APL64 Project Details

Contents

Overview of the APL64 Project

APL2000 is developing the APL64 Project for the 64-bit hardware and operating system environment. The APL64 Project will incorporate virtually all the features of APL+Win. The project was conceived in August 2015 and has continued until the present with significant results. Major investment by APL2000 of resources and team members' efforts has enabled the APL64 Project progress thus far achieved.

Continue reading to learn about the significant results achieved by the APL64 Project team and understand that the continuation of the APL64 Project will depend upon the level of interest from APL2000 customers. In this document the name 'APL64 Project' emphasizes the role of APL2000 customers in determining if the APL64 Project will become an APL2000 product.

Primary APL64 Project Goals

- Design for 64-bit hardware and operating systems
- Provide a seamless, high-compatibility transition from APL+Win to the APL64 Project
- Establish level of customer interest to determine project continuation and direction
- Create an interpreter compatible with Windows, Android, iOS and Linux
- Improve the productivity and ease-of-use for customers and APL64 Project developers
- Develop the APL64 Project using current, best-of-breed programming tools
- Increase the potential for future enhancements of the APL64 Project
- Improve quality control

APL64 Project Comparison with APL+Win

<table>
<tr><th>Comparison</th><th>APL+Win</th><th>APL64 Project</th></tr>
<tr><td>Maximum

Workspace Size</td><td>Less than 3.7 GB</td><td>Up to available workstation memory permitting the creation and processing of multiple large APL variables</td></tr>
<tr><td>Maximum

Homogeneous

Variable Size</td><td>Less than 2 GiB and in practice 500 Mib or less</td><td>Up to 2,146,435,071 elements in a variable. Nested arrays may recursively contain up to this number of elements at each nesting level.

<table><tr><th>Type</th><th>Max GiB</th></tr><tr><td>Boolean</td><td>2</td></tr><tr><td>Integer</td><td>8</td></tr><tr><td>Double</td><td>16</td></tr><tr><td>Unicode Char</td><td>4</td></tr><tr><td>APL+Win Char</td><td>2</td></tr></table></td></tr>
<tr><td>Native Execution mode</td><td>32-bit</td><td>64-bit</td></tr>
<tr><td>APL Programmer Session Enhancements</td><td>n/a</td><td>Multiple user-selected Session formats to display APL statements and results.</td></tr>
<tr><td>Editing of APL variables</td><td>Limited editing support for non-nested, homogeneous variables of rank 2 or less with separate editor for numeric and text variable structures.</td><td>Unified APL variable editor transparently handling homogeneous, heterogeneous and nested variable structures.</td></tr>
<tr><td>Compatibility with APL+Win</td><td></td><td>Excellent – virtually all APL+Win features available</td></tr>
<tr><td>Multi-row APL executable statements</td><td>n/a</td><td>Facilitate entering APL ‘scripts’ in the APL Programmer Session</td></tr>
<tr><td>Multi-threaded primitive function operation</td><td>n/a</td><td>Programmer-controlled using ☐TL</td></tr>
<tr><td>APL64 Project built from inception using test-driven development protocols</td><td>n/a</td><td>Assures that initial implementations and subsequent modifications will produce correct results</td></tr>
<tr><td>APL User-Defined Function Enhancements</td><td>n/a</td><td>New Control Structures, Header options, ‘inner functions’, …</td></tr>
<tr><td>Installation Target</td><td>Not in Program Files (x86)</td><td>Not limited</td></tr>
</table>

See here for the definition of ‘GiB’ and here for the definition of ‘GB’.

Feasibility Phase of the APL64 Project is completed

The initial phase of the project determined that it was possible to develop the APL64 Project using currently-available programming tools. Some elements of the APL+Win product were developed using programming tools which have been deprecated by Microsoft, so current programming tools have been identified which are applicable to the APL64 Project.

Programming styles and design criteria for complex software have significantly changed since APL+Win was created, so the APL64 Project incorporates the best design elements of APL+Win and combines that with up-to-date technology, years of APL2000 developer experience and customer input.

The interpreter portion of the APL64 Project was constructed using Microsoft .Net, which is supported on Windows, Android, iOS and Linux. The APL 'programmer session' or programmer graphical user interface (GUI) was constructed using Microsoft .Net and Microsoft Windows Presentation Foundation (WPF) for the Windows environment.

In this completed feasibility phase the Primary Project Goals have been accomplished:

- A new APL64 Project parser, tokenizer and interpreter were created based on those elements of APL+Win.
- Substantial new and expanded developer documentation was incorporated into the interpreter source code to improve subsequent maintenance by APL2000.
- Interfaces from the 64-bit to the 32-bit environment were developed for compatibility with native 32-bit, APL+Win components such as □WI, □WCALL and the APL+Win Grid Control.
- The system was built using Test Driven Development (TDD) protocols and as such includes an extensive unit test component to enhance the verification of results.
- Many APL functions, operators and system functions were implemented in the APL64 Project to establish the pattern for this work.
- The enhanced APL programmer GUI was implemented providing several 'skins'.

Customer Phase of the APL64 Project

As part of the announcement of the APL64 Project, APL2000 customers are requested to provide feedback on their level of interest in and need for the APL64 Project features. Customer input over many years has already been used to design and implement APL2000 products, but the response of current APL2000 customers will determine the future of the APL64 Project.

Transition to 64-bit Environment

Although virtually all new computer hardware available today incorporates 64-bit hardware and operating system, it is recognized that APL2000 customers may not have completed the transition from the 32-bit environment to the 64-bit environment. The APL+Win [32-bit] product will remain available and maintained during the development of the APL64 Project and while a significant APL+Win subscriber base continues to exist. If current APL2000 customer input indicates that the APL64 Project should be continued to completion, that project will receive a high priority.

Features and Enhancements Based upon Customer Feedback

The APL64 Project features and enhancements are derived from years of customer experience and constructive customer feedback. The APL64 Project is designed to be highly-compatible with APL+Win, adds many new features and enhancements which are not feasible in APL+Win and resolves some limitations of APL+Win.

The APL64 Project's look and feel will be very familiar to existing APL+Win users because it uses the same workspace, programmer session, function and variable metaphors as APL+Win. The APL64 Project is an interpreted rather than compiled language, which uses its own built-in APL-oriented editors, and does not require installation of third party keyboard drivers or compiler software such as Microsoft Visual Studio.

Some Important APL64 Project Features

Implemented features and potential features are indicated separately. Not all APL64 Project features are described here. Features illustrated are subject to subsequent enhancement and modification.

Access to All Available Workstation Memory

The APL64 Project is designed for the 64-bit operating system environment which supports much larger memory allocations to applications running on a workstation.

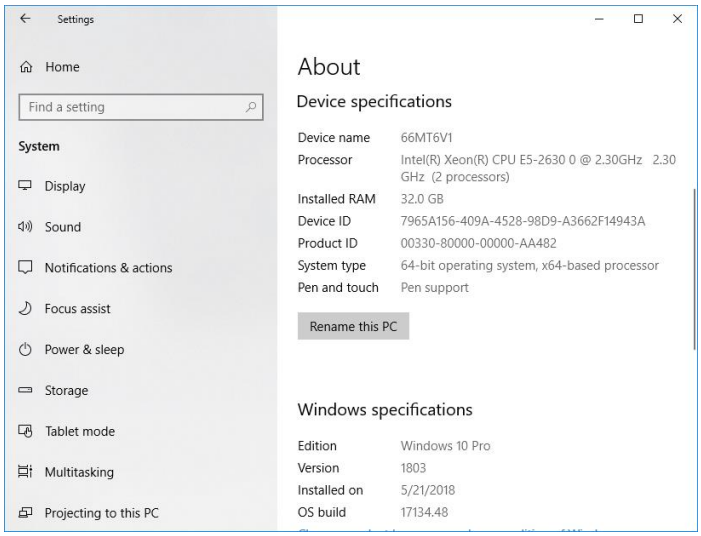
Workspace Size Limited Only by the Available Workstation Memory

The available workstation memory is available to be used by data and functions in an APL64 Project workspace. In the 64-bit operating system environment the available workstation memory, exclusive of the memory required by the operating system, may be allocated to an application running on that workstation.

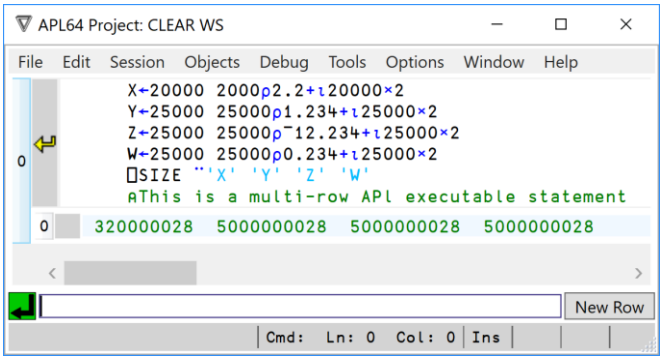
Variables can use All Available Workstation Memory

There is no restriction other than workstation memory on the combined size of all homogeneous variables and nested arrays that contain large homogeneous elements. Homogeneous variables (such as integer, double and char arrays) are constrained by element count limits, as outlined in an earlier table.

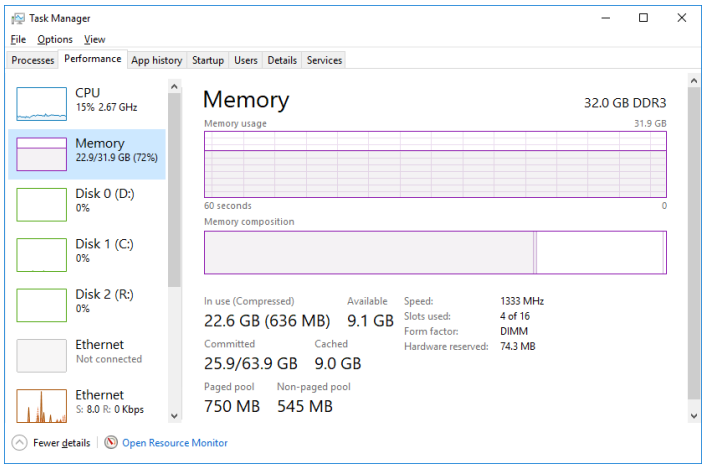
Machine configuration example: 32 GB memory



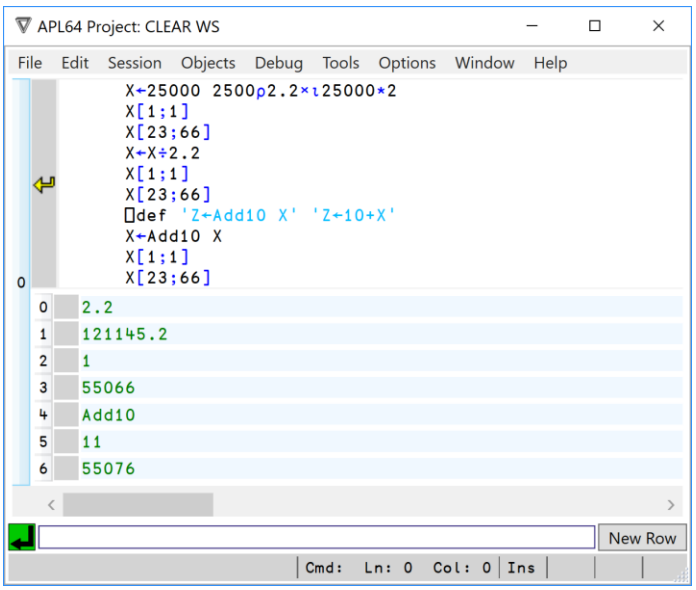
APL64 Project: Execute a multi-row APL statement to create four homogeneous variables, each with 'size' greater than 2 GB:



In this example the APL64 Project workspace memory is entirely contained in the workstation memory, without disk caching required:



A ‘very large’ APL homogeneous variable may be manipulated using APL primitive functions, user-defined functions, etc., just like any other APL variable. In this example a multi-row APL statement is executed to modify a ‘large’ APL homogeneous variable.



Comparison to APL+Win:

APL+Win workspace size is limited to approximately 3.7 GB. Homogeneous variables are limited to less than 2 GiB each. In APL+Win, if a variable is created which approaches the 2 GiB limit, it is unlikely that any processing can be done on it as there would not be sufficient space for a result of the same dimension, other than a boolean result. In practice, the maximum usable variable size in APL+Win is much smaller than 2 GiB, e.g., 1 GiB – 512 MiB.

The APL64 Project and APL+Win both support non-homogeneous and nested variables.

A non-homogeneous APL variable may be considered an ordered collection of homogeneous variables.

Virtually all APL+Win features will be available in the APL64 Project

APL primitive functions, APL system variables, APL operators, legacy Win32 features and virtually all APL system functions will be available in the APL64 Project.

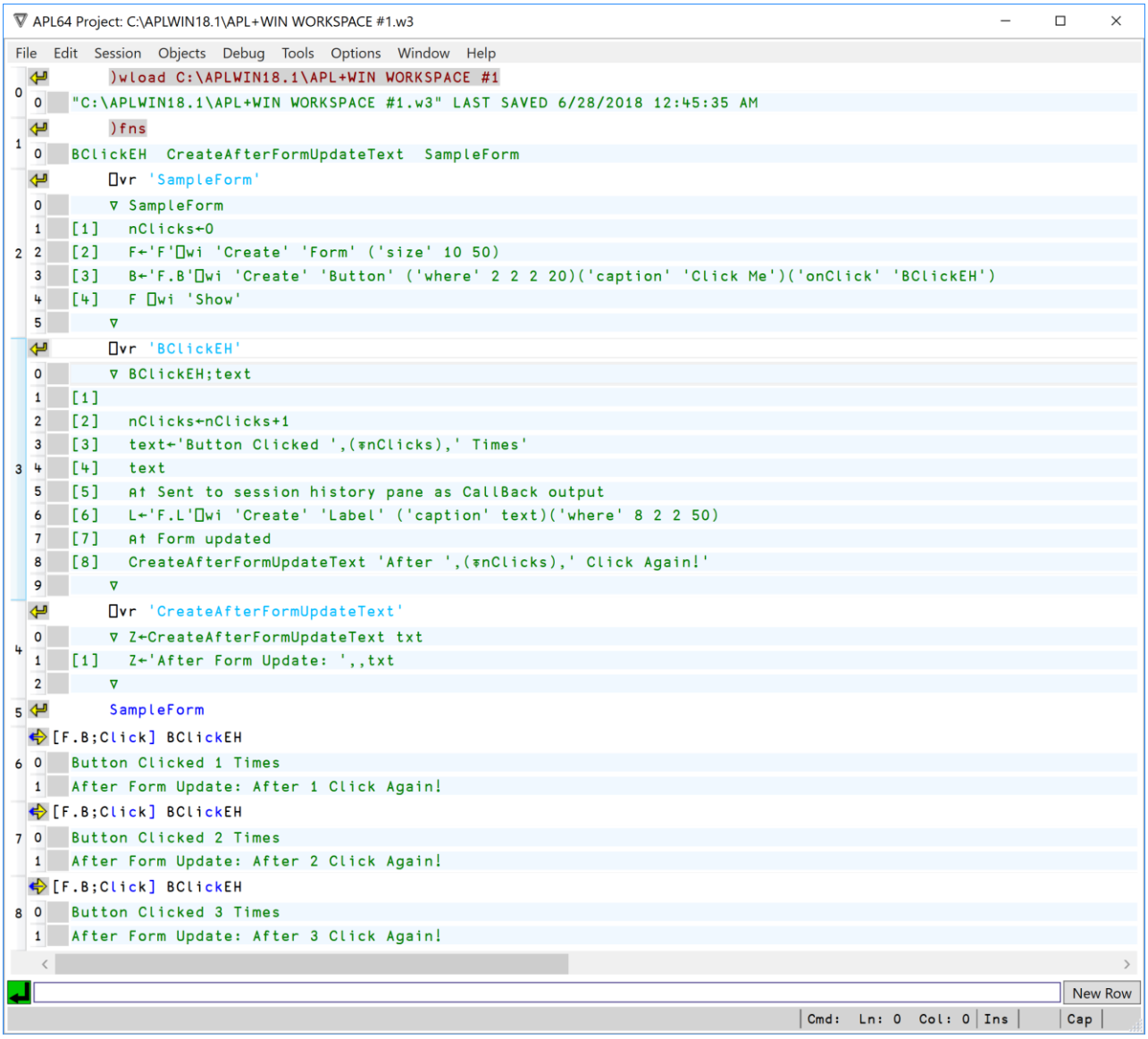
Transparent Bridge to APL+Win 32-bit □WI and □WCALL

A transparent bridge to APL+Win 32-bit features has been implemented in the interpreter for seamless and fully-compatible access to □WI for APL+Win-style user interfaces, □WCALL for access to Win32 APIs, APL+Win Grid Control, APL+Win ActiveX Engines and other custom ActiveX assemblies using the □WCALL and □WI ActiveX interface.

Easy Transition from APL+Win to the APL64 Project:)WLOAD & Run an APL+Win Workspace

To make the user transition as simple as possible, it is designed so that an APL+Win workspace may be loaded into the APL64 Project and run in the 64-bit Windows environment with little or no modification.

In this example the new ')WLOAD' system command loads the 'APL+Win workspace #1.w3' workspace into the APL64 Project environment. The 'SampleForm' APL function is executed and the □WI-based form is displayed. When the user clicks the 'Click Me' button the form is updated and the programmer session displays the event handler results sent to the session by the BClickEH button click event handler function.



Native and APL Component File Compatibility

The ⎕N..., ⎕XN..., ⎕F..., ⎕XF... and ⎕CF... file functions can compatibly read and write data between APL+Win and the APL64 Project.

Wrapl/UnWrapl Compatibility

The APL64 Project wrapl and unwrapl serialization and the APL+Win analogues are fully compatible.

Multi-Threaded APL Primitive Function Execution

The APL64 Project ☐TL system variable supports multi-threaded execution in the work session to enhance performance. Once the APL programmer sets the arguments to ☐TL, the APL64 Project interpreter will apply multi-threaded execution to the APL application without further effort by the APL programmer.

The APL programmer, using ☐TL, can determine the array size and number of threads which will be used when executing scalar primitive functions, such as plus, minus, times, divide, etc., on arrays. Once ☐TL is programmer-specified for multi-threading, the APL64 Project interpreter automatically distributes the scalar primitive function execution on arrays to multiple threads.

☐TL may be specified as a two-element vector or a multi-row, two-column matrix. The first element or column specifies the threshold number of array elements and the second element or column specifies the number of threads to be used for the calculations on the specified arrays. When ☐TL is specified as a matrix with more than one row, the first column should contain increasing values.

Example: ☐TL←10000 2 means that one thread should be used unless the number of elements of the array is greater than or equal to 10000.

Example: ☐TL←2 2⍴10000 2 20000 4 means that one thread should be used for arrays with less than 10000 elements, two threads for arrays with 10000 to 19999 elements and four threads for arrays with 20000 or more elements.

Thread allocation to the physical hardware processors is performed by the operating system. Multi-threaded primitive function execution is disabled by default. An empty vector or matrix (the default) specifies that single threading should be used throughout the work session.

APL 'Programmer Session'

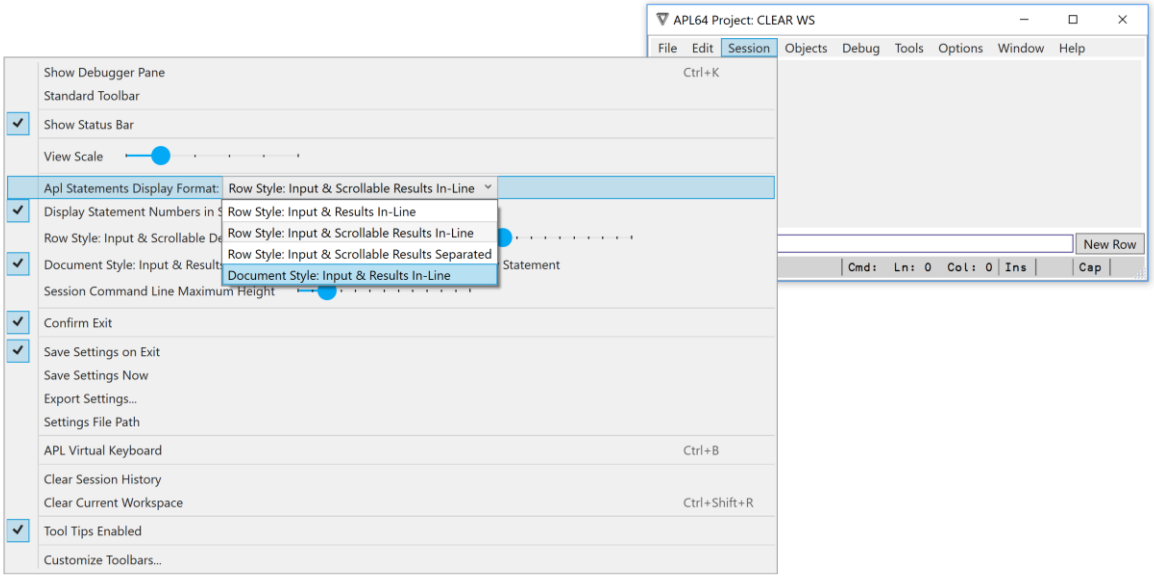
The graphical user interface [GUI] for the 'programmer session', used when the APL64 Project is used in a command-line style, has been significantly enhanced.

- The programmer session and the interpreter now run in separate execution threads.
- The programmer session accesses the interpreter asynchronously so that the APL programmer may continuously-enter executable statements without waiting for the prior executable statement to complete execution or for output to be displayed in the session history pane.
- The 'Pause' and 'Stop' session options will now have a reliable and virtually-immediate effect on the interpreter. The programmer session display is arranged in panes, e.g., command line, session history, debugger, state indicator, function and variable editing instances.
- The programmer session panes may be separately and independently docked or floated in locations on single or multiple monitors.
- Several 'skins' are supported in the programmer session. The user-preferred session 'skin' may be selected at any time during an APL64 Project session. This selection affects the display and organization of previously-executed APL statements and interpreter output in the Session History Pane.

- The APL ‘document-style’ skin is analogous to the APL+Win ‘programmer session’ GUI. The session history pane contains APL executed statements and interpreter output displayed in the order received from the interpreter as a continuously flowing text with new line separators.
- For each session skin the session history pane includes a tooltip describing the user operations available for that skin.
- The row-oriented skins display the session history pane as a grid.
    - Each APL executable statement is a ‘row’ in the session history pane.
    - Each ‘row’ has a ‘row header’ on the left which may be used to select the entire APL executable statement associated with that row.
    - Each interpreter result, e.g., from execution of an APL executable statement or a function callback is a ‘row’ in the session history pane.
    - Options for the display of interpreter results order in the session history pane.
        - In the order received from the interpreter.
        - In a scrollable list following the applicable APL executable statement.
        - In a separate list which may be docked or floated on any workstation display.

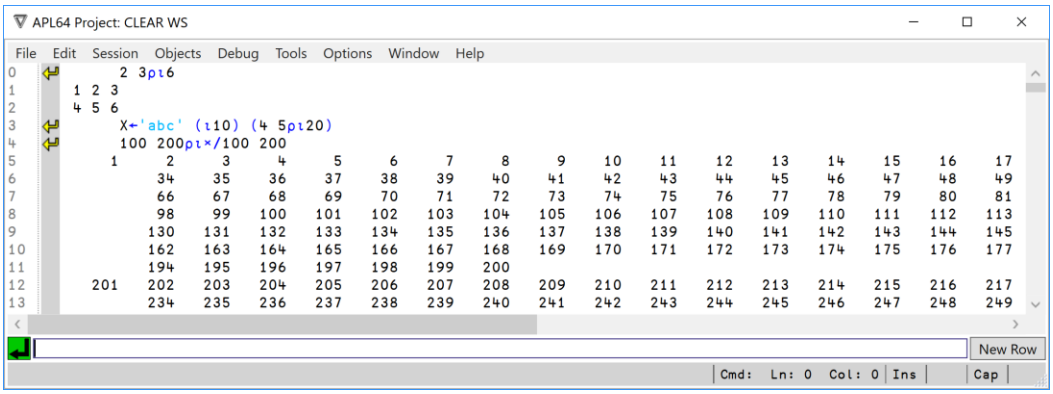
Selecting the Session History Pane Format

In the APL64 Project the 'APL Statements Display Format' menu item in the 'Session Options' menu provides the option to select a GUI skin without restarting the programmer session.



Document-style APL Session Format

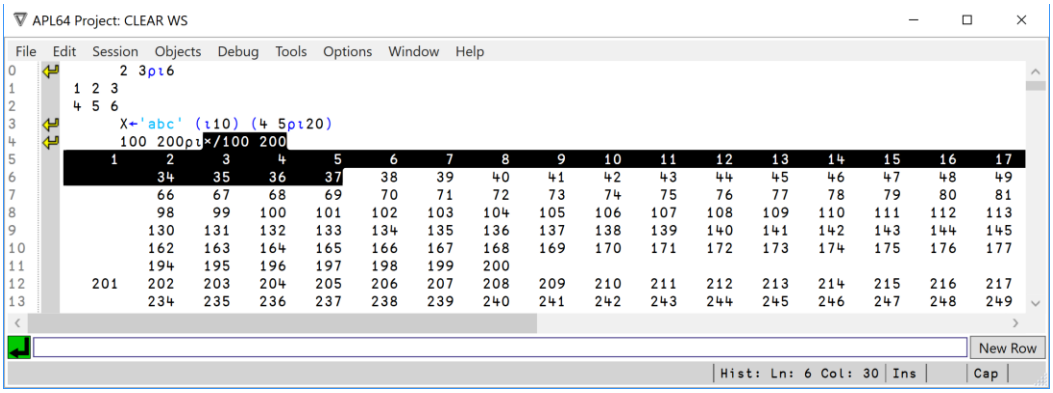
The APL+Win programmer session graphical user interface is available in the APL64 Project. It is called the 'Document Style: Input & Results In-Line' skin. The scrollable portion of the session history pane illustrates the APL executable statements and the APL result statements in the order as emitted by the interpreter.



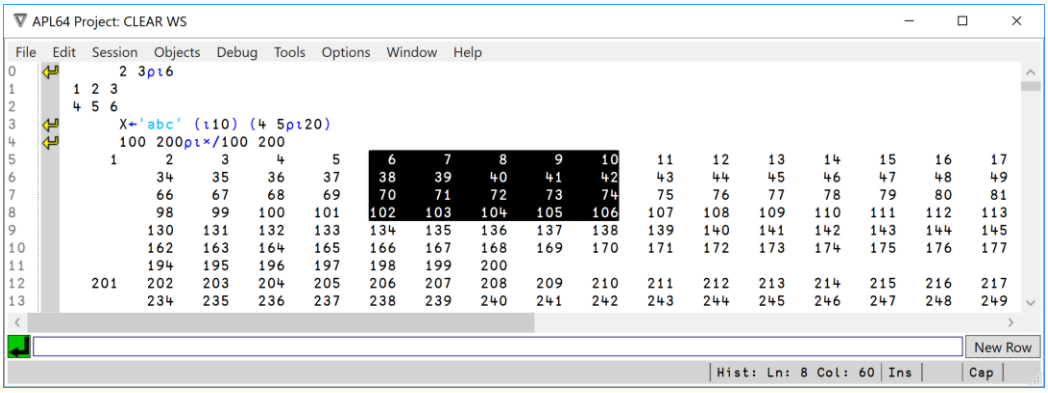
Document-style APL Session Format: Selecting Linear or Rectilinear Text Blocks

The 'Document Style: Input & Results In-Line' skin supports two selection modes:

The 'Selection Start and Selection Length' mode is the document-style APL format enabled using the Shift + cursor movement keystrokes or pointer operations. A contiguous portion of the text in the session history pane may be selected.

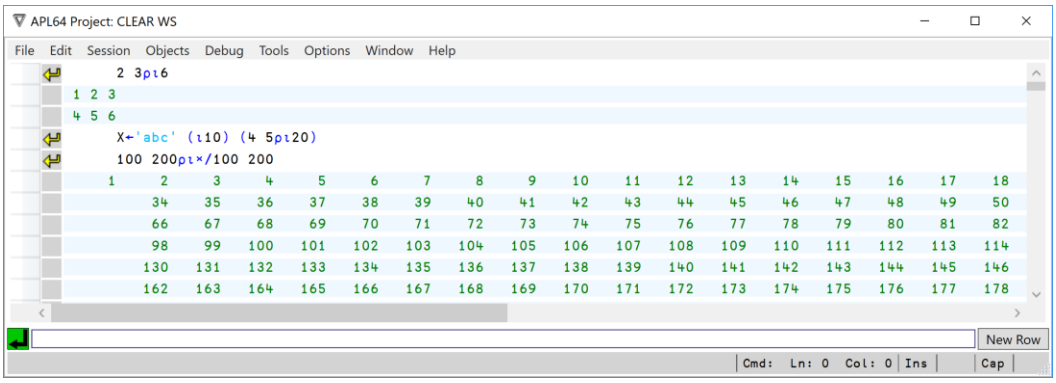


The ‘Columnar Selection’ mode is enabled using the Shift + Alt + cursor movement keystrokes or pointer operations. A rectilinear portion of the text in the session history pane may be selected, deleted, pasted in and copied.



Row Style: Input and Results In-Line Format

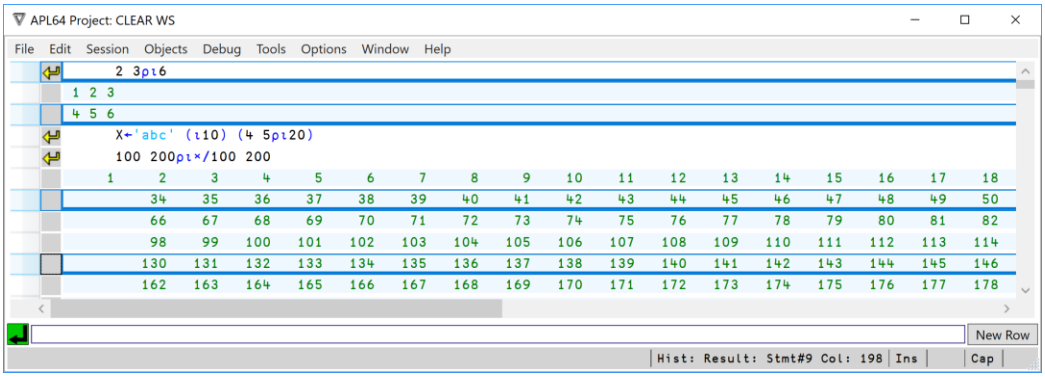
The text in the session history pane is composed of rows for APL executable statements and interpreter results. Statements are illustrated in the order executed or emitted by the interpreter.



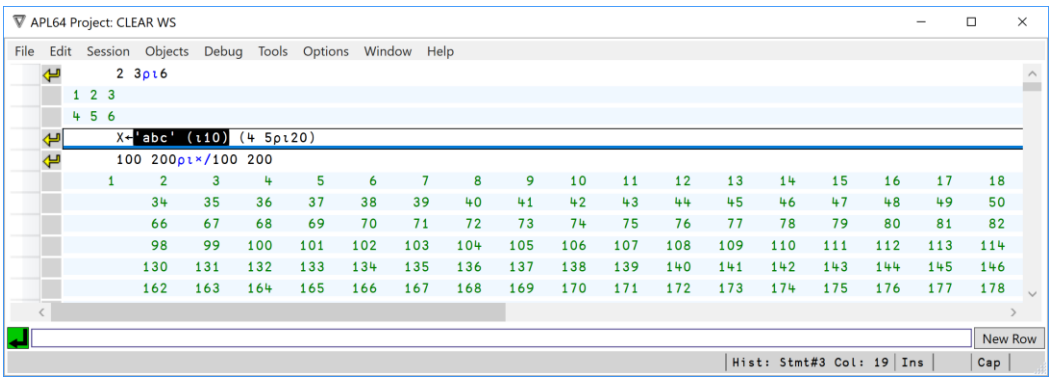
Row Style: Input and Results In-Line Format: Selecting Text

Two selection modes are supported.

The entire contents of one or more, possibly non-contiguous, rows may be selected using the Ctrl + Click key stroke on the applicable row headers.



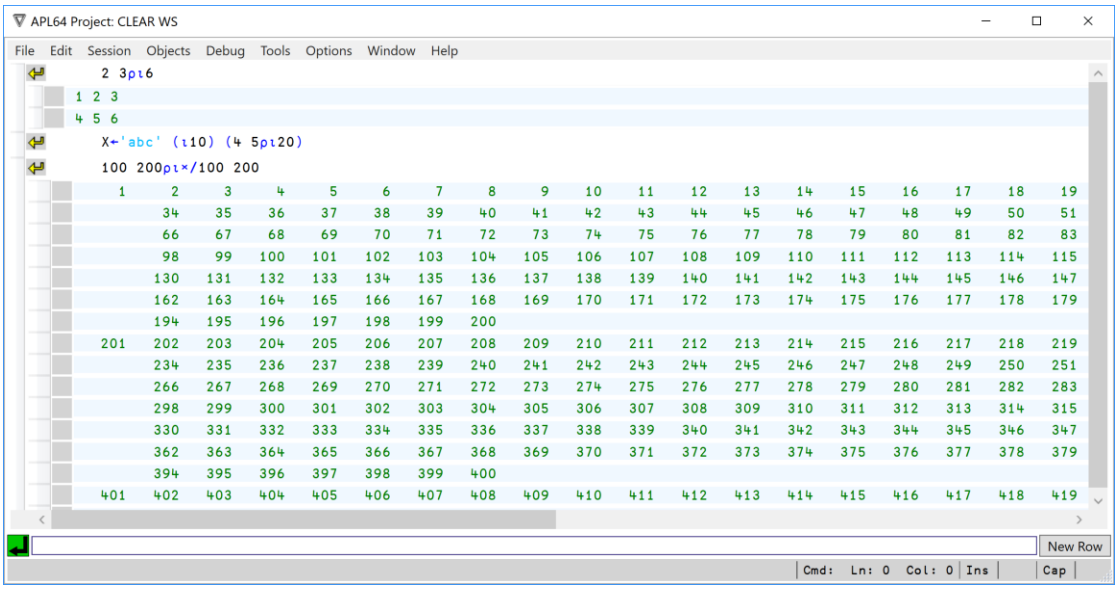
All or part of one row may be selected using the Shift + cursor movement keystrokes or pointer operations within the APL statement column of the applicable row.



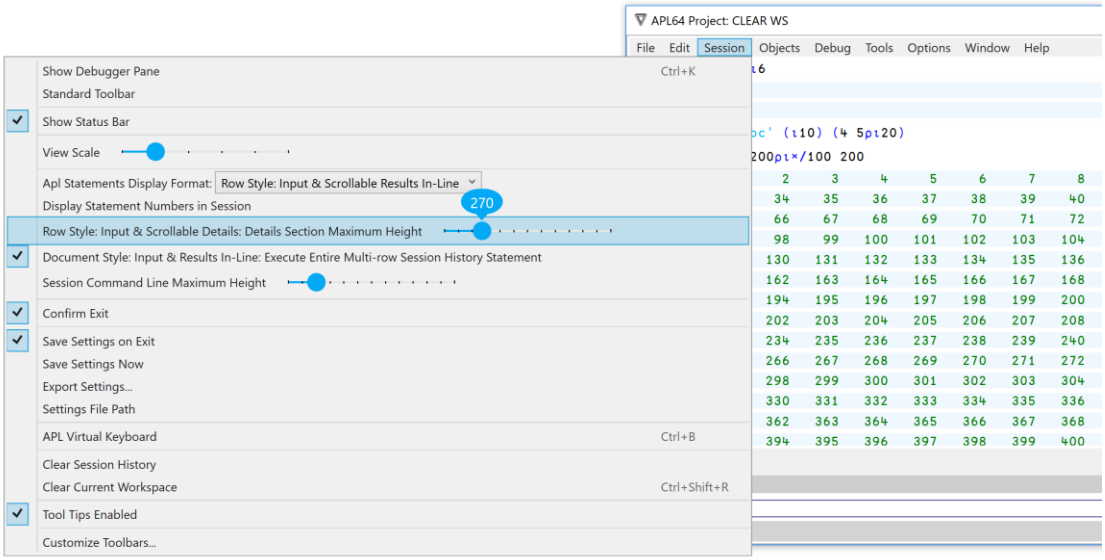
Row Style: Input & Scrollable Results In-Line Format

The APL executable statements or callback statements are displayed in the session history pane in the order executed. Interpreter results are displayed in a scrollable grid immediately following the associated APL executable or callback statement. As the interpreter emits a result it is appended to the scrollable list below the associated APL statement.

A benefit of this format is that the interpreter output for APL statements which would generate many rows of output is scrolled and the session history pane is not scrolled, keeping the associated APL executable or callback statement which caused the interpreter output to remain visible.



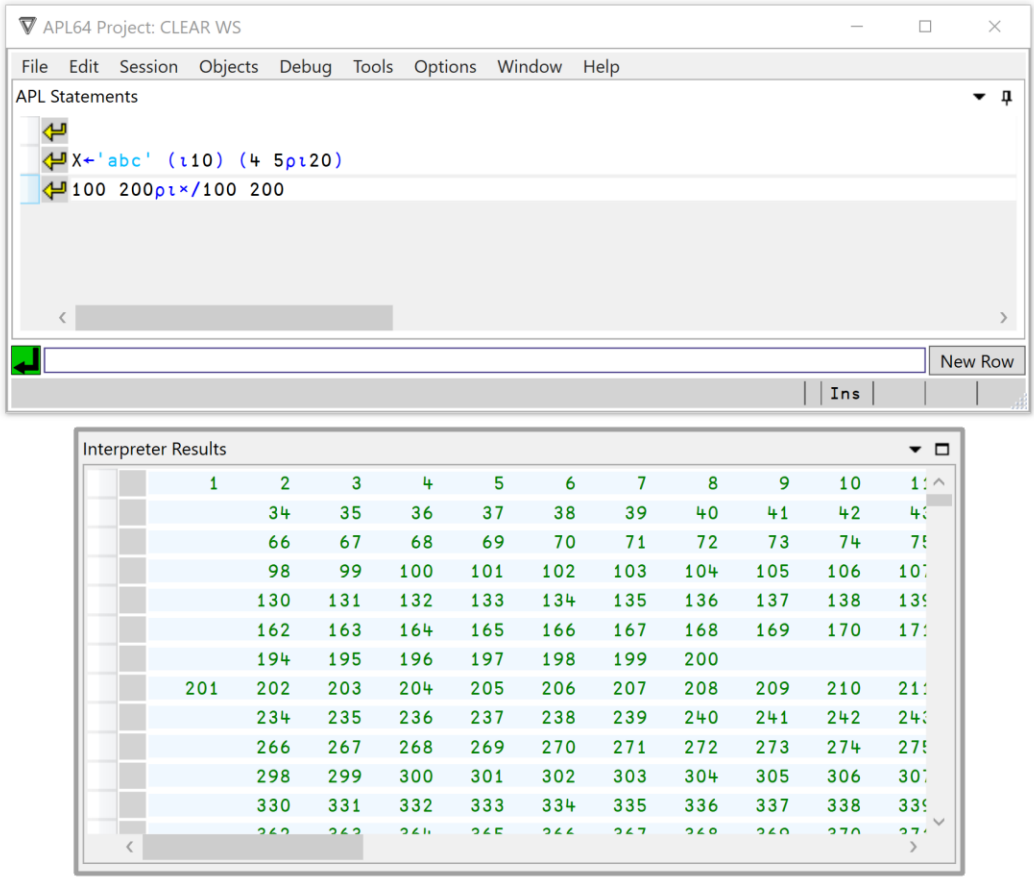
The height of the scrollable list of interpreter results before vertical scrolling is available is user-controlled:



The selection options for the 'Row Style: Input & Scrollable Results In-Line' skin are the same as those for the 'Row Style: Input & Results In-Line' skin.

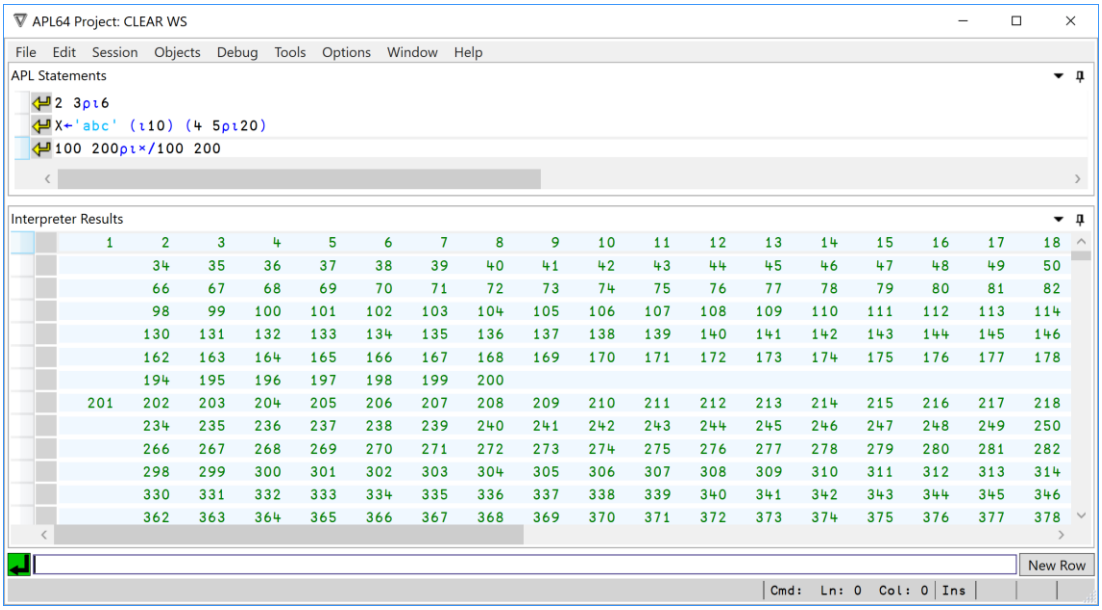
Row Style: Input & Scrollable Results Separated Format

This format displays the APL executable or callback statements in a grid with the interpreter results for a selected statement in a separate grid which can be independently docked or floated on any workstation display. An APL statement is selected by left clicking on the row header of the desired statement.



The selection options for the 'Row Style: Input & Scrollable Results Separated' skin are the same as those for the 'Row Style: Input & Results In-Line' skin.

When the results grid is docked to the main session window, the vertical or horizontal splitter bar can be used to modify the relative space consumed by the two panes of this skin.



Accessing the Session History Pane Text

Selected text in the session history pane may be copied to the Windows Clipboard using Ctrl + C or the right-click context menu options.

Selected text may be executed by using the Enter keystroke.

Selected text may be copied to the Session Command Line by using the Shift + Enter keystroke.

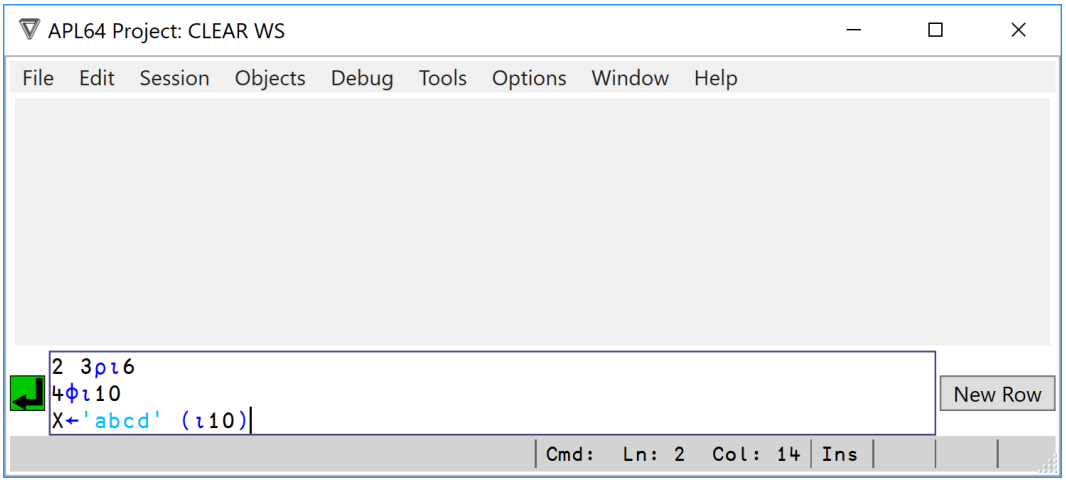
If there is no selected text, the contents of the row containing the cursor will be the 'effective' selection.

The content of the session history pane is not editable. It may be cleared using the Session > Clear Session History menu item.

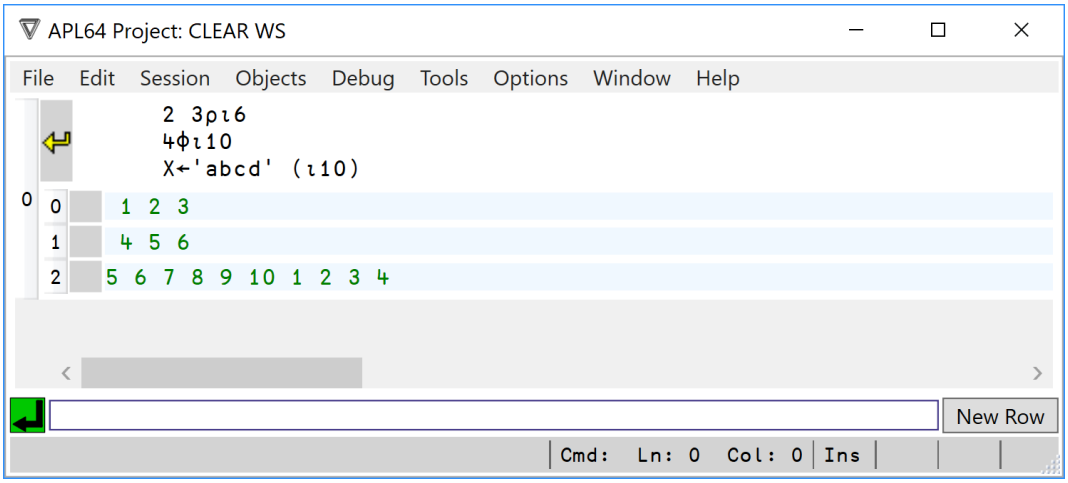
Syntax Coloring in the Session Command Line and Session History Pane

Syntax coloring is applied to the APL executable statements in the Session Command Line and Session History Pane. The APL executable statements include APL programmer-generated statements and interpreter-generated callback statements. All Session skins support this syntax coloring.

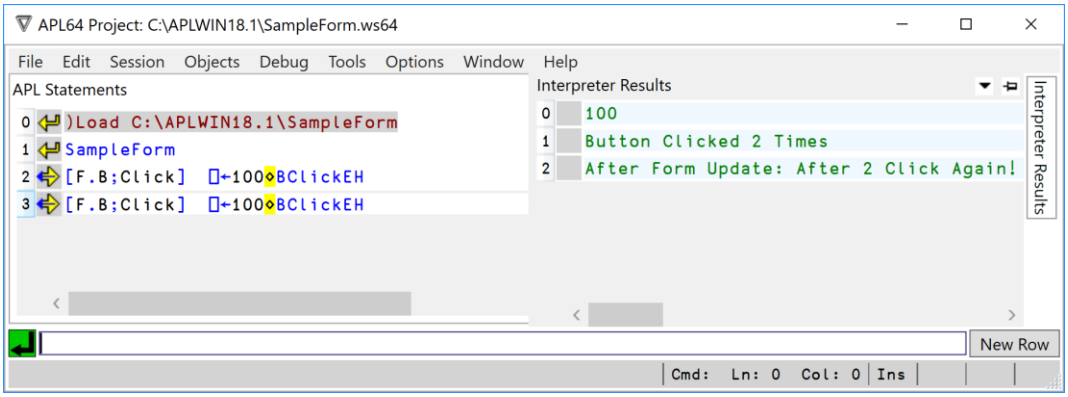
Syntax coloring in the Session Command Line:



Syntax Coloring of APL programmer-generated statements in the Session History Pane:

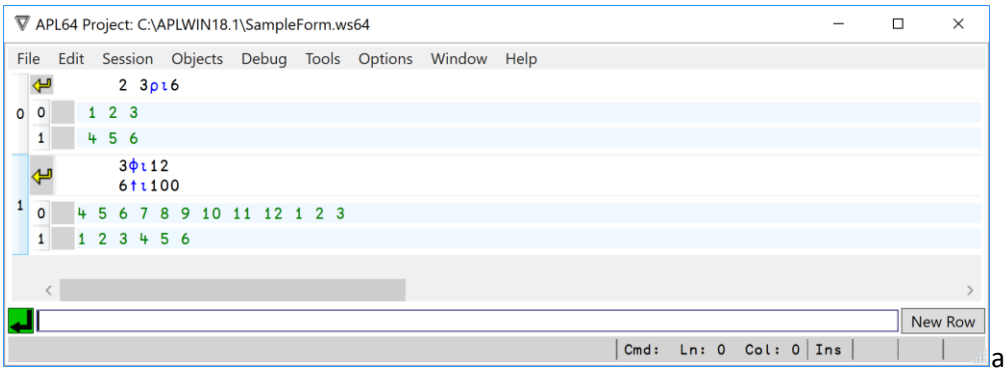


Syntax Coloring of callback statements in the Session History Pane:



Background Color varies with APL Statement Type in Row style Skins

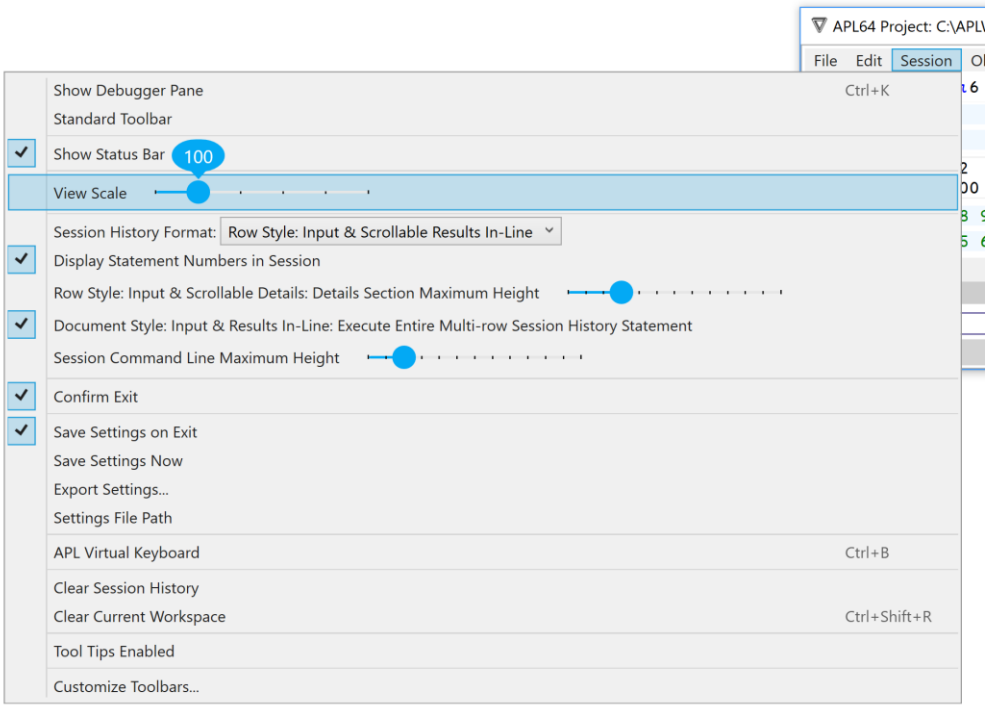
For the Row style skins, the background color of the Session History Pane rows differentiates between result type APL statements and other, e.g., input or callback, APL statements.



a

Session View Scale

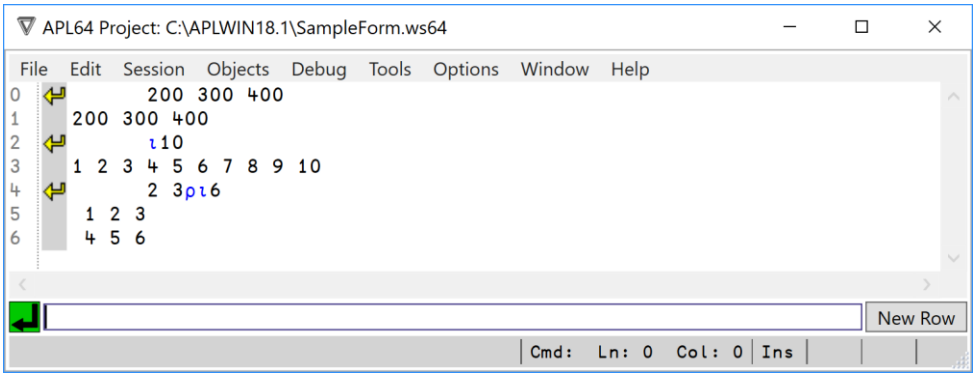
The scale of the session panes is user-controlled via the Ctrl + mouse wheel or the Session > View Scale menu item.



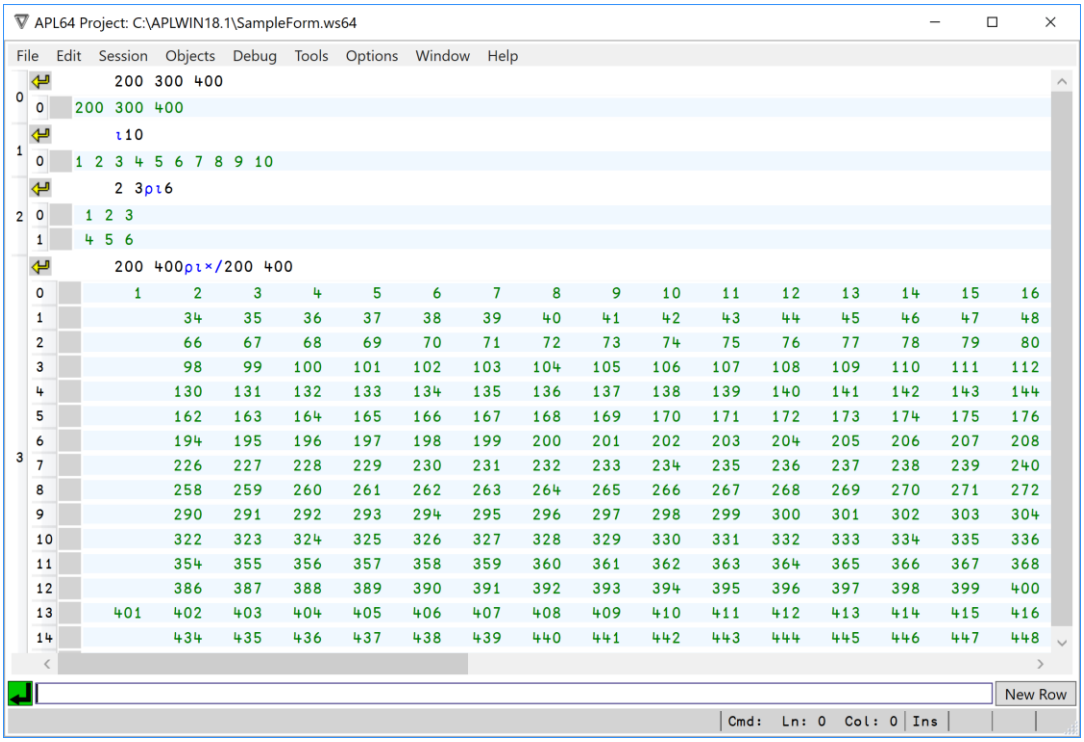
Statement Numbers in the Session History Pane

An option is provided to display statement numbers in the session history pane. For team programming in the APL64 Project, this feature improves group code review sessions.

In the document-style' session skin:

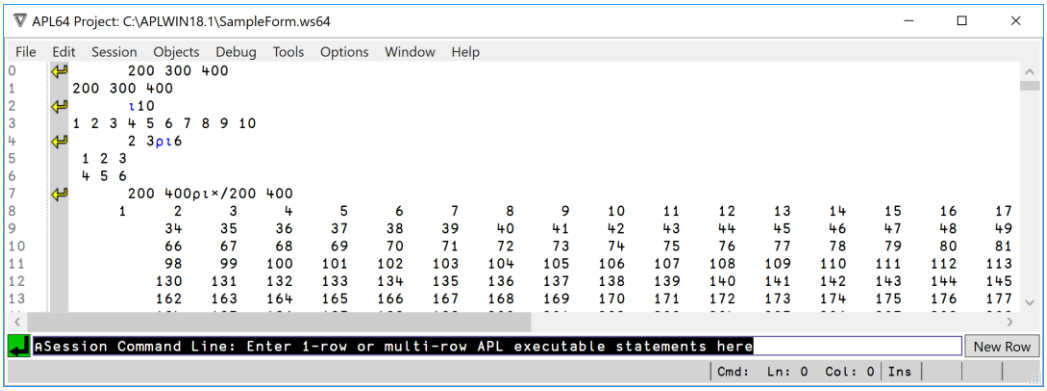


In the 'row-style' session skin with scrollable results in 'master/detail' format:



Session Command Line is separate from the Session History Pane

The Session Command Line is separated from and displayed at the bottom of the Session History Pane.

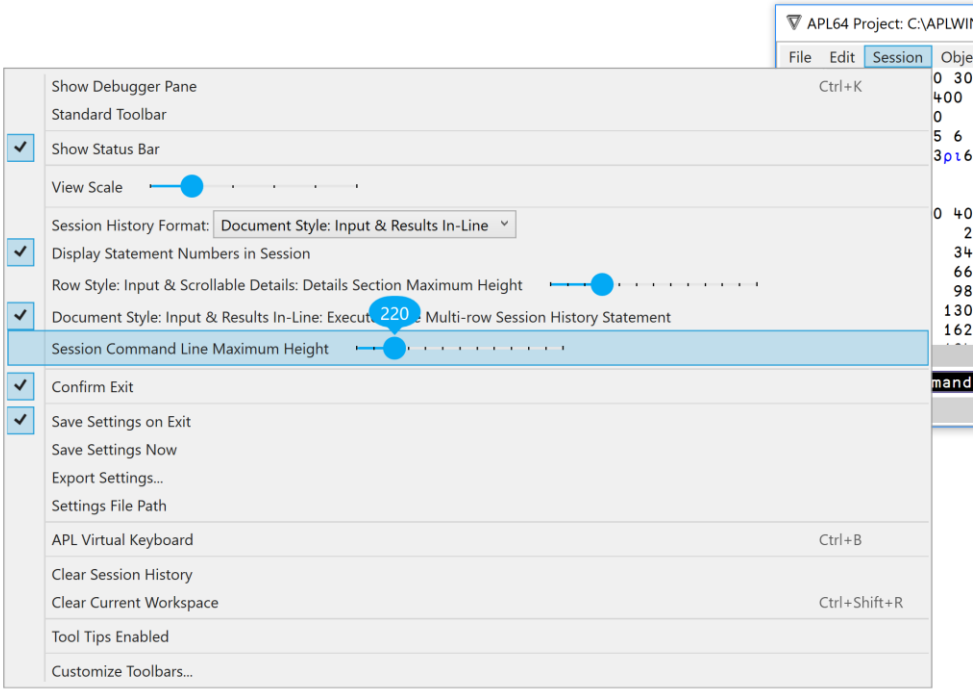


Enter Keystroke executes the APL statements in the Session Command Line

When the cursor is on the Session Command Line the Enter keystroke will execute the APL executable statement(s) in the Session Command Line.

Session > Session Command Line Maximum Height

The height of the Session Command Line before vertical scrolling is enabled is user-controlled via the Session menu. This feature applies when a multi-row APL executable statement is entered or pasted into the Session Command Line.



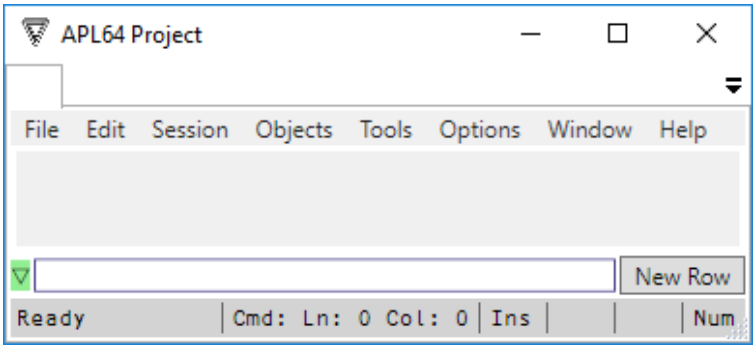
Continuation Lines in the Session Command Line

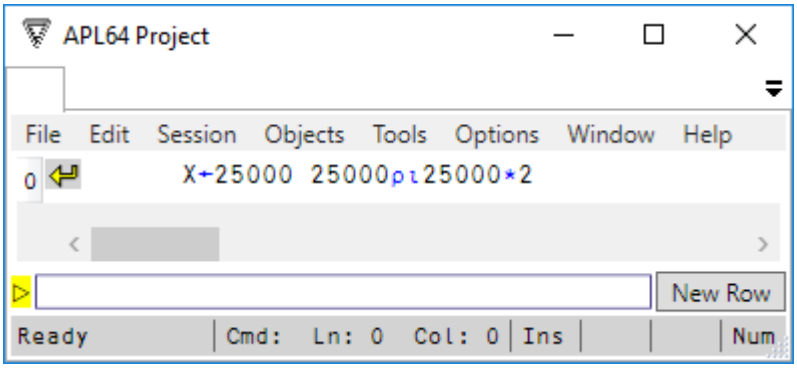
APL executable statements may be continued across multiple lines in the Session Command Line by entering & as the last non-comment character of the line(s) to be continued. Continuation lines are also supported in user-defined functions.

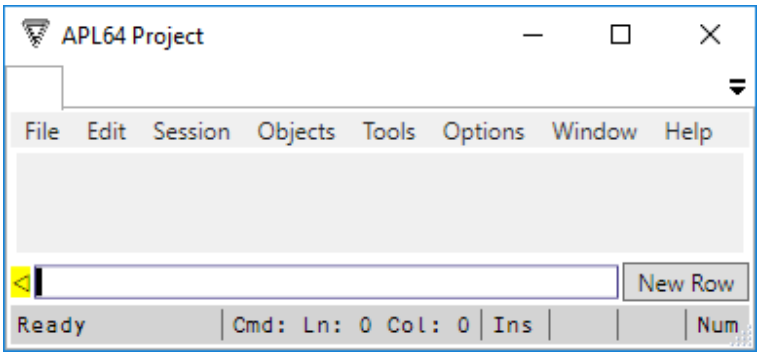
Interpreter Execution State Indicator

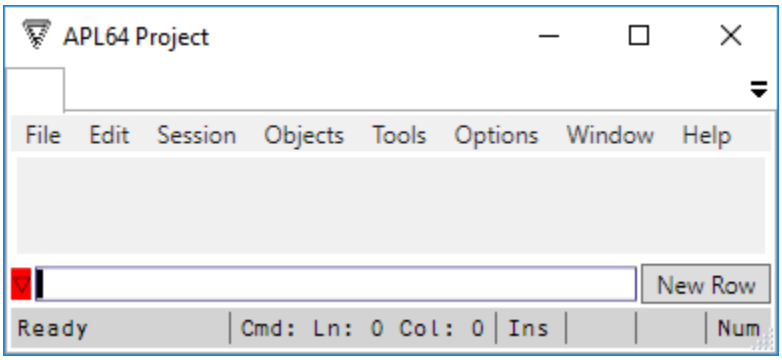
The interpreter execution state is indicated to the left of the session command line.

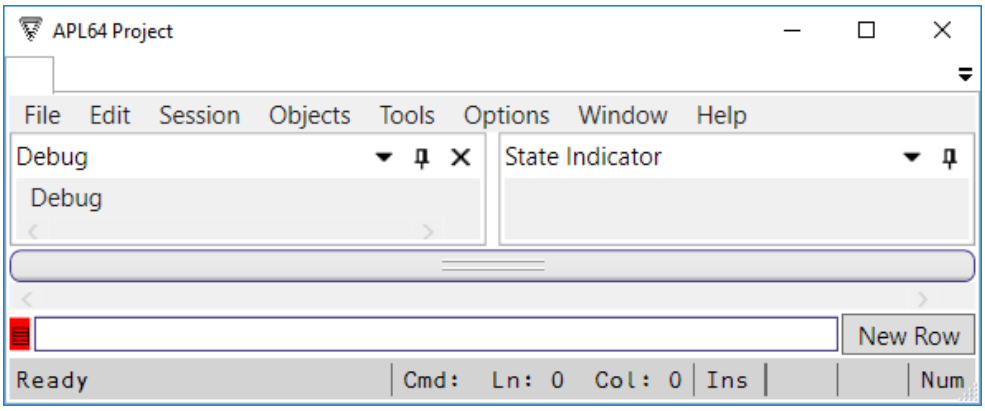
Interpreter Execution State	Input Possible	Glyph	Unicode (Hex)	Background Color
Ready	Yes	▽	u25BD	Light Green
Running User Input	Yes, Injected Execution	▷	u25B7	Yellow
Running Callback	Yes, Injected Execution	◁	u25C1	Yellow
Suspended	Yes	▽	u25BD	Red
Suspended in Debugger	Yes	▤	u25A4	Red







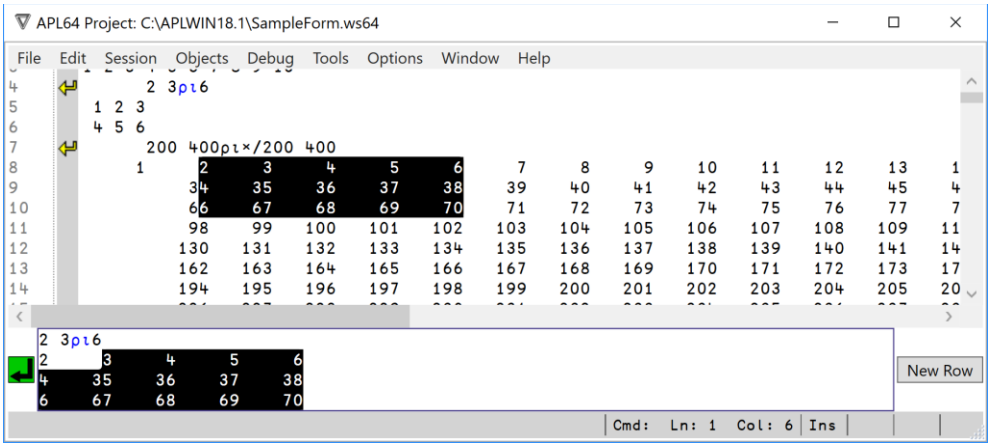




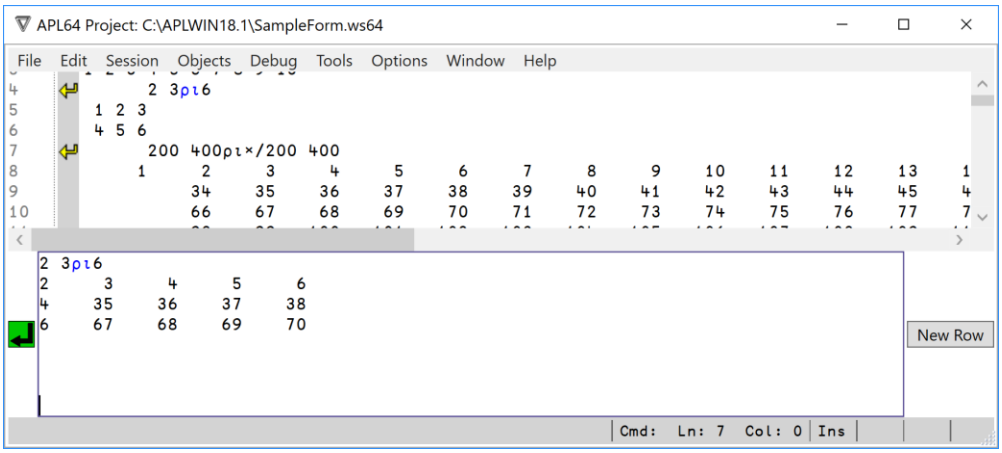
Multi-row APL Executable Statements in the Programmer Session

Often it is convenient to associate multiple APL executable statements into a 'script' which is executed *en masse* in the Programmer Session. Separating the statements in such a 'script' using a 'new line' character, ⎕TCNL, improves readability.

Multi-row APL executable statements may be manually-entered or pasted into the APL64 Project Session Command Line. Shift + Cursor keystrokes will select a contiguous, linear block of text in the Session Command Line. Alt + Shift + Cursor keystrokes will select a contiguous, rectilinear block of text in the Session Command Line.

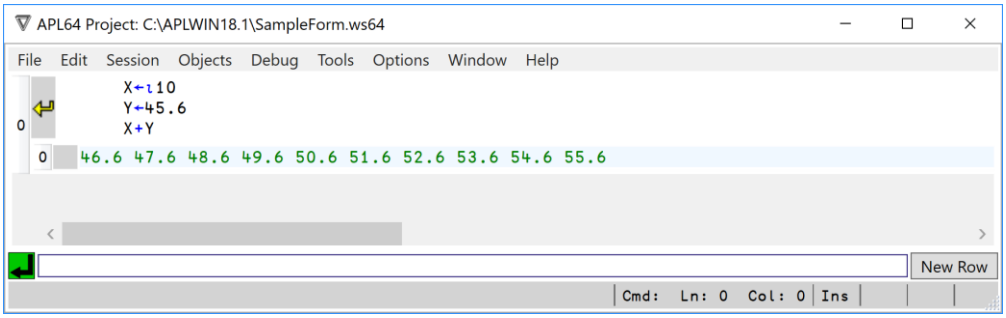


Additional rows in the statement may be created using the ‘New Row’ button to the right of the Session Command Line, the Ctrl + Enter keystroke, or pressing the down arrow on the last row:

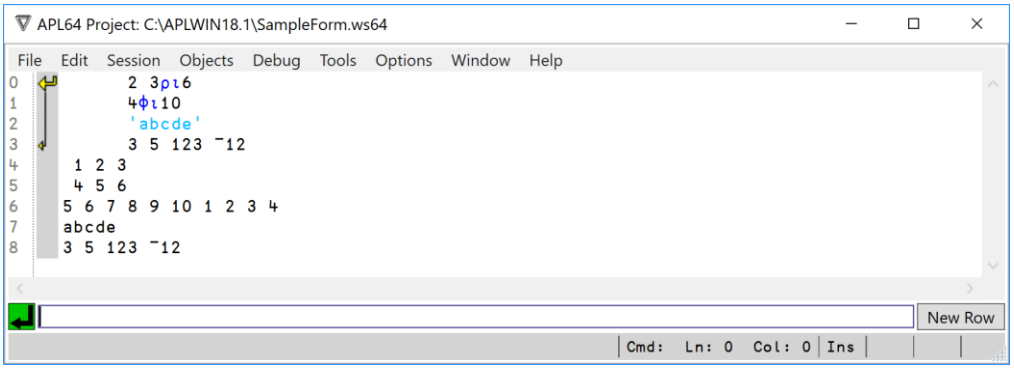


To execute a multi-row APL statement in the Session Command Line, use the Enter keystroke. Once executed, multi-row APL executable statements are identified in the Session History pane using an enlarged row height in the row style skins and by an extended ‘APL executable statement glyph’ in the document style skin.

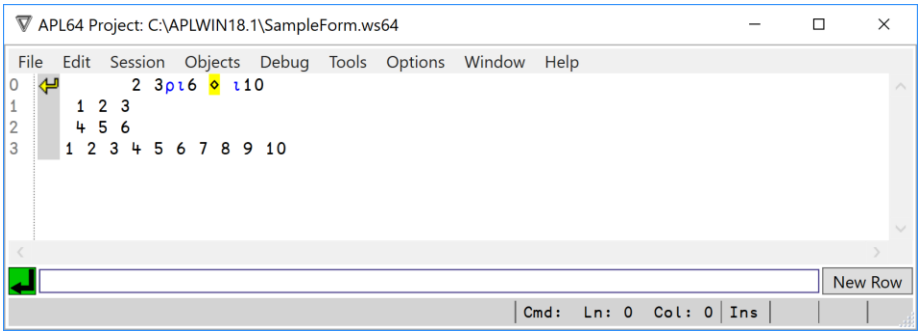
Enlarged row height in a row style skin for a multi-row APL statement:



Extended ‘APL executable statement glyph’ in the document style skin for a multi-row statement:



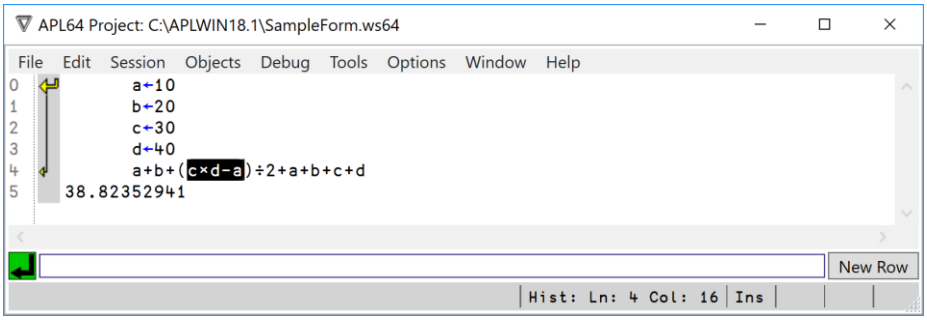
The APL ‘diamond’ statement separator continues to be supported in the APL64 Project.



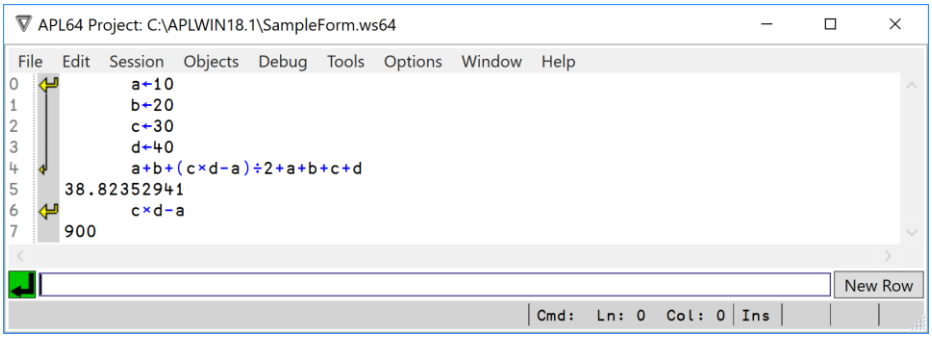
Execute the Selected Part of an APL statement in the Session History Pane

Example:

Enter and execute some APL statements which are the illustrated in the session history pane. Select a portion of an APL statement in the session history pane.



Click the Enter button to execute the selected portion of the applicable APL statement:

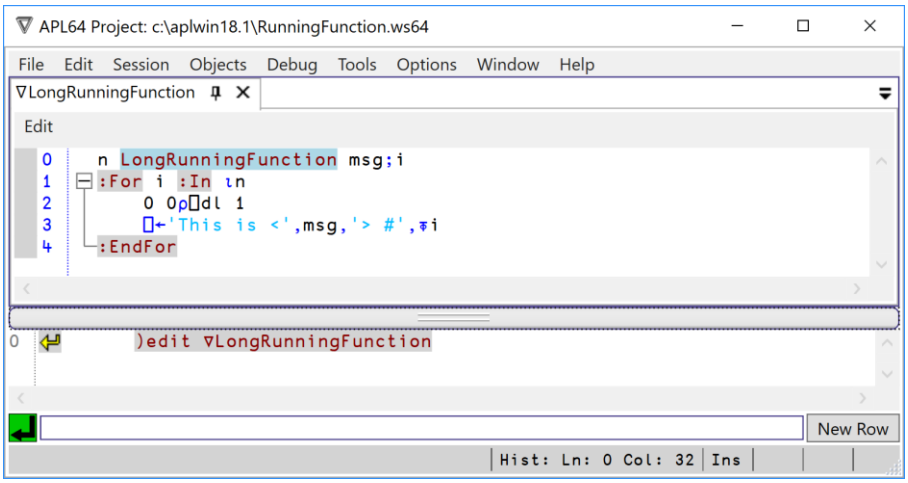


Execution Injection

The APL64 Project interpreter supports 'execution injection' so that the programmer may enter additional APL executable statements in the session. The execution of these additional APL statements is 'injected' into the processing sequence of the interpreter. Since interpreter output from previous APL executable statements may be streaming to the programmer session GUI, to facilitate the programmer entry of additional APL executable statements the session 'command line' is a separate entry field positioned at the bottom edge of the session history pane.

A programmer 'injected' APL executable statement will cause the interpreter to stop processing prior APL executable statements until the most recent 'injected' APL statement processing is completed by the interpreter. Interpreter processing of prior APL statements will resume thereafter in the order of the 'injection' of those prior APL statements.

Example: The Document Style session skin is illustrated



The user executes: 10 LongRunningFunction 'One' and the Session History Pane illustrates the interpreter output of the execution of the 10 LongRunningFunction 'One' statement.

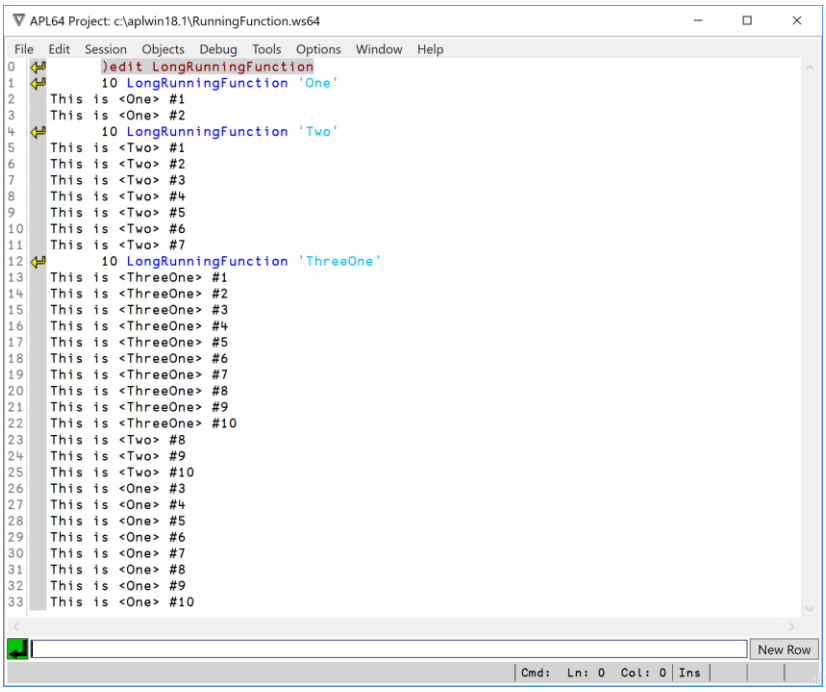
Before that output is complete, the user executes: 10 LongRunningFunction 'Two' and the interpreter suspends the execution of the 10 LongRunningFunction 'One' statement and commences the execution of the 'injected' 10 LongRunningFunction 'Two' statement. The Session History Pane illustrates the interpreter output of the 10 LongRunningFunction 'Two' statement.

Before that output is complete, the user executes: 10 LongRunningFunction 'Three' and the interpreter suspends the execution of the 10 LongRunningFunction 'Two' statement and commences the execution of the 'injected' 10 LongRunningFunction 'Three' statement. The Session History Pane illustrates the interpreter output of the 10 LongRunningFunction 'Three' statement.

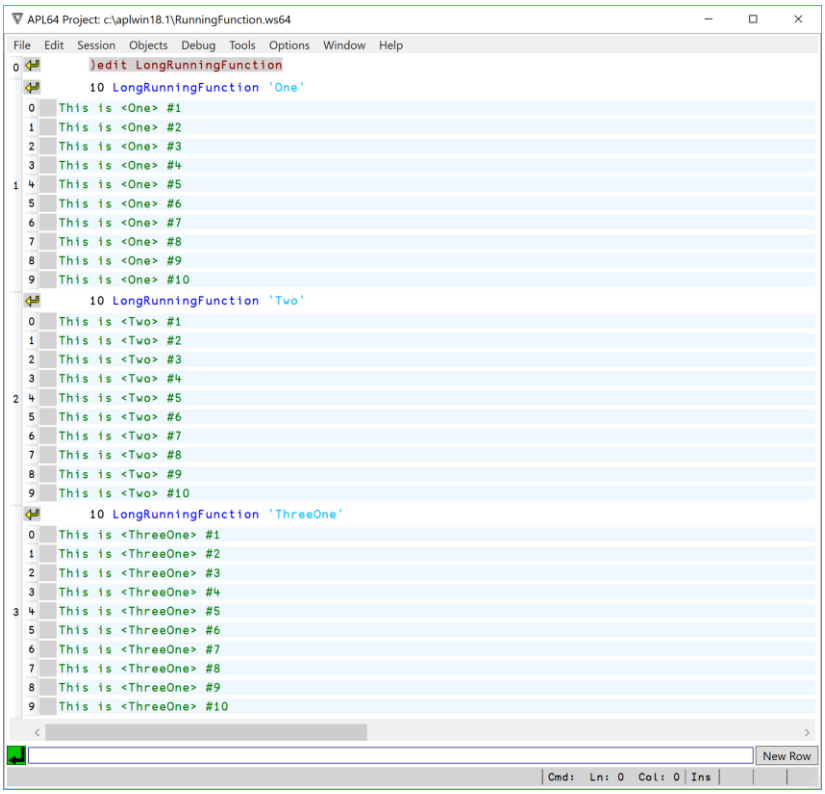
When the interpreter output of the 10 LongRunningFunction 'Three' statement is complete, the interpreter resumes the execution of the 10 LongRunningFunction 'Two' statement and the Session History Pane illustrates the remainder of the output of the 10 LongRunningFunction 'Two' statement.

When the interpreter output of the 10 LongRunningFunction 'Two' statement is complete, the interpreter resumes the execution of the 10 LongRunningFunction 'One' statement and the Session History Pane illustrates the remainder of the output of the 10 LongRunningFunction 'One' statement.

Because the Document Style session skin is selected, the interpreter output is illustrated in the session history pane in the order emitted.



If the Row Style: Input & Scrollable Results In-Line session skin had been selected, the scrollable results detail grids for each APL executable statement would be updated as interpreter output was emitted:



If 'execution injection' processing by the interpreter is underway, the interpreter output will be displayed differently depending on the session 'skin' currently visible.

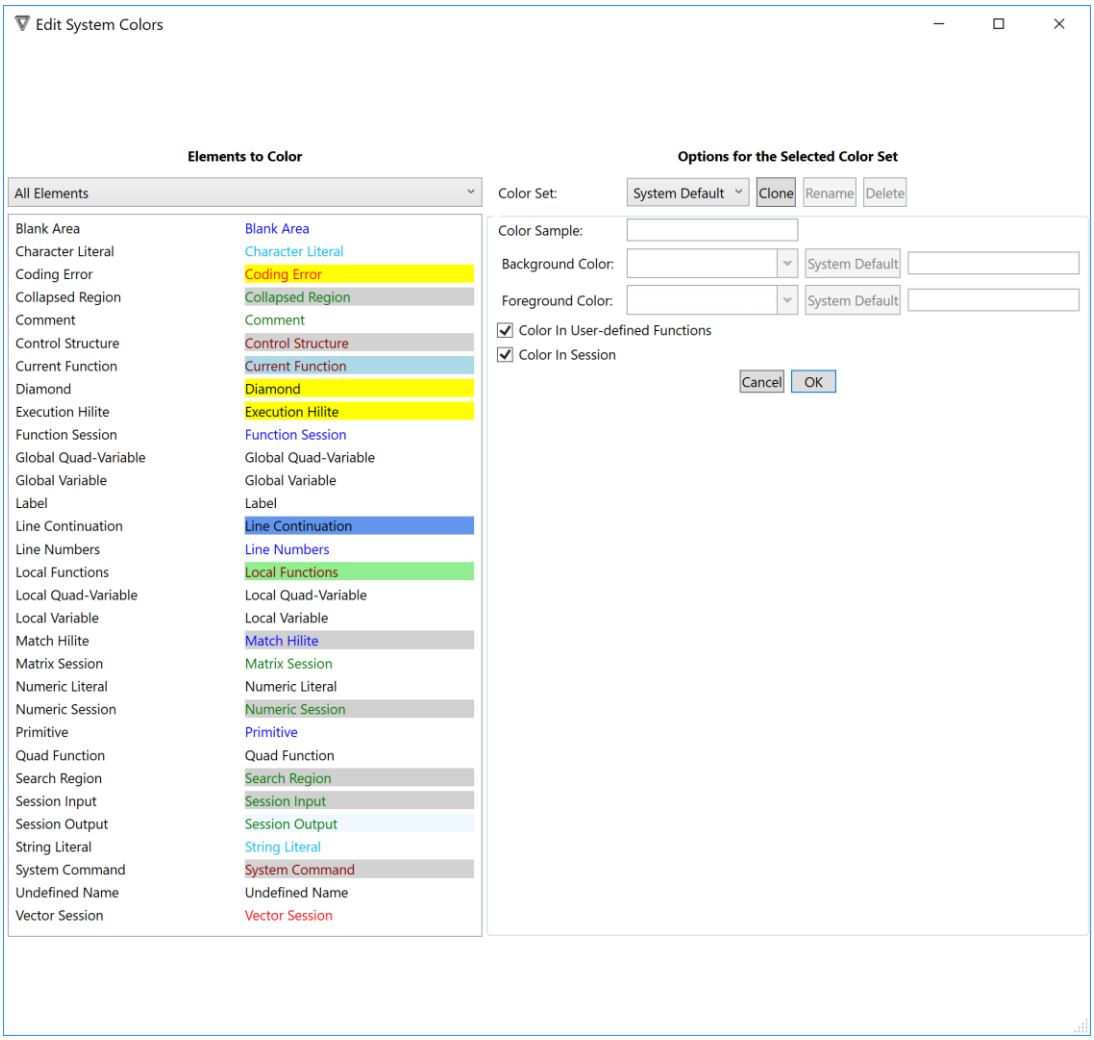
Skin Description	Interpreter Output Display Order in Session	Interpreter Output In Scrolling List
Row Style: Input & Results In-Line	Emitted Order	N
Row Style: Input & Scrollable Results In-Line	Associated with APL Stmt	Y
Row Style: Input & Scrollable Results Separated	Associated with APL Stmt	Y
Document Style: Input & Results In-Line	Emitted Order	N

Colors Dialogue Enhanced

The Colors dialogue provides foreground and background color options within the Session History Pane, the Session Command Line, instances of user-defined function or APL variable editors and the debugger.

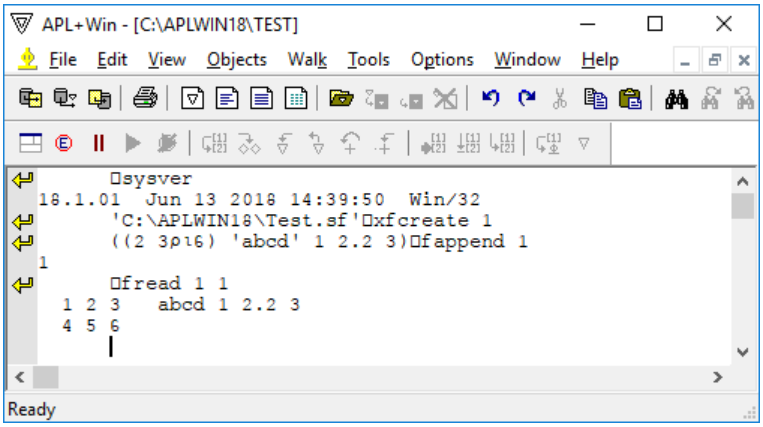
In the APL64 Project the dialogue for the APL programmer to select these color options has been enhanced:

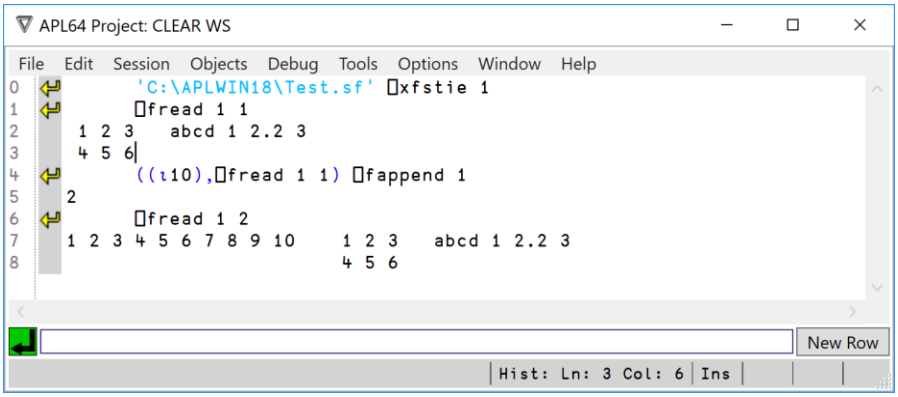
- Multiple color sets are supported
- All color selections are simultaneously visible
- Color sets may be cloned

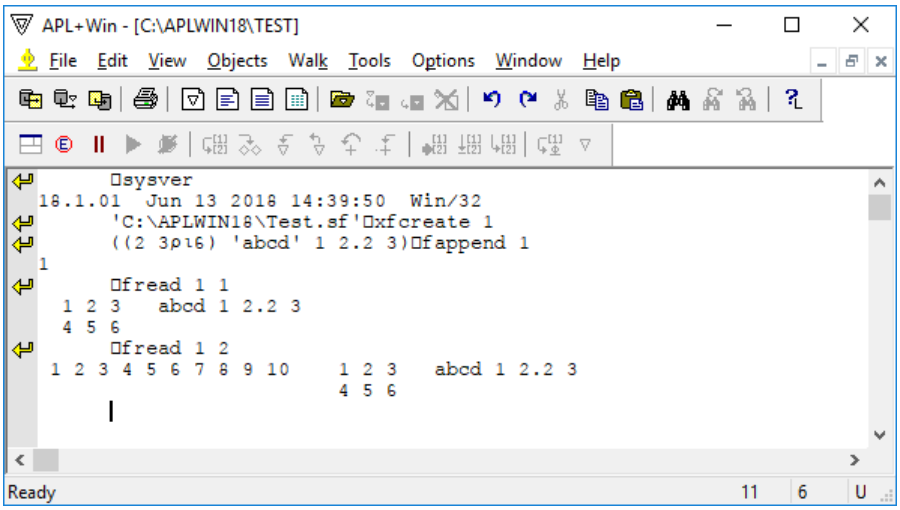


Native and APL Component File Functions

The file functions in the APL64 Project are fully compatible with their analogues in APL+Win. Component files created in APL+Win or the APL64 Project can be read and modified in the APL64 Project or APL+Win. The number of native and component files which can be simultaneously tied in the APL64 Project has been increased.

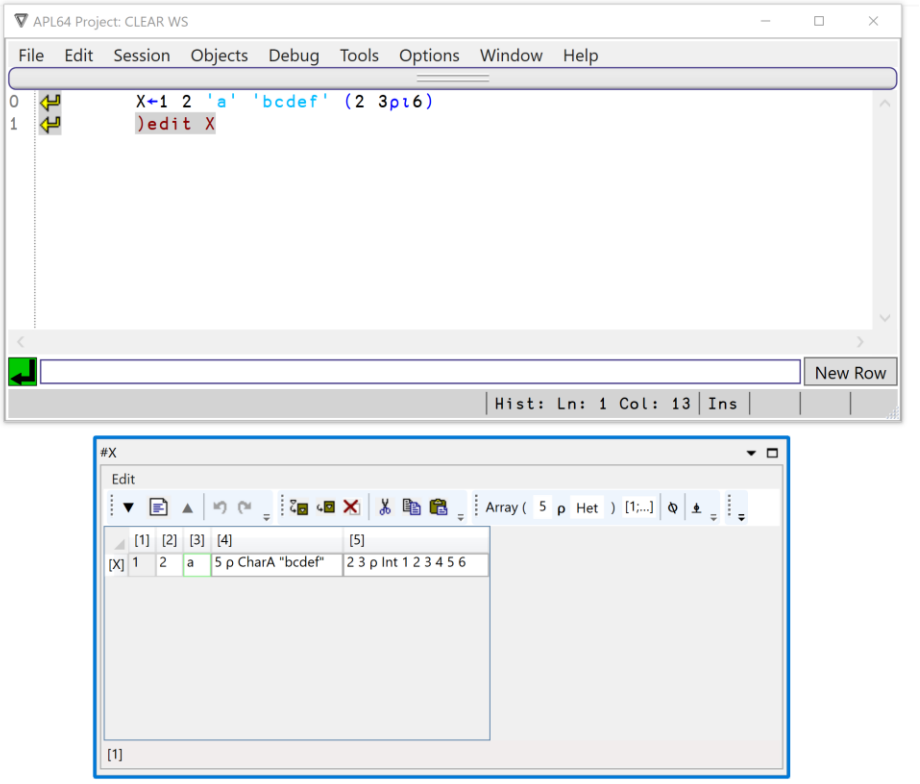






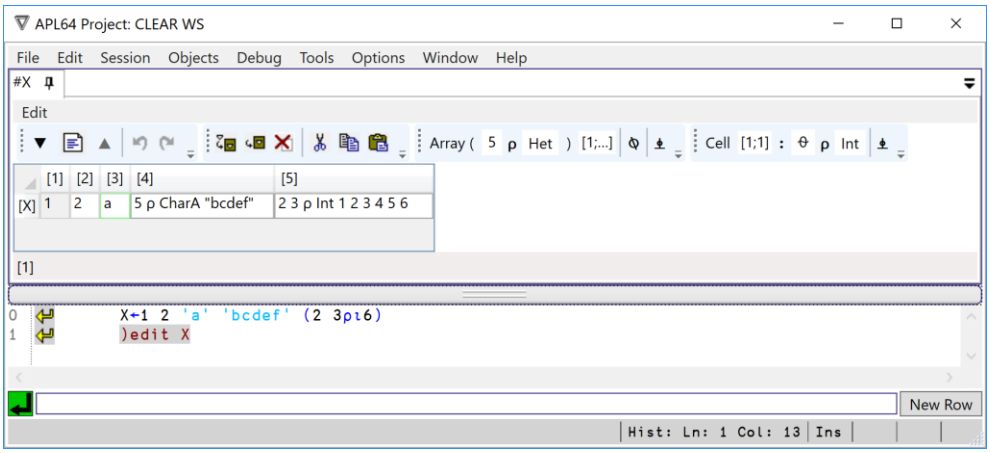
APL Variable Editor: Unified editing of all APL64 Project variable types

The APL variable editor, accessed via the ‘)edit’ system command unifies browsing and editing of any APL64 Project variable including homogeneous, non-homogeneous and nested variables with unlimited redo and undo of user edits. An instance of the APL variable editor is contained within its own pane which can be independently docked or floated on any workstation monitor.



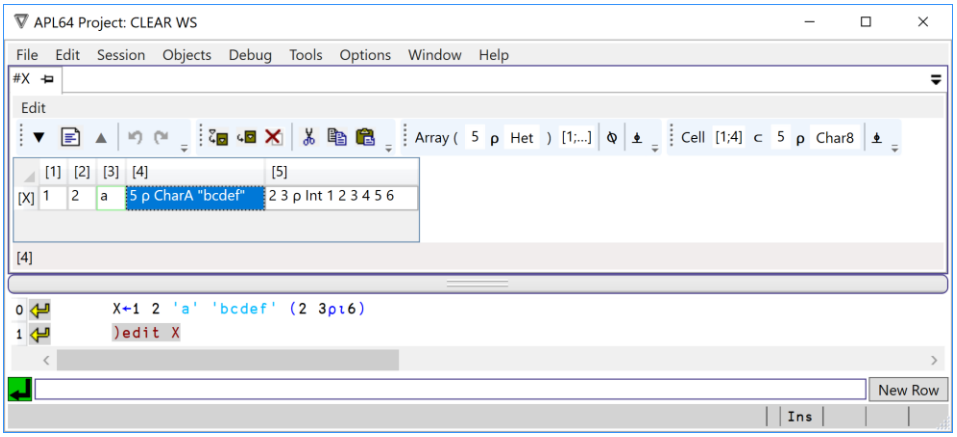
The variable elements and characteristics are displayed in an instance of the APL variable editor pane. The elements of the variable are contained in the cells of a grid with rows and columns representing the shape of the variable.

Value types, i.e. the basic APL64 Project data types of char, string, Int32 and double are directly editable in the containing cell of the APL variable editor.

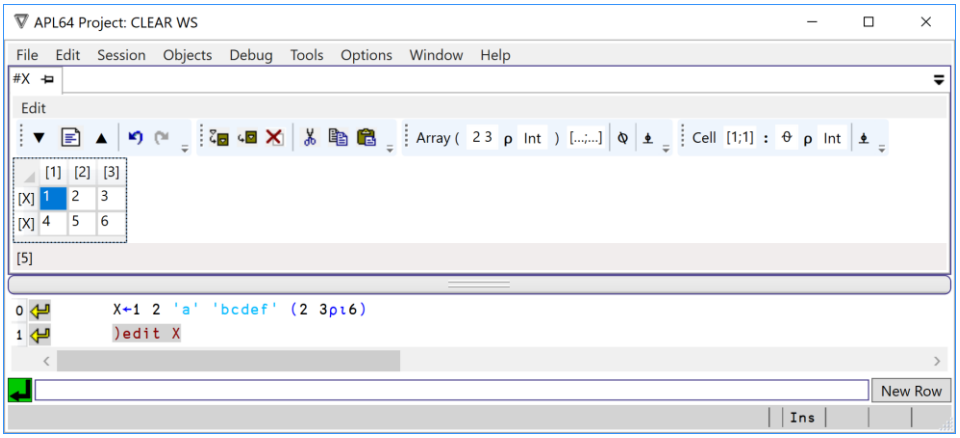


For enclosed elements of a nested variable the user can drill-down into the cells to the level of a value type which can then be directly edited.

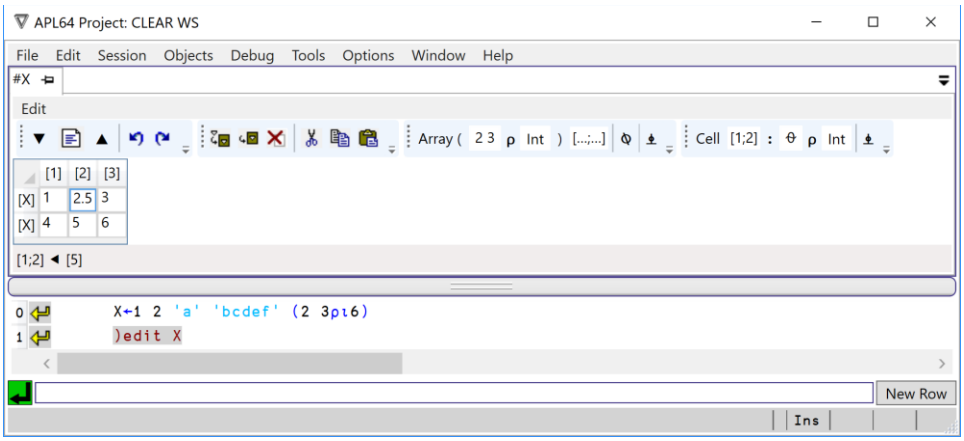
In simple cases, clicking on the enclosed cell, X[1;4] in the example, will expose the contained value type:



For more complex enclosed cells, the drill-down tool bar or clicking on the cell may be used to expose the next level of nesting, e.g., X[1;5] in the example:



Value types can be modified to a different type within the APL variable editor, e.g., changing an integer to a double.



Edit menu provides options to insert or delete columns or rows of the variable.

The tool bar provides options to drill-down to or return from a nested element, modify the shape of the array or its elements, enclose or disclose the array or an element, rotate the display of the variable values, execute APL functions on the variable, view selected planes of a variable with rank greater than two and use a selected text editor when appropriate for a variable.

Microsoft .Net Implementation

Cross-platform APL64 Project Interpreter

The APL64 Project interpreter is compatible with Microsoft .Net Core which enables a cross-platform APL interpreter.

APL64 Project Interpreter is a .Net Assembly

The APL64 Project interpreter is packaged as a .Net assembly so it may be directly integrated into other .Net solutions.

.Net Very Large Objects Supported

The interpreter is compiled in C# using the 'gcAllowVeryLargeObjects' switch which vastly increases available workspace memory.

Automatic Encoding of Character Data to most compact form

Character data is automatically encoded in compact 1-byte (classic APL+Win), standard 2-byte (UTF-16), or extended 4-byte (UTF-32) formats

Simplified □UTF Conversions

Simplified and easy to understand conversion between APL and external character encoding formats via □UTF

Transparent Unicode Support in Arrays and Scalars

Transparent support for Unicode characters in character scalars, vectors, matrices and arrays.

New APL String Data Type

String Definition

An APL string is an ordered group of characters treated as a scalar, with delimiters «…».

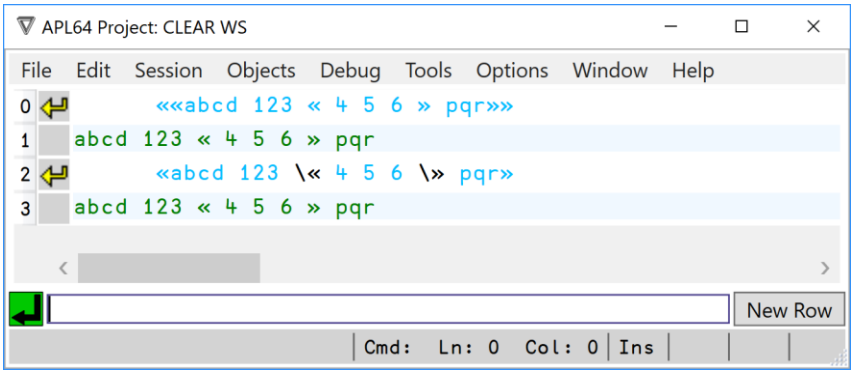
Defining a String Variable in the APL64 Project

The 'chevron' string delimiters are available on the APL64 Project physical and virtual keyboards as Shift + Alt + < for « and Shift + Alt + > for ».

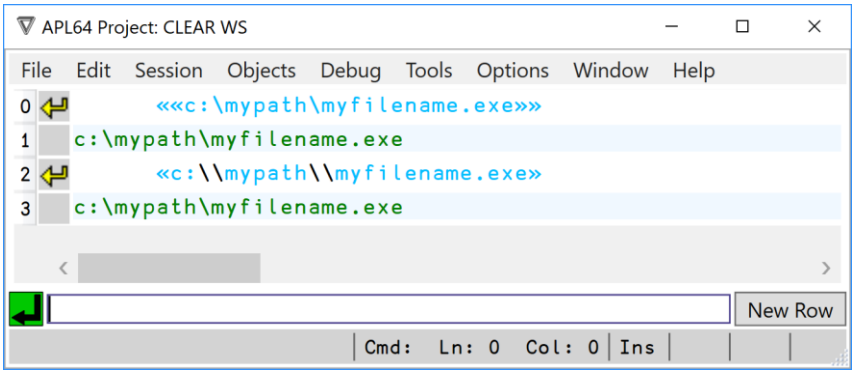
Handling Special Characters in a String Variable

Special characters (such as newlines, tabs) in Strings can be 'escaped, e.g., «123\t456\r» rather than requiring concatenation, e.g., ("123",□TCHT,"456",□TCNL).

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

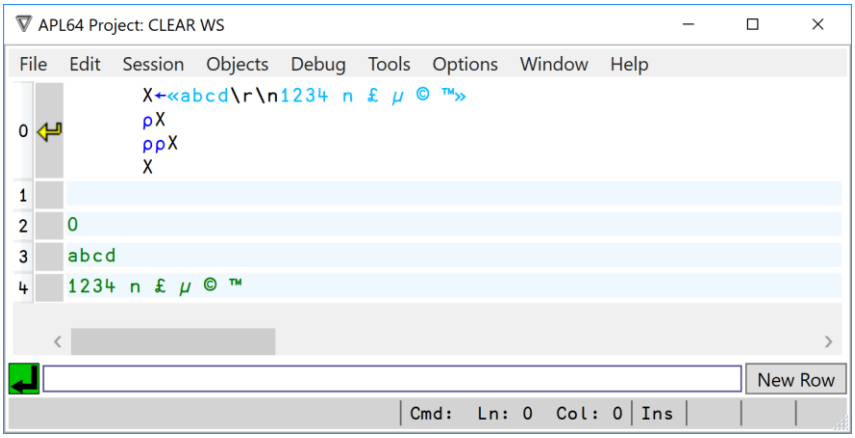


- Paths which contain backslash characters can be done without escaping the backslash if double enclosing 'chevrons' are used.

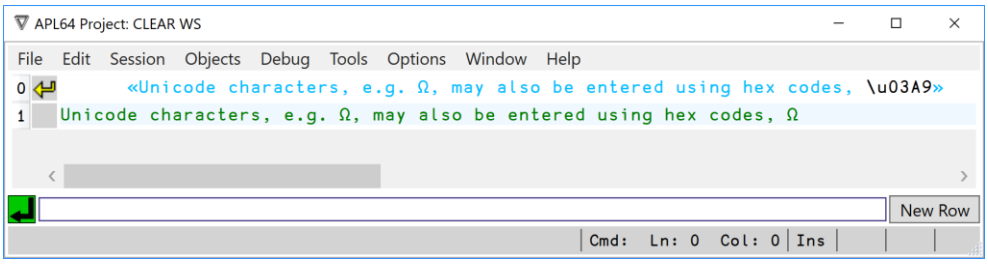


String Variables support Unicode Character Elements

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

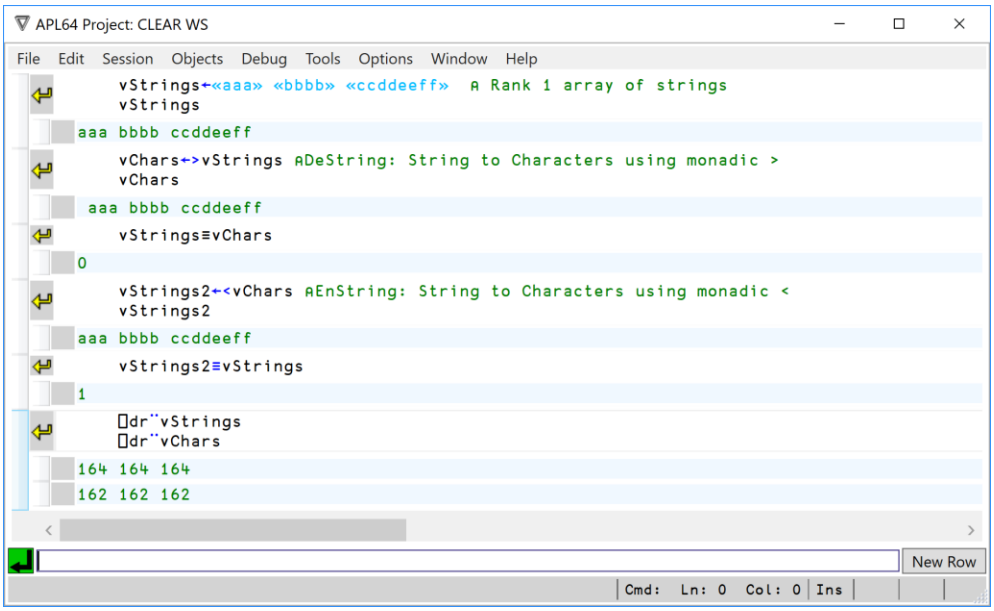


- Unicode characters may be entered using their hex codes via the \uHHHH sequence or \xHH for 2-hex digit codes.



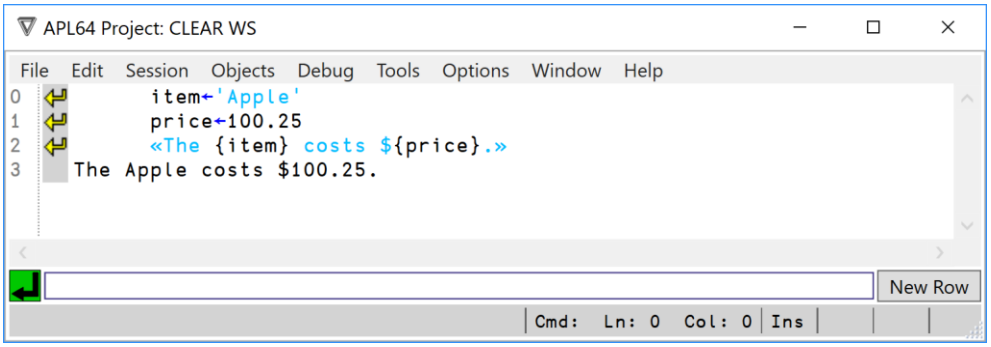
Conversion between String and Character Variables

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



String Substitution

String substitution to simplify text formatting, e.g.



Execution of User-defined Extension Methods

The APL64 Project ⎕na system function may be used to associate an APL function name with a method in an external, dynamic-linked library (dll) so that external method may be used directly from the APL64

Project. This technology is sometimes used to access proprietary commercial methods and performance-enhanced methods. In the APL64 Project this technology is the alternative to the legacy ☐CALL which is not supported in .Net.

Access Win32 Methods in Unmanaged C++ DLLs

The 'DLL …MethodName' ☐NA 'APLFnName' syntax is used if the external dll is available in APL+Win and the APL64 Project, e.g., based on unmanaged C++ and Win32.

Example: Access the Win32 Kernel.32 dll directly from the APL64 Project:

Access C# Methods in .Net Assemblies

In the APL64 Project ☐na has been enhanced to support the 'EXT …MethodName' ⧈NA 'APLFnName' syntax so that methods in an external .Net assembly using C#, VB.Net or Managed C++ may be used directly from the APL64 Project. This syntax can also be used to access an unmanaged C++ method which the APL programmer has exposed within a managed C++ method.

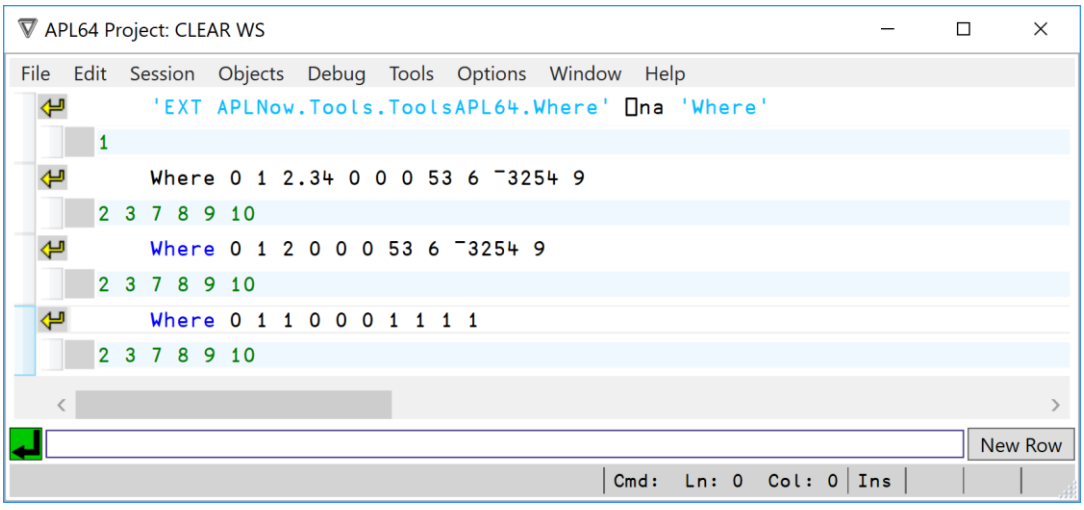
Any .Net programming language, e.g., C#, VisualBasic, Python, etc., may be used in the external .Net assembly.

Example: Access a user-defined C# assembly method directly from the APL64 Project:

A user-defined C# assembly containing the Where() method which returns the indices of the location of non-zero elements of a numeric or Boolean array of rank less than 2.

```
public static Aval Where(IExtContext context, Aval left, Aval right)
{
    if (left != null) throw new ValenceError();
    if (right.Rank > 1) throw new RankError();
    var res = new List<int>();
    var io = context.QuadIO;
    switch (right.Type)
    {
        case DT.Bool:
            var bv = right.BoolValues;
            for (var i = 0; i < bv.Length; ++i) if (bv[i]) res.Add(i + io);
            break;
        case DT.Int:
            var iv = right.IntValues;
            for (var i = 0; i < iv.Length; ++i) if (iv[i] != 0) res.Add(i + io);
            break;
        case DT.Double:
            // This is doing "⎕CT tolerant compare" of floating point values.
            // Could run faster using less APL-ish (dv[x] != 0.0) exact comparison.
            var ct = context.QuadCT;
            var dv = right.DoubleValues;
            for (var i = 0; i < dv.Length; ++i) if (Num.qctcomp(dv[i], 0.0, ct) != 0) res.Add(i + io);
            break;
        default:
            throw new DomainError();
    }

    return new Aval(res.ToArray());
}
```

In this example the user-defined .Net assembly has a reference to the .Net APLNow.Data assembly as a cloud-based Nuget package which is a component of the APL64 Project so that the appropriate APLNow.Data.IExtContext can be incorporated into the arguments of the method to be exposed as a user-defined function in the APL64 Project. The compiled user-defined C# assembly DLL has been placed in the same folder as the APLNow.Data.dll.

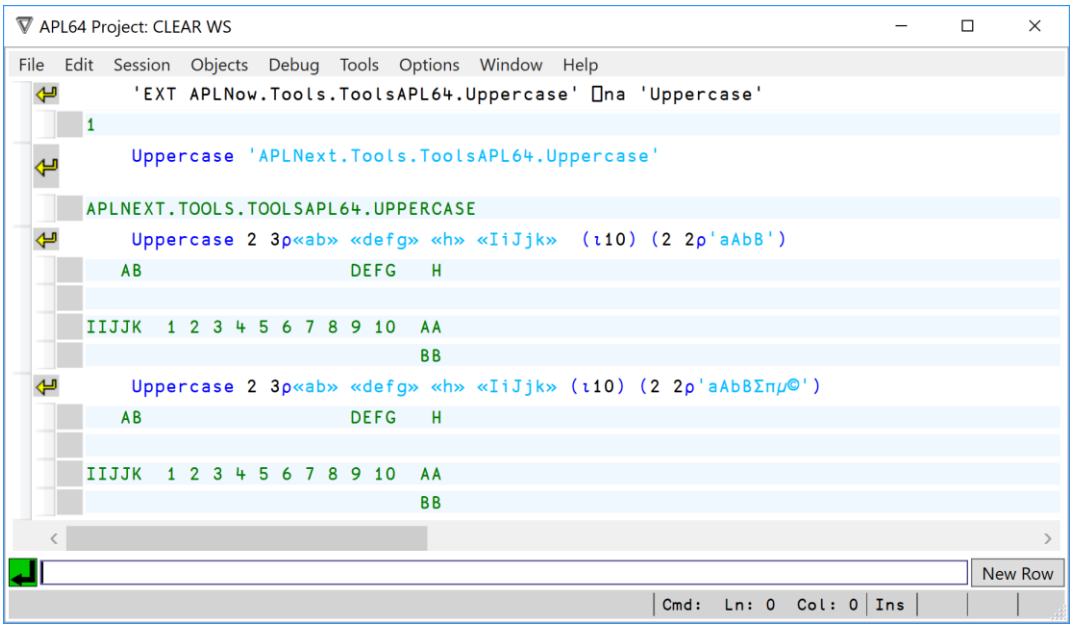


Example: Access a user-defined C# assembly method directly from the APL64 Project:
A user-defined C# assembly containing the Uppercase() method which returns input values in uppercase.

This user-defined .Net assembly has a reference to the .Net APLNow.Data assembly as a cloud-based Nuget package which is a component of the APL64 Project so that the appropriate

APLNow.Data.IExtContext can be incorporated into the arguments of the method to be exposed as a user-defined function in the APL64 Project workspace. The compiled user-defined C# assembly DLL has been placed in the same folder as the APLNow.Data.dll. This method handles heterogeneous, nested APL64 Project variables.

```
public static Aval Uppercase(IExtContext context, Aval left, Aval right)
{
    if (left != null) throw new ValenceError();
    switch (right.Type)
    {
        case DT.Zc:
            return new Aval(right.ZcValues.Select(Zc.ToUpper), right.Shape);
        case DT.Char:
            return new Aval(right.CharValues.Select(char.ToUpper), right.Shape);
        case DT.Ucs:
            return new Aval(right.UcsValues.Select(Ucs.ToUpper), right.Shape);
        case DT.String:
            return new Aval(right.StringValues.Select(c => c.ToUpper()), right.Shape);
        case DT.Aval:
            return new Aval(right.AvalValues.Select(c => Uppercase(context, null, c)), right.Shape);
        default:
            return right;
    }
}
```



Replacements for Deprecated APL System Functions

Assembler functions accessed via ☐CALL and ☐ARBIN are not supported but can be easily and efficiently replaced by 64-bit ☐NA extension functions.

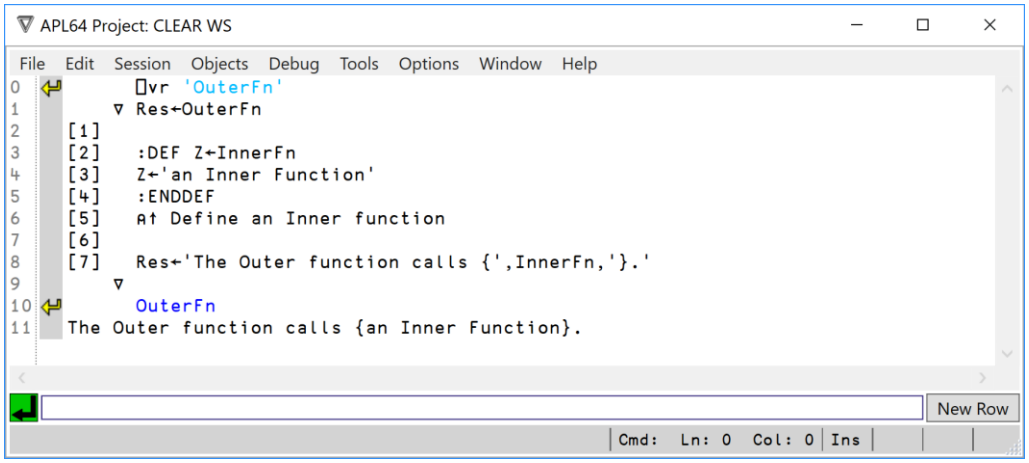
User-defined APL Functions

Local Inner Functions

Local (Inner) functions can be defined inside of 'outer' functions via the new :DEF control structure declaration. While the outer function is executing, it can make calls to the inner function. Unlike using ☐DEF to dynamically define a local function inside another function, the :DEF control structure does not have any execution overhead associated with it.

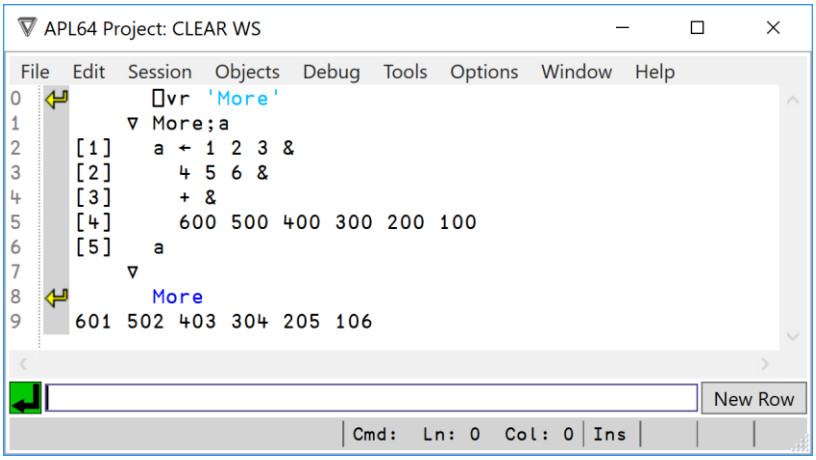
- The inner function is statically defined once, when the outer function is defined and doesn't incur any execution overhead to redefine the inner function each time it is needed. When ☐DEF is used to define a local function, there is a substantial execution cost incurred each time the function is defined, even before it is first executed.
- The :DEF control statement is easier to read and easier to code than functions specified using ☐DEF or ☐FX since quoted strings are not needed.
- The :DEF control statement supports setting of stop and trace lines via the session manager editor, just like in non-inner functions.
- Functions declared via :DEF are local to the functions they are defined within and don't "pollute" the global workspace with specialized functions that are only applicable to the context of the containing function where they are defined.

Example:



Continuation Lines

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. In this case, lines [1], [2], [3], and [4] are treated as a single statement.



Function Header Enhancements

- Optionally use space, instead of semi-colon between local variables in a function header, e.g.,

  larg Foo rarg;local1 local2 local3…

- Suffixed or multiple, sequential semi-colons permitted in a function header to avoid parsing warnings when editing a function header, e.g.

  larg Foo arg;local1;;;local2 local3;

- Suffixed comment permitted in a function header, e.g.,

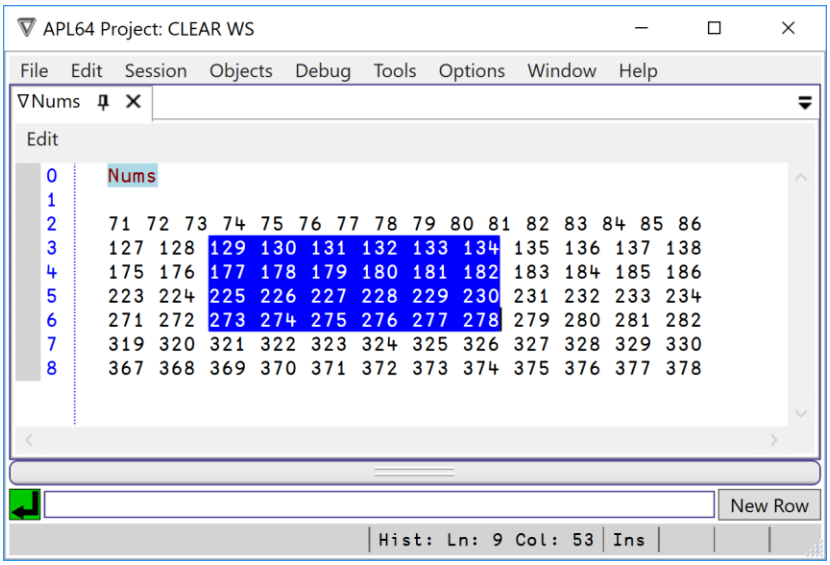
  larg Foo rarg… ⍝Fn header comment

Selection of Linear or Rectilinear Text Blocks in Function Source Code

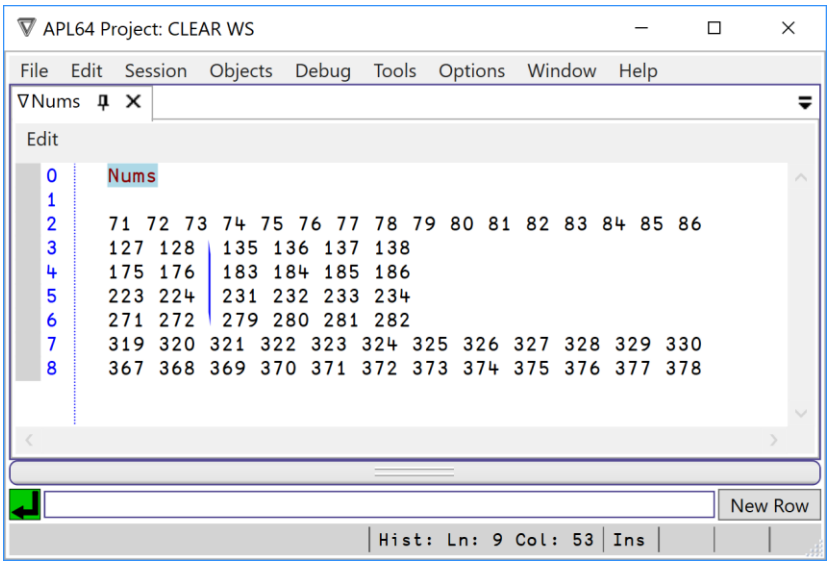
The Shift + Cursor keystrokes select a contiguous, linear block of text in the source code of a function being edited. The Alt + Shift + Cursor keystrokes select a contiguous, rectilinear block of test in the source code of a function. Selected text may be deleted, pasted-in or copied.

Example:

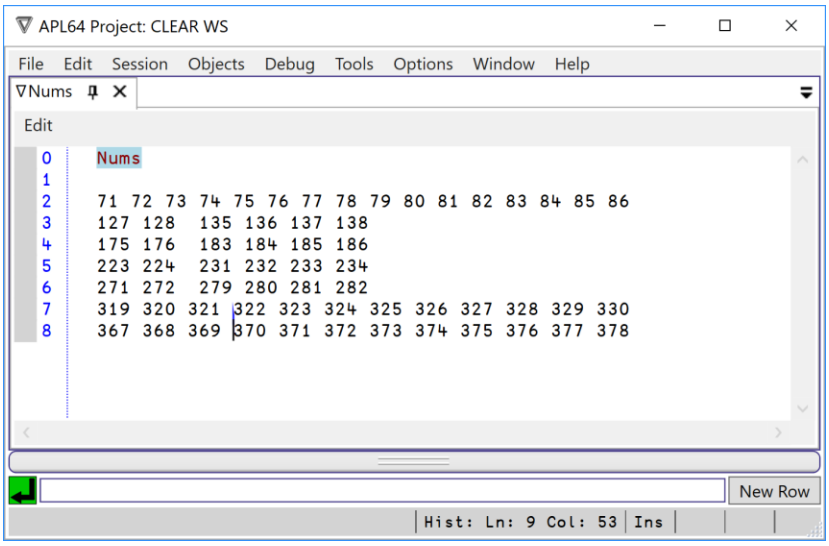
Select a rectilinear block of function source code text:



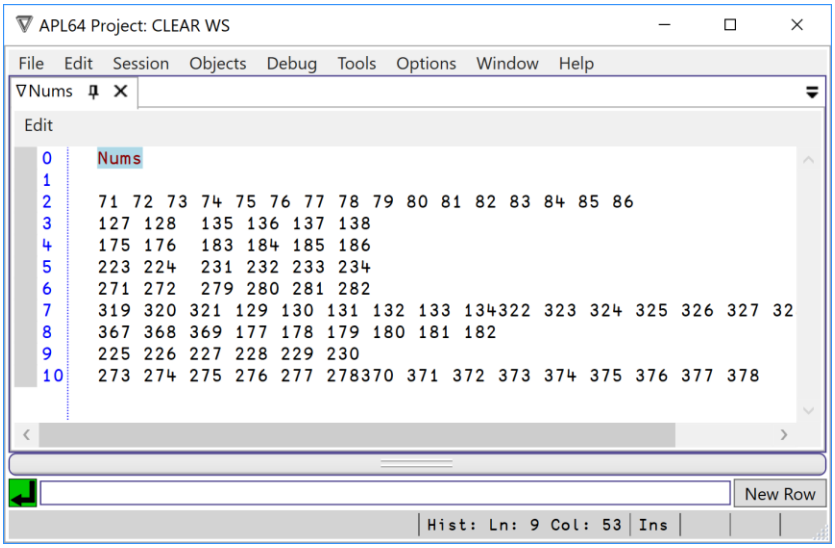
Delete the selected block of text from the function source code:



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



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



Extended State Indicator □SIX

The □SIX system function provides extended state indicator information controlled by its right argument.

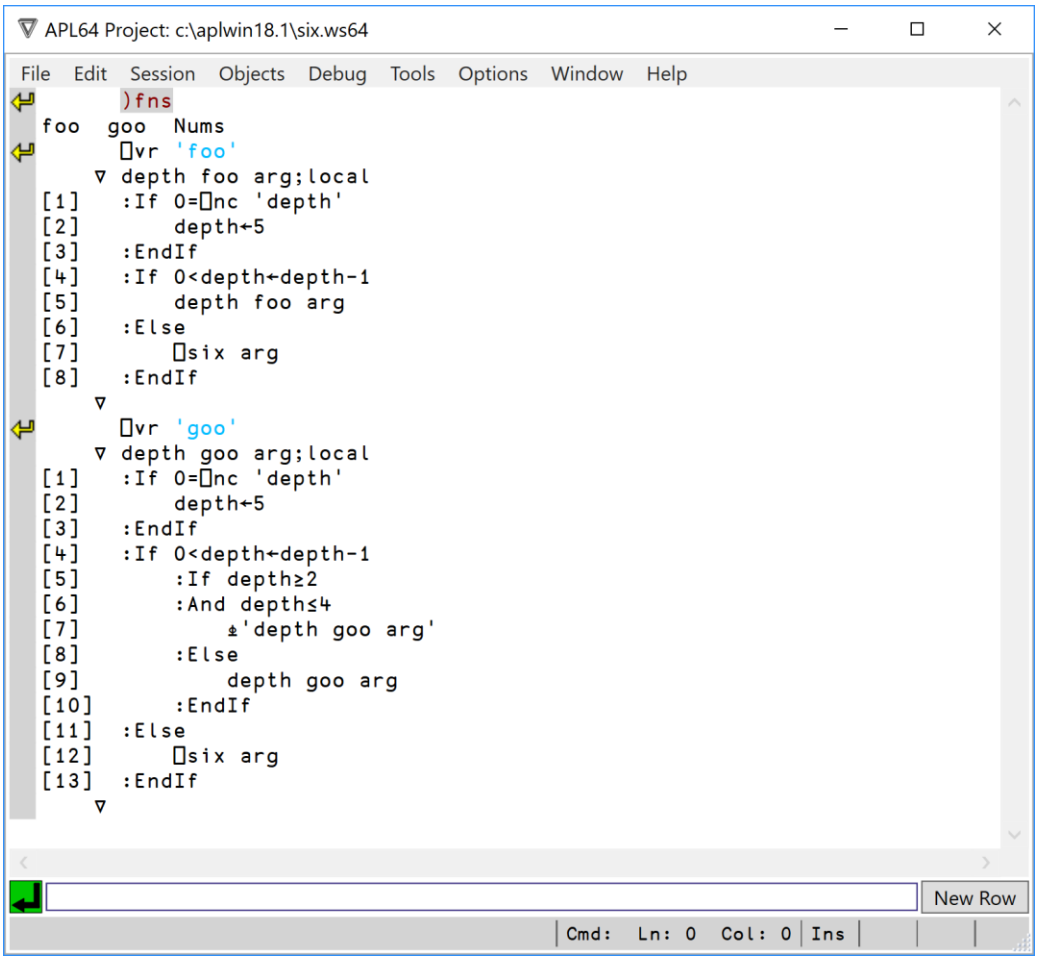
The □SIX system function output includes the statement that is being executed at each level of the state indicator with the [Imm] prefix or the ellipsis if there is more state indicator information that is not being displayed.

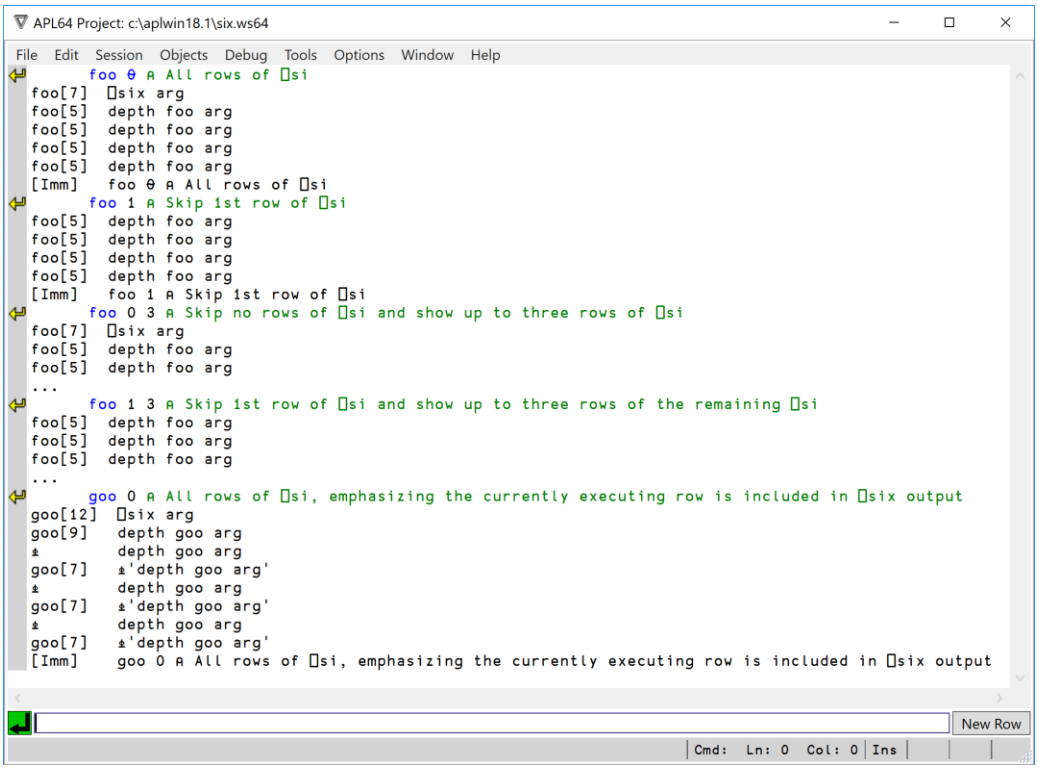
□SIX Right Arguments	**□SIX Output Description**
'L'	List local variable names and value tips that show the values of the locals, like the APL+Win crash dump file content
'S'	List local variable names and value tips for the current and shadowed values of locals

'G'	List global variable names and value tips for any globals that are not shadowed by locals on the state indicator
1st Numeric Arg	# rows of state indicator to skip
2nd Numeric Arg	# rows of state indicator to display; default 0 indicates all rows
3rd Numeric Arg	# columns of value tip information to display; default 100

□SIX system function can be useful when the □SI stack is very deeply-nested and the important portion of the stack is one or a few levels back from the current state. Using □SIX the APL programmer has precise control over what portion of the state indicator is returned.

Example: The 'foo' function will recursively build up the state indicator stack so that the results of □SIX can be illustrated.





High-resolution Timer

- ☐AI[2] has been enhanced to use the .Net Stopwatch class when available
- ☐TT returns the number of timer ticks since the APL+Win session started
- ☐TF returns the number of timer ticks per second, which for most workstations will be 1E7.

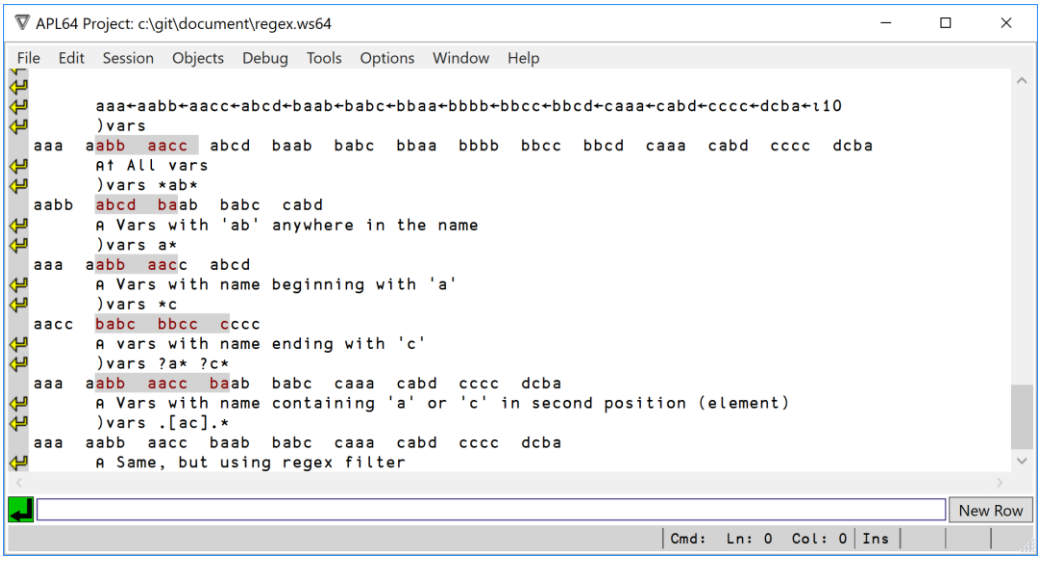
Example which illustrates the use of ☐TT and ☐TF which eliminate the overhead of computing the other elements of ☐AI or the need to index to obtain ☐AI[2].

```
      ⎕TT
87716865

      ⎕TF
10000000

      ⎕TT ÷ ⎕TF
18.8115512

      ⎕TT ÷ ⎕TF ⋄ ⎕AI[2]
25.6503921
25.6626072
```

Enhanced System Commands:

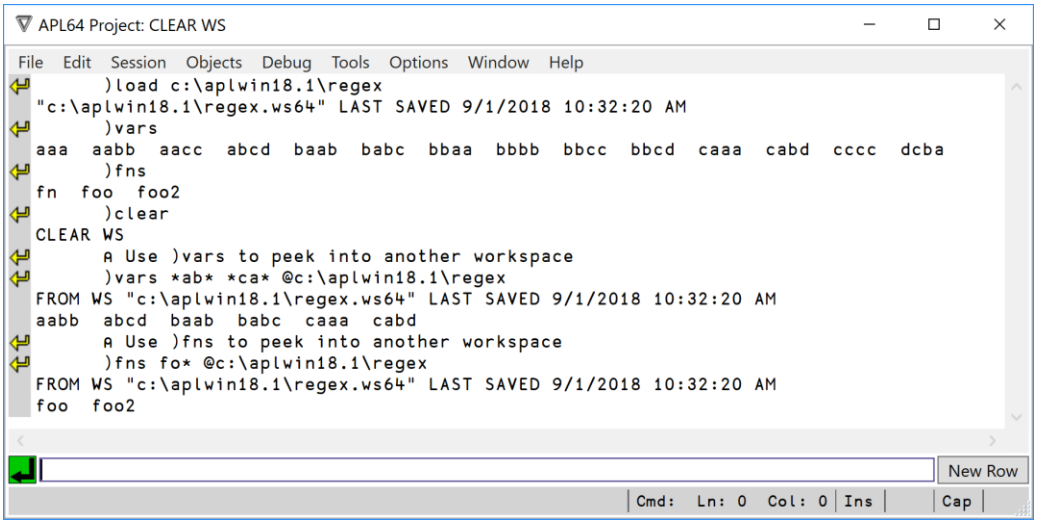
Regex Wildcard Filters for)VARS,)NAMES,)FNS,)COPY,)PCOPY System Commands

Improved wildcard options with regex filters for)NAMES,)COPY,)FNS and)VARS system command arguments.



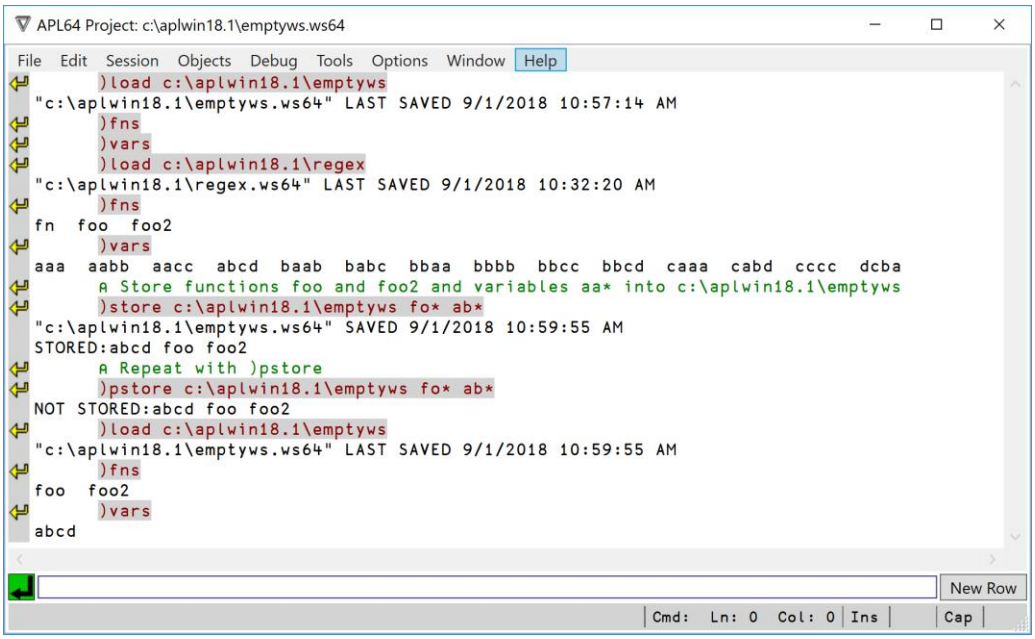
Peek into Saved Workspaces for Variables, Functions and Object Names

Peek into saved workspaces for variables, functions and names by adding @wsname as the last argument element



)STORE &)PSTORE System Commands are Inverses of)COPY &)PCOPY

)STORE and)PSTORE commands are inverses of)COPY and)PCOPY. Rather than copying objects from the argument workspace into the active workspace, they store objects from the active workspace into the argument workspace.



Potential APL64 Project Features

The APL64 Project development design significantly facilitates APL2000's implementation of enhancements in the interpreter including new and improved APL primitive functions with new glyphs and new components to improve customer's productivity and ease-of-use.

Microsoft .Net Features

- APL64 Project syntax could be extended to directly access .Net objects using object oriented [OOP] syntax, e.g., obj1.prop1, obj1.method1, including the APLNext Supervisor, the APLNext C# Script Engine and custom .Net assemblies.
- APL64 Project syntax could be extended to support APL libraries, e.g., namespaces and classes, based on the ':DEF' local function mechanism which has been implemented in the APL64 Project.
- The APL64 Project could be extended to support additional .Net data types such as decimal, uint, etc.
- The APL64 Project could be extended to export Microsoft Common Intermediate Language (MSIL) so that APL64 Project .Net assemblies would be interoperable with other .Net languages.
- The APL64 Project could be extended to optionally save workspaces in xml-format source code for use in contemporary and powerful source code control tools like Git**.** GitHub is a commercial implementation of Git.
- APL Variable Editor
  - The APL variable editor ")EDIT" could be extended to browse an APL component file.
  - The APL variable editor ")EDIT" could be configured for use in customer-developed applications.

Multi-threaded Primitive Function Execution

The action of □TL could be extended to other APL64 Project primitives such as iota, reshape, take, drop, etc., whereby sections of the resulting arrays would be filled by different threads.

APL Component Files

APL component files will be enhanced to store Unicode text, with an appropriate indication that such files would not be compatible with APL+Win.

APL 'Programmer Session'

- The APL 'programmer session' can be configured, with appropriate security considerations, to run on a local workstation to access an APL64 Project interpreter running on a remote workstation.
- Code completion features, e.g., 'intellisense', can be implemented.

Improved User Command Support

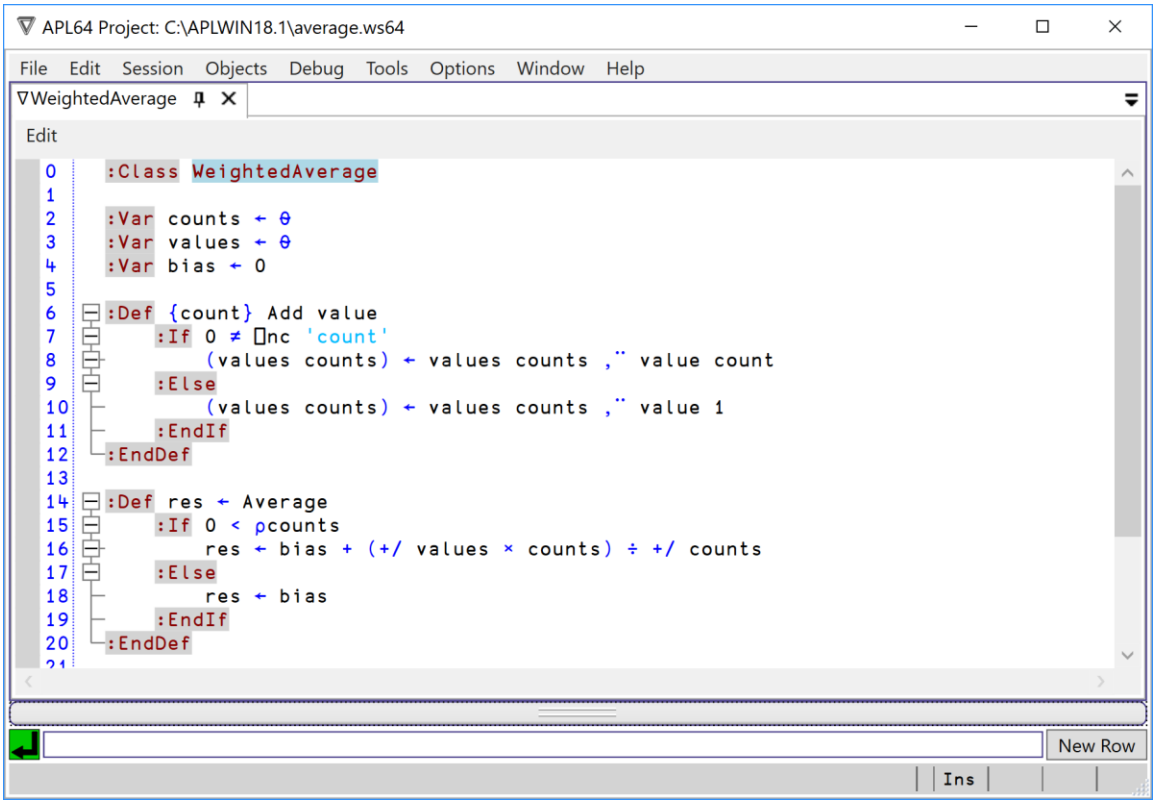
The APL64 Project could be extended to support reflective developer tools, so that with appropriate security considerations, external APL64 Project code running on a separate thread could interact with the current active workspace to eliminate limitations of APL+Win user commands such as object shadowing and intertwined execution states.

Object-Oriented Features

New control structures for declaring user-defined classes, properties, enumerations, etc., will be implemented. The specifications for this feature are being reviewed, so the examples below provide a general indication of the implementation direction of object-oriented control structures in the APL64 Project environment.

The WeightedAverage class is defined and an instance of it is created. First three values are added (each equally weighted by a factor of 1) and the 'Average' is queried. Next a new value with 10,000 times weighting is added, and the 'Average' is queried again:

Class definition:



Add values: 1 10 15 to the c.values property of the instance of the WeightedAverage class, c. For each of these values the corresponding c.counts property elements are default to 1. The class instance variables (counts, values, and bias) are maintained separately for each instance of the WeightedAverage class. After using the c.Average method, another value, 10,000 x 100 is added to the c.values and c.count properties and the c.Average method is again used: