The first coordinate represents the real parts of the complex numbers and the second coordinate represents the imaginary parts.

  For example, the complex scalar 3J4 (3+*i*4) is represented by the vector 3 4. The complex vector 1J2 3J4 5 6J7 0J8 is represented by: 2 5⍴1 3 5 6 0 2 4 0 7 8. An *m*-by-*n* complex matrix is represented by a 2-by-*m*-by-*n* numeric array; and so on.
- Functions that operate on arbitrary complex arrays are prefixed CX; functions that operate only on complex scalars (two-element numeric vectors) are prefixed Z.
- The term "alternating string" refers to a numeric vector with an even number of elements where the odd elements are the real parts of a vector of complex numbers and the even elements are the imaginary parts of the complex numbers.

The CX function filters an APL array to ensure it is a valid representation of a complex array. It also takes an alternating string and organizes it as a representation of a complex vector. You can use the CXRESHAPE function to create higher dimensional arrays of complex numbers.

Once you establish representations of arrays of complex numbers, you can apply a variety of computational functions to them. For example, the function CXTIMES multiplies two complex arrays. To multiply the complex vector 1J2 3 4J5 by the vector 1J¯1 2J¯2 3J¯3, you can execute:

```
      1 2 3 0 4 5 CXTIMES 1 ¯1 2 ¯2 3 ¯3
```

When both operands are complex scalars, use ZTIMES. For example, the following statement multiplies 6J7 by 8J9.

```
      6 7 ZTIMES 8 9
```

Other functions are available to compute:

- sums, differences, and quotients
- square roots, powers, logarithms, and exponentials
- trigonometric and hyperbolic functions and their inverses
- conjugates, magnitudes, and amplitudes
- inverses and transposes of complex matrices.

Use the Summary function in the COMPLEX workspace to display the name, syntax, and purpose of each of these functions.

The CXROOTS function finds the roots of a function of a single complex variable. This function must be named CXfn and must be monadic and return an explicit result. Both the argument and the result must be a complex scalar (two-element numeric vector). After you define CXfn, execute

roots ← CXROOTS *guesses*

where the argument is an alternating string or complex vector of guesses. The number of supplied guesses limits the number of roots that will be returned. The CXstate global variable controls the behavior of the CXROOTS function — the maximum number of iterations, the convergence tolerance, the display of intermediate results, and so forth. Execute CXSTATE for a review and interpretation of the elements of CXstate.

The *DATES* Workspace

The DATES workspace contains functions that you can use to manipulate dates. With these functions, you can:

- convert dates from vector to scalar and scalar to vector
- format dates into a readable form
- verify dates.

Most of the functions handle any number of dates. The functions that accept the vector form of dates require that the argument be in `⎕ts` timestamp form. The following conventions are used in the function descriptions.

date is an integer array whose last dimension is `3` (`3=¯1↑⍴`*date*); typically in `3↑⎕ts` form:

- *date*`[1]` is a two- or four-digit year (`1900`s are assumed for two-digit representations)
- *date*`[2]` is an integer (`1` to `12`) that represents the month
- *date*`[3]` is an integer (`1` to `31`) that represents the day of the month.

ts is a matrix with one date or time per row, or an integer array whose last dimension is `7` (`7=¯1↑⍴`*ts*); typically in `7↑⎕ts` form; typically the functions also work with trailing elements missing:

- *ts*`[1]` is a two- or four-digit year (`1900`s are assumed for two-digit representations)
- *ts*`[2]` is an integer (`1` to `12`) that represents the month
- *ts*`[3]` is an integer (`1` to `31`) that represents the day of the month
- *ts*`[4]` is an integer (`0` to `23`) that represents the hour
- *ts*`[5]` is an integer (`0` to `59`) that represents the minute
- *ts*`[6]` is an integer (`0` to `59`) that represents the second
- *ts*`[7]` is an integer (`0` to `999`) that represents the millisecond.

In describing output formats, a single character means one or two output characters, whereas a double character means the output is zero-filled. Thus, D standing for a day of the month would be `3` or `30`; DD would be represented as `03` or `30`. Uppercase H means `24`-hour cycle; lowercase h is `12`-hour cycle. M means a one or two digit month, MM a zero-filled two digit month, MMM the three-letter abbreviation, and MMMM the full name of the month.

Function Reference

DATEBASE Determine Elapsed Days

Purpose:
Returns an integer array of shape `¯1↓⍴`*date* that represents the number of elapsed days since January `1`, `1900`. Elements of *result* may be negative. (A companion function, `DATEREP`, reverses the representation.)

Syntax: *result* `←` `DATEBASE` *date*

Example:
Find the number of days between February `28`, `2000`, and March `2`, `2000`.

```
      DATEBASE 2 3⍴2000 3 2 2000 2 28
36585 36582
      36585 - 36582
3
```

DATECHECK Check Valid Date

Purpose: Returns a Boolean vector of shape `¯1↓⍴`*date*, in which `1`s indicate valid dates.

Syntax: *result* `←` `DATECHECK` *date*

Example:
This example shows that February `29`, `2000`, is a valid date (since `2000` was a leap year), but February `29`, `1900`, is not (since `1900` was not a leap year).

```
      DATECHECK 2 3⍴2000 2 29 1900 2 29
1 0
```

DATEOFFSET Add Days to Date

Purpose: Adds a specified number of days to each date in the right argument and returns the new dates.

Syntax: *result* `←` *days* `DATEOFFSET` *date*

The arguments must conform in shape. If neither of the arguments is a scalar, `⍴`*days* must equal `¯1↓⍴`*date*.

Example:
Add, 30, 60, and 90 days to the date November 15, 2001. The resulting dates are December 15, 2001; January 14, 2002 and February 13, 2002.

```
      30 60 90 DATEOFFSET 2001 11 15
 2001 12 15
 2002  1 14
 2002  2 13
```

DATEREP   Represent Elapsed Days as Timestamp

Purpose:
Converts scalars representing number of elapsed days since January 1, 1900 to dates in (3↑⎕ts) format. This is the companion function to DATEBASE.

Syntax: *date* ← DATEREP *elapsed*

Example:

```
      DATEBASE 2001 1 23
36912
      DATEREP 36912
2001 1 23
```

DATESPELL   Format Date

Purpose: Returns a timestamp formatted in a specified style.

Syntax: *result* ← *code* DATESPELL *ts*

code is a one- or two-element vector in which the first element is the display style and the second element is an optional hour offset. The default hour offset is 0.
ts is a full or partial timestamp; if you use the hour offset, however, you must specify at least the hour.

If you use more than 3 elements in *ts*, you can display time in AM/PM or 24-hour (military) style. The base codes display time in AM/PM style; add 8 to each code to display time in 24-hour style. The table below shows the codes and the format each displays. Note that the first two digits of the year display only if they are part of the argument.

Date Formats for DATESPELL

Code	Result
0 or 8	4 MAR 2002
1 or 9	MAR 4, 2002
2 or 10	4 MARCH 2002
3 or 11	MARCH 4, 2002
4 or 12	MON 4 MAR 2002
5 or 13	MON, MAR 4, 2002
6 or 14	MONDAY 4 MARCH 2002
7 or 15	MONDAY, MARCH 4, 2002

Examples:
The last two examples demonstrate using the hour offset to show the difference in times between Washington, D.C, and Los Angeles.

```
      0 DATESPELL 2001 12 31 12
31 DEC 2001  12 N
      5 DATESPELL TS←2002 1 1 2 10
TUE, JAN 1, 2002  2:10 AM
      13 ¯3 DATESPELL TS
MON, DEC 31, 2001  23:10
```

DAYOFWK Determine Day of the Week from Date

Purpose: Returns the day of the week (1, Sunday through 7, Saturday).

Syntax: *result* ← DAYOFWK *date*

The *result* has one element for each date in *date*.

Example:
The example shows that January 1, 1999, was a Friday; January 1, 2000, was a Saturday; and January 1, 2001, was a Monday.

```
      DAYOFWK 3 3⍴1999 1 1 2000 1 1 2001 1 1
6 7 2
```

DAYOFYR Determine Day of the Year from Date

Purpose: Returns the day of the year (1 through 366). The *result* has 1 element for each date in *date*.

Syntax: *result* ← DAYOFYR *date*

Example:

```
      T←2000 12 31 2001 1 3 2001 12 31
      DAYOFYR 3 3⍴T
366 3 365
```

DAYSDIFF Determine Difference in Days

Purpose:
Returns an integer array that contains the difference, in days, between the corresponding dates supplied in the arguments. Note that this function does not count both the start and the end date. The difference between the last day of the year and the first day of the year is 364 for non-leap years.

Syntax: *result* ← *date1* DAYSDIFF *date2*

Example:

```
      L←3 3⍴2000 3 2 2001 2 28 2002 12 31
      R←3 3⍴2000 2 28 2001 3 2 2002 1 1
      L DAYSDIFF R
3 ¯2 364
```

DSPELL Display Date and Time

Purpose: Displays the date and time in the argument in the form: D MMM YY H:mm:ss:nnn.

Syntax: *text* ← DSPELL *ts*

The time precision of the result depends on the length of the last dimension of the argument. DSPELL displays the time in 24-hour (military) style.

Example:

```
      DSPELL 2001 3 26 14
26 MAR 2001  14:00
```

FTIMEBASE Convert Dates to Elapsed Microseconds

Purpose:
Converts the dates and times in the argument to single numbers that represent elapsed microseconds since 00:00:00.000, January 1, 1900.

Syntax: *result* ← FTIMEBASE *ts*

Example:

```
      ⎕pp←16
      FTIMEBASE 2001 3 26 11 19 53 518
3194594393518000
```

FTIMEFMT Convert Elapsed Microseconds to Date and Time

Purpose:
Converts scalars that represent elapsed microseconds since 00:00:00.000, January 1, 1900, and formats the result in the form: M/DD/YY HH:mm:ss:nnn

FTIMEFMT displays the time in 24-hour (military) style.

Syntax: *text* ← FTIMEFMT *elapsed*

Example:

```
      FTIMEFMT 3200000000000000
 5/28/01  00:53:20.000
```

FTIMEREP Convert Elapsed Microseconds to Timestamp

Purpose:
Converts scalars that represent elapsed microseconds since 00:00:00.000, January 1, 1900, to dates in ⎕ts timestamp form.

Syntax: *result* ← FTIMEREP *elapsed*

The *result* is an integer array of dates that correspond to the elements of *elapsed.*

Example:

```
      FTIMEREP 3200000000000000
2001 5 28 0 53 20 0
```

HOURBASE Convert to Elapsed Hours

Purpose:
Converts the dates and hours in the argument to single numbers that represent the elapsed hours since 00, January 1, 1900.

Syntax: *result* ← HOURBASE *dateshours*

dateshours is an integer array whose last dimension is 4; typically, a vector in the form 4↑⎕ts.

Example:

```
      HOURBASE  2001 3 26 11
887387
```

HOURREP Convert Elapsed Hours to Dates

Purpose:
Converts scalars that represent the elapsed hours since 00, January 1, 1900, to dates and times in 4↑⎕ts format.

Syntax: *result* ← HOURREP *elapsed*

Example:

```
      HOURREP 887387
2001 3 26 11
```

LEAPYR Determine Leap Year

Purpose: Determines if a specified year is a leap year.

Syntax: *result* ← LEAPYR *year*

The argument is the year in two- or four-digit form; 1900s are assumed when two digits are used. The *result* is 1 if the year is a leap year.

Example:

```
      LEAPYR 1990+⍳10
0 1 0 0 0 1 0 0 0 1
```

`MDYTOYMD` Convert Month-Day-Year Format

Purpose:
Converts dates that are in the form month-day-year to dates that are in the form year-month-day.

Syntax: *result* ← `MDYTOYMD` *mdy*

The argument *mdy* is an array of dates represented as MDDYY or MDDYYYY.

Example:

```
      T←2 2⍴20599 4252002 102099 6102002
      MDYTOYMD T
 990205 20020425
 991020 20020610
```

`MINBASE` Convert Dates to Elapsed Minutes

Purpose:
Converts dates and times to single numbers that represent the elapsed minutes since `00:00`, January `1`, `1900`.

Syntax: *result* ← `MINBASE` *datestimes*

The argument *datestimes* is an integer array of dates whose last dimension is `5`; typically, a vector in `5↑⎕ts` form.

Example:

```
      MINBASE 2001 3 26 11 53
53243273
```

`MINREP` Convert Elapsed Minutes to Dates

Purpose:
Converts scalars that represent the elapsed minutes since `00:00`, January `1`, `1900`, to dates and times in `5↑⎕ts` format.

Syntax: *result* ← `MINREP` *elapsed*

Example:

```
      MINREP 53243273
2001 3 26 11 53
```

`SECBASE` Convert Dates to Elapsed Seconds

Purpose:
Converts dates and times to single numbers that represent elapsed seconds since `00:00:00`, January `1`, `1900`.

Syntax: *result* ← `SECBASE` *datestimes*

The argument *datestimes* is an integer array of dates whose last dimension is `6`; typically, a vector in `6↑⎕ts` form.

Example:

```
      SECBASE 2001 3 26 11 53 19
3194596399
```

`SECREP` Convert Elapsed Seconds to Dates

Purpose:
Converts scalars that represent the elapsed seconds since `00:00:00`, January `1`, `1900`, to dates and times in `6↑⎕ts` format.

Syntax: *result* ← `SECREP` *seconds*

Example:

```
      SECREP 3194596399
2001 3 26 11 53 19
```

TIMEBASE Convert Date to Elapsed Milliseconds

Purpose:
Convert the date specified in the argument to the number of elapsed milliseconds since 00:00:00.000, January 1, 1900.

Syntax: *result* ← TIMEBASE *ts*

Example:

```
      TIMEBASE 2001 3 26 11 53 19 518
3194596399518
```

TIMEFMT Format Dates and Times

Purpose:
Format dates and times specified in the argument in the form: M/DD/YY HH:mm:ss:nnn.

Syntax: *result* ← TIMEFMT *ts*

The precision of the time depends on whether the last four elements of *ts* are present.

Example:

```
      TIMEFMT 2001 3 26 13
 3/26/01  13:00
```

TIMEREP Convert Elapsed Milliseconds to Dates

Purpose:
Convert scalars that represent elapsed milliseconds since 00:00, January 1, 1900, to dates and times in ⎕ts form.

Syntax: *result* ← TIMEREP *elapsed*

Example:

```
      TIMEREP 3194596399518
2001 3 26 11 53 19 518
```

WKDAYSDIFF Determine Number of Weekdays between Dates

Purpose:
Calculates the number of weekdays between the corresponding dates in the arguments. Note that this function does not count the last day; that is, Monday to Friday returns 4, as does Sunday to Friday. Monday to Sunday (or Saturday) returns 5. (You can also think of it as counting from the beginning of each day.)

Syntax: *result* ← *date1* WKDAYSDIFF *date2*

Example:

```
      2002 10 15 WKDAYSDIFF 2002 10 1
10
```

YMDTOMDY Convert Year-Month-Day Format

Purpose: Converts dates in the form year-month-day to dates in the form month-day-year.

Syntax: *result* ← YMDTOMDY *ymd*

Example:
In the second example, the dates are put in the month-day-year form and then formatted with ⎕fmt.

```
      YMDTOMDY 20020915
9152002
      T←20020527 20020303 20020424 20021216
      FSTR←'G<ZZ/ZZ/ZZZZ >'
      FSTR ⎕fmt YMDTOMDY 2 2⍴T
 5/27/2002  3/03/2002
 4/24/2002 12/16/2002
```

The *EIGENVAL* Workspace

The `EIGENVAL` workspace contains a variety of functions that find eigenvalues and eigenvectors of real matrices. Real and complex eigenvalues and eigenvectors are derived. Different routines can be used depending upon whether or not the real matrix is symmetric.

A real scalar `S` is an eigenvalue for a matrix `M` if there is a non-zero vector `V` such that: `S×V=M+.×V`
That is, the scalar multiplied by the vector equals the matrix multiplied by the vector. The vector is then the eigenvector for the eigenvalue.

The functions `EIG` and `SYMEIG` find eigenvalues and eigenvectors of a real *n*×*n* matrix. The latter operates only on symmetric matrices. The function `EIGVEC` finds an eigenvector corresponding to a real eigenvalue. This section describes `EIG`, `SYMEIG`, and `EIGVEC` in detail. The `FMTEIGS` function displays eigenvalues and eigenvectors in an easy-to-read format. Use `Summary` in the workspace to see the name, purpose, and syntax of each of the other functions.

Note: The functions generate random numbers to produce eigenvectors, so results may vary from run to run. However, for a given eigenvalue, results will all be (complex) linear combinations of each other.

Function Reference

`EIG` Find Eigenvalues and Eigenvectors for a Non-Symmetric Matrix

Purpose: Finds eigenvalues and eigenvectors in a real, non-symmetric, *n*×*n* matrix.

Syntax: *result* ← *mode* `EIG` *matrix*

The left argument is optional. If it is `0` or absent, only eigenvalues are sought. If it is `1`, corresponding eigenvectors are found if all the eigenvalues are real. If it is `2`, complex eigenvectors are found for complex eigenvalues.

Result:
The shape of the result depends on *mode* and on the success of the calculation.

- If all the eigenvalues are real and *mode* is omitted or zero, ⍴*result* is *n*.
- If some of the eigenvalues are complex and *mode* is not `2`, ⍴*result* is `2×`*n*. The first row contains the real parts; the second row contains the imaginary parts. A warning message is displayed.
- If all the eigenvalues are real and *mode* is `1`, ⍴*result* is `(`*n*`+1)×`*n*. The first row is the vector of eigenvalues and the column below each eigenvalue is the corresponding eigenvector of unit length.
- If some of the eigenvalues are complex and *mode* is `2`, ⍴*result* is `2×(`*n*`+1)×`*n*. The result is the complex analogue of the `(`*n*`+1)×`*n* array of the previous case, containing eigenvalues in the first rows of the real and complex planes, and eigenvectors in the remaining rows.

If only *k* of the eigenvalues are found, a warning message displays and the last dimension of the result has length *k* in each of the above cases.

Example:

```
      ⎕pp←5
      M←?4 4⍴10
      M
 2  8  5  6
 3  1  7  7
10  4  6  9
 1  1  6  7
      ⎕←Z←2 EIG M
 20.693    ¯3.0552   ¯3.0552     1.4174
  0.48185   0.45264  ¯0.095578   0.31915
  0.43867   0.01845   0.35075    0.14712
  0.66777  ¯0.58160  ¯0.53997    0.5949
  0.35983   0.20981   0.3315    ¯0.72291

  0         3.2311   ¯3.2311     0
  0        ¯0.2435    0.50499    0
  0         0.3621   ¯0.09162   0
  0        ¯0.3782    0.44044   0
  0         0.28128   0.11519   0
```

Check the complex third eigenvalue and complex eigenvector.

```
      ⎕←EVAL3←Z[;1;3]
¯3.0552 ¯3.2311
      ⎕←EVEC3←0 1↓Z[;;3]
¯0.0956  0.3508 ¯0.5360 0.3315
 0.5050 ¯0.0916 ¯0.4404 0.1152
```

Use the CXMATTIMES and CXTIMES functions to verify the test for complex eigenvalues and eigenvectors.

```
      (M,[⎕io-.5] 0) CXMATTIMES EVEC3
 1.9237 ¯1.3677 0.2145 ¯0.6052
¯1.2341 ¯0.8534 3.0775 ¯1.4229
      EVAL3 CXTIMES EVEC3
 1.9237 ¯1.3677 0.2145 ¯0.6052
¯1.2341 ¯0.8534 3.0775 ¯1.4229
```

EIGVEC Find Eigenvector Corresponding to Eigenvalue

Purpose:
Find the unit length eigenvector corresponding to a particular real eigenvalue of a real matrix. This function is useful when:

- you omitted *mode* for the EIG or SYMEIG functions or
- you use 1 for *mode* and the result of EIG contains both complex and real eigenvalues.

Syntax: *result* ← *matrix* EIGVEC *eigval*

Example:
This example finds the eigenvector for an eigenvalue derived from the preceding example. Note that some eigenvectors are complex

```
      ⎕←X←1 EIG M
20.693 ¯3.0552 ¯3.0552 1.4174
 0       3.2311 ¯3.2311 0
      ⎕←EV1←M EIGVEC X[1;1]
¯0.48185 ¯0.43867 ¯0.66777 ¯0.35983
```

Check the test for real eigenvalues and eigenvectors.

```
      EV1×X[1;1]
¯9.9709 ¯9.0774 ¯13.818 ¯7.4459
      M+.×EV1
¯9.9709 ¯9.0774 ¯13.818 ¯7.4459
```

POLYROOT Find Roots of a Real Polynomial

Purpose:
Find the roots of a real polynomial by finding the eigenvalues of the companion matrix.

Syntax: *result* ← POLYROOT *coefvec*

The argument is a vector of coefficients in descending power order.

Result:
The result is a vector of all the real roots, or a 2-row matrix of complex roots.

Example:

```
      POLYROOT 1 6 11 6
¯3 ¯2 ¯1
      POLYROOT 1 0 1
 0  0
 1 ¯1
```

SYMEIG Find Eigenvalues and Eigenvectors for a Symmetric Matrix

Purpose: Find eigenvalues and eigenvectors in a real, symmetric, *n*×*n* matrix; that is, *mat* ≡ ⍉*mat*.

Syntax: *result* ← *mode* SYMEIG *matrix*

The left argument is optional. If it is 0 or absent, only eigenvalues are sought. If it is 1, corresponding eigenvectors are found. (All the eigenvalues will be real for a symmetric matrix.)

The shape of the result depends on *mode* and on the success of the calculation. The possible shapes are:

- If *mode* is omitted or zero, ⍴*result* is *n*. The result is the vector of (real) eigenvalues.
- If *mode* is 1, ⍴*result* is (*n*+1)×*n*. The first row is the vector of (real) eigenvalues and the column below each eigenvalue is the corresponding eigenvector of unit length.

If only *k* of the eigenvalues are found, the system displays a warning message and the last dimension of the result has length *k* in each of the above cases.

Example:

```
      ⎕←M←3 3⍴1 2 3 2 8 5 3 5 4
1 2 3
2 8 5
3 5 4
      ⎕←X←1 SYMEIG M
 1.65     ¯1.0282  12.378
 0.59223  0.75369 ¯0.28499
−0.60531  0.18268 ¯0.77474
 0.53185 ¯0.63133 ¯0.5644
```

Verify the test for eigenvalues and eigenvectors.

```
      EV1←1↓X[;1]
      EV1×X[1;1]
0.97717 ¯0.99875 0.87755
      M+.×EV1
0.97717 ¯0.99875 0.87755
```

The *FFT* Workspace

The FFT workspace contains functions that compute forward and inverse discrete Fourier Transforms of real or complex data. It also contains utilities that let you apply windowing to control end-effect leakage. The workspace requires the companion APL file, also named FFT, to be in the same directory as the active workspace. The main routines require the FFTTN function, which ties this file.

Run FFTDETAILS to see detailed documentation on using this workspace. The FFTDEMO function shows a series of examples using the key functions in the workspace.

Function Reference

FFT Compute Fast Fourier Transform

Purpose: Calculates complex-valued Fast Fourier Transforms.

Syntax: rDV←*x* ⋄ iDV←*y* ⋄ FFT

The function requires FFTTN and ∆FT as subroutines. It operates on two global variables:

- rDV is a numeric vector of real parts of the complex argument.
- iDV is a numeric vector of imaginary parts of the complex argument.

These variables pass data to the function and return results from the function; the results overwrite the initial values. The variables must have equal lengths; the lengths must equal a power of 2 and be in the range 8 through 4096; for example,

```
rDV←(2*⌈2⍟⍴rDV)↑rDV
```

If the variables are not equal in length, you can pad one of them to the left or right with zeros; for example:

```
iDV←(⍴rDV)↑iDV
```

The results are returned in normal order. (See `FFTDETAILS` for a definition of normal order.)

After execution, the Fourier coefficients in `rDV` and `iDV` are scaled high by `⍴rDV`. To generate absolute Fourier coefficients, divide `rDV` and `iDV` by `⍴rDV`.

If the input is a linear time spacing `TIME←∆T×¯1+⍳`*n*, you can multiply `rDV` and `iDV` by `∆T` to show the equivalence of the discrete and continuous Fourier Transforms.

The `FFTDEMO` function shows examples using this function.

`FFT2` Perform Two Real-Valued Fast Fourier Transforms

Purpose:
Performs two *n*-point, real-valued Fast Fourier Transforms, using only one call to the *n*-point complex-valued `FFT` routine.

Syntax: `rDV←`*x1* `⋄ iDV←`*x2* `⋄ FFT2`

The function requires `FFT`, `FFTTN`, and `∆FT` as subroutines. It operates on two global variables:

- `rDV` is the first real numeric vector.
- `iDV` is the second real numeric vector.

The variables must have equal lengths; the lengths must equal a power of `2` and be in the range `8` through `4096`. If the variables are not equal in length, you can pad one of them to the left or right with zeros; for example:

```
rDV←(2*⌈2⍟⍴rDV)↑rDV ⋄ iDV←(2*⌈2⍟⍴iDV)↑iDV
```

The function assigns the results to the following global variables:

- `rDV1` contains the real parts of the Fast Fourier Transform of the first vector, `rDV`.
- `iDV1` contains the imaginary parts of the Fast Fourier Transform of the first vector, `rDV`.
- `rDV2` contains the real parts of the Fast Fourier Transform of the second vector, `iDV`.
- `iDV2` contains the imaginary parts of the Fast Fourier Transform of the second vector, `iDV`.

When execution completes, `FFT2` erases `rDV` and `iDV`.

`IFFT` Calculate Inverse Fast Fourier Transform

Purpose: Calculates complex-valued inverse Fast Fourier Transforms.

Syntax: `rDV←`*x* `⋄ iDV←`*y* `⋄ IFFT`

The function requires `FFTTN` and `∆FT` as subroutines. It operates on two global variables:

- `rDV` is a numeric vector of real parts of the complex argument.
- `iDV` is a numeric vector of imaginary parts of the complex argument.

These variables pass data to the function and return results from the function; the results overwrite the initial values. The variables must have equal lengths, equal a power of `2`, and be in the range `8` through `4096`. If the variables are not equal in length, you can pad one of them to the left or right with zeros; for example:

```
rDV←(2*⌈2⍟⍴rDV)↑rDV ⋄ iDV←(2*⌈2⍟⍴iDV)↑iDV
```

Input should be stored in normal frequency order and should be scaled high by `⍴rDV`; that is: *absolute Fourier coefficients* `× ⍴rDV`

The `FFTDEMO` function shows examples using this function.

`RFFT` Compute a Single Real-Valued Fast Fourier Transform

Purpose:
Calculates a single, real-valued Fast Fourier Transform using an internal *n*`÷2` complex algorithm.

Syntax: `rDV←`*x* `⋄ RFFT`

The function requires `FFTTN` and `∆FT` as subroutines. You store the vector you want to transform in the global variable `rDV`. The function stores the results in the global variables `rDV` and `iDV`. `rDV` contains the real parts of the result; `iDV` contains the imaginary parts. The result is the same as produced by `FFT` with `iDV←(⍴rDV)⍴0`. The difference is that the calculation is performed on an array that is only half as large.

The *UTILITY* Workspace

The `UTILITY` workspace contains functions that can help you develop effective APL applications. These functions demonstrate techniques for input and data conversion, screen display modification, and workspace and function maintenance. Some of the functions are described here. Use `Summary` in the workspace for a complete listing of the available utilities.

`ALLBUT` List Selected Functions and Variables

Purpose:
Returns a list of all the functions and variables in the workspace that are not named in the argument.

Syntax: *names* `← ALLBUT '`*namelist*`'`

The argument is a list of valid identifier names. The list can be a character matrix with one name per row or a character vector with names separated by spaces. The result is a character matrix of the names of all the functions and variables in the active workspace that are not explicitly named in the argument.

Example:

```
      ⎕erase ALLBUT 'ANNUALRPT MONTHLYRPT'
```

`ALLFNS` List Functions

Purpose: Returns the list of all functions in the workspace.

Syntax: *fnlist* `← ALLFNS`

The result is a character vector of all the functions in the workspace, listed alphabetically, separated by spaces.

Example:

```
      ALLFNS
ALLFNS MYFN REPORT ENTER UPDATE
```

`COMPAREVRS` Compare Two Functions

Purpose: Compares the visual representations of two functions and displays the differences.

Syntax: *vr1* `COMPAREVRS` *vr2*

The arguments are the visual representations of the two functions that you want to compare. The result displays the function lines that are different for the two functions. The function specified by the left argument appears first, followed by a dashed line, then the function specified by the right argument.

Example:

```
      (⎕vr 'MONTH') COMPAREVRS ⎕vr 'YEAR'
```

DIAM Combine Function Lines with Diamonds

Purpose:
Uses diamonds to combine lines of one or more functions into single lines. Labeled lines remain labeled. Before combining lines, DIAM removes all comments except those that begin with lamp del (⍝∇), lamp del-tilde (⍝⍫), or lamp right-brace (⍝{).

Syntax: `DIAM '`*fnlist*`'`
`DIAM` *tieno compno passno*

The argument can be a character scalar, vector, or matrix that contains the names of the functions you want to modify; or, the argument can be a two- or three-element numeric vector that specifies a file component. To modify multiple file components, use a two- or three-column numeric matrix as the argument.

DISPLAY Display Array

Purpose:
Displays an array graphically. This function is particularly useful in illustrating the structure of a nested array.

Syntax: *result* ← `DISPLAY` *array*

The *result* is the pictorial representation of an array. In the borders of items, the theta (⊖) and rotate (⌽) symbols indicate zero dimensions. Arrows (→ or ↓) indicate separate dimensions. An epsilon (∊) indicates nested data. A tilde (~) indicates a numeric item. A plus sign (+) indicates simple data that are heterogeneous.

Example:
```
       DISPLAY ⍳¨⍳4
.→--------------------------------.
| .-.   .---.   .-----.   .-------. |
| |1|   |1 2|   |1 2 3|   |1 2 3 4| |
| '~'   '~--'   '~----'   '~------' |
'∊--------------------------------'
```

FILEHELPER Override File Access Matrix

Purpose: Overrides a file access matrix and grants full access to the file owner.

Syntax: *result* ← `FILEHELPER` *filename*

Grants the owner of the specified file full explicit access with no passnumber. FILEHELPER returns the original access matrix as the explicit result. The owner can then modify the access matrix as appropriate and restore it to the file.

FNCOMP Compare Functions

Purpose: Compares two functions and displays the lines that differ.

Syntax: *result* ← *fnspec1* `FNCOMP` *fnspec2*

The arguments are either the names of functions in the active workspace or a file specification that consists of file tie numbers, component numbers, and optional passnumbers.

If you specify a file component, it must contain a visual representation of a function. In the resulting display, the function specified in the left argument appears first. A dashed line separates the functions.

Example:
The first examples compares two functions in the active workspace. The second compares a function in the workspace to its representation in component 35 of the file tied to 90.
```
       'MONTHLY' FNCOMP 'YEARLY'
       'MONTHLY' FNCOMP 90 35
```

`FNIDS` Search a Function for Identifiers

Purpose: Searches a specified function for various types of identifiers.

Syntax: *idlist* `←` *fnname* `FNIDS '`*specs*`'`

The left argument can be a character scalar or vector that contains the name of the function to search, or it can be a two- or three-element vector that contains a file tie number, component number, and optional passnumber. If *fnname* is numeric, `FNIDS` treats the character vector in the component as the function listing. `FNIDS` does not search text in comments or character constants.

The right argument contains the specification of the search categories:

- A semicolon (`;`) specifies identifiers that are in the header and local to the function; this includes the name of the function only if it is explicitly localized.
- A colon (`:`) specifies labels.
- A right tack (⊣) specifies identifiers that are index assigned
- A left arrow (←) specifies identifiers that are directly assigned
- A quad (⎕) specifies identifiers that are system functions or system variables.

To specify combinations of categories, use the following set operators:

- A cup (∪) specifies union
- A cap (∩) specifies intersection
- A tilde (~) specifies set difference. You can also use the tilde (~) monadically to mean the complement of a set with respect to all the identifiers used in the function.

The result is normally a character matrix of identifier names in alphabetic order that satisfies the specifications in the right argument. If either argument is in error, the result is a character vector that describes the problem.

Examples:
Find all identifiers.

```
      'FNAME' FNIDS ''
```

Find all identifiers in the header that are local to the function. Include the name of a function only if it is explicitly localized.

```
      'FNAME' FNIDS ';'
```

Find all identifiers that the function assigns directly. Include the name of the result only if it is explicitly assigned in the body.

```
      'FNAME' FNIDS '←'
```

Find all global identifiers that the function uses.

```
      'FNAME' FNIDS '~ ⎕ ∪ : ∪ ;'
```

Find all identifiers that are directly or index assigned, but are not in the header.

```
      'FNAME' FNIDS '(← ∪ ⊣) ~ ;'
```

Find all identifiers in the header that are system functions or system variables.

```
      'FNAME' FNIDS '⎕ ∩ ;'
```

`FNREPL` Change String in Function

Purpose: Displays or changes all occurrences of *string* in *fnlist*.

Syntax: `'`*fnlist*`' FNREPL '`*string*`'`
`'`*fnlist*`' FNREPL '`*oldstring*`' BY '`*newstring*`'`

The left argument contains the names of the functions that you want to search. The right argument is a character vector containing the syntactic element you want to display. `FNREPL` prints the function name, the number of occurrences of the string in the function, and the text of each line that contains the string. To replace one syntactic element with another, use `FNREPL` with the companion function `BY`.

Examples:
Print all the occurrences of PRINT in all the functions in the active workspace, and replace all the occurrences of PRINT with NEWPRINT.

```
      ALLFNS FNREPL 'PRINT'
      ALLFNS FNREPL 'PRINT' BY 'NEWPRINT'
```

LISTER List Functions and Variables

Purpose: Lists all the functions and variables in the workspace.

Syntax: LISTER

LOCKEDFNS List Locked Functions

Purpose:
Returns a character matrix that contains the names of all the locked functions in the active workspace.

Syntax: *fnlist* ← LOCKEDFNS

Example:

```
      LOCKEDFNS
BUILD
CENTER
LOCKEDFNS
```

ORDLOC Reorder Local Identifiers

Purpose:
Reorders the local identifiers in the headers of one or more functions in alphabetical order, system variables first, with no duplicates.

Syntax: ORDLOC '*fnlist*'
ORDLOC *tieno compno passno*

The argument is a character scalar, vector, or matrix that contains the names of the functions you want to modify; or, the argument is a two- or three-element numeric vector that contains a file tie number, component number, and optional passnumber. If the argument is numeric, ORDLOC treats the character vector in the component as the function listing. After reordering the local identifiers, ORDLOC replaces the function listing in the component. To modify multiple file components, use a two- or three-column numeric matrix argument.

Example:

```
      ⎕crl 'DRIVER[0]'
DRIVER;STOP;⎕PW;COUNT;GO
      ORDLOC 'DRIVER'
Locals reordered in <DRIVER>
      ⎕crl 'DRIVER[0]'
DRIVER;COUNT;GO;STOP;⎕PW
```

PACKVR Compress Function

Purpose:
Compresses the visual representation of a function by removing excess spacing and by combining lines with diamonds.

Syntax: *vr2* ← PACKVR *vr1*

Example:

```
      ⎕def PACKVR ⎕vr 'SPACIOUS'
```

RELABEL Use Consistent Labels

Purpose: Modifies one or more functions to use a consistent set of labels (L1, L2, etc.).

Syntax: RELABEL '*fnlist*'
RELABEL *tieno compno passno*

The argument is a character scalar, vector, or matrix that contains the names of the function you want to modify; or, the argument is a two- or three-element numeric vector that contains a file tie number, component number, and optional passnumber. If the argument is numeric, RELABEL treats the character vector in the component as the function listing. After relabeling the identifiers, RELABEL replaces the function listing in the file component. To modify multiple file components, use a two- or three-column numeric matrix argument.

Example:

```
      RELABEL 'SCRAMBLED'
⋆N.B.:  Label <GOON> occurs only once in <SCRAMBLED>; relabeled as <L6>
⋆N.B.:  Label <END> occurs only once in <SCRAMBLED>; relabeled as <L9>
```

RELOCALVR Replace Local Variable Names

Purpose: Replaces local variable names in the visual representation of a function.

Syntax: *vr2* ← {*new*} {*saved*} RELOCALVR *vr1*

There are two optional left arguments. The argument *new* is a list of new local variable names or a list of prefixes. RELOCALVR uses either the new names or constructs the new names from the list of prefixes. The argument *saved* is a list of local variables names to ignore. RELOCALVR does not change these names.

Effect:
Replaces local variables names in the visual representation of a function, using the names or prefixes that you specify in *new*. RELOCALVR does not change any names listed in the optional argument *saved*.

REPLACEVR Replace Identifiers in Function

Purpose: Replaces selected identifiers in the visual representation of a function.

Syntax: *vr2* ← '/*old1*/*new1*/*old2*/*new2*/…' REPLACEVR *vr1*

The left argument is a segmented string that consists of delimiters and pairs of identifier names. The first character is the delimiter. The succeeding pairs consist of an identifier name to replace followed by the replacement name. The right argument is the visual representation of the function whose identifiers you want to replace.

SCAN Perform General Workspace Search and Replace

Purpose: Performs a variety of function search and replace operations throughout a workspace.

Syntax: *specs1* SCAN *specs2*

See the variable SCANhow for documentation on this function.

SHARE Grant File Access

Purpose: Modifies the access matrix of a file to grant various forms of access to other users.

Syntax: SHARE '*filename*'
SHARE *tieno*

This interactive function modifies the access matrix of a file to grant different forms of access to different users. For explicit directions, see the variable SHAREhow.

SNUFF Remove Standard Comments

Purpose:
Removes standard comments from one or more functions, except those that begin with lamp del (⍝∇), lamp del-tilde (⍝⍫), or lamp right-brace (⍝{).

Syntax: SNUFF '*fnlist*'
SNUFF *tieno compno passno*

The argument is a character scalar, vector, or matrix that contains the names of the functions you want to modify; or, the argument is a two- or three-element numeric vector that contains a file tie number, component number, and optional passnumber. If the argument is numeric, SNUFF treats the character vector in the component as the function listing. After removing the comments, SNUFF replaces the function listing in the file component. To modify multiple file components, use a two- or three-column numeric matrix argument.

Example:

```
      SNUFF 'PRODUCTION'
Comments removed from <PRODUCTION>
```

STORAGE List Object Sizes

Purpose:
Shows the amount of storage, in decreasing order of size, that the objects in the workspace use.

Syntax: STORAGE

Example:

```
      STORAGE
⎕wa = 1449128
3392 FN  CUMSUM
1904 FN  ROWFIND
1424 FN  LTIMES
1232 FN  SStoMAT
880  FN  MFHIRES
720  FN  EXT
688  FN  DEB
416  FN  OVER
400  FN  MFRESOLUTION
336  FN  ROVER
160  VAR GRPMFFNS
16   VAR CUMPCT
16   VAR wsid
```

UNCOMMENTVR Remove Comments

Purpose: Removes the comments from the visual representation of a function.

Syntax: *vr2* ← UNCOMMENTVR *vr1*

UNDIAM Remove Diamonds

Purpose:
Removes the diamonds from one or more functions so that each statement appears on a separate line.

Syntax: UNDIAM '*fnlist*'
UNDIAM *tieno compno passno*

The argument is a character scalar, vector, or matrix that contains the names of the functions you want to modify; or, the argument is a two- or three-element numeric vector that contains a file tie number, component number, and optional passnumber.

If the argument is numeric, UNDIAM treats the character vector in the component as the function listing. After removing the diamonds, UNDIAM replaces the function listing in the component. To modify multiple file components, use a two- or three-column numeric matrix.

Example:

```
      UNDIAM 'CALC REPORT'
<CALC> undiamondized
<REPORT> undiamondized
```

UNLABELVR Remove Labels

Purpose: Replaces labels with line numbers in the visual representation of a function.

Syntax: *vr2* ← UNLABELVR *vr1*

UNLOCKEDFNS List Unlocked Functions

Purpose:
Returns a character matrix that contains the names of all the unlocked functions in the active workspace. If all the functions are locked, the result is an empty matrix.

Syntax: *fnlist* ← UNLOCKEDFNS

Example:

```
      UNLOCKEDFNS
ACCUM
AVERAGE
FIND
STAT
WRITE
```

UNPACKVR Expand Function

Purpose:
Splits function lines at diamonds and insert spaces for readability. This function reverses the effect of PACKVR

Syntax: *vr2* ← UNPACKVR *vr1*

Example:

```
      ⎕def UNPACKVR ⎕vr 'COMPRESSED'
```

UNTAINT Restore File after Network Crash

Purpose: Restores the usability of a file that was left "tainted" by a network crash.

Syntax: UNTAINT '*filename*'

USEDBY Return Global Objects

Purpose:
Returns the list of the global objects in the workspace that are referenced by a list of unlocked functions.

Syntax: *namelist* ← USEDBY '*fnlist*'

The argument is a list of function names. The list can be a character matrix with one name per row, or a character vector with names separated by spaces.

The result is a matrix of function and variable names currently in the workspace whose definitions are global to any of the functions in the right argument, including the functions in *fnlist*. The names are in alphabetical order, with one name per row. If none of the functions in the right argument exists, the result is a matrix with no rows.

Example:

```
      USEDBY 'ADDRECORD DELETERECORD'
ADDRECORD
CHECKADD
CHECKDELETE
DELETERECORD
FILE
GETRECORD
PUTRECORD
      ⎕ERASE ALLBUT USEDBY 'DRIVER'
```

∆VR Return Standard Visual Representation

Purpose:
Modifies the visual representation of a function to conform to the ⎕vr format of other APL64 systems that do not support arbitrary statement spacing.

Syntax: *visrepr* ← ∆VR '*fnname*'

Example:

```
      ⎕vr 'TEST'
    ∇ TEST ;  A  ;  B
[1]   A ← 1    ⋄   B ← 2
[2]       ⍝ COMMENT
[3]   C←3⋄D←4⍝COMMENT 2
[4]     LABEL  :  STATEMENT
[5]       '  QUOTED   STRING   '  ⍝  SPACED   O U T  COMMENT
[6]   E  [   5  6  ]  ⋄ ⎕ , ⎕
    ∇
      ∆VR'TEST'
    ∇ TEST;A;B
[1]   A←1 ⋄ B←2
[2]  ⍝ COMMENT
[3]   C←3 ⋄ D←4 ⍝COMMENT 2
[4]  LABEL:STATEMENT
[5]   '  QUOTED   STRING   ' ⍝  SPACED   O U T  COMMENT
[6]   E[5 6] ⋄ ⎕,⎕
    ∇
```

WSSHOW Display Character String

Purpose:
Prints a report that shows occurrences of a character string in all the unlocked functions in the active workspace.

Syntax: WSSHOW '*string*'

The argument is a character vector containing the string you want to show. WSSHOW searches the entire display form of each function in the active workspace. In particular, WSSHOW scans line numbers, comments, character constants, and line-ending newline characters. WSSHOW prints the complete text of each line containing the character string.

Example:

```
      WSSHOW '⎕FRE'
Searching 36 functions
1 IN <CHECKFILE>:
[3]  STATUS←⎕FREAD TN,2
2 IN <PROCESS>:
[7]  (UPDATE ⎕FREAD TN,5) ⎕FREPLACE TN,5
```

XREF Cross Reference Identifiers

Purpose:
Displays a cross-reference of a function, beginning with the header and followed by an alphabetical list of all the identifiers in the function.

Syntax: XREF '*fnname*'
XREF *tieno compno passno*

The argument can be character scalar or vector that contains the name of the function to cross reference, or it can be a two- or three-element numeric vector that contains a file tie number, component number, and optional passnumber. If the argument is numeric, XREF treats the character vector in the component as the function listing.

XREF indicates the usage of each identifier as follows:

- A semicolon (`;`) indicates a local identifier
- A colon (`:`) indicates a label
- A blank indicates a global identifier.

Example:

```
      ⎕vr 'CENTER' ◇ XREF 'CENTER'
    ∇ R←A CENTER B
[1]   B←,B ◇ →(0=1↑0⍴A)⍴L1 ◇ A←RWTD A
[2]  L1:A←⌈/0,+\×/ 0 ¯1 ↓A
[3]   R←(1,A)⍴A↑((⌈0.5×0⌈A-⍴B)↑' '),B
    ∇
    ∇ R←A CENTER B
A;           1←    1     2←    2     3
B;           1←    1     3
L1:          2:    1
R;           3←
RWTD         1
→            1
:            2
```

Using the *WFCARE* Workspace

The `WFCARE` workspace contains programs to help you maintain workspace files, which are file images of workspaces. The table shows the organization of a workspace file. These files are useful as:

- a temporary input file for workspace documentation or workspace comparison procedures
- a backup facility for workspaces
- a storage facility for unlocked functions and variables
- a medium for transferring workspaces between APL systems on different computers.

Workspace File Organization

Component	**Information**
`1` to `3`	Reserved for `WSDOC` state information
`4`	`⎕alx`
`5`	`⎕elx`
`6`	`⎕sa`
`7` to `9`	Reserved for `WSDOC` state information
`10`	`⎕lx`
`11`	`⎕wssize, ⎕wsowner, ⎕wsts`
`12`	`⎕wsid`
`13`	`⎕si`
`14`	`⎕wa, ⎕io, ⎕pp, 0, 0, ⎕ct, ⎕rl, ⎕symb, 1↑⎕ai`
`15`	Function list as a character matrix
`16`	Corresponding component locations
`17`	Corresponding function sizes
`18`	Variable list as a character matrix
`19`	Corresponding component locations
`20`	Corresponding variable sizes
`21` to `30`	Reserved
`31` to `wfoffset`	Reserved for user if `wfoffset` defined
`wfoffset+1` on	Visual representations of unlocked functions and data for variables

The functions in `WFCARE` store functions and variables in, or retrieve them from, an existing workspace file. A general purpose search-and-replace utility operates on all unlocked functions in a workspace file.

Function Descriptions

ALPHABETIZE Alphabetize Function or Variable Lists

Purpose: Alphabetizes the function or variable lists (or both) in a workspace file.

As you add new functions (with FNADD and FNPUT) and variables (with VARADD and VARPUT), the lists in components 15 and 18 of a workspace file get out of order. If you access these components directly in your configuration software, you may want to keep them in alphabetical order. ALPHABETIZE reorders the lists in a workspace file relating to functions (components 15 to 17) or variables (components 18 to 20), or both, so that the namelists are in alphabetical order.

Syntax: *tienos* ALPHABETIZE *codes*

The left argument is a numeric vector of tie numbers of workspace files whose lists you want to alphabetize. The right argument contains codes that indicate which lists are to be alphabetized and optionally whether identifiers with the same alphabetic prefix but different numeric suffixes should be maintained in alphabetic order (ABC15 before ABC2) or numerical order (ABC2 before ABC15).

Use FNS to indicate the function list, VARS to indicate the variable list, and FNS,VARS to indicate both. Supplement with NUMERICS to request numerical order rather than the default alphabetical order.

Example:
Alphabetize the functions in the workspace files tied to 90 and 92.

```
       90 92 ALPHABETIZE FNS
```

Alphabetize the functions and variables in the file tied to 91 but preserve numerical order in similar names.

```
       91 ALPHABETIZE FNS,VARS,NUMERICS
```

FILETOWS Re-Create Workspace from File Image

Purpose: Re-creates a workspace that has been captured as a workspace file.

Syntax: FILETOWS *tieno*

The argument is the file tie number of a workspace file originally created by the WSTOFILE function.

Effect:
If possible, FILETOWS defines in the active workspace all the variables stored in the file . If possible, it also defines in the workspace all the unlocked functions in the file . FILETOWS sets the system variables ⎕io, ⎕pp, ⎕ct, ⎕rl, ⎕lx, ⎕alx, ⎕elx, ⎕sa, and ⎕wsid.

FILETOWS prints error messages for any variables or functions it is unable to define in the workspace. Locked functions can never be defined, and it may not be possible to define some unlocked functions in the workspace due to WS FULL or other problems. Diagnostic messages describe where problems occur.

Example:

```
       )CLEAR
CLEAR WS
       )COPY 11 WFCARE
SAVED . . .
       '2 REPORTWF' ⎕FTIE 90
       FILETOWS 90
2 REPORT SAVED . .
       )FNS
```
functiona functionb functionc . . .
```
       )VARS
```
variablea variableb variablec . . .

`FNADD` Add Functions to Workspace File

Purpose:
Adds functions in the active workspace to a workspace file. The functions must be unlocked and cannot already exist in the file.

Syntax: *message* ← *tieno* `FNADD` `'`*fnlist*`'`

The left argument is the tie number for the workspace file. The right argument is a list of functions you want to add; the list can be either a character vector with the names separated by spaces, commas, or newline characters (`⎕tcnl`), or a character matrix with one name per row, left justified.

The result is a character vector that describes the errors for any functions that could not be added. An empty vector means that all the functions were added successfully.

Example:

```
      90 FNADD 'RPT3 RPT4 ALLREPORTS'
```

`FNERASE` Erase Functions from Workspace File

Purpose: Erases functions from a workspace file.

Syntax: *message* ← *tieno* `FNERASE` `'`*fnlist*`'`

The left argument is the tie number for the workspace file. The right argument is a list of functions you want to erase; the list can be either a character vector with the names separated by spaces, commas, or newline characters, or a character matrix with one name per row, left justified.

The result is a character vector that describes the errors for any functions that could not be erased. An empty vector means that all the functions were erased successfully.

Note:
To prevent inadvertent erasures, `FNERASE` deletes only the function directory entries, not their visual representations. To recover all the file storage from discardable functions, there are four steps: run `WFDIR` to determine their location in the workspace file; run `FNERASE`; replace the identified components with empty vectors; use `⎕fdup` to compact the file.

Example:

```
      90 FNERASE 'REPORT4 REPORT5'
```

`FNGET` Define Functions from Workspace File

Purpose: Retrieves functions from a workspace file and defines them in the active workspace.

Syntax: *message* ← *tieno* `FNGET` `'`*fnlist*`'`

The left argument is the tie number for the workspace file. The right argument is a list of functions to define; the list can be either a character vector with the names separated by spaces, commas, or newline characters, or a character matrix with one name per row, left justified.

The result is a character vector that describes the errors for any functions that could not be defined. An empty vector means that all the functions were defined successfully.

Example:

```
      90 FNGET 'REPORT1 REPORT5'
```

FNPUT Add Functions to Workspace File

Purpose:
Adds functions to or replaces functions in a workspace file. Functions must be unlocked, but a function can already exist in the file.

Syntax: *message* ← *tieno* FNPUT '*fnlist*'

The left argument is the tie number for the workspace file. The right argument is a list of functions you want to add or replace; the list can be either a character vector with the names separated by spaces, commas, or newline characters, or a character matrix with one name per row, left justified.

The result is a character vector that describes the errors for any functions that could not be added or replaced. An empty vector means that all the functions were added or replaced successfully.

Example:

```
      90 FNPUT 'REPORT1 REPORT5'
```

FNREPLACE Replace Functions in Workspace File

Purpose:
Replaces functions in the active workspace in a workspace file. The functions must be unlocked and must already exist in the file.

Syntax: *message* ← *tieno* FNREPLACE '*fnlist*'

The left argument is the tie number for the workspace file. The right argument is a list of functions you want to replace; the list can be either a character vector with the names separated by spaces, commas, or newline characters, or a character matrix with one name per row, left justified.

The result is a character vector describing the errors for any functions that could not be replaced. An empty vector means that all functions were successfully replaced.

Example:

```
      90 FNREPLACE 'RP1 RPT5 ALLREPORTS'
```

VARADD Add Variables to Workspace File

Purpose:
Adds variables in the active workspace to a workspace file. The variables cannot already exist in the file.

Syntax: *message* ← *tieno* VARADD '*varlist*'

The left argument is the tie number for the workspace file. The right argument is a list of variables you want to add; the list can be either a character vector with the names separated by spaces, commas, or newline characters, or a character matrix with one name per row, left justified.

The result is a character vector describing the errors for any variables that could not be added. An empty vector means that all the variables were successfully added.

Example:

```
      90 VARADD 'STATEVEC RPRTFMT FNAMES'
```

VARERASE Erase Variables from Workspace File

Purpose: Erases variables from a workspace file.

Syntax: *message* ← *tieno* VARERASE '*varlist*'

The left argument is the tie number for the workspace file. The right argument is a list of variables to erase; the list can be either a character vector with the names separated by spaces, commas, or newline characters, or a character matrix with one name per row, left justified.

The result is a character vector describing the errors for any variables that could not be erased. An empty vector means that all variables were successfully erased.

Note:
To prevent inadvertent erasures, `VARERASE` only deletes variable directory entries, not their values. To recover all the file storage from discardable variables, there are four steps: run `WFDIR` to determine their location in the workspace file; run `VARERASE`; replace the identified components with empty vectors; use `⎕fdup` to compact the file.

Example:

```
      90 VARERASE 'STATEVEC FNAMES'
```

`VARGET` Assign Variables Stored in Workspace File

Purpose: Assigns variables in the active workspace that are stored in a workspace file.

Syntax: *message* ← *tieno* `VARGET` `'`*varlist*`'`

The left argument is the tie number for the workspace file. The right argument is a list of variables to define; the list can be either a character vector with the names separated by spaces, commas, or newline characters, or a character matrix with one name per row, left justified.

The result is a character vector describing the errors for any variables that could not be defined. An empty vector means that all variables were successfully defined.

Example:

```
      90 VARGET 'STATEVEC RPRTFMT FNAMES'
```

`VARPUT` Add Variables to Workspace File

Purpose:
Adds variables to or replaces variables in a workspace file. A variable can already exist in the file.

Syntax: *message* ← *tieno* `VARPUT` `'`*varlist*`'`

The left argument is the tie number for the workspace file. The right argument is a list of variables you want to add or replace; the list can be either a character vector with the names separated by spaces, commas, or newline characters, or a character matrix with one name per row, left justified.

The result is a character vector describing the errors for any variables that could not be added or replaced. An empty vector means that all variables were successfully added or replaced.

Example:

```
      90 VARPUT 'NEWREPORTFMT FILENAMES'
```

`VARREPLACE` Replace Variables in Workspace File

Purpose: Replaces variables in a workspace file. The variables must already exist in the file.

Syntax: *message* ← *tieno* `VARREPLACE` `'`*varlist*`'`

The left argument is the tie number for the workspace file. The right argument is a list of variables you want to replace; the list can be either a character vector with the names separated by spaces, commas, or newline characters, or a character matrix with one name per row, left justified.

The result is a character vector describing the errors for any variables that could not be replaced. An empty vector means that all variables were successfully replaced.

Example:

```
      90 VARREPLACE 'STATEVEC FILENAMES'
```

`WFDIR` Return Component Directory

Purpose: Returns a function and variable component directory for a workspace file.

Syntax: *directory* `←` `WFDIR` *tieno passno*

The argument is the tie number for the workspace file. An optional second element can contain a file pass number. The result is a character matrix containing the file name, an alphabetical list of functions and their component locations, and an alphabetical list of variables and their component locations. The lists are folded into three columns for compactness. If the tied file is not a workspace file, the result is a character vector error message.

`WFSEARCH` Search Workspace File for Matches

Purpose:
Searches some or all unlocked functions in a workspace file for one or more strings, and displays or replaces all matches. Searching is done only with APL code strings, but can be expanded to include character constants or comments. You can search for syntactic or non-syntactic elements. You can also optionally define your own functions for displaying matches.

Syntax: *message* `←` `'`*tieno fnlist*`'` `WFSEARCH` `'`*strings*`'`

The left argument indicates the file tie number of the workspace file, the functions in the workspace to be searched, the portions of these functions to be searched, whether to search for syntactic or non-syntactic elements, and whether to display or replace matches. If the left argument contains only the names of functions, searching is done only with the APL code strings of the functions, the searching includes non-syntactic elements, and matches are displayed (not replaced). This behavior can be modified by including any of the following five characters in the left argument:

`"` (double quote) - also search character constants

`⍝` (comment symbol) - also search comments

`⌈` (Alt-S) - search for syntactic elements

`⍴` (Alt-R) - replace strings

`⎕` (Alt-L) - display strings (automatic if `⍴` absent).

The left argument can be either a character vector or matrix. If the argument is a vector, use spaces, commas, or newline characters to separate the names of the functions to be searched. Use spaces to separate the file tie number of the workspace file from the names and special characters. If the argument is a matrix, it must have one name per row. The file tie number of the workspace file and any of the special characters should be in one separate row of the matrix. If the argument is a vector containing only spaces, the file tie number of the workspace file, and some of the special characters, then all functions in the workspace file are searched. If the argument contains only the tie number, it can be numeric.

The right argument contains a list of one or more strings to search for and replace (optional). The first character is the delimiter character. If only searching is done, then matches are found to any of the strings supplied in the argument. If strings are to be replaced, the strings come in pairs: first the search string, then the replacement string.

Replacements are made in the order of appearance of the search strings. However, once a string has been replaced, the replacement string is not searched when further searches are performed on the same function. Thus, a value for *strings* of `'/A/B/B/A'` interchanges the names of `A` and `B` in the function.

The result *message* is a character vector containing the names of the functions containing matches (if display alone was requested), or containing changes (if replacement was requested).

Effect:
If you want to display the matches in a special format, WFSEARCH looks for a function named search. You can write this function to create the special format. It must be dyadic: the left argument is the ⎕vr form of the function being searched; the right argument is a bit vector, of the same length as the left argument, that points to the matches.

If replacements have been done, search is called only after the function has been successfully redefined. The right argument to search then points to the first character of each insertion. If any insertion was empty, it points to the first character following the insertion.

If display alone is requested, no change occurs in the global environment. If replacement is requested, the changed function is defined locally. If this function definition creates a function not already in the workspace file, the new function is added to the file, and the function being searched is left unchanged. If the ⎕def creates a function with the same name as a function already in the file, the function is replaced and the function being searched is left unchanged.

WFSEARCH cannot be used to make replacements in a function on the workspace file named search. WFSEARCH displays appropriate error messages for names that are not the names of functions on the workspace file. WFSEARCH signals a FILE TIE ERROR to the calling environment if no file tie number is supplied, or if there is no file tied to the file tie number supplied. WFSEARCH signals a DOMAIN ERROR − RIGHT ARGUMENT to the calling environment if one or more of the strings being searched for is empty, or if an odd number of strings is present when doing replacement.

Example:
The first statement below finds 'FOO' and 'GOO', but due to non-syntactic searching, the second does not.

```
      '90 FOO GOO' WFSEARCH '/OO'
      '90 FOO GOOΓ' WFSEARCH '/OO'
```

To interchange 'A' and 'B', use the statement:

```
      '90 FOO GOOΓρ' WFSEARCH '/A/B/B/A'
```

WSTOFILE Capture File Image of Workspace

Purpose: Captures a file image of a workspace.

Syntax: WSTOFILE

Effect:
When called, WSTOFILE prompts for the name of the workspace file you want to create. It creates the file and displays the tie number and the number of functions and variables that it writes to the file. If the file name you supply is incorrectly formed or duplicates the name of an existing non-empty file, WSTOFILE prints a diagnostic message and prompts for a different name. Use empty or all blank input to exit.

Locked functions cannot be stored in a workspace file. The names of the locked functions in a workspace where WSTOFILE is executing are included in the file's function directory, but empty vectors are placed in the components allocated for their visual representations.

You can reserve a number of components on the file for your own use. If the variable wfoffset exists when you run WSTOFILE and the variable is a numeric singleton greater than 30, all components between 31 and wfoffset are reserved.

WSTOFILE can only approximate the values for ⎕wa and ⎕symb in the workspace being captured, since the process of copying WSTOFILE into the workspace changes their values. WSTOFILE references the eight system variables: ⎕io, ⎕ct, ⎕pp, ⎕rl, ⎕lx, ⎕elx, ⎕alx, and ⎕sa. The approximated value for ⎕symb[2] differs by one for each of these system variables that has not been previously referenced in the workspace. The approximated value of ⎕wa differs by 16 for each of ⎕alx and ⎕elx that has not been previously referenced in the workspace.

Example:

```
      )LOAD 2 REPORT
2 REPORT SAVED . . .
      )COPY 11 WFCARE WSTOFILE
```

```
SAVED . . .
      WSTOFILE
Workspace file   2 REPORTWF
Tied to 90
37 functions
16 variables
```

Using the *WSCOMP* Workspace

The WSCOMP workspace allows you to compare the functions and variables in two workspaces. The main function, also called WSCOMP, operates on file images of workspaces the WSTOFILE or WSTOFILENO functions create. These file images are known as workspace files.

By default, WSCOMP lists the definitions of the function and the values of the variables that differ, and lists the names of functions and variables that are identical in both workspaces or that are in only one workspace.

For each function that is different in the two workspaces, WSCOMP displays the function lines that are different or missing from the function in one workspace. WSCOMP does not display lines that are identical, even if their numbers differ because of insertions or deletions. When a function is locked in one workspace, WSCOMP notes that fact and does not display the unlocked counterpart in the other workspace.

State settings in the workspace determine WSCOMP options. You can review and change these settings with the full-screen WCSTATE function. The possible choices follow.

- You can compare functions, variables, or both.
- You can display lists of the chosen objects (functions or variables) that are equal in both workspaces.
- For the objects you choose to compare, you can display a list of the chosen objects in the first workspace that are not present in the second, or vice versa.
- You can check if the referenced files are actually workspace files.
- You can ignore special comments (⍝⍫) in function comparisons. You may want to use these comments to store configuration information that may naturally vary from workspace to workspace.
- You can print timestamps that indicate when functions and variables were stored in their respective workspace files for functions and variables that differ.

WSCOMP normally list the items in the following order.

- the names of the functions that are in only one workspace.
- the lines that are different in the first workspace for functions that are in both workspaces.
- the names of the functions that are identical in both workspaces.
- the names of the variables that are only in one workspace.
- the names of the variables that are identical in both workspaces.

Using WSCOMP to Compare Workspaces

Follow these steps to compare two workspaces:

1. Prepare workspace files for all the workspaces you want to compare. Load each workspace, copy the WSTOFILE function into the workspace, and create the workspace file. Note that the files are tied to tie numbers 90 and 91.

```
      )XLOAD COLOR
9 COLOR SAVED . . .
      )COPY WFCARE WSTOFILE
SAVED . . .
      WSTOFILE
Workspace file  COLORWF
Tied to 90
53 functions
6 variables
      )XLOAD SETUP
9 SETUP SAVED . . .
```

```
      )COPY WFCARE WSTOFILE
SAVED . . .
      WSTOFILE
Workspace file  SETUPWF
Tied to 91
26 functions
10 variables
```

2. Load the WSCOMP workspace and check the comparison settings. Select the comparison options with the WCSTATE function. To restore the default settings, execute WCDEFAULT.

```
      )LOAD WSCOMP
SAVED . . .
      WCSTATE
```

3. Run the WSCOMP function. WSCOMP first compares the functions in the workspaces and then the variables in the workspaces. Note that you are comparing COLOR to SETUP by virtue of the time numbers you specify as the arguments.

```
      90 WSCOMP 91
Workspace comparison of <9 COLOR> to <9 SETUP>
         Printed at 3/01/90 at 15:37:03

45 functions in <9 COLOR> not in <9 SETUP>
.
.  List of function names
.
18 functions in <9 SETUP> not in <9 COLOR>
.
.  List of function names
.
===== Function -- ATTRNUM in <9 COLOR>
.
.  List of lines that are different
.
===== Function -- DEB in <9 COLOR>
.
.  List of lines that are different
.
===== Function -- UPPERCASE in <9 COLOR>
.
.  List of lines that are different
.
5 functions identical in both workspaces:
  BG16    BLINKING    CBE    CBE16    GRAPHICSCARD

9 variables in <9 COLOR> not in <9 SETUP>:
  ATTRIBUTES    HOW_COLORIZE_FNS     wsid
  DESCRIBE      ∆ui

9 variables in <9 SETUP> not in <9 COLOR>:
  COLORS  atP351P dlP351P  ftP351P  ttP351P
  atFX80  dlFX80     ftFX80     ttFX80

1 variable identical in both workspaces:
  CBEpanel
```

4. Erase the workspace files if you have no further use for them.

```
      'COLORWF' ⎕FERASE 90
      'SETUPWF' ⎕FERASE 91
```

Comparing Workspaces Automatically

Rather than follow the preceding steps, you can use the `EZWC` function to create and tie temporary workspace files and call `WSCOMP` automatically. `EZWC` also erases the files automatically when the workspace comparison is complete.

```
             'COLOR' EZWC 'SETUP'
      WS files:
      V3350376
      W3350376
      .
      .
      .
      Comparison details
      .
      .
      .
```

Function Reference

`EZWC` Generate Automatic Workspace Comparisons

Purpose: `EZWC` is a simplified packaging of the `WSCOMP` function.

Syntax: `'`*wsname1*`' EZWC '`*wsname2*`'`

The arguments are the names (including the nondefault library number or directory) of the two workspaces to be compared. The saved copy of the active workspace must contain the `WSCOMP` workspace.

Effect:
`EZWC` is an unlocked prototype function that you can modify to your requirements. It creates two temporary workspace files to store the file images of the two named workspaces. These file names are based on `⎕ts` to minimize the danger of conflicts with existing files. `EZWC` uses the `WSTOFILENO` function to build the workspace files.

Example:

```
      '10 PROD' EZWC 'TEST'
```

`WCDEFAULT` Restore Default Settings

Purpose: Restores default settings for workspace comparisons.

Syntax: `WCDEFAULT`

`WCSTATE` Review and Change Settings

Purpose: Reviews and changes the `WSCOMP` state settings.

Syntax: `WCSTATE`

Requires: `SELECT`, `WC∆Grps`, `WC∆Params`, `WC∆Screen`, `WCDEFAULT`, or `env`

Effect:
Displays a screen containing all possible `WSCOMP` options. You can use the keys on the cursor pad to change the settings. The current settings are always highlighted.

If an option group allows exactly one choice, use the left and right cursor keys to toggle among the choices. When your selection is highlighted, press Enter to move to the next option group.

For groups with multiple choices, use the left and right cursor keys to move through the choices. To add an option, move to the option and press Ins. To remove an option, move to the option and press Del. When you complete your selections, press End.

`WSCOMP` Compare Two Workspaces

Purpose: Compares the functions and variables in two workspaces and displays the differences.

Syntax: *tieno1* `WSCOMP` *tieno2*

Effect:
`WSCOMP` uses the state settings you establish with the `WCSTATE` function. Its arguments are the tie numbers of the two workspace files corresponding to the workspaces to compare.

To direct output to a printer or file, copy the `OUTPUT` workspace and run the appropriate functions. `WSCOMP` uses the new destination. If output is to file, you must tie or create the output file before running `WSCOMP`.

`WSTOFILENO` Create Workspace File

Purpose: Creates a workspace file from the active workspace.

Syntax: `WSTOFILENO` *tieno*

tieno is the tie number of an empty component file.

Example:

```
      'NEWWSFL' ⎕FCREATE 19
      WSTOFILENO 19
53 functions
6 variables
```

Chapter 7: Packaging a Runtime System

The runtime system is a special version of the APL64 system that can run an application but lacks the facilities to develop new APL programs. You use the APL64 runtime interpreter to build packaged Windows applications that you can redistribute freely.

The features that are removed from the runtime system are:

- the APL64 session manager windows, including the History pane
- the editor windows for editing functions and variables
- Debugger
- the "This will end your APL64 session" dialog box.

All of the APL64 windows interface features are available in the runtime system, and applications are expected to rely completely on the facilities provided by `⎕wi`, `⎕ni`, `⎕wcall`, and `⎕na` for their user interfaces. An APL application does not need to have a visible user interface, however. It can be an invisible server program that communicates with other applications through an ActiveX interface, TCP/IP or other means.

The runtime system works as though `⎕sa←'OFF'` permanently. At any point that the APL system would require immediate execution input, or ⎕ or ⍞ input, the runtime system ends execution. Any implicit or explicit APL output from ⎕← or ⍞← or an unassigned result is simply ignored. `⎕inbuf` has no effect. The user sees only the user interface, if any, of your application. A runtime application typically creates and waits on its main form, then terminates execution when the user closes this form.

Any of the following also causes the runtime system to terminate with no error:

- no initial workspace specified
- an empty `⎕lx` in any workspace the application loads
- normal completion of the latent expression (`⎕lx`)
- a suspension of APL execution
- insufficient Windows memory

Note: Refer to Help | Developer Version GUI | Create Windows Runtime Executable for additional information.

Creating the APLNOW32.INI File for a Runtime System

The various aspects of tailoring a system and setting parameters for APLNOW32.INI files are applicable if you are packaging an application as a runtime system. There are three important issues that are relevant, especially to runtime systems:

- You can create additional sections in your application's `.INI` file to save application-specific information. When APL64 saves settings, it leaves any sections you create untouched, as well as any additional entries in existing sections except for the [Libraries] section, which it saves as a whole.
- In a runtime application, do not include a [User] section. If you do, you defeat the security associated with runtime workspaces; it allows anyone with a development system to load the runtime workspace.
- You can have the name of your application appear in some dialog boxes and window titles with the [Config]ApplicationName parameter set in your application's `.INI` file.

A Sample Runtime Application

The following example shows how to package a very simple APL64 application that displays the current time in a window. You can try the process yourself by creating the following two functions in a clear workspace:

The function Clock creates a form and then begins a Wait on the form. This Wait continues for as long as the application is running. When the Wait ends by the user closing the form, the main function ends, and the runtime system shuts down. Note that in the APL64 development system, this function will work without explicitly invoking the Wait method, since the implicit wait loop of the APL64 session manager allows timer events to invoke the ShowTime callback function. In the runtime system, no session manager wait loop is running.

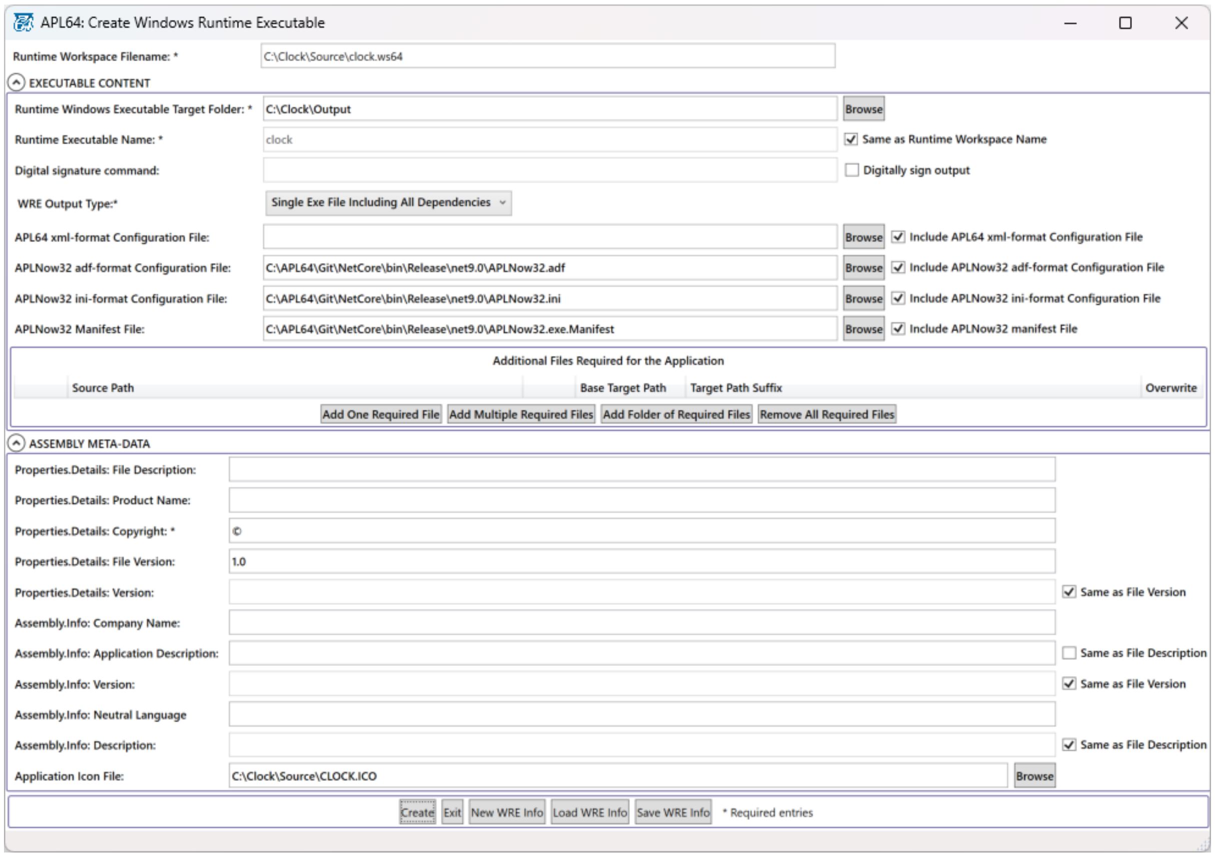
```
     ∇ Clock;f;l;t
       ⍝ Simple "Digital Clock" program that
       ⍝   demonstrates packaging a runtime
       ⍝   application for APL64

       f←'fmClock' ⎕wi 'Delete'
       f←'fmClock' ⎕wi 'New' 'Form' 'Close'
       f ⎕wi 'caption' 'Clock'
       f ⎕wi 'border' 17
       f ⎕wi 'where' 0 0 4 14

       l←'fmClock.lTime' ⎕wi 'New' 'Label'
       l ⎕wi 'caption' ''
       l ⎕wi 'where' .1 1 2 14
       l ⎕wi 'font' 'Arial' 1.5 1 'ansi'

       t←'fmClock.t1' ⎕wi 'New' 'Timer'
       t ⎕wi 'interval' 1000
       t ⎕wi 'onTimer' 'ShowTime'

       f ⎕wi 'Wait'
     ∇

     ∇ ShowTime;t
       ⍝ Show the time in the label field
       t←100⊥3↑3↓⎕ts
       'fmClock.lTime' ⎕wi 'caption' (,'G<Z9:99:99>' ⎕fmt t)
     ∇
```

Packaging the Application

The following steps illustrate how you can make the `Clock` function into a standalone Windows application:

1. Prepare the workspace to be self-starting by setting `⎕lx`, and save it:
   ```
   ⎕mkdir 'c:\clock\source'
   ⎕mkdir 'c:\clock\output'
   ⎕lx←'Clock'
   )save c:\clock\source\clock
   ```
2. Create an icon for the application using an icon editor. This step is optional; you can use the sample icon `CLOCK.ICO` that is supplied with APL64 in the Examples folder:

   C:\Users\programmer\AppData\Roaming\APLNowLLC\APL64\Examples to c:\clock\source\clock.ico
3. Select **Options | Create Runtime .Net Assembly | Create Windows Runtime Executable** menu to create the Windows runtime application.


APL64: Create Windows Runtime Executable

Runtime Workspace Filename: *	C:\Clock\Source\clock.ws64

EXECUTABLE CONTENT

Runtime Windows Executable Target Folder: *	C:\Clock\Output	
Runtime Executable Name: *	clock	✓ Same as Runtime Workspace Name
Digital signature command:		☐ Digitally sign output
WRE Output Type:*	Single Exe File Including All Dependencies	
APL64 xml-format Configuration File:		✓ Include APL64 xml-format Configuration File
APLNow32 adf-format Configuration File:	C:\APL64\Git\NetCore\bin\Release\net9.0\APLNow32.adf	✓ Include APLNow32 adf-format Configuration File
APLNow32 ini-format Configuration File:	C:\APL64\Git\NetCore\bin\Release\net9.0\APLNow32.ini	✓ Include APLNow32 ini-format Configuration File
APLNow32 Manifest File:	C:\APL64\Git\NetCore\bin\Release\net9.0\APLNow32.exe.Manifest	✓ Include APLNow32 manifest File

Additional Files Required for the Application

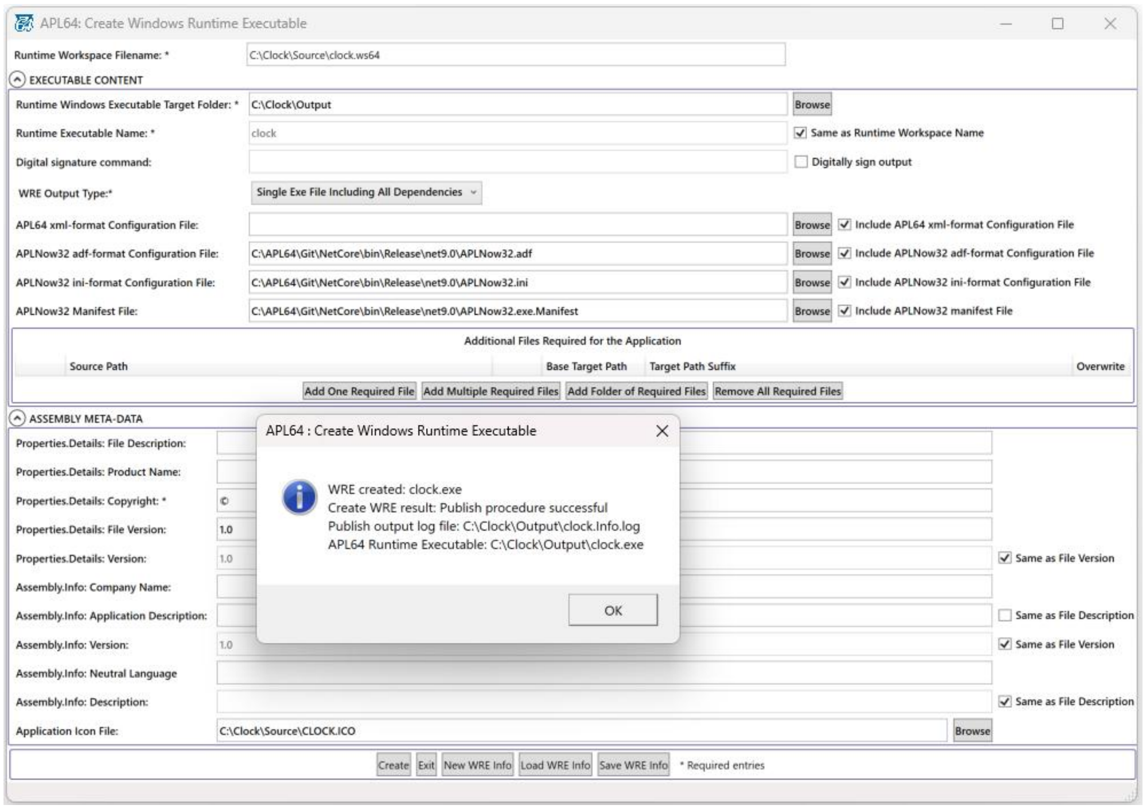
Source Path	Base Target Path	Target Path Suffix	Overwrite

ASSEMBLY META-DATA

Properties.Details: File Description:		
Properties.Details: Product Name:		
Properties.Details: Copyright: *	©	
Properties.Details: File Version:	1.0	
Properties.Details: Version:		✓ Same as File Version
Assembly.Info: Company Name:		
Assembly.Info: Application Description:		☐ Same as File Description
Assembly.Info: Version:		✓ Same as File Version
Assembly.Info: Neutral Language		
Assembly.Info: Description:		✓ Same as File Description
Application Icon File:	C:\Clock\Source\CLOCK.ICO	

Note: The folder c:\clock\output specified in the field **Runtime Windows Executable Target Folder** must exist.

4. Complete the form as shown above then click Create.
5. When successful, this should appear:


APL64: Create Windows Runtime Executable

Runtime Workspace Filename: *	C:\Clock\Source\clock.ws64	
EXECUTABLE CONTENT		
Runtime Windows Executable Target Folder: *	C:\Clock\Output	Browse
Runtime Executable Name: *	clock	☑ Same as Runtime Workspace Name
Digital signature command:		☐ Digitally sign output
WRE Output Type:*	Single Exe File Including All Dependencies	
APL64 xml-format Configuration File:		Browse ☑ Include APL64 xml-format Configuration File
APLNow32 adf-format Configuration File:	C:\APL64\Git\NetCore\bin\Release\net9.0\APLNow32.adf	Browse ☑ Include APLNow32 adf-format Configuration File
APLNow32 ini-format Configuration File:	C:\APL64\Git\NetCore\bin\Release\net9.0\APLNow32.ini	Browse ☑ Include APLNow32 ini-format Configuration File
APLNow32 Manifest File:	C:\APL64\Git\NetCore\bin\Release\net9.0\APLNow32.exe.Manifest	Browse ☑ Include APLNow32 manifest File

Additional Files Required for the Application

Source Path	Base Target Path	Target Path Suffix	Overwrite

Add One Required File | Add Multiple Required Files | Add Folder of Required Files | Remove All Required Files

ASSEMBLY META-DATA		
Properties.Details: File Description:		
Properties.Details: Product Name:		
Properties.Details: Copyright: *	©	
Properties.Details: File Version:	1.0	
Properties.Details: Version:	1.0	☑ Same as File Version
Assembly.Info: Company Name:		
Assembly.Info: Application Description:		☐ Same as File Description
Assembly.Info: Version:	1.0	☑ Same as File Version
Assembly.Info: Neutral Language		
Assembly.Info: Description:		☑ Same as File Description
Application Icon File:	C:\Clock\Source\CLOCK.ICO	Browse

Create | Exit | New WRE Info | Load WRE Info | Save WRE Info | * Required entries

APL64 : Create Windows Runtime Executable

WRE created: clock.exe
Create WRE result: Publish procedure successful
Publish output log file: C:\Clock\Output\clock.Info.log
APL64 Runtime Executable: C:\Clock\Output\clock.exe

OK

6. Click Ok to close the dialog.
7. Test the newly created Clock.exe application in c:\clock\output folder.

Files to Distribute with a Runtime Application

The bare minimum requirements for a runtime application is the Windows runtime executable created by the Windows Runtime Executable.

Techniques for Polishing Applications

You may wish to display some sort of banner immediately as your application is initializing. This can be done with the command line arguments SIP, SIT, or SIC and the ⎕SPLASH system function to hide it. Refer to the System Function manual for additional information.

You can set a return code while your runtime application is running by setting ⎕sys[22]. The calling application can capture this code using the Windows GetExitCodeThread API or the DOS batch errorlevel parameter.

Chapter 8: Transferring Applications between APL Systems

This chapter introduces the concepts and techniques that can help you transfer applications between various APL systems. It discusses compatibility issues and identifies the mechanisms for transferring your applications, workspaces, and files from other APL or APL+ systems to APL64, and in some cases, from APL64 to other systems. Additionally, the chapter includes some discussion of converting applications.

APL+ Systems Compatibility

Because APL as mathematical notation was carefully defined, the primitive functions in all APL systems are highly compatible and generally provide the same result. Of course, APL as a programming language has evolved over the years, sometimes in major ways; for example, nested arrays were added after APL+PC. The basic language of APL+ systems continues to evolve as well; for example, additional primitive functions in APL64 allow an axis designation since version `3`, and the Replicate and Expand primitives allow negative arguments in version `4.0`.

In general, except for the specific differences for a few functions between Evolution levels `1` and `2`, the computational aspect of any application that you transfer from an older system to a newer version continues to perform as expected. What differs among systems are the interfaces to the platform and the host operating system. The specifications for saving information in files and workspaces differs somewhat among systems, and the construction of the user interface differs greatly.

APL+DOS, APL+UNIX/LINUX (Version `4` or later), APL★PLUS II for VAX/VMS, and APL+PC all have the same `⎕av` order, so no character set conversions are necessary when moving native files among them. APL+Win added a number of vowels with diacritical marks used in European alphabets to `⎕av`. These characters replaced many of the pseudo line-drawing characters, some character positions that were undefined, and several obsolete APL symbols. In general, you should not require character set conversions when moving native files from the systems listed above to APL64. When you use native files, you have the highest degree of compatibility possible. If the order of characters in `⎕av` is identical between systems, you can use a native file on both systems without needing a conversion utility.

Workspace and Component File Compatibility

The internal representation of data in saved APL workspaces varies among systems because of hardware- or system-dependent characteristics. These variations mean, for example, that a workspace saved in the APL+PC and APL+DOS Systems cannot be loaded or copied directly by APL64.

A higher degree of compatible internal structure exists with APL component files.

- The `.SF` file formats that APL64, APL+Win and APL+DOS use are identical. Files you create on one system are usable on both.
- Using an appropriate file architecture switch, APL+UNIX/LINUX can create and use `.SF` files in the APL64 format. Those files that you create under APL+UNIX/LINUX in this format is fully usable with APL64.
- Using the APL utility functions in the `PCFILES` workspace, APL64 can read and write to the `.ASF` component files of APL+PC.

Transferring Applications

APL64 provides utility workspaces and commands to help you exchange workspaces and files with other APL systems. This table can help you decide which techniques to use to transfer your files and workspaces to APL64.

Transferring Files and Workspaces into APL64

If your source system is...	And you are transferring a ...	On the source system, use	To create a ...	Load into APL64 with
APL+DOS (Note 1)	Workspace (`.WS` file)	`WStoFILE` in the `CONVERT` workspace		`FILEtoWS` in the `CONVERT` workspace

If your source system is...	And you are transferring a ...	On the source system, use	To create a ...	Load into APL64 with
	Component file (`.SF` file)	nothing necessary		`⎕fread`
APL+UNIX/LINUX	Workspace	`WSTOFILE` in the `UTILITY` workspace	`.SF` file (Note 2)	`FILETOWS` in the `WFCARE` workspace
	Component file (`.SF` file) – Note 2	nothing necessary		`⎕fread`
APL+PC	Workspace (`.AWS` file)	`DTFN`/`DTFNALL` in the `UTILITY` workspace	`.T` file	`LFFN` in the `LFFN` workspace
		`WSTOFILE` in the `UTILITY` workspace	`.ASF` file	`PCFILETOWS` in the `PCFILES` workspace
	Component file (`.ASF` file)	nothing necessary		`N/A`
APL★PLUS for VM or MVS Version 7 or later	Workspace	`∆∆DUMPWS` in the `SLT` workspace	`.SLT` file	`∆∆LOADWS` in the `SLT` workspace
	Component file	`∆∆DUMPFILE` in the `SLT` workspace	`.SLT` file	`∆∆LOADFILE` in the `SLT` workspace
APL★PLUS for VAX/VMS	Workspace	`DUMPWS` in the `SLT` workspace	`.SLT` file	`∆∆LOADWS` in the `SLT` workspace
	Component file	`DUMPFILE` in `SLT` workspace	`.SLT` file	`∆∆LOADFILE` in the `SLT` workspace
APL2	Workspace	`)OUT` system command	`.ATF` file	`]in` user command

Notes:

1. To reverse the process, use `SAVEII` in the `CONVERT` workspace or `WSTOFILE` in the `WFCARE` workspace in APL64 and then `FILETOWS` (`WFCARE`) in APL+DOS. Alternatively, you can use `∆∆DUMPWS` and `∆∆LOADWS` in the `SLT` workspaces in both systems.
2. Applicable for a component file created for the APL+Win/APL64 architecture (format). When migrating a component file from one APL system to another, you may run into an issue with the file access matrix set in the component file. This is because the internal representation of a file owner is set via the user number, rather than username. Updating the access matrix is a common practice when migrating files between APL systems. This is partly due to the default user number that may be different on different systems. So, ensure that the file access matrix is set correctly in the APL component file prior to migrating to the other APL system. Refer ⎕FCREATE and ⎕FSTAC in the APL+UNIX/LINUX documentation for additional information.

Utilities for Transferring Applications

You can use the following workspaces and commands for transferring applications and data between systems.

- `]pccopy` User Command
  You can use this command to copy objects from a modest-sized APL+PC workspace directly into APL64. It uses `⎕cmd` to start an APL+PC runtime system that invokes the `DTFN` facility on the `PC` workspace and then runs `LFFN` to import the resulting transfer file into APL64. For large APL+PC workspaces, use one of the techniques listed below that write a file.
- `WFCARE` Workspace
  You can use `WSTOFILE` and `FILETOWS` in this workspace to store and retrieve images of workspace objects in component files, which you can then transfer between systems.
- `DTFN` and `LFFN` Workspaces
  You can use these workspaces to create a native file to move all or part of a workspace to another APL system with the same `⎕av`.

- `]xlate` User Command
  You can use this command to write data to a native file when a `⎕av` translation is necessary. See the User Command Descriptions chapter in this manual.
- `CONVERT` Workspace
  You can use this workspace to move workspaces between APL64 and APL+PC. This workspace contains facilities to diagnose areas of your applications where some conversion effort may be required.
- `PCFILES` Workspace
  You can use this workspace to read, write, or copy component files created on APL+PC. These files have `.ASF` extensions.
- `SLT` Workspace
  You can use this workspace to perform a Source Level Transfer between different APL systems.
- `]in` and `]out` User Commands
  You can use these commands to convert IBM's APL2 workspaces to APL64 workspaces, and vice versa.

You can also use the `]apllib` user command to identify what files and workspaces from various APL systems are in a particular library or directory. For example:

```
      ]apllib C:\APLWin19\MIGRATE
------ APL+PC
DTFN      LFFN      PCCOPY    SLT     W2FF2W
------ APL+PC .ASF Files ------
SLT
------ APL+DOS .WS Workspaces ------
CONVERT   DTFN      LFFN      PCFILES   SERUPLD   SERXFER   SLT
------ APL+DOS/Win .SF Files ------
SERMFFNS  SERPLFNS  SERUPFNS  SLT
------ APL+Win .W3 Workspaces ------
CONVERT   DTFN      LFFN      PCFILES   SLT  SLTX
------ APL64 .WS64 Workspaces ------
UCMDUTIL
```

Simple Tools for Transferring the Contents of Workspaces

To transfer the variables and unlocked functions of workspaces from other systems, you can use the techniques this section describes. You can write the contents of a workspace to a component file and back to a workspace using two functions from the `WFCARE` workspace. In addition, this workspace has several functions you can use to maintain workspace files. See the Programming Tools chapter in this manual for descriptions of these functions.

You can also write the contents of a workspace to a native file and read it back using the `DTFN` and `LFFN` workspaces.

Using `WSTOFILE` and `FILETOWS`

You can use the `WSTOFILE` and `FILETOWS` functions in the `WFCARE` workspace to transfer workspaces between systems with compatible component files. These functions provide a fast and convenient way to transfer workspaces among APL+PC, APL+UNIX/LINUX, APL+DOS and APL64.

Note: On APL+UNIX/LINUX and APL+PC, these functions are in the `UTILITY` workspace.

To use `WSTOFILE`, follow these steps.

1. On the source system, enter:
   ```
         )load myws
         )copy path\WFCARE WSTOFILE
   WSTOFILE
   ```
2. Respond to the `Workspace-file name:` prompt with the name for a new or empty component file; for example, *path*`\MYWSWF`. The `WSTOFILE` function writes all the unlocked functions, variables, and system variables in the workspace into the file.

3. Transfer the new component file to the receiving system.
4. On the receiving system, enter:

```
      )clear myws
)copy path\WFCARE FILETOWS
'mywswf' ⎕ftie 99
FILETOWS 99
)erase FILETOWS
)save mynewws
```

FILETOWS reads and defines the functions, variables, and system variables from the file into the workspace, sets ⎕wsid to its original value, and erases itself.

Note that SAVEPC in the CONVERT workspace (on this system) or PCWSTOFILE in the PCFILES workspace creates a .ASF file that can be used by FILETOWS in APL+PC. Conversely, LOADPC (CONVERT workspace) or PCFILETOWS (PCFILES workspace) does the equivalent of FILETOWS on a .ASF file created by WSTOFILE on APL+PC.

Using the DTFN and LFFN Workspaces

The functions in the DTFN and LFFN workspaces use native files to move applications to other APL+ systems with the same ⎕av, such as APL+UNIX/LINUX, APL+DOS, or APL+PC. Follow these steps to use the native file transfer technique.

1. Build the transfer file:

```
   )xload wsname
)copy path\DTFN
'wsname.t' ⎕ncreate ¯1
DTFNALL ¯1
⎕nuntie ¯1
```

2. Use any file transfer program that transfers unmodified binary data to move the file to the target system.
3. In APL on the target system, tie the native transfer file and reconstruct the workspace:

```
   )clear
)copy path\LFFN
'wsname.t' ⎕ntie ¯1
LFFN ¯1
)erase LFFN
)save myws
```

Function Reference – DTFN and LFFN

DTFN Dump Workspace Objects to Native File

Purpose:
Appends APL functions and variables to a native file for transfer to another APL system. When necessary, you have the option of performing ⎕av translations on either APL system.

Syntax:
objlist DTFN *ntn*

Example:

```
      'A:FINMODEL.XFR' ⎕ncreate ¯1
      'MODELDOC RPTDOC REPORTFN' DTFN ¯1
Starting size is 0
MODELDOC dumped
RPTDOC dumped
REPORTFN dumped
      ⎕nuntie ¯1
```

DTFNALL Dump Entire Workspace to Native File

Purpose:
Writes an entire workspace to a native file for transfer to another APL system. When necessary, you have the option of performing ⎕av translations on either APL system.

Syntax:
DTFNALL *ntn*

Example:
```
      'A:MYWS.XFR' ⎕ncreate ¯1
      DTFNALL ¯1
```

LFFN Load Workspace Objects from Native File

Purpose:
Recovers the functions and variables moved to this APL+ System after you dump them to a native file (with DTFN or DTFNALL) on another APL+ System.

Syntax:
LFFN *ntn*

Example:
```
      'A:FINMODEL.XFR' ⎕ntie ¯2
      LFFN ¯2
MODELDOC←
RPTDOC←
⎕def REPORTFN
```

Transferring Workspaces as Workspace Files

You can transfer entire workspaces between systems.

Using the CONVERT Workspace

The CONVERT workspace contains conversion facilities that make transferring workspaces between APL64 and APL+PC easier. Functions in the workspace flag known problem areas during the conversion. You can also write APL64 workspaces to APL+DOS using this workspace.

Converting from APL+PC to APL64

To convert a workspace from APL+PC to APL64, follow these steps.

1. Start APL+PC and load the workspace you want to convert.
```
      )xload MYWS
```
2. Copy the WSTOFILE function from the UTILITY workspace. Run WSTOFILE to save an image of your workspace in a special component file known as a workspace file.
```
      )copy UTILITY WSTOFILE
      WSTOFILE
 Workspace-file name:  MYWSWF
```
3. End APL+PC and start APL64.
4. Load the CONVERT workspace.
```
      )load CONVERT
```
5. Run the LOADPC function, using the DOS name of the workspace file as an argument. LOADPC generates a list of system-specific areas you may need to convert manually.
```
      LOADPC '\PC\MYWSWF'
```
6. Make any necessary modifications to the functions and variables in the new workspace and save it. See the "Using the UCMDS2 File" section, earlier in this chapter, for some commands you can use.

Converting from APL64 to APL+PC

To convert a workspace from APL64 to APL+PC, follow these steps.

1. Start APL64 and load the CONVERT workspace.

```
        )load CONVERT
```

2. Run the SAVEPC function from the CONVERT workspace to save an APL64 workspace into an APL+PC workspace file. Use the name of the workspace you want to move to APL+PC as an argument. SAVEPC loads the workspace, searches for potential problems, and prompts you for the name of a workspace file

```
        SAVEPC 'MYWSIII'
  Workspace-file name: \PC\MYWSPCWF
```

3. End APL64 and start APL+PC.
4. Copy the FILETOWS function from the UTILITY workspace into a clear workspace.

```
       )clear
CLEAR WS
       )copy UTILITY FILETOWS
SAVED...
```

5. Tie the workspace file.

```
        'MYWSPCWF' ⎕ftie 1
```

6. Run the FILETOWS function using the tie number as an argument.

```
        FILETOWS 1
```

   FILETOWS re-creates the workspace objects based on the images in the workspace file.

7. Erase FILETOWS, name the workspace, modify the workspace as suggested by SAVEPC, and save the converted workspace.

Transferring a Workspace from APL64 to APL+DOS

To move a workspace from APL64 to APL+DOS, you can use the SAVEII function. You use this function (which is a cover for the WSTOFILE function) as you use SAVEPC described above. It searches the workspace for code that may require manual conversion. CONVERT also contains simulations of the functions in the QPACK workspace from APL+PC. These functions create simulated packages that use nested arrays to bind a collection of functions and variables together. They emulate all the capabilities of QPACK except that they cannot contain locked functions. Run the Describe and Summary functions for more information on the tools in the CONVERT workspace.

Function Reference -- CONVERT

LOADPC Convert and Re-create a Workspace

Purpose:
Converts a workspace file saved under APL+PC and re-creates the workspace under APL64.

Syntax:
msgs ← LOADPC *pcfilename*

pcfilename is the name of the APL+PC workspace file to be converted into an APL64 workspace; *msgs* is messages and warnings about the success of the conversion

Comments:
Before using LOADPC, copy the WSTOFILE function from the APL+PC UTILITY workspace. Use WSTOFILE to convert the workspace to a workspace file.

As LOADPC converts the workspace file and re-creates the workspace, it displays messages about the contents of the workspace and warns you about the contents of the workspace that it could not convert; for example, functions that do not exist in APL64.

SAVEPC Convert and Save a Workspace as a PC Workspace File

Purpose:
Converts a workspace saved under APL64 as a workspace file compatible with APL+PC.

Syntax:
`SAVEPC` *workspacename*

workspacename is the name of the APL64 workspace you want to save as an APL+PC workspace file.

Comments:
`SAVEPC` loads an APL64 workspace, prompts you for a name, and saves it in an APL+PC workspace file. To re-create the workspace with APL+PC, run the `FILETOWS` function (from the `UTILITY` workspace) in a clear workspace. `SAVEPC` warns you about system differences as it saves the workspace.

SAVEII Convert and Save a Workspace as a II Workspace File

Purpose:
Converts a workspace saved under APL64 as a workspace file compatible with APL+DOS.

Syntax:
`SAVEII` *workspacename*

workspacename is the name of the APL64 workspace you want to save as an APL+DOS workspace file.

Comments:
`SAVEII` loads an APL64 workspace, prompts you for a name, and saves it in an APL+DOS workspace file. To re-create the workspace with APL+DOS, run the `FILETOWS` function (from the `WFCARE` workspace) in a clear workspace. `SAVEII` warns you about system differences as it saves the workspace.

∆pack Create a Simulated Package

Purpose:
Simulates the `PACK` function from the APL+PC `QPACK` workspace.

Syntax:
pkg `←` `∆pack` *namelist*

Comments:
This function packs the names you specify into a package. Each name must be a valid identifier name, but should not identify a locked function. Names cannot be labels. The system ignores multiple occurrences of the same name.

∆pchk Check if an Array is a Simulated Package

Purpose:
Determines if an array is a simulated package.

Syntax:
result `←` `∆pchk` *array*

The result is `1` if the array is a simulated package; otherwise, the result is `0`.

∆pdef Define a Simulated Package in a Workspace

Purpose:
Defines all the names in a simulated package in the workspace.

Syntax:
```
            ∆pdef pkg
namelist    ∆pdef pkg
```

Comments:
Defines the objects in the simulated package in the workspace. All the names in the optional left argument must be in the package. None of the names can be labels and none can reference a function appearing in the state indicator.

Note: If successful, `∆pdef` replaces workspace copies of the objects you specify with versions from the package. If you specify a name in the workspace that is not in the package, the system erases that object from the workspace.

`∆pex` Create a New Package that Excludes Parts of an Old Package

Purpose:
Creates a new simulated package by excluding a list of names from an existing package.

Syntax:
pkg2 ← *namelist* `∆pex` *pkg1*

`∆pins` Insert a Second Package into Another Package

Purpose:
Creates a new package by inserting a second package into the first package.

Syntax:
pkg3 ← *pkg1* `∆pins` *pkg2*

Comments:
This function inserts one package into another and returns the combined package. The combined package is the union of the first and second. If both original packages contain the same name, `∆pins` uses the value from the package you insert. The order of names in the combined package can be random.

`∆pnames` Return the Names in a Simulated Package

Purpose:
Lists the names in a simulated package.

Syntax:
namelist ← `∆pnames` *pkg*

Comments:
This function returns a matrix list of names in a simulated package. If the argument is not a package, the result is an empty vector.

`∆pnc` Return the Name Classes in a Simulated Package

Purpose:
Returns an integer vector indicating the name classes of the contents of a simulated package.

Syntax:
intvec ← `∆pnc` *pkg*
intvec ← *namelist* `∆pnc` *pkg*

Comments:
When you use this function monadically, it returns an integer vector that contains the name classes in a simulated package, in the same order that `∆pnames` returns the names in the package. When you use this function dyadically, it returns an integer vector containing the name classes of the names in the left argument, one element per name. The name classes are:

¯1	name used without referent
0	name not in package
2	variable
3	defined function
4	ill-formed name

∆ppdef Protectively Define a Simulated Package

Purpose:
Defines the contents of a simulated package in the workspace without replacing existing objects with objects in the package. It returns the list of objects that it could not define.

Syntax:
namelist ← ∆ppdef *pkg*
namelist2 ← *namelist1* ∆ppdef *pkg*

Comments:
When you use this function monadically, it defines all the names in a simulated package except those that already exist in the workspace. When you use this function dyadically, it defines all the names you specify in the left argument except those that already exist in the workspace.

∆psel Create a Second Simulated Package by Selecting a Subset

Purpose:
Creates a new simulated package from an existing simulated package by selecting a list of names from the existing package.

Syntax:
pkg2 ← *namelist* ∆psel *pkg1*

Comments:
The left argument contains the names you want in the new package. Each name in the list must exist in the first package.

∆pval Return the Value of a Variable in a Simulated Package

Purpose:
Returns the value of a variable you specify in a simulated package.

Syntax:
array ← *varname* ∆pval *pkg*

Comments:
A DOMAIN ERROR occurs if the variable you specify in the left argument is not in the simulated package you specify in the right argument.

Using the PCFILES Workspace

The PCFILES workspace contains a suite of utility functions you can use to access .ASF files that APL+PC recognizes as APL component files. This workspace allows you to use APL+PC component files without changing their format. You can also copy APL+PC files into the APL64 component file format. The following table lists the functions you use to work with .ASF files. These functions use native file operations in APL64, although they use the syntax of the component file functions.

Functions Available to Access .ASF Files as Native Files

Function	Description
FAPPEND	Appends a component to a .ASF file.
FCREATE	Creates and ties an empty .ASF file.
FDROP	Drops components from either end of an .ASF file.
FERASE	Erases tied .ASF files.
FLIB	Lists .ASF files.
FNAMES	Lists names of tied .ASF files.
FNUMS	Lists tie numbers of tied .ASF files.
FRDAC	Reads access matrix of .ASF files.
FRDCI	Reads component information of .ASF files.

Function	Description
FREAD	Reads components of .ASF files.
FRENAME	Renames a .ASF file.
FREPLACE	Replaces components in a .ASF file.
FRESIZE	Sets the size limit for a .ASF file.
FSIZE	Finds the size of a .ASF file.
FSTAC	Sets the access matrix of a .ASF file.
FTIE	Ties a .ASF file.
FUNTIE	Unties a .ASF file.

The syntax of each function is the same as the corresponding system function for APL64. Keep the following guidelines in mind.

- When you use a positive tie number with FCREATE, FTIE, and so on, the functions use the negative of that number as a native tie number. Make sure those negative values are not tie numbers for some other non-.ASF native files.
- The PCFILES utility functions do not observe the file access matrix stored in the .ASF files.
- FCREATE creates a .ASF file with user number ⎕ai[1] (default =1) as the owner; the default user number on the APL+PC System is 0. If you want to access a .ASF file you create in APL64 from APL+PC, you must either sign on as user number 1 in APL+PC or allow access to the file for user number 0. To accomplish the latter, execute the following statement from this workspace:

  0 ¯1 0 FSTAC *tieno*

Copying .ASF Files as Component Files

You can also use the WINcopyPC and PCcopyWIN functions to copy file data between APL64 component files and APL+PC component files.

Use WINcopyPC to copy data from APL+PC files to APL64 files. Create or tie a .SF file with one of the appropriate APL64 system functions and tie the .ASF file you want to copy using FTIE from this workspace. Then use

aplwinfile WINcopyPC *aplpcfile*

where *aplwinfile* is the tie number of the .SF file and *aplpcfile* is the tie number of the .ASF file. If the .SF file already exists, WINcopyPC appends the contents of the .ASF file to it.

Use PCcopyWIN to copy data from APL64 files to APL+PC files. Do not attempt to transfer nested or heterogeneous data to APL+PC. Create or tie the .ASF file using FCREATE or FTIE from this workspace. Tie the .SF file with a system function. Then use

aplpcfile PCcopyWIN *aplwinfile*

where *aplpcfile* is the tie number of the .ASF file and *aplwinfile* is the tie number of the .SF file. If the .ASF already exists, PCcopyWIN appends the contents of the .SF file to it.

Using a Source Level Transfer (SLT)

You can move your applications from one computer to another through a source level transfer. The source level transfer facility provided in APL64 conforms to the Workspace Interchange Convention (level 1) for converting APL functions, variables, and environmental information into character variables called canonical representation vectors (CRVs).

In contrast to the tools discussed earlier, SLT is the tool of choice when you need to use ⎕av translations. When you use this facility, you may want to use the file extension .SLT for the SLT native files. They are simply DOS files that you can transfer manually or with a file transfer program to another machine with an APL system that supports the Workspace Interchange Convention. Using the functions in this workspace, you can put the image of a component file or a workspace into an SLT file, and you can build a component file or a workspace from its image in an SLT file. You can also load translate tables from an SLT file into global variables in the workspace. This obviates the need to build them during each SLT operation as long as the sending and receiving APL systems do not change. Other functions list the workspace images and their offsets in an SLT file and list the parameters of the SLT package.

Installing the SLT Workspace

Before you can use the transfer functions, you must install the SLT workspace. You need to install the workspace only once. The SLT workspace supplied with your system contains a single function, ∆∆SLTINSTALL. This function loads the other functions and variables in the facility from a companion component file. Its argument identifies the companion file by tie number or name. If the argument is empty, the system uses the value of ⎕wsid as the file name.

The installation makes any necessary name substitutions when it brings in the functions and variables from the SLT file. All the functions and variables you install begin with the first two characters of ∆∆SLTINSTALL to avoid conflicts with existing identifiers in the workspace you want to transfer. If the ∆∆ characters will not avoid name conflicts in your application, change them to two other legal characters by editing the name of ∆∆SLTINSTALL before you execute it. The system adds the new prefix automatically to all installed functions and variables.

The following example shows how you install the workspace.

```
      ⎕libd '2 C:\APL'
      )load 2 SLT
      ∆∆SLTINSTALL '2 SLT'
TYING SLT AUXILIARY FILE <2 SLT> TO 1
∆∆BITS
∆∆DESCRIBE
∆∆END
∆∆ESCAPE
∆∆ISOAPL
.
.
.
∆∆SLTLIST
∆∆LOADTRANS
SLT PACKAGE LOADED
```

Now set the workspace name to one of your choice and save the installed workspace; for example:

```
      )wsid MYSLT
WAS 2 SLT
      )save
2 MYSLT SAVED. . .
```

The install function makes rough checks for workspace free area and unused symbol table slots adequate to allow the installation process to run to completion. It also checks for objects already present in the workspace with the same function or variable names as those in the SLT package. If the function detects any of these situations, it pauses after a descriptive message to allow you to interrupt the process and correct the problem.

You can set several parameters and options for the SLT package to enhance its performance and effectiveness. The transfer functions usually ask if you want to review the parameters before proceeding. Use the expression ∆∆SLTPARMS 0 to obtain a list of the current parameters. Edit ∆∆SLTPARMS to change any of the parameters embedded in the function.

To avoid incomplete character translation, be sure you set the parameter that signals the existence of identical ⎕av's in the source and target APL systems, if appropriate. To do this, edit the line labeled ∆1 in the ∆∆SLTPARMS function. The default parameter, 0, means different ⎕av tables. Change the 0 to 1 to indicated identical ⎕av's.

If you do not have identical ⎕av's on both ends of the transfer, but often deal with a known pair of diverse APL systems, you can prepare a specialized version of the workspace that contains appropriate ⎕av translation tables. Then, you do not have to rebuild the translation tables within each of the five major functions. Instead, you run

```
     ∆∆LOADTRANS  SLTfilename
```

for any incoming SLT file. Then you save the tables it produces in the SLT workspace and reuse them.

You may also want to adjust the parameter controlling whether the main functions prompt you to review the parameter list.

Once you install the workspace, you can get online documentation for each of its functions.

- Type ∆∆SUMMARY for a summary of all the functions.
- Type ∆∆EXPLAIN '*fnname*' for information on a specific function.

Transferring a Workspace from APL64

1. Condition the workspace you want to transfer. Load it using)xload if appropriate. All the functions should be unlocked and all the global variables should have correct values.

```
      )xload MYXFER
```

2. Copy the SLT workspace into the active workspace:

```
      )copy MYSLT
SAVED . . .
```

3. Send the workspace to file:

```
        ∆∆DUMPWS 'A:MYXFER.SLT'
  Review parameter settings in function ∆∆SLTPARMS?  N
  OFFSET: 1644 NAME: WORKSPACE 2 XFER  TO: A:MYXFER.SLT
  ⎕PP
  ⎕ALX
  ⎕CT
  ⎕ELX
  ⎕IO
  ⎕LX
  ⎕RL
  ⎕SA
  MAINFUNCTION
  SUBFUNCTION
  VARIABLE
  END OF FILE MARK OFFSET: 22248
```

   Record the name and starting offset along with the dump to allow the most efficient retrieval of the information.

4. Repeat these steps for each workspace you want to transfer and then move the SLT file to the other APL system. You can place multiple workspace and file dumps in a single SLT file, but placing a single dump in an SLT file allows the most efficient retrieval.

Transferring a Workspace to APL64

1. If you do not have a record of the dumps and starting offsets in an SLT file, use ∆∆SLTLIST to produce a catalog:

```
        )load MYSLT
  SAVED 17:15:54 08/13/87
        ∆∆SLTLIST 'A:MYXFER.SLT'
  Review parameter settings in function ∆∆SLTPARMS? N
  NORMALIZING  COMPOSITE  SEQUENCES...
  NORMALIZATION  COMPLETE
  BUILDING  TRANSLATE  TABLE...
  TRANSLATE  TABLE  COMPLETE
  OFFSET: 1644     NAME: WORKSPACE  2 MYXFER
  OFFSET: 22248    END OF FILE MARK
```

2. Bring in a workspace from the file. Assume you have edited ∆∆SLTPARMS to specify identical ⎕av's and to suppress the parameter review prompt.

```
      ∆∆LOADWS '⋄A:MYXFER.SLT⋄2 MYXFER'
MATCHING ⎕AV'S SPECIFIED, TRANSLATION BYPASSED
OFFSET: 1644     NAME: WORKSPACE 2 MYXFER
SAVED: ...
SYSTEM:  APL★PLUS  . . .
⎕PP
⎕ALX
⎕CT
⎕ELX
⎕IO
⎕LX
⎕RL
⎕SA
MAINFUNCTION
SUBFUNCTION
VARIABLE
OFFSET: 22248    END OF FILE MARK
      )erase ∆∆LOADWS  ⍝ Not necessary on some systems.
      )wsid MYXFER
```

Since no name was explicitly provided for the retrieved workspace, the process pauses with the cursor following the derived workspace name on the)wsid line. You can then edit the workpace name. Press Enter to save the workspace. The function erases all the SLT functions and variables.

3. Save the new workspace.
4. Repeat these steps for each workspace on the transfer file.

Transferring Files with SLT

1. Use the ∆∆DUMPFILE function to store an image of an APL component file.

```
      '2 MYXFER' ∆∆DUMPFILE 'A:MYXFER.SLT'
OFFSET: 1644 FILE: 2 MYXFER TO: 'A:MYXFER.SLT'
COMPONENT...
END OF FILE MARK OFFSET:  2967
```

2. Repeat step 1 for each component file you want to transfer out of APL64.
3. Use ∆∆SLTLIST to produce a catalog of the dumped files and to see the offsets of the files. You will need to run ∆∆SLTLIST especially if you have more than one file in the SLT file.

```
      ∆∆SLTLIST 'A:MYXFER.SLT'
Review parameter settings in function ∆∆SLTPARMS? N
NORMALIZING  COMPOSITE  SEQUENCES...
NORMALIZATION  COMPLETE
BUILDING  TRANSLATE  TABLE...
TRANSLATE  TABLE  COMPLETE
OFFSET:  1644    NAME:  FILE 2 MYXFER
OFFSET:  2967    END OF FILE MARK
```

4. Move the file and use the ∆∆LOADFILE function to access the transfer file and write it as a component file. Note the use of file offset to identify the section of the file you want to retrieve.

```
      '0 NEW' ∆∆LOADFILE 'A:MYXFER.SLT⋄1644'
USING PREVIOUS COMPOSITE NORMALIZATION RESULT
USING PREVIOUSLY GENERATED TRANSLATE TABLE
OFFSET:  1644  NAME:  FILE   2 MYXFER
SYSTEM:   APL★PLUS  . . .
...........
OFFSET:  2967      END OF FILE MARK
```

Function Reference -- SLT

∆∆DUMPFILE Create the Image of a Component File

Purpose:
Puts the image of a component file into an SLT file.

Syntax:
compfile ∆∆DUMPFILE *sltfile*

The left argument is the tie number or name of the component file; the right argument is the tie number or name of the SLT native file. If you use file names instead of tie numbers, enclose the names in single quotes.

∆∆DUMPWS Create the Image of a Workspace

Purpose:
Puts the image of the current workspace in an SLT file.

Syntax:
∆∆DUMPWS *sltfile*

The right argument is the tie number or name of the SLT native file. If you use the file name instead of the tie number, enclose the name in single quotes.

∆∆LOADFILE Build a Component File from an Image

Purpose:
Builds a component file from its image in an SLT file.

Syntax:
compfile ∆∆LOADFILE *sltfile⋄sltimage*

The left argument is the tie number or name of the component file you want to load; the right argument is a delimited string that identifies the tie number or name of the SLT native file, followed by the image of the file that contains the components you want to load. You can specify the image in the file by offset or name substring. The string delimiter must be a ⋄. If you use a file name for either parameter, enclose the right argument in single quotes.

∆∆LOADTRANS Load Translate Tables

Purpose:
Loads translate tables from an SLT file into global variables in the workspace.

Syntax:
∆∆LOADTRANS *sltfile*

The right argument is the tie number or name of the SLT native file. If you use the file name instead of the tie number, enclose the name in single quotes.

Effect:
Using this function eliminates the need to build translate tables during each SLT operation as long as the sending and receiving APL systems do not change.

`ΔΔLOADWS` Create a Workspace from an Image

Purpose:
Builds a workspace from its image in an SLT file.

Syntax:
compfile `ΔΔLOADWS '`*sltfile*`⋄`*sltws*`⋄`*wsname*`'`

The optional left argument is the tie number or name of a component file you want to use for error logging. The right argument is a delimited string that identifies the SLT native file, the workspace name in the SLT file, and the name you want to assign to the retrieved workspace. You can specify the SLT file by tie number or name, and the workspace in the file by offset in the file or name substring. If you omit the segment that identifies the workspace in the file, the system processes the first image in the SLT file. The string delimiter must be a `⋄`. Enclose the right argument in single quotes.

`ΔΔSLTLIST` List the Contents of an Image File

Purpose:
Lists the names of the images and their offsets in an SLT file that contains more than one workspace or file.

Syntax:
`ΔΔSLTLIST` *sltfile*

The right argument is the tie number or name of an SLT native file. If you use the file name instead of the tie number, enclose the name in single quotes.

`ΔΔSLTPARMS` Source Level Transfer Parameters

Purpose:
Lists the parameters for the source level transfer package.

Syntax:
parameter `← ΔΔSLTPARMS` *parm*

When you use this function with a right argument of `0`, it lists the parameters currently in effect for the source level transfer. To change the parameters, edit the function.

Transferring Workspaces between APL2 and APL64

The user commands `]in` and `]out` allow you to transfer workspaces between APL64 and IBM's APL2/370, APL2/6000, or APL2/PC. These user commands use transfer files to move data to and from workspaces in APL64. `]in` converts the objects in an APL2 transfer file into objects in the current APL64 workspace. `]out` converts the objects in the current workspace into a transfer file that you can export to APL2. To physically move transfer files between computers, you must use a third party binary transfer program, such as DCA's IRMA FT3270.

Note: Before trying to use `]in` and `]out`, make sure that the user command processor is installed on your system. Type:

```
        ]?
```

If a list of user commands appears on the screen, the command processor is installed. If the list does not appear, install the command processor. See the User Command Processor chapter in this manual for information on installing and using user commands.

Transferring APL2 Workspaces to APL64

The user command `]in` copies and converts the objects in an APL2 `.ATF` transfer file into the current APL64 workspace. You can create APL2 transfer files using the APL2 `)OUT` system command.

Follow these steps to convert and transfer an APL2 workspace to APL64. If you are transferring a workspace from APL2/370, start APL2 in CASE(2) mode.

1. In APL2, type

```
        )CLEAR
    )COPY  workspace
    )OUT  filename
```

where *workspace* is the name of the APL2 workspace you are transferring and *filename* is the name of the transfer file. On the IBM 370, the transfer file will have an APLTF filetype.

2. Move the file to the computer you use to run APL64. If you are moving the file from a mainframe or RS/6000, you need a third party software application, such as IRMA FT3270, to transfer the file physically between computers. Transfer the file in binary form, not in ASCII form.
3. In APL64, type

```
        )clear
CLEAR WS
      ]in  filename
```

where *filename* is the name of the APL2 transfer file you created on the host system. `]in` copies and converts the objects in the transfer file into the active workspace.

Occasionally, differences in APL2 syntax, system function names, and system features prevent `]in` from re-creating an object from the transfer file. If `]in` cannot convert an object in a transfer file, it assigns the object to one of the following global variables. `]in` substitutes the name of the APL2 object for *name*.

- `inFnDef`*name* contains the `⎕cr` form of the APL2 function ∇*name*
- `inQuad`*name* contains data values from the APL2 system variable ⎕*name*
- `inVarDef`*name* contains an expression to produce values for the APL2 variable *name*

You can edit the variable to change any statements or functions that are incompatible with APL64 and then re-create the object manually.

Transferring APL64 Workspaces to APL2

You can convert workspaces from APL64 to APL2 using the `]out` user command. `]out` creates a transfer file with the extension `.ATF`, performs character `⎕av` translations, and writes all the objects in the active workspace to the transfer file. The APL2 system command `)IN` converts the transfer file created by `]out` into the active APL2 workspace.

Because there are differences between some system functions and system variables in APL64 and APL2, the APL2 system command `)IN` may not be able to convert some APL64 functions. You can either modify these before transferring them, or transfer their `⎕cr`'s to APL2 and then edit these matrix representations.

`]out` cannot convert locked objects, `⎕na` functions, or objects that are local to the user command processor; these objects begin with delta underscore (⍙).

Follow these steps to convert and transfer an APL64 workspace to APL2.

1. In APL64, type

```
        )xload  workspace
    )reset
    ]out  filename
```

where *workspace* is the name of the workspace you want to transfer and *filename* is the name of the transfer file. `]out` adds the file extension `.ATF` to the filename you enter.

2. Move the transfer file to the computer you use to run APL2. If you are moving the file to an IBM 370 or an IBM RS/6000, you need a third party software application, such as IRMA FT3270, to transfer the file physically between computers. Be sure to transfer the file in binary form—not ASCII or character form. Transfer files should have a file type of APLTF on the IBM 370 and a file extension of `.ATF` on both the IBM PC and the IBM RS/6000.
3. In APL2, type

```
)CLEAR
)IN filename
)SAVE workspace
```

where *filename* is the name of the transfer file, and *workspace* is the name of the APL2 workspace you want to use to save the converted objects.

Converting Applications from APL+DOS

Once you load or copy a workspace that originated on an APL+DOS system, the following commands can be useful:

- The `]iicheck` command searches functions in the workspace for any features specific to APL+DOS. These features may either be different (for example, `⎕arbin`, `⎕call`, `⎕inbuf`, `⎕na`) or absent (for example, `⎕edit`, `⎕gline`, `⎕inkey`, `⎕poke`, `⎕mouse`, `⎕sound`, `⎕wput`) in APL64. This command can also be useful when run in a workspace imported from APL+PC.

  If you find functions you need to convert, use `]wsloc` or `]loc` to search the workspace for references to the functions. You can use `]usedby` to obtain a list of workspace objects that a set of functions will ultimately use, or you can use `]xref` to obtain an identifier cross-reference for a particular function.
- The `]iiasmvars` command returns a list of the names of all the numeric variables in the workspace that appear to be APL+DOS `⎕call` right arguments.
- The `]asmfix` command searches the workspace for functions that invoke `⎕call` and that match a name of a function in the `ASMFNS` workspace. It then replaces those functions with APL64 versions of the `ASMFNS` utilities. Run this command in the workspaces you transfer from either APL+DOS or APL+PC.

APL+DOS Items not in APL64

This list shows the APL+DOS System Commands, Functions, Variables and facilities not in APL64.

System	⎕KEYLOG	System	⎕GPAINT	⎕MKDIR	⎕WNUMS
Commands	⎕KEYSRC	Functions	⎕GPRINT	⎕MLOAD	⎕WOPEN
)[recall line]	⎕LINELOG	⎕CGI	⎕GSHADE	⎕PEEK	⎕WPUT
)DEBUG	⎕MOUSE	⎕CRT	⎕GSTATUS	⎕PFKEY	
)EDIT	⎕PFKEYS	⎕ECTRL	⎕GTYPE	⎕POKE	Facilities
)GO	⎕POKES	⎕ED	⎕GVIEW	⎕RMDIR	VDI Graphics
)HELP	⎕SEG	⎕EDIT	⎕GWINDOW	⎕SOUND	Paradox IF
	⎕WINDOW	⎕FSTATUS	⎕GWRITE	⎕WCLOSE	
System	⎕WKEYS	⎕GCIRCLE	⎕GZOOM	⎕WCTRL	
Variables	⎕WSTATE	⎕CINIT	⎕HELP	⎕WGET	
⎕CURSOR		⎕GLINE	⎕INKEY	⎕WIN	
⎕HNAMES		⎕GMASK	⎕INT	⎕WKEY	

Techniques for Converting Printing Routines from APL+PC/DOS to APL64

There are a variety of techniques that APL+PC/DOS users used to print text and APL characters. For each of these, you can accomplish the same thing in APL64.

Printing APL characters through the Windows queue – `10 ⎕arbin 'text'`

Windows treats printing as jobs instead of allowing a program like APL64 to completely control the printer. This means that you must build a whole page or report in memory and send your results to the printer when you perform the Close method on your printer object.

The use of port `10` in `⎕arbin` referenced a special driver loaded on `LPT1` that handled the printing of APL characters. If you need to print APL characters, you can use a Printer object and the Print method. If your code were:

```
[12]  10 ⎕arbin 'Text String 1', ⎕tcnl
[13]  10 ⎕arbin 'Text String 2', ⎕tcnl
```

You can accomplish the same thing by creating a printer object and setting the font:

```
[12]  prt ← 'prt' ⎕wi 'New' 'Printer' ('caption' 'Print_Job')
[13]  prt ⎕wi 'Set' ('font' 'APL+WIN' 1 0 'DEFAULT')
```

```
[14]  prt ⎕wi 'Print' ('Text String 1', ⎕tcnl)
[15]  prt ⎕wi 'Print' ('Text String 2', ⎕tcnl)
[16]  prt ⎕wi 'Close'
```

While this may seem like more work, remember that you no longer need to have a special printer driver loaded when you start APL and you don't need the APL initialization step that was necessary to download the APL character set to your printer. In APL64, you can print APL characters without any special initialization. You can also mix graphics in with your printing, allowing more flexible printing mechanism.

Using ⎕POKE 125, 126 and 166

These pokes were used when trying to print sections of your session. While session output is no longer useable in a runtime application, you may have tools that print just for your own use. If you used these, you can replace them.

POKE 125 was used for selecting a port (usually port 10); in APL64, use the Setup option on the Print dialog.

POKE 126 was used to Set Epson graphics printing; in APL64 select the Font for printer item from the Options menu in your APL64 session.

POKE 166 was used to toggle Ctrl+PrintScreen. If you want to do this manually, print the desired text in the session, tag it with the mouse and select Print from the File menu.

If you used ⎕poke 166 under program control to automatically print session output and you don't want to manually tag or print the text, you can use either of the techniques described above.

Chapter 9: Error Messages

This appendix lists error messages you can encounter in the APL64 interpreter and in the APL Windows Interface (WI). You can usually trap these messages with ⎕elx. You can also cause certain errors or create conditions in the session that invoke message windows; you cannot trap these messages. You can get a complete list of the errors associated with the network interface by invoking ⎕ni 'Errors'.

In the case of a system failure, you may see a box entitled APL64 System Failure. The text in this box can be useful to APL2000 in determining the cause of the problem. Please capture the screen or copy the information and report it, along with any information you have as to the sequence of actions that led up to the failure. Other catastrophic failures may generate an operating system failure message box; this will have "APL64" or the name of the interpreter in the title bar. An edit control in this window contains information about the operating system registers at the time the error occurred. You can copy this information to the clipboard and paste it into a message. In this case, it is even more important to know the steps that led to the failure, since the error is occurring outside of APL64. It is best if you have a reproducible scenario that generates the error.

Some of the messages listed below are specific; others are given as a category with sample text.

System Error Messages

- `AXIS ERROR`
  You are specifying an axis that does not exist or that cannot be created through lamination.
  vec1,[2]vec2

- `CANNOT EDIT ASSOCIATED FUNCTION`
  The system is unable to edit the ⎕na associated function.

- `DESTINATION ERROR`
  You attempted to branch illegally into a control structure.

- `DISK ERROR`
  A file or workspace operation is failing at an unexpected operating system error.

- `DISK FULL ERROR`
  The disk selected for creating a file, appending to or replacing a file, or saving a workspace has insufficient space to hold the object.

- `DOMAIN ERROR`
  A function argument is not within the domain of the function, possibly with regard to the other argument.
  10 × 'A'

- `DRIVE NOT READY`
  The disk drive selected for a read or write has no disk, has the door open, or is unready for use in some other way.

- `ENTRY POINT NOT FOUND`
  A module's symbol table does not contain the entry point indicated by the associated function that ⎕na creates.

- `ERROR WRITING TEMPORARY COPY FILE`
  The system is unable to write to the special file that is used during the copy process.

- `EVOLUTION ERROR`
  The feature behaves differently depending on the evolution level. You should update your program to match a particular evolution level or change the)evlevel setting.

- `EXTERNAL DATA ERROR`
  This error can occur during the processing of ⎕na.

- `FILE ACCESS ERROR`
  Either you are attempting to execute a file function on a tied file without using the passnumber with which the file was tied, or you are attempting to tie a file improperly. The latter problem can be caused by trying to tie a file :
  - that was created from another user number
  - that has an access matrix that limits or denies access to you
  - using an incorrect passnumber.

- `FILE ARGUMENT ERROR`
  You are attempting to reference a file with a fileid that does not conform to the correct format.

- `FILE DATA ERROR`
  You are attempting to read a component from a component file that does not contain a validly formed APL variable. This is most likely to occur if the component file is accessed by non APL2000 programs.

- `FILE FULL`
  You are attempting to append or replace a component in a file that requires the total size of the file to exceed the number of bytes specified in the file size limit (⎕fsize n)[4].

- `FILE INDEX ERROR`
  You are attempting to access a nonexistent component from a component file.

- `FILE NAME ERROR`
  You are attempting either to create or rename a file with a name that is already in use.

- `FILE NAME TABLE FULL`
  You have exhausted the system storage for tied file names.

- `FILE NOT FOUND`
  You are attempting to reference a file that does not exist in the specified directory.

- `FILE RESERVATION ERROR`
  You are attempting to resize a file to a size that is smaller than the minimum necessary to hold its contents.

- `FILE TIE ERROR`
  You are attempting either to tie a file with a tie number that is already in use or to execute a file operation with a tie number that is not in use.

- `FILE TIE QUOTA EXCEEDED`
  You have as many files tied as the system allows.

- `FILE TIED`
  You are attempting to tie a component file that is already exclusively tied by another user.

- `FORMAT ERROR`
  You have an invalid left argument for ⎕fmt. This message is normally preceded by a qualifier that indicates which part is in error.

- `HOST ACCESS ERROR`
  The operating system does not allow or is unable to execute the requested operation. For example, you are attempting to erase a file that allows read only access.

- `HOST ERROR`
  An error directly associated with the failure of a host operating system call has occurred during the execution of an input/output function for the session manager or the traditional file system functions. The error report contains the information available from the operating system about the immediate circumstances of the error, including: the Windows system or library call that failed; a designator for a place in the interpreter; the C library errno number; the number returned by Windows GetLastError(); and the text supplied by Windows for that error number.

- `INCOMPATIBLE WS`
  You are attempting to load a workspace that cannot be loaded with your current configuration. This usually occurs when you attempt to load a workspace that was saved in too old of a version, a newer version of APL64 than the current version and a workspace that originated on an APL+DOS system.

- `INCORRECT COMMAND`
  You are attempting to execute a line that starts with a right parenthesis ")" but that does not constitute a valid system command.

- `INDEX ERROR`
  You are attempting to extract elements from an array using indexing, but one or more of the indices falls outside the shape of the array.
  ```
  a←4 6ρ'B' ⋄ A[1 3 5;2]
  ```

- `INSUFFICIENT HANDLES`
  There are insufficient handles defined to tie any more files. Untie the unneeded files or edit the FILES= statement in the CONFIG.SYS file to increase the number of handles available and restart your system.

- `INSUFFICIENT MEMORY`
  The system is unable to complete the operation due to memory constraints outside of the workspace.

- `INTERFACE CONVERSION ERROR`
  The system is unable load or copy the named objects from a workspace when unsupported in APL64.

- `INTERNAL REPRESENTATION OF ASSOCIATED FN OBSOLETE`
  This error can occur during the processing of ⎕na.

- `INTALX`
  An error has occurred during the handling of an attention.

- `INTERRUPT`
  An interrupt has been issued by the user pressing the Break key combination (Ctrl+Break twice in rapid succession under the right conditions, for example, during a long-running primitive function).

- `INVALID MODULE STRUCTURE`
  The internal structure of the variable is not a well-formed module for calling into a DLL using ⎕na.

- `INVALID OR DAMAGED COMPONENT FILE`
  You are attempting to tie a file as a component file when the file does not have the appropriate structure.

- `LANGUAGE INTERFACE NOT INSTALLED`
  This error can occur during the processing of ⎕na.

- `LENGTH ERROR`
  You are attempting to use an argument to a function where the length does not conform properly along a relevant dimension.

  ```
      (3 4⍴⍳12)×4 4⍴⍳16
  ```

- `LIBRARY NAME ERROR`
  The directory you are attempting to associate with a library number using ⎕libd is invalid.

- `LIBRARY NAME TABLE FULL`
  There are too many directories associated with library numbers.

- LIBRARY NOT FOUND
  You referenced a library number that has no current definition.

- `LIBRARY NUMBER ERROR`
  The number you are attempting to associate with a directory using ⎕libd is invalid.

- `LIBRARY QUOTA EXCEEDED`
  You have as many active library definitions as the system allows. You can release existing library definitions to provide room for new definitions.

- `LIMIT ERROR`
  You are attempting to perform an operation that exceeds one of the limits of the system.

  ```
    999*999
  ```

- `MODULE NOT FOUND`
  You are attempting to call an associated function created with ⎕na whose variable name (the library) does not exist.

- ⎕NA INTERNAL ERROR
  This error can occur during the processing of ⎕na.

- `NO SPACE FOR ⎕DM`
  There is not enough space in the workspace to record the diagnostic message.

- `NONCE ERROR`
  You are attempting to execute something that is not possible with the current system but that may be included in a future release.

- `NOT ...:`
  COPIED, ERASED, FOUND, SAVED, etc. These messages are, strictly speaking, not errors, but information about a system function or system command that you have invoked. It usually means that a name you supplied has not been found or the name already exists and cannot be replaced.

- `OUTER SYNTAX ERROR`
  You are attempting to execute a function that has an ill-formed control structure.

- `RANK ERROR`
  You are attempting to use a function that requires arguments of conforming rank but the ranks of the arguments differ.
  ```
      2 3↑5 5 5⍴⍳125
  ```

- `SI DAMAGE`
  A suspended, or pendent function has been modified in such a way that the system cannot resume executing it.

- `SYNTAX ERROR`
  You are attempting to execute a statement that does not obey the rules of APL syntax; for example, you have unmatched parentheses or quote marks.

- `SYSTEM ERROR`
  An error has occurred in the APL interpreter code, possibly caused by damage to the workspace's internal data structures. The APL64 session terminates after a system error.

- `TYPE ERROR`
  You are attempting to assign a value to a defined function.

- `UCMDS ...`
  Many errors are prepended with UCMDS or contain UCMDS in the text. These errors have occurred in the user command processor. You can frequently salvage the situation by using the]ufile command to display and change the search list or by correctly installing the user command processor.

- `VALUE ERROR`
  You are attempting to derive a value from something that does not have any result defined; for example, you referenced a name that is not defined at this stack level, or you attempted to assign (or use in a calculation) the result of a function that does not return a result or that has not had a result assigned during execution.

- `WS ARGUMENT ERROR`
  The workspace identifier is ill-formed or is too long to process.

- `WS DAMAGED`
  The source workspace is not in the correct form for a saved workspace, or a disk error has occurred.

- `WS FULL`
  You are attempting to execute a statement for which there is insufficient room in the workspace. There is no room to assign a variable or work area for temporary use and calculation.

- `WS NAME ERROR`
  You are attempting to change a workspace name to an invalid name with)save or)wsid.

- `WS NOT FOUND`
  The workspace you are attempting to load or copy from does not exist in the specified (or implicit) directory.

- `WS TOO LARGE`
  There is insufficient space in memory to load the specified workspace.

- `XHFOST ERROR`
  An error directly associated with the failure of a host operating system call has occurred during the execution of an extended file function. The error report contains the information available from the operating system about the immediate circumstances of the error, including: the Windows system or library call that failed; a designator for a place in the interpreter; the C library errno number; the number returned by Windows GetLastError(); and the text supplied by Windows for that error number.

APL WI Error Messages

These error messages, which come from the Windows interface, show an ellipsis where the system provides additional information about the specific error. The line below the error message gives an example of the type of error that can generate the message. In some cases, an additional explanation is given.

- `⎕WCALL LIMIT ERROR: ...`
  example: Recursion limit (nnnn) exceeded
- `⎕WGIVE LIMIT ERROR: ...`
  example: Recursion limit (nnnn) exceeded
- `⎕WI ACTION ERROR: ...`
  example: Property-set not permitted in this context
- `⎕WI ACTION NAME ERROR: ...`
  example: Improperly formed action name
  An action is a portion of the right argument to `⎕wi`; it could be a method, a property, or an event-handler property.
- `⎕WI ARGUMENT ERROR: ...`
  example: Argument required for method
- `⎕WI ATTRIBUTE ERROR: ...`
  example: Duplicate or conflicting attributes specified
- `⎕WI CLASS ERROR: ...`
  example: Class not found
- `⎕WI CLASS NAME ERROR: ...`
  example: Improperly formed class name
- `⎕WI CREATION ERROR: ...`
  example: Unable to create window; reason=nnnn
  In this context, "window" means a window in the Windows operating system sense of the word; this is roughly equivalent to a control object in the APL64 interface.
- `⎕WI DDE ERROR: ...`
  example: Need application|topic in ddeTopic
- `⎕WI DEF ERROR: ...`
  example: Definition cannot change class
- `⎕WI DOMAIN ERROR`
  A Domain Error in the traditional APL sense that occurred in APL WI.
- `⎕WI DRAW ERROR: ...`
  example: Invalid brush style

- `⎕WI ERROR: ...`
  example: Unexpected result from Windows OS
- `⎕WI EVENT NAME ERROR: ...`
  example: Improperly formed event name
- `⎕WI FOCUS ERROR: ...`
  example: Window not open
- `⎕WI HOST ERROR: ...`
  The operating system is unable to execute the requested operation; it returns the action that generated the error; the operating system function that failed; the number returned by Windows `GetLastError();` and the text supplied by Windows for that error number.
- `⎕WI LENGTH ERROR`
  A Length Error in the traditional APL sense that occurred in APL ⎕WI.
- `⎕WI LIMIT ERROR`
  A Limit Error in the traditional APL sense that occurred in APL ⎕WI.
- `⎕WI LIMIT ERROR: ...`
  example: Unable to create new timer
- `⎕WI MENU ERROR: ...`
  example: Menu bar items or popup menus can't be checked
- `⎕WI OBJECT ERROR: ...`
  example: Parent not found (too many ..'s)
- `⎕WI OBJECT NAME ERROR: ...`
  example: Invalid result object name following ">"
  The greater than sign is used in redirection syntax for creating ActiveX children.
- `⎕WI OPEN ERROR: ...`
  example: Parent must be opened before child
- `⎕WI PARENTAGE ERROR: ...`
  example: An object cannot be its own parent
- `⎕WI PRINTER ERROR: ...`
  example: Print Setup Error
- `⎕WI RANK ERROR`
  A Rank Error in the traditional APL sense that occurred in APL ⎕WI.
- `⎕WI REGISTRY ERROR: ...`
  example: Cannot open registry key

- `⎕WI RESOURCE ERROR: ...`
  A resource in this context is usually an image file, such as a bitmap, icon, or cursor.
- `⎕WI RESULT ERROR: ...`
  example: Result redirection can only handle ActiveObject return values
- `⎕WI SIZE ERROR: ...`
  example: Cannot set size when scale is proportional; Use extent or where instead
- `⎕WI WAIT ERROR: ...`
  example: Unable to wait on disabled object
- `APL32 DAMAGED`
  The APLNow32 Runtime System (32 bit) process is not accessible.
- `APX initialization failure: ...`
  example: Unable to load Resource DLL
- `APX SAVE ERROR: ...`
  example: Unable to save
- `APX SYSTEM ERROR: ...`
  example: Undefined case
- `Initialization error: ...`
  example: Unable to start OLE library
- `LENGTH ERROR: ...`
  example: Wrong number of index arguments
- `MEMORY ERROR: ...`
  example: Insufficient Windows memory
- `NONCE ERROR: ...`
  example: Planned feature not yet implemented
- `SYSTEM ERROR: ...`
  example: Internally inconsistent system state
- `Unable to register CommandDockWrapper (reason nnnn)`
  The command bar mechanism failed.
- `[name:] Help not available`
  The Help resource for an ActiveX object is not defined or not present.

Chapter 10: System Limits and Characteristics

This appendix lists some characteristics of APL64 and the system limits for the implementation parameters identified in the ISO standard.

APL64 uses the following internal data representations:

- Boolean. One bit per element, range 0-1.
- Integer. 32-bit 2's complement integers, range ¯2146435071 to 2146435071.
- Floating point. 64-bit IEEE standard double-precision; range from positive-number limit to negative-number limit.
- Character. 8-bit characters.
- Nested. 4 bytes per element at outermost level (4-byte pointer).
- Heterogeneous. 10 bytes per element (1 byte datatype, 1 byte reserved, 8 bytes to hold the largest possible value).

The following table lists the system limits. A system limit of "none" means that the system has no constant value that is enforced as a limit. In these cases, available workspace area is the only limit that applies. The limits and characteristics in the table are those required to be documented by the international standard for the APL language.

System limits

Limit	Characteristic
Positive-Number-Limit	1.7976931348623158E308
Negative-Number-Limit	¯1.7976931348623158E308
Positive-Counting-Number-Limit	9007199254740991 (¯1+2*53)
Negative-Counting-Number-Limit	¯9007199254740991 (-¯1+2*53)
Index-Limit (coordinate length):	2146435071
Count-Limit	2146435071
String-Limit	1073741791
Rank-Limit	127
Identifier-Length-Limit	100
Quote-Quad-Output-Limit	None
Comparison-Tolerance-Limit	1E¯13
Integer-Tolerance (system fuzz) for non-zero integer *int*	(2*¯41+⌊2⍟|int) < fuzz < 2*¯40+⌊2⍟|int
Print-Precision-Limit	17
Full-Print-precision	17
Exponent-Field-Width	4
Indent-Prompt	6ρ' '
Quad-Prompt	'⎕:',⎕tcnl,6ρ' '
Function-Definition-Prompt	7↑'[',(⍕line-number),']'
Line-Limit	32767
Printing-Width-Limit	30 to 2147483647
Symbols-Limit	32767
Input-Line-Length-Limit	1014
Function-Line-Length-Limit	None
Execute-Argument-Length-Limit	2146435071
Workspace size	Dependent on the installed physical memory

INDEX