DYALOG

The tool of thought for expert programming

Dyalog™ for Windows

User Guide

Version: 14.0

Dyalog Limited

email: support@dyalog.com

http://www.dyalog.com

Version: 14.0

Revision: 20150302

Contents

Chapter 1:

The APL Environment

Introduction

The Dyalog APL Development Environment includes a Session Manager, an Editor, and a Tracer all of which operate in windows on the screen. The session window is created when you start APL and is present until you terminate your APL session. In addition there may be a number of edit and/or trace Windows, which are created and destroyed dynamically as required. All APL windows are under the control of Windows and may be selected, moved, resized, maximised and minimised using the standard facilities that Windows provides.

APL Keyboards

The Classic and Unicode Editions of Dyalog APL for Windows use different techniques for mapping keystrokes to APL characters and to special command shortcuts.

The Classic Edition uses a proprietary technique for these mappings. The Unicode Edition uses Microsoft's IME (Input Method Editor) technology. Many other applications use the same technology, which means that the Dyalog Unicode IME may be used not only with *Dyalog APL for Windows Unicode Edition*, but also with word processing applications, spreadsheets, terminal emulators etc. Therefore with the Dyalog Unicode IME installed, and with a suitable font selected, APL characters can be entered and viewed in many other applications.

In both Classic and Unicode Editions APL characters are generated when the user presses certain combinations of *meta keys* in conjunction with the normal character keys. Meta keys include Shift, Ctrl and Alt.

For both input techniques it is possible to alter the mapping of keystrokes to APL characters, and to add support for new languages. It is also possible to alter the keystrokes which define special command keyboard shortcuts. For further details, see *Unicode Edition Keyboard* on page 6 or *Classic Edition Keyboard* on page 11

Unicode Edition and the Dyalog Unicode IME

The Dyalog Unicode IME is the input mechanism for generating APL characters for Unicode editions of Dyalog APL. The version of the IME supplied with version 14.0 can be used with version 12.1 and later, provided that they are patched to a version created on or after 1st April 2011.

The Dyalog Unicode IME defines the mapping of keystrokes to Unicode characters. Only keystrokes which resolve to characters that either do not appear on the standard keyboard or which differ from those that appear on the standard keyboard are included in the selectable translate table. In effect the Dyalog Unicode IME can be regarded as an overlay of the standard keyboard which contains just APL characters.

The Dyalog Unicode IME supplied with Version 14.0 includes support for Belgian, Danish, Finnish, French, German, Italian, Spanish Swedish and British and American English keyboards, based on the Dyalog hardware keyboard layout; these keyboard layouts are described at http://dfns.dyalog.com/n_keyboards.htm. Note that for Danish, British and American English keyboards the older layouts, based on the Dyalog APL Ctrl Keyboard, are included in the UnicodeIME\aplkeys directory.

The default keyboard mapping for unsupported languages is American English.

The IME translate tables include mappings for the special command keystrokes used by Dyalog APL.

These command keystroke mappings are ignored by applications unless the application is explicitly named in the Dyalog Unicode IME configuration. It is expected that only terminal emulators used for running UNIX-based versions of Dyalog APL will use this feature.

In particular, Dyalog APL for Windows Unicode Edition does not use the mappings in the translate tables; the mappings are defined under Options/Configure/Keyboard Shortcuts (see *Installation & Configuration Guide: Configuration Dialog: Keyboard Shortcut Tab*).

Note that the Dyalog Unicode IME replaces any previous IME, as well as the Dyalog Ctrl and Dyalog AltGr keyboards.

Classic Edition

The mapping for each of the ⎕AV positions and its associated keystroke is defined by a selectable translate table. ⎕AV includes all the APL symbols used by Dyalog APL as well as all the (non-APL) characters which appear on a standard keyboard. This mapping only works with Classic Edition.

The Classic Edition installation also includes the Dyalog Unicode IME (described below) so that users may enter APL characters into other applications; the Dyalog Unicode IME is however not used by the Classic Edition itself.

The Classic Edition includes support for Danish, Finnish, French, German, Italian, Swedish, and both British and American English keyboards. The default keyboard mapping for unsupported languages is American English.

Session Manager

The Dyalog APL/W session is fully configurable. Not only can you change the appearance of the menus, tool bars and status bars, but you can add new objects of your choice and attach your own APL functions and expressions to them. Functions and variables can be stored in the session *namespace*. This is *independent* of the active workspace; so there is no conflict with workspace names, and your utilities remain permanently accessible for the duration of the session. Finally, you may set up different session configurations for different purposes which can be saved and loaded as required.

The session window is defined by an object called `⎕SE`. This is very similar to a Form object, but has certain special properties. The menu bar, tool bar and status bars on the session window are in fact MenuBar, ToolControl and StatusBar objects owned by `⎕SE`. All of the other components such as menu items and tool buttons are also standard GUI objects. You may use `⎕WC` to create new session objects and you may use `⎕WS` to change the properties of existing ones. `⎕WG` and `⎕WN` may also be used with `⎕SE` and its children.

Components of the session that perform actions (MenuItem and Button objects) do so because their Event properties are defined to execute system operations or APL expressions. System operations comprise a pre-defined set of actions that can be performed by Dyalog APL/W. These are coded as keywords within square brackets. For example, the system operation `[WSClear]` produces a `clear ws`, after first displaying a dialog box for confirmation. You may customise your session by adding or deleting objects and by attaching system operations or APL expressions to them.

Like any other object, `⎕SE` is a namespace that may contain functions and variables. Furthermore, `⎕SE` is independent of the active workspace and is unaffected by `)LOAD` and `)CLEAR`. It is therefore sensible to store commonly used utilities, particularly those utilities that are invoked by events on session objects, in `⎕SE` itself, rather than in each of your application workspaces.

The possibility of configuring your APL session so extensively leads to the requirement to have different sessions for different purposes. To meet this need, sessions are stored in special files with a .DSE (Dyalog Session) extension. The default session (i.e. the one loaded when you start APL) is specified by the **session_file** parameter. You may customise this session and then save it over the default one or in a separate file. You can load a new session from file at any stage without affecting your active workspace.

Positioning the Cursor

The cursor may be positioned within the current APL window by moving the mouse pointer to the desired location and then clicking the Left Button. The APL cursor will then move to the character under the pointer.

Selection

Dragging the mouse selects the text from the point where the mouse button is depressed to the point where the button is released. When you select multiple lines, the use of the left mouse button always selects text from the start of the line. A contiguous block of text can be selected by dragging with the right mouse button.

Double-clicking the left mouse button to the left of a line selects the whole line, including the end-of-line character.

Scrolling

Data can be scrolled in a window using the mouse in conjunction with the scrollbar.

Invoking the Editor

The Editor can be invoked by placing the mouse pointer over the name of an editable object and double-clicking the left button on the mouse. If you double-click on the empty Input Line it acts as "Naked Edit" and opens an edit window for the suspended function (if any) on the APL stack. For further details, see *Invoking the Editor* on page 114. See also "Installation and Configuration Guide: DoubleClickEdit".

The Current Object

If you position the input cursor over the name of an object in the session window, that object becomes the current object. This name is stored in the CurObj property of the Session object and may be used by an application or a utility program. This means that you can click the mouse over a name and then select a menu item or click a button that executes code that accesses the name.

The Session Pop-up Menu

Clicking the right mouse button brings up the Session pop-up menu. This is described later in this chapter.

Drag-and-Drop Editing

Drag-and-Drop editing is the easiest way to move or copy a selection a short distance within an edit window or between edit windows.

To *move* text using drag-and-drop editing:

1. Select the text you want to move.
2. Point to the selected text and then press and hold down the left mouse button. When the drag-and-drop pointer appears, drag the cursor to a new location.
3. Release the mouse button to drop the text into place.

To *copy* text using drag-and-drop editing:

1. Select the text you want to move.
2. Hold down the Ctrl key, point to the selected text and then press and hold down the left mouse button. When the drag-and-drop pointer appears, drag the cursor to a new location.
3. Release the mouse button to drop the text into place.

If you drag-and-drop text within the Session window, the text is copied and not moved whether or not you use the Ctrl key.

Interrupts

To generate an interrupt, click on the Dyalog APL icon in the Windows System Tray; then choose *Weak Interrupt* or *Strong Interrupt*. To close the menu, click *Cancel*. Alternatively, to generate a weak interrupt, press Ctrl+Break, or select *Interrupt* from the *Action* menu on the Session Window.

Unicode Edition Keyboard

Introduction

Unicode Edition supports the use of standard Windows keyboards that have the additional capability to generate APL characters when the user presses Ctrl, Alt, AltGr (or some other combination of meta keys) in combination with the normal character keys.

Dyalog APL is supplied with the Dyalog Unicode IME keyboard for a range of different languages as listed below. The intention is that only APL characters and characters that appear in locations different from the underlying keyboard are defined; any other keystroke is passed through *as is*.

Installation

During the Installation of Dyalog APL Unicode Edition, setup installs the Dyalog Unicode IME (IME). For any given Input Language the IME consists of an additional service for that Input Language, and a translate table which maps keystrokes for the appropriate keyboard to Unicode code points for APL characters

An IME service is installed for every Input Language that the user who installs Dyalog APL has defined, as well as every Input Language for which Dyalog has support.

The keyboard mappings are defined for the following national languages: Belgian, Danish, Finnish, French, German, Italian, Spanish, Swedish and British and American English

These mappings are described at http://dfns.dyalog.com/n_keyboards.htm.

For any other Input Language the American English translate table is selected. Note that some Input Languages are defined to be *substitutes* for other Input Languages; for example Australian English Input is a substitute for American English Input, Austrian German Input a substitute for German German Input. In these cases the IME will install the appropriate translate table. It is also possible to create support for new languages or to modify the existing support. See the *IME User Guide* for further details.

Configuring the Dyalog APL IME

The following description uses screenshots taken from a Windows 7 PC with three Input Languages configured for the current user: English (United Kingdom) - the default Input Language, Danish (Denmark) and English (United States).

The Dyalog Unicode IME is added as an additional service to all keyboards defined to the user and the administrator at the time that the IME was installed.

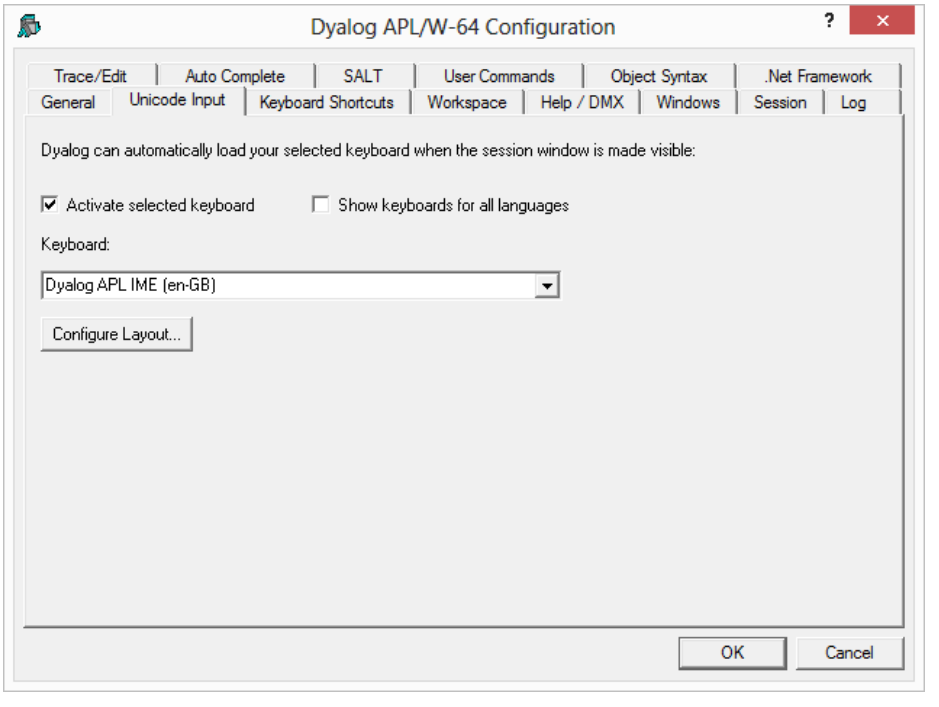
For each IME the underlying keyboard layout file will be the same as that defined for the base keyboard. The layout file is a DLL created by Microsoft.

The language specified in the description of the IME is the name of the IME translate table that has been associated with the IME for the specific keyboard. In the case of languages not supported by the IME the keyboard will default to en-US. With the IME as supplied with Version 13.2 altering this text requires editing the appropriate Registry value.

The IME may be configured from within APL or from Windows.

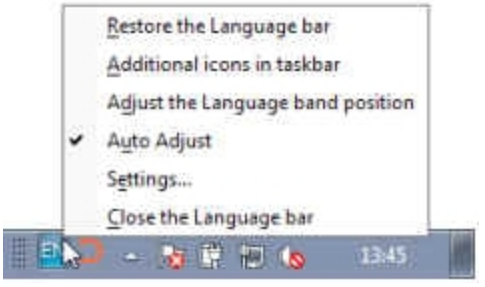
From within Dyalog APL

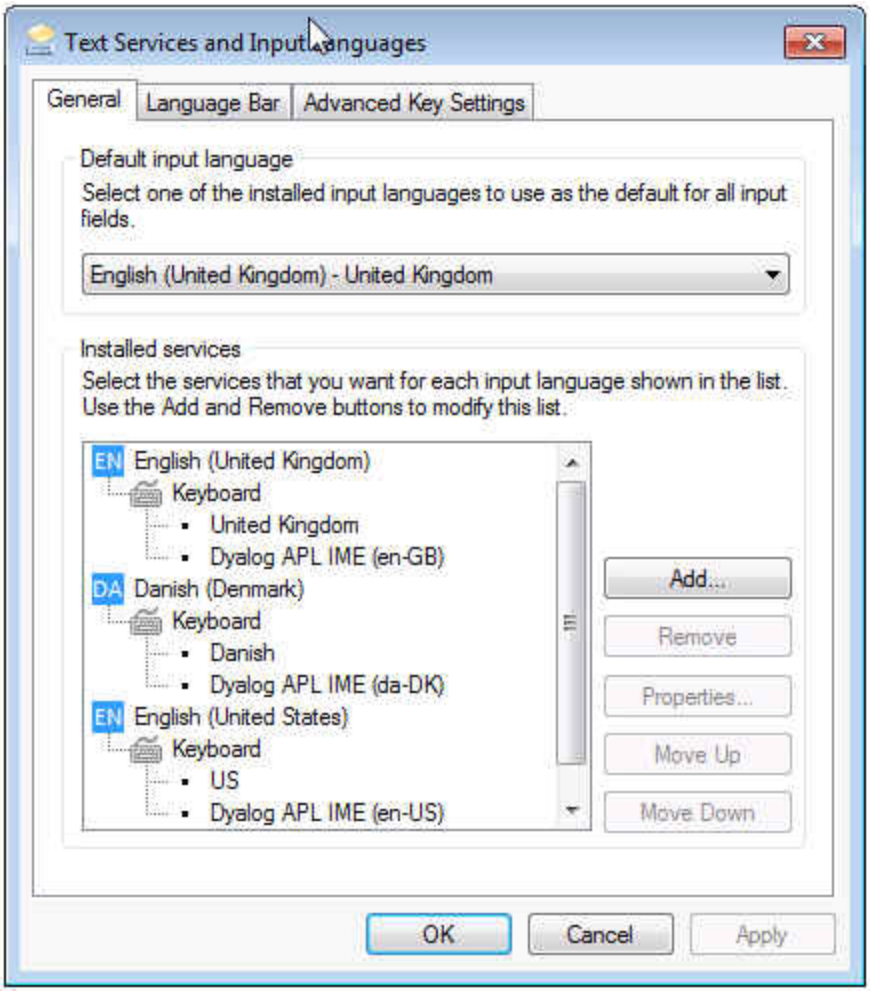
To change the properties of the IME go to *Options/Configure/Unicode Input* tab and select *Configure Layout*:



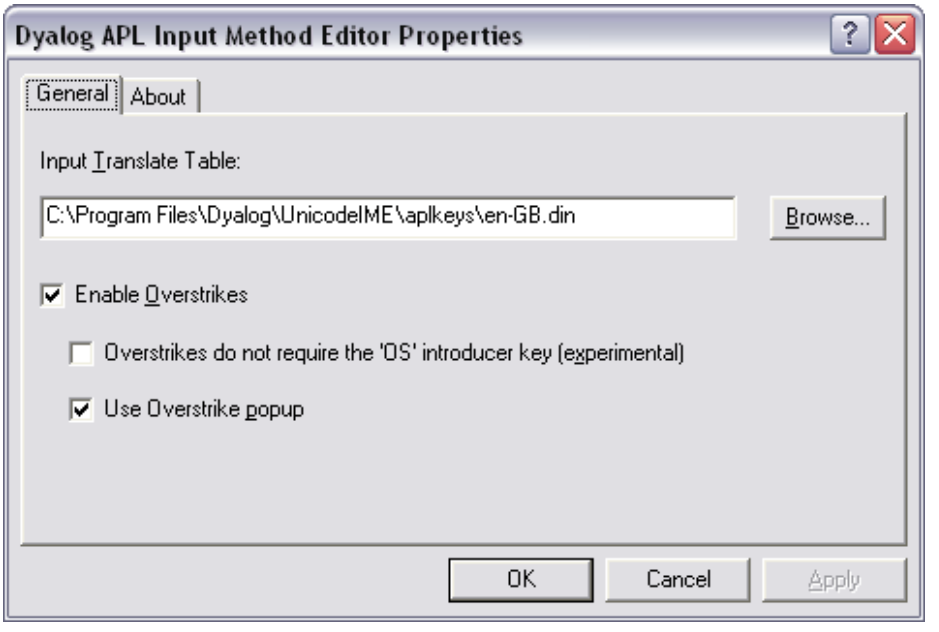
From Windows

Right click on either the Input Language icon or the Keyboard layout icon in the TaskBar and select *Settings*:





To alter the configuration of any of the installed IMEs, select that IME and click on *Properties*:



Input translate table:

The translate table defines the mapping between APL characters and the keystrokes that generate those APL characters. It is possible to alter the mapping or to create support for new keyboards by altering the translate table, or by selecting a different translate table. See the *IME User Guide* for more details.

Overstrikes:

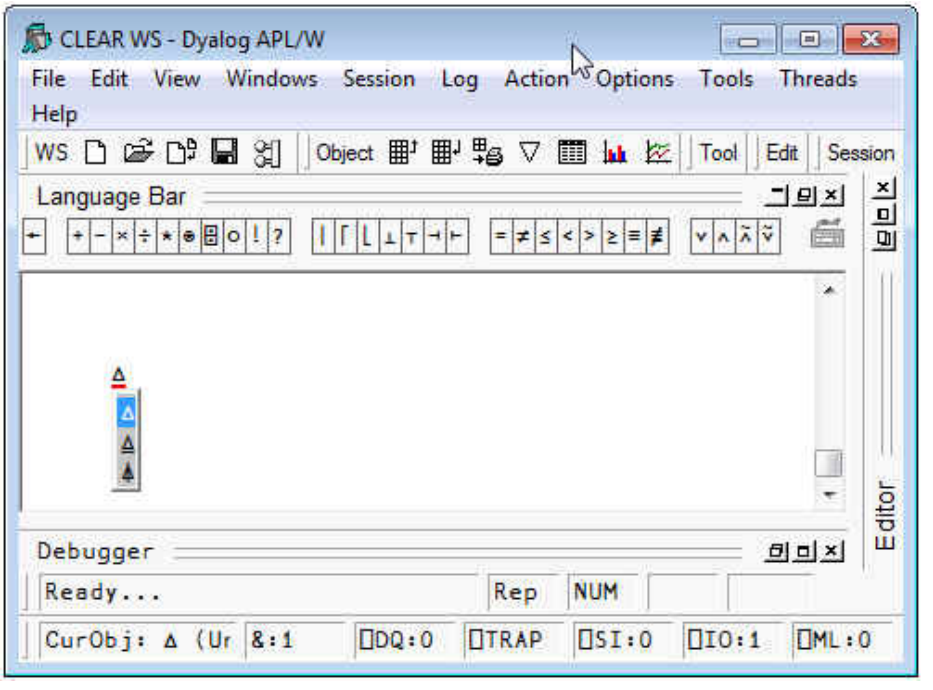
In the original implementations of APL, many of the special symbols could only be generated by overstriking one character on top of another as is reflected in the appearance of the glyphs. For example, the symbol for Grade Up (⍋) is actually the symbol for delta (⍙) superimposed on the symbol for vertical bar (|)

In Dyalog APL such symbols can be generated either by a single keystroke, or (in *Replace* mode) by overtyping one symbol with another. For example ⍋ may be generated using Shift+Ctrl+4, or by switching to *Replace* mode and typing the three keystrokes Ctrl+h, Left-Cursor, Ctrl+m.

Using the Dyalog Unicode IME the character can also be entered by pressing Ctrl+Bksp, Ctrl+m, Ctrl+h. Note that Ctrl+Bksp is the default *Overstrike Introducer Key* (key code OS).

Use Overstrike popup:

With this option selected, when the character following the Overstrike Introducer Key is pressed, a popup box displays all the overstrikes which contain the last character typed: in the example below Ctrl+Bksp has been followed by Ctrl+h:



Note the fine (red) line under the Δ in the Session window. This indicates that an overstrike creation operation is in progress.

The input of the symbol ⍙ can be completed by pressing Ctr+m, or by moving the selection up and down the pop-up list using Cursor-Up or Cursor-Down.

Overstrikes do not require the OS introducer key (experimental):

With this option selected, the IME identifies characters which are part of a valid overstrike, and when such a character is entered into the session, begins an overstrike creation operation. This mode is experimental in the IME supplied with Version 13.2.

Classic Edition Keyboard

The standard Version 13.2 Classic Edition keyboard tables are files supplied in the aplkeys sub-directory named cc.din where cc is the standard 2-character country code, e.g. `uk.din`.

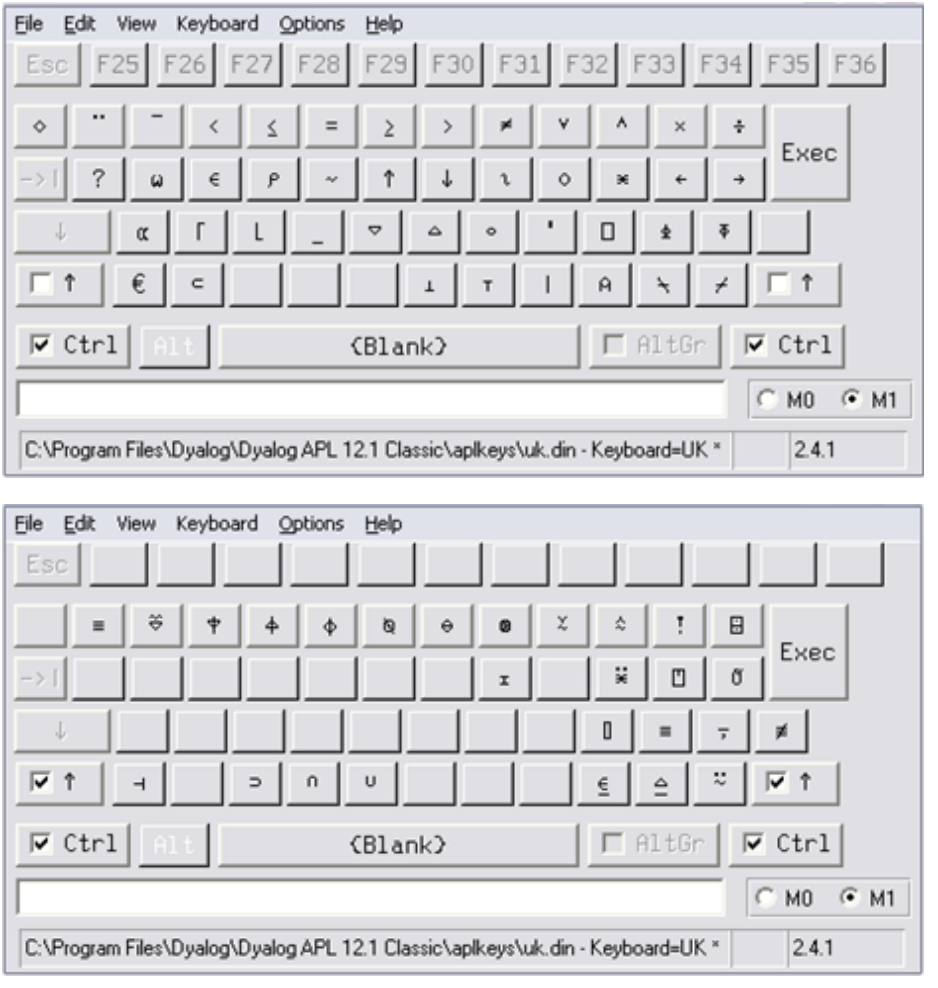
Note that the standard tables do not support the entry of APL underscored characters ΔABCDEFGHIJKLMNOPQRSTUVWXYZ.

The standard table supports two modes of use; traditional (mode 0) and unified (mode 1). The keyboard starts in mode 1 and may be switched between modes by clicking the *Uni/Apl* field in the status bar or by keying * on the Numeric-Keypad.

Unified Layout

The following picture illustrates the standard UK keyboard Unified layout.

APL symbols are entered using the Ctrl and Ctrl+Shift keys as illustrated below.



Traditional Layout

The following picture illustrates the standard UK keyboard Traditional layout.

APL symbols are entered using the Shift and Ctrl+Shift keys as illustrated below.

Line-Drawing Symbols

Classic Edition includes 12 single-line graphics characters for drawing lines and boxes. Line-drawing characters are entered using the keys on the numeric keypad in conjunction with the Ctrl key as shown below. Num Lock must be on.

Normal		
7	8	9
4	5	6
1	2	3
0		.

Ctrl		
┌	┬	┐
├	┼	┤
└	┴	┘
	│	─

Note:to accommodate other characters, line-drawing symbols are located in the non-printable area of the font layout. Although these characters can normally be used in output to the session (function: `DISP` in the UTIL workspace uses them), **many printer drivers and some display drivers will not display characters from these positions in the font.**

Keyboard Shortcuts

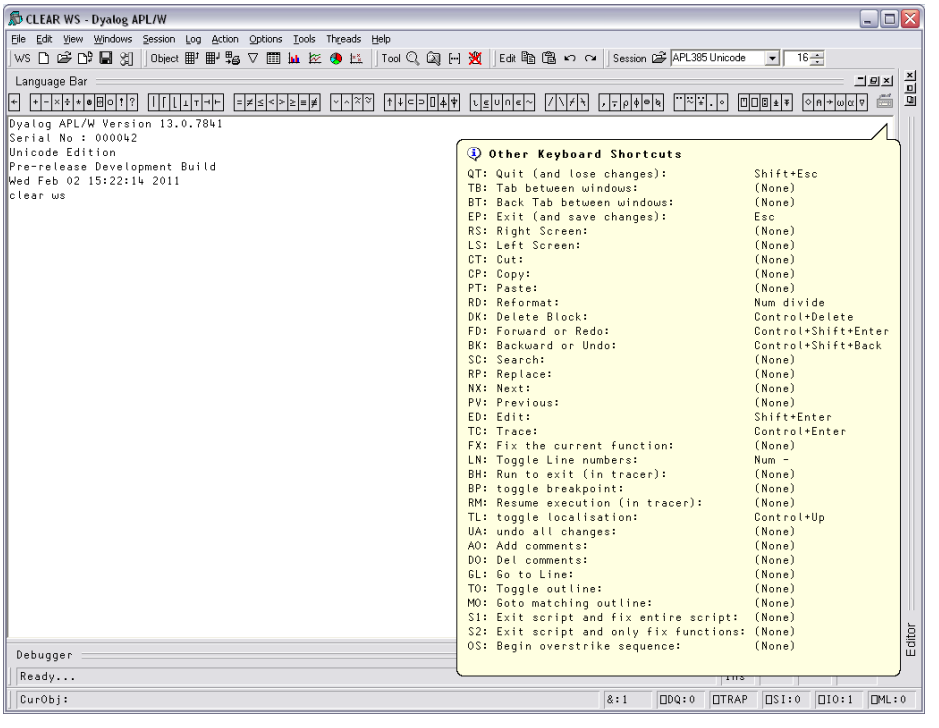
The terms keyboard shortcut (Unicode Edition) and command (Classic Edition) are used herein to describe a keystroke that generates an action, rather than one that produces a symbol.

Unicode Edition

Unicode Edition provides a number of shortcut keys that may be used to perform actions. For compatibility with Classic Edition and with previous Versions of Dyalog APL, these are identified by 2-character codes; for example the action to start the Tracer is identified by the code <TC>, and mapped to user-configurable keystrokes.

In the Unicode Edition, Keyboard Shortcuts are defined using Options/Configure/Keyboard Shortcuts and stored in the Windows Registry. Note that the Unicode IME translate tables have definitions for the Keyboard Shortcuts too; these are ignored by the interpreter, and are intended for use with terminal emulators being used in conjunction with non-GUI versions of Dyalog APL.

To the right of the last symbol in the Language Bar is the Keyboard Shortcut icon  . If you hover the mouse over this icon, a pop-up tip is displayed to remind you of your keyboard shortcuts as illustrated below.



Classic Edition

Commands fall into four categories, namely cursor movement, selection, editing directives and special operations, and are summarised in the following tables. The input codes in the first column of the tables are the codes by which the commands are identified in the Input Translate Table.

Table 1: Cursor Movement Commands

Input Code	Keystroke	Description
LS	Ctrl+PgUp	Scrolls left by a page
RS	Ctrl+PgDn	Scrolls right by a page
US	PgUp	Scrolls up by a page
DS	PgDn	Scrolls down by a page
LC	Left Arrow	Moves the cursor one character position to the left
RC	Right Arrow	Moves the cursor one character position to the right
DC	Down Arrow	Moves the cursor to the current character position on the line below the current line
UC	Up Arrow	Moves the cursor to the current character position on the line above the current line
UL	Ctrl+Home	Move the cursor to the top-left position in the window
DL	Ctrl+End	Moves the cursor to the bottom-right position in the window
RL	End	Moves the cursor to the end of the current line
LL	Home	Moves the cursor to the beginning of the current line
LW	Ctrl+Left Arrow	Moves the cursor to the beginning of the word to the left of the cursor
RW	Ctrl+Right Arrow	Moves the cursor to the end of the word to the right of the cursor
TB	Ctrl+Tab	Switches to the next session/edit/trace window
BT	Ctrl+Shift+Tab	Switches to the previous session/edit/trace window

Table 2: Selection Commands

Input Code	Keystroke	Description
Lc	Shift+Left Arrow	Extends the selection one character position to the left
Rc	Shift+Right Arrow	Extends the selection one character position to the right
Lw	Ctrl+Shift+Left Arrow	Extends the selection to the beginning of the word to the left of the cursor
Rw	Ctrl+Shift+Right Arrow	Extends the selection to the end of the word to the right of the cursor
Uc	Shift+Up Arrow	Extends the selection to the current character position on the line above the current line
Dc	Shift+Down Arrow	Extends the selection to the current character position on the line below the current line
Ll	Shift+Home	Extends the selection to the beginning of the current line
Rl	Shift+End	Extends the selection to the end of the current line
Ul	Ctrl+Shift+Home	Extends the selection to the beginning of the first line in the window
Dl	Ctrl+Shift+End	Extends the selection to the end of the last line in the window
Us	Shift+PgUp	Extends the selection up by a page.
Ds	Shift+PgDn	Extends the selection down by a page

Table 3: Editing Directives

Input Code	Keystroke	Description
DI	Delete	Deletes the selection
DK	Ctrl+Delete	Deletes the current line in an Edit window. Deletes selected lines in the Session Log.
CT	Shift+Delete	Removes the selection and copies it to the clipboard
CP	Ctrl+Insert	Copies the selection into the clipboard
FD	Ctrl+Shift+Enter	Reapplies the most recent undo operation
BK	Ctrl+Shift+Bksp	Performs an undo operation
PT	Shift+Insert	Copies the contents of the clipboard into a window at the location selected
OP	Ctrl+Shift+Insert	Inserts a blank line immediately after the current one (editor only)
HT	Tab	Indents text
TH	Shift+Tab	Removes indentation
RD	Keypad-slash	Reformats a function (editor only)
TL	Ctrl+Alt+L	Toggles localisation of the current name
GL	Ctrl+Alt+G	Go to [line]
AO	Ctrl+Alt+,	Add Comments
DO	Ctrl+Alt+.	Delete Comments
AC		Align Comments

Table 4: Special Operations

Input Code	Keystroke	Description
IN	Insert	Insert on/off
LN	Keypad-minus	Line numbers on/off
ER	Enter	Execute
ED	Shift+Enter	Edit
TC	Ctrl+Enter	Trace
EP	Esc	Exit
QT	Shift+Esc	Quit

The Session Colour Scheme

Within the Development Environment, different colours are used to identify different types of information. These colours are normally defined by registry entries and may be changed using the Colour Configuration dialog box as described later in this chapter. In the Classic Edition, colours may alternatively be defined in the Output Translate Table (normally WIN.DOT). This table recognises up to 256 foreground and 256 background colours which are referenced by colour indices 0-255. These colour indices are mapped to physical colours in terms of their Red, Green and Blue intensities (also 0-255). Foreground and background colours are specified independently as Cnnn or Bnnn. For example, the following entry in the Output Translate Table defines colour 250 to be red on magenta.

```
C250: 255 0 0   + Red foreground
B250: 255 0 255 + Magenta background
```

The first table below shows the colours used for different session components. The second table shows how different colours are used to identify different types of data in edit windows.

Table 5: Default Colour Scheme - Session

Colour	Used for	Default
249	Input and marked lines	Red on White
250	Session log	Black on White
252	Tracer : Suspended Function	Yellow on Black
253	Tracer : Pendent Function	Yellow on Dark Grey
245	Tracer : Current Line	White on Red

Table 6: Default Colour Scheme Edit windows

Colour	Array Type	Editable	Default
236	Simple character matrix	Yes	Green on Black
239	Simple numeric	No	White on Dk Grey
241	Simple mixed	No	Cyan on Dk Grey
242	Character vector of vectors	Yes	Cyan on Black
243	Nested array	No	Cyan on Dk Grey
245	⎕OR object	No	White on Red
248	Function or Operator	No	White on Dk Cyan
254	Function or Operator	Yes	White on Blue

Syntax Colouring in the Session

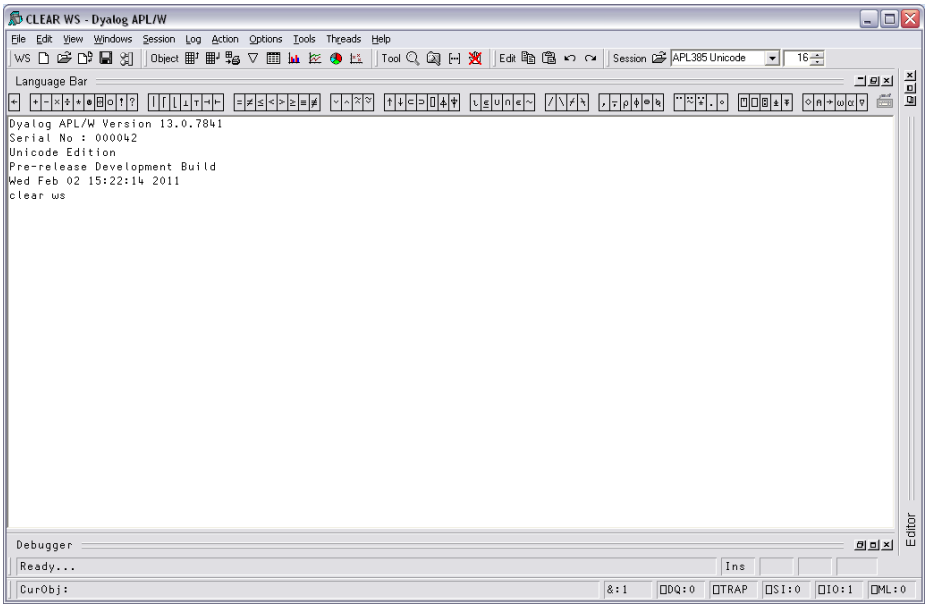
As an adjunct to the overall Session Colour Scheme, you may choose to apply a *syntax colouring scheme* to the current Session Input line(s). You may also extend syntax colouring to previously entered input lines, although this only applies to input lines in the current session; syntax colouring information is not remembered in the Session Log.

Syntax colouring may be used to highlight the context of names and other elements when the line was entered. For example, you can identify global names and local names by allocating them different colours.

See *Installation & Configuration Guide: Colour Selection Dialog* for further details.

The Session Window

The primary purpose of the session window is to provide a scrolling area within which you may enter APL expressions and view results. This area is described as the *session log*. Normally, the session window will have a menu bar at the top with a tool bar below it. At the bottom of the session window is a status bar. However, these components of the session may be extensively customised and, although this chapter describes a typical session layout, your own session may look distinctly different. A typical Session is illustrated below.



A typical Session window

Window Management

When you start APL, the session is loaded from the file specified by the **session_file** parameter. The position and size of the session window are defined by the Posn and Size properties of the Session object `⎕SE`, which will be as they were when the session file was last saved.

The name of the active workspace is shown in the title bar of the window, and changes if you rename the workspace or `)LOAD` another.

You can move, resize, minimise or maximise the Session Window using the standard Windows facilities.

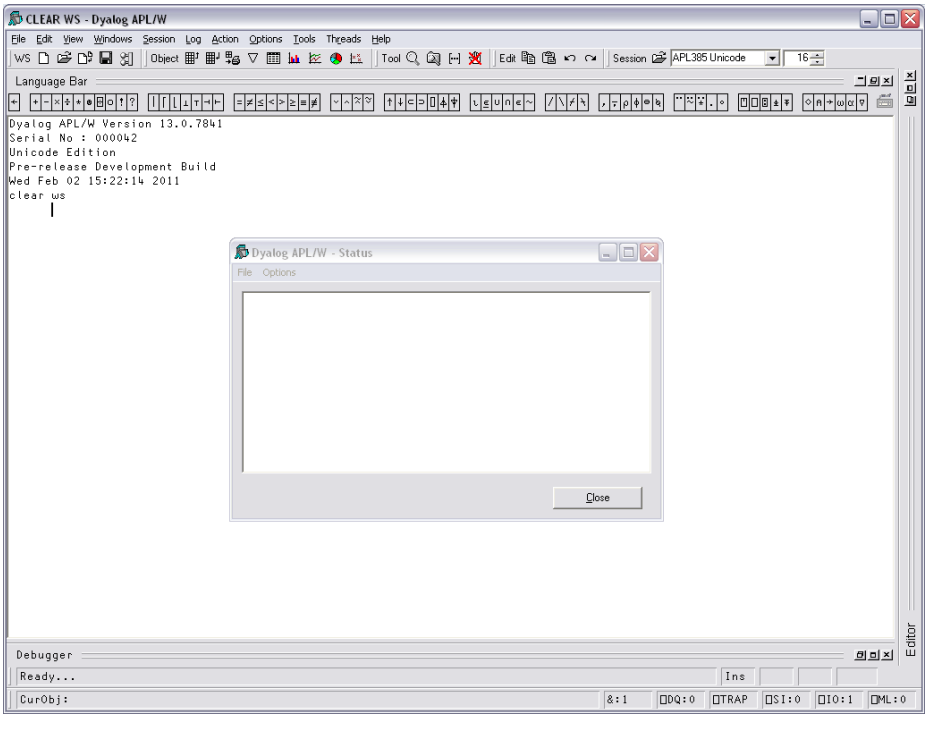
In addition to the Session Window itself, there are various subsidiary windows which are described later in the Chapter. In general, these subsidiary windows may be docked inside the Session window, or may be stand-alone floating windows. You may dock and undock these windows as required. The standard Session layout illustrated above, contains docked Editor, Tracer and SIStack windows.

Note that the session window is only displayed **when** it is required, i.e. when APL requests input from or output to the session. This means that end-user applications that do not interact with the user through the session, will not have an APL session window.

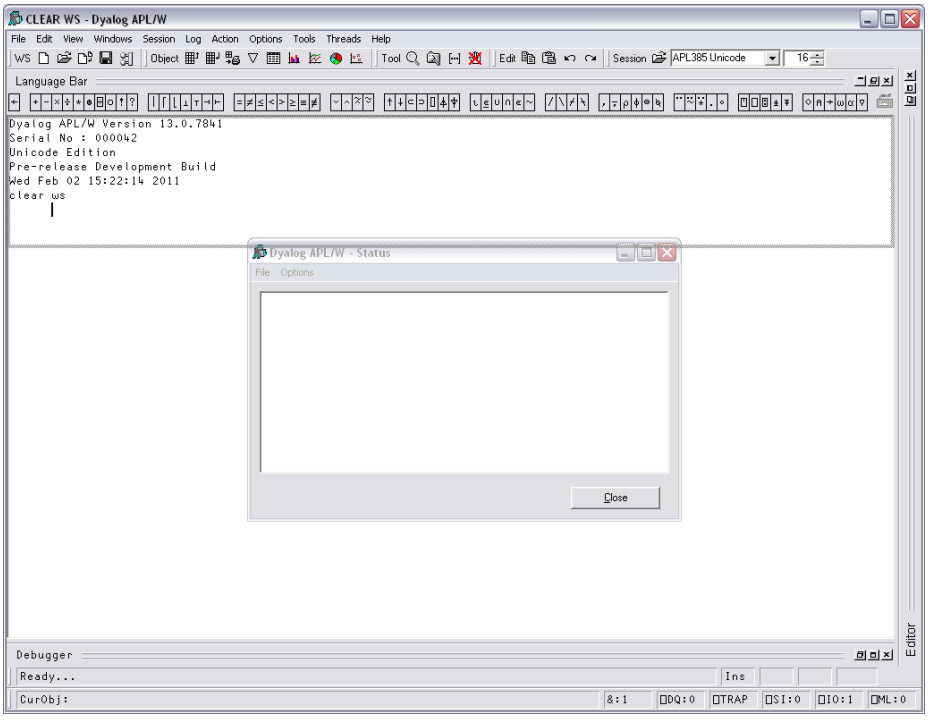
Docking

Nearly all of the windows used in the Dyalog APL IDE may be docked in the Session window or be stand-alone floating windows. When windows are docked in the Session, the Session window is split into resizable panes, separated by splitters. The following example, using the Status window, illustrates the principles involved. (The use of the Status window is described later in this Chapter.)

To start with, the Status window is hidden. You may display it by selecting the *Status* menu item from the *Tools* menu. It initially appears as a floating (undocked) window as shown below.

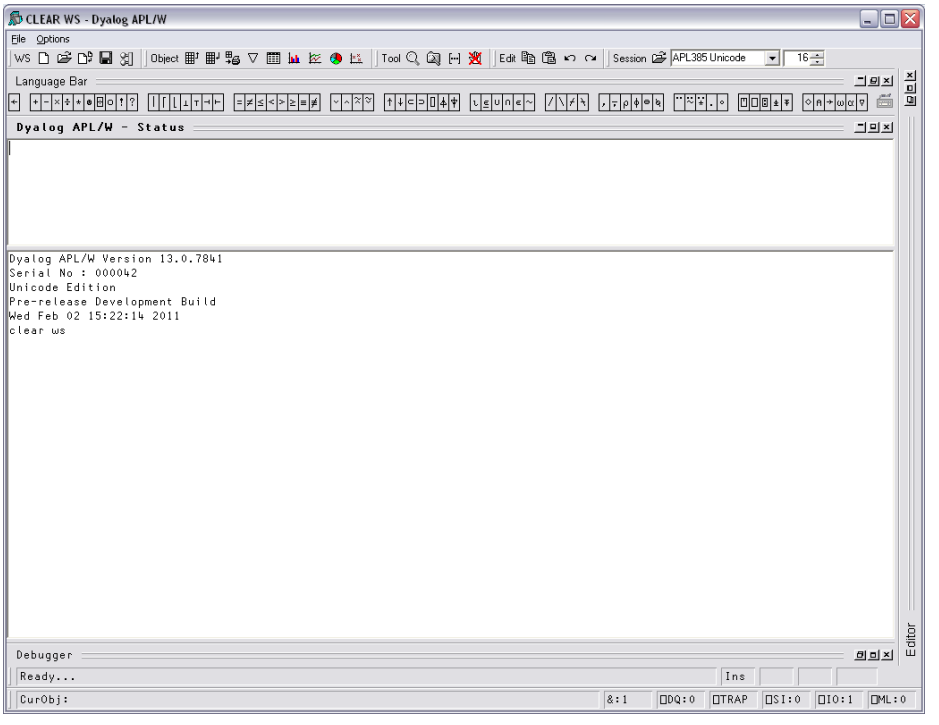


If you press the left mouse button down over the Status window title bar, and drag it, you will find that when the mouse pointer is close to an edge of the Session window, the drag rectangle indicates a docking zone as shown below. This indicates the space that the window will occupy if you now release the mouse button to dock it.

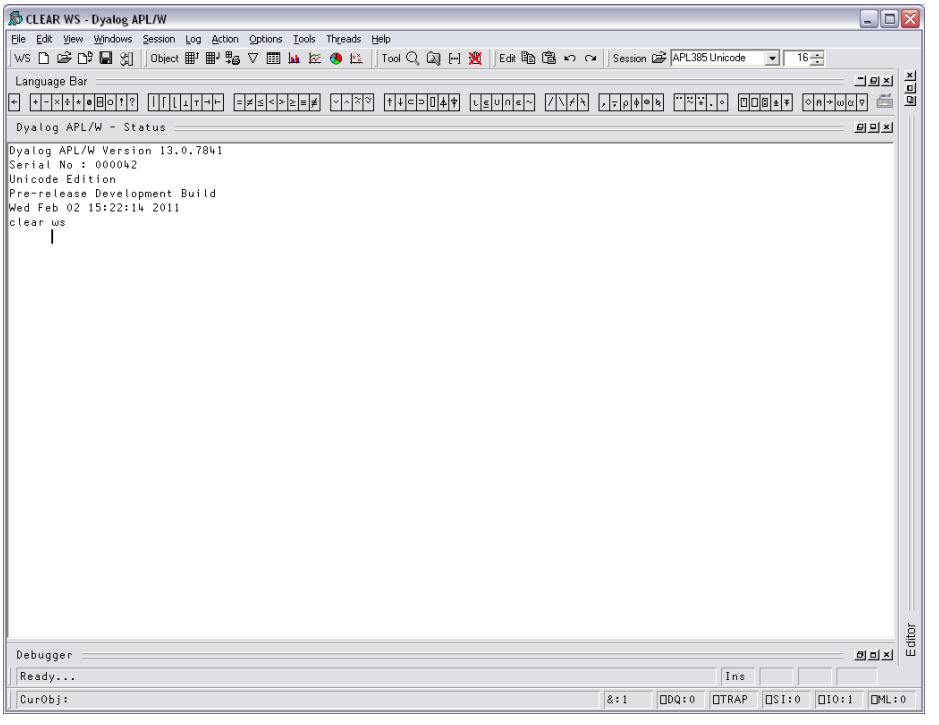


The next picture shows the result of the docking operation. The Session window is now split into 2 panes, with the Status window in the upper pane and the Session log window in the lower pane. You can resize the panes by dragging with the mouse.

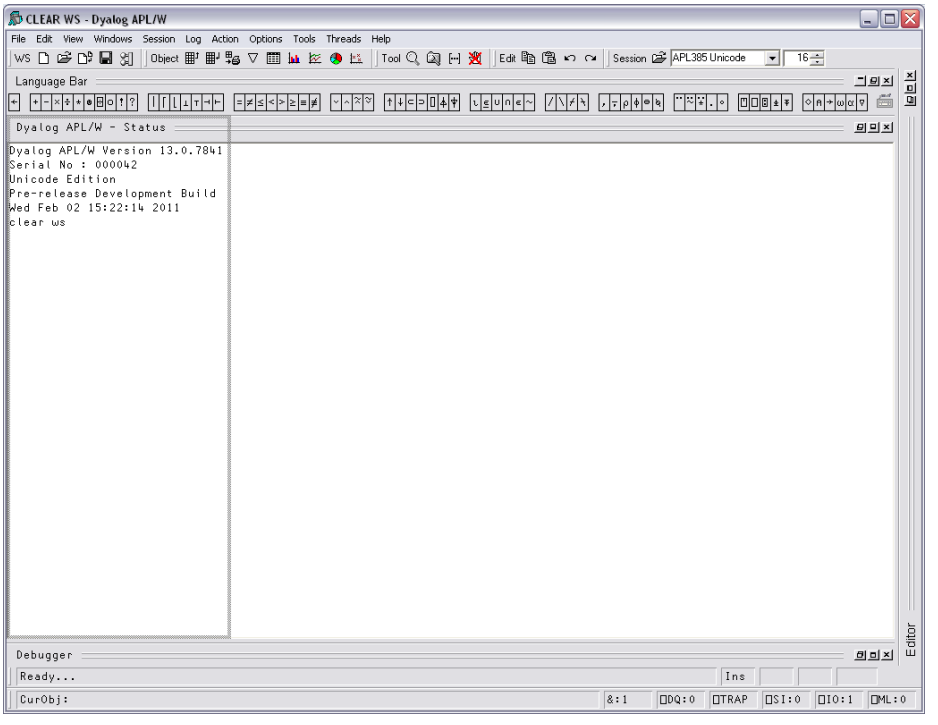
You will notice that a docked window has a title bar (in this case, the caption is *Status*) and 3 buttons which are used to *Minimise*, *Maximise* and *Close* the docked window.



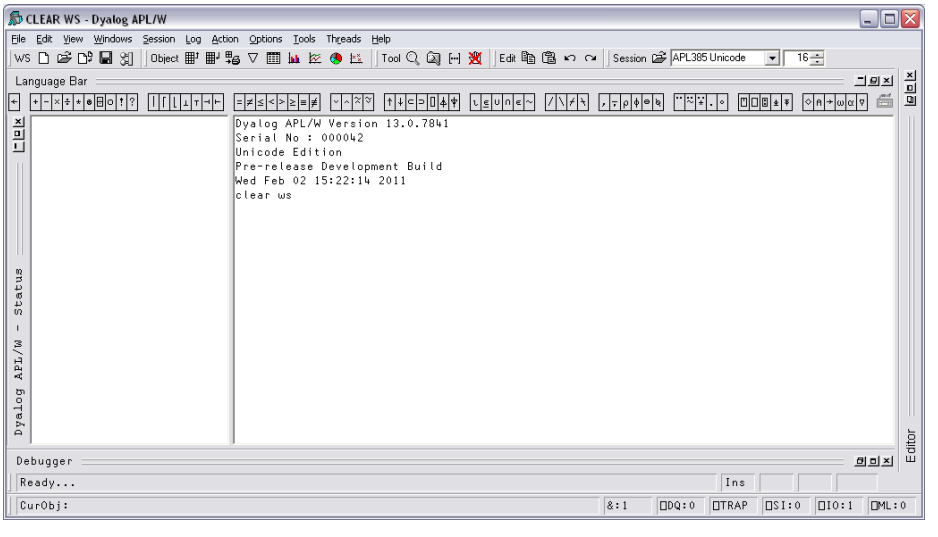
The next picture shows the result of minimising the Status window pane. All that remains of it is its title bar. The Minimise button has changed to a Restore button, which is used to restore the pane to its original size.



You can pick up a docked window and then re-dock it along a different edge of the Session as illustrated below.



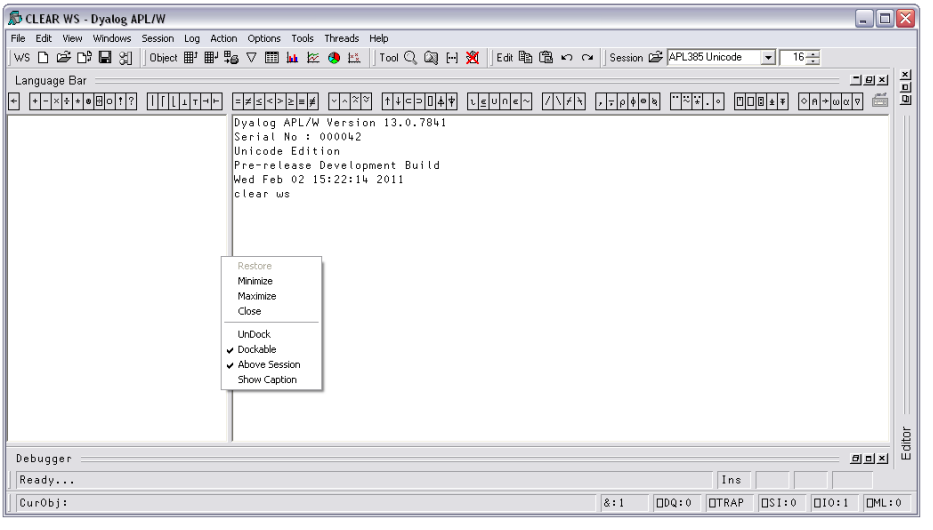
Docking the Status window along the left edge of the Session causes the Session window to be split into two vertical panes. Notice how the title bar is now drawn vertically.



If you click the right mouse button over any window, its context menu is displayed. If the window is dockable, the context menu contains the following options:

Undock	Undocks the docked window. The window is displayed at whatever position and size it occupied prior to being docked.
Hide Caption	Hides the title bar of the docked window,
Dockable	Specifies whether the window is currently dockable or is locked in its current state. You can use this to prevent the window from being docked or undocked accidentally.

The last picture shows the effect of using Hide Caption to remove the title bar. In this state, you can resize the pane with the mouse, but the Minimise, Maximise and Close buttons are not available. However, you can restore the object's title bar using its context menu.



Entering and Executing Expressions

Introduction

The session contains the *input line* and the *session log*. The input line is the last line in the session, and is (normally) the line into which you type an expression to be evaluated.

The session log is a history of previously entered expressions and the results they produced.

If you are using a log file, the Session log is loaded into memory when APL is started from the file specified by the **log_file** parameter. When you close your APL session, the Session log is written back out to the log file, replacing its previous contents.

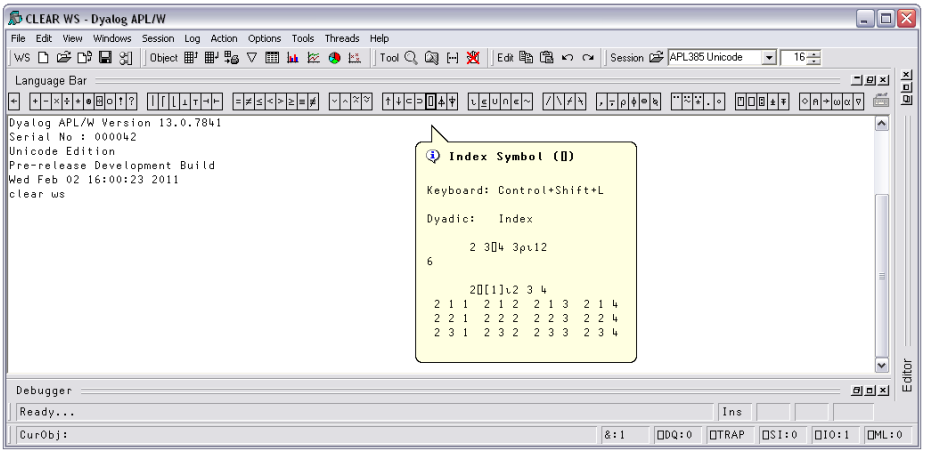
In general you type an expression into the input line, then press Enter (ER) to run it. After execution, the expression and any displayed results become part of the session log.

You can move around in the session using the scrollbar, the cursor keys, and the PgUp and PgDn keys. In addition, Ctrl+Home (UL) moves the cursor to the beginning of the top-line in the Log and Ctrl+End (DL) moves the cursor to the end of the last (i.e. the *current*) line in the session log. Home (LL) and End (RL) move the cursor to the beginning and end respectively of the line containing the cursor.

Language Bar

The Language Bar is an optional window which is initially docked to the Session Window, to make it easy to pick APL symbols without using the keyboard.

If you hover the mouse pointer over a symbol in the APL Language Bar, a pop-up tip is displayed to remind you of its usage. If you click on a symbol in the Language Bar, that symbol is inserted at the cursor in the current line in the Session.



Auto Complete

As you start to enter characters in an APL expression, the *Auto Complete* suggestions pop-up window (AC for short) offers you a choice based upon the characters you have already entered and the current context.

For example, if you enter a ⎕, AC displays a list of all the system functions and variables. If you then enter the character `r`, the list shrinks to those system functions and variables beginning with the letter r, namely ⎕`refs`, ⎕`rl`, and ⎕`rtl`. Instead of entering the remaining characters, you may select the appropriate choice in the AC list. This is done by pressing the right cursor key or (in PocketAPL) by tapping the choice in the list.

If you begin to enter a name, AC will display a list of namespaces, variables, functions, operators that are defined in the current namespace. If you are editing a function, AC will also include names that are localised in the function header.

If the current space is a GUI namespace, the list will also include Properties, Events and Methods exposed by that object.

As an additional refinement, AC remembers a certain number of previous auto complete operations, and uses this information to highlight the most recent choice you made.

For example, suppose that you enter the two characters `)c`. AC offers you `)clear` thru' `)cs`, and you choose `)cs` from the list. The next time you enter the two characters `)c`, AC displays the same list of choices, but this time `)cs` is pre-selected.

You can disable or customise Auto Completion from the *Auto Complete* page in the Configuration dialog box which is described later in this chapter.

Executing an Expression

To execute an expression, you type it into the input line, then press Enter (ER). Alternatively, you can select *Execute* from the *Action* menu. Following execution, the expression and any displayed results become part of the session log.

Instead of entering a new expression in the input line, you can move back through the session log and re-execute a previous expression (or line of a result) by simply pointing at it with the cursor and pressing Enter. Alternatively, you can *select Execute* from the *Action* menu. You may alter the line before executing it. If you do so, it will be displayed using colour 249 (Red on White), the same as that used for the input line. When you press Enter the new line is copied to the input line prior to being executed. The original line is restored and redisplayed in the normal session log colour 250 (Black on White).

An alternative way to retrieve a previously entered expression is to use Ctrl+Shift+Bksp (BK) and Ctrl+Shift+Enter (FD). These commands cycle backwards and forwards through the *input history*, successively copying previously entered expressions over the current line. When you reach the expression you want, simply press Enter to re-run it. These operations may also be performed from the *Edit* menu in the session window.

Executing Several Expressions

You can execute several expressions, by changing more than one line in the session log before pressing Enter. Each line that you change will be displayed using colour 249 (Red on White). When you press Enter, these *marked* lines are copied down and executed in the order they appear in the log.

Note that you don't actually have to *change* a line to mark it for re-execution; you can mark it by overtyping a character with the same character, or by deleting a leading space for instance.

It is also possible to execute a contiguous block of lines. To do this, you must first select the lines (by dragging the mouse or using the keyboard) and then copy them into the clipboard using Shift+Delete (CT) or Ctrl+Insert (CP). You then paste them back into the session using Shift+Insert (PT). Lines pasted into the session are always marked (Red on White) and will therefore be executed when you press Enter. To execute lines from an edit window, you use a similar procedure. First select the lines you want to execute, then cut or copy the selection to the clipboard. Then move to the session window and paste them in, then press Enter to execute them.

Session Print Width (PW)

Throughout its history, APL has used a system variable `⎕PW` to specify the width of the user's terminal or screen. Session output that is longer than `⎕PW` is automatically wrapped and split into multiple lines on the display. This feature of APL was designed in the days of hard-copy terminals and has become less relevant in modern Windows environments.

Dyalog APL continues to support the traditional use of `⎕PW`, but also provides an alternative option to have the system wrap Session output according to the width of the Session Window. This behaviour may be selected by checking the Auto PW checkbox in the Session tab of the Configuration dialog box.

Using Find/Replace in the Session

The search and replace facilities work not just in the Editor as you would expect, but also in the Session. For example, if you have just entered a series of expressions involving a variable called `SALES` and you want to perform the same calculations using `NEWSALES`, the following commands will achieve it:

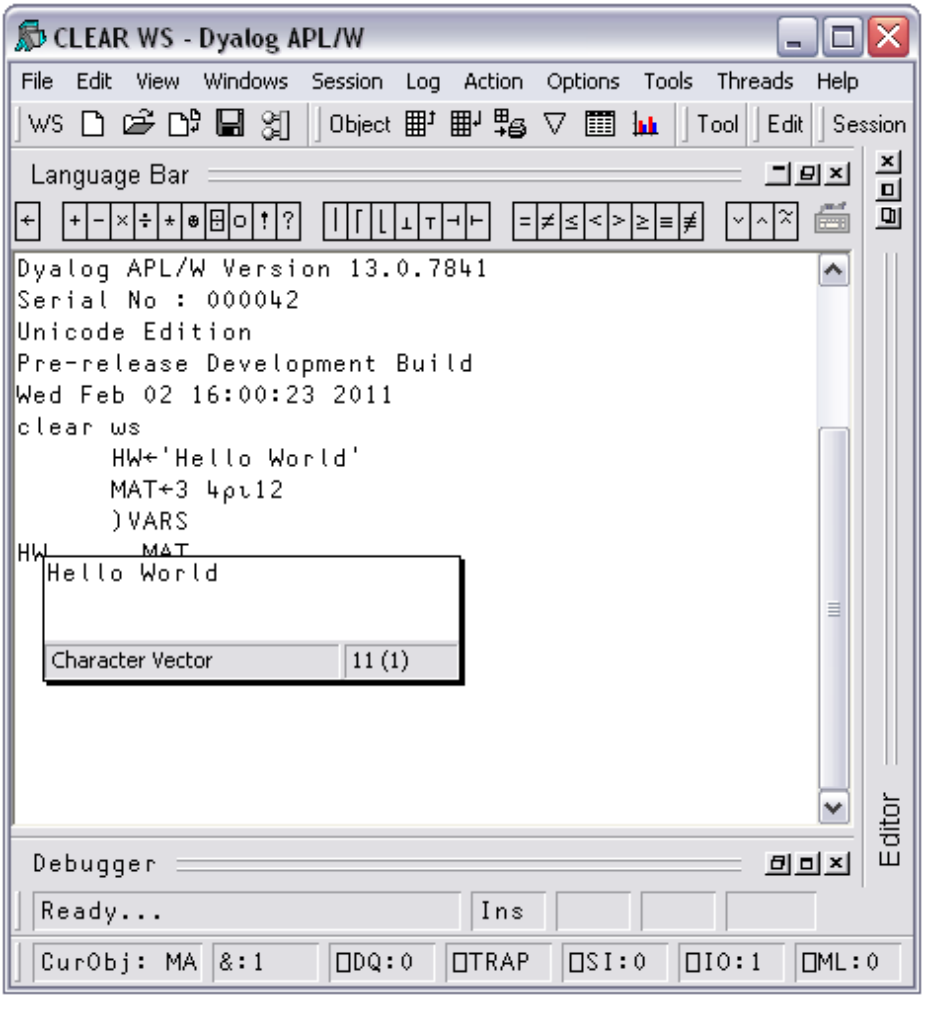
Enter `SALES` in the *Find* box, and `NEWSALES` in the *Replace* box. Now click the *Replace All* button. You will see all occurrences of `SALES` change to `NEWSALES`. Furthermore, each changed line in the session becomes marked (Red on White). Now click on the session and press Enter (or select *Execute* from the *Action* menu).

Once displayed, the *Find* or *Find/Replace* dialog box remains on the screen until it is either closed or replaced by the other. This is particularly convenient if the same operations are to be performed over and over again, and/or in several windows. *Find* and *Find/Replace* operations are effective in the window that previously had the focus.

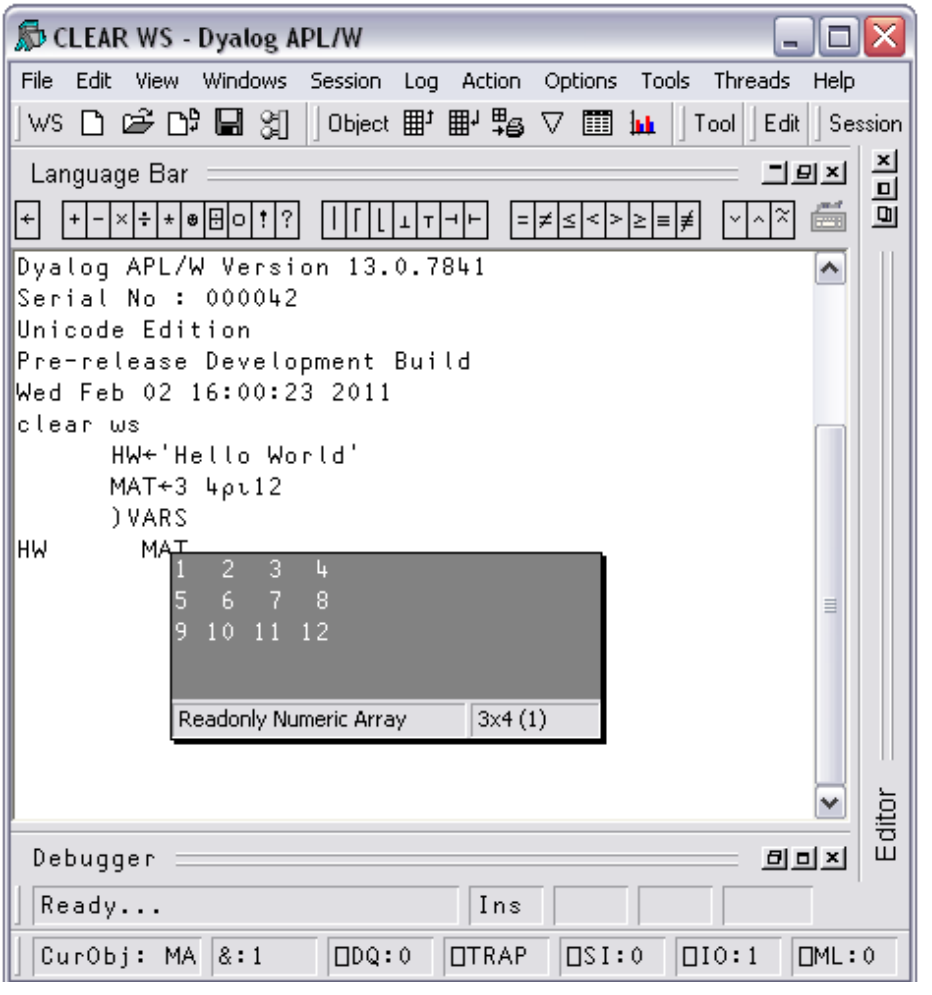
Value Tips

If you hover the mouse pointer over a name in the Session or Debugger window, APL will display a pop-up window containing the value of the symbol under the mouse pointer.

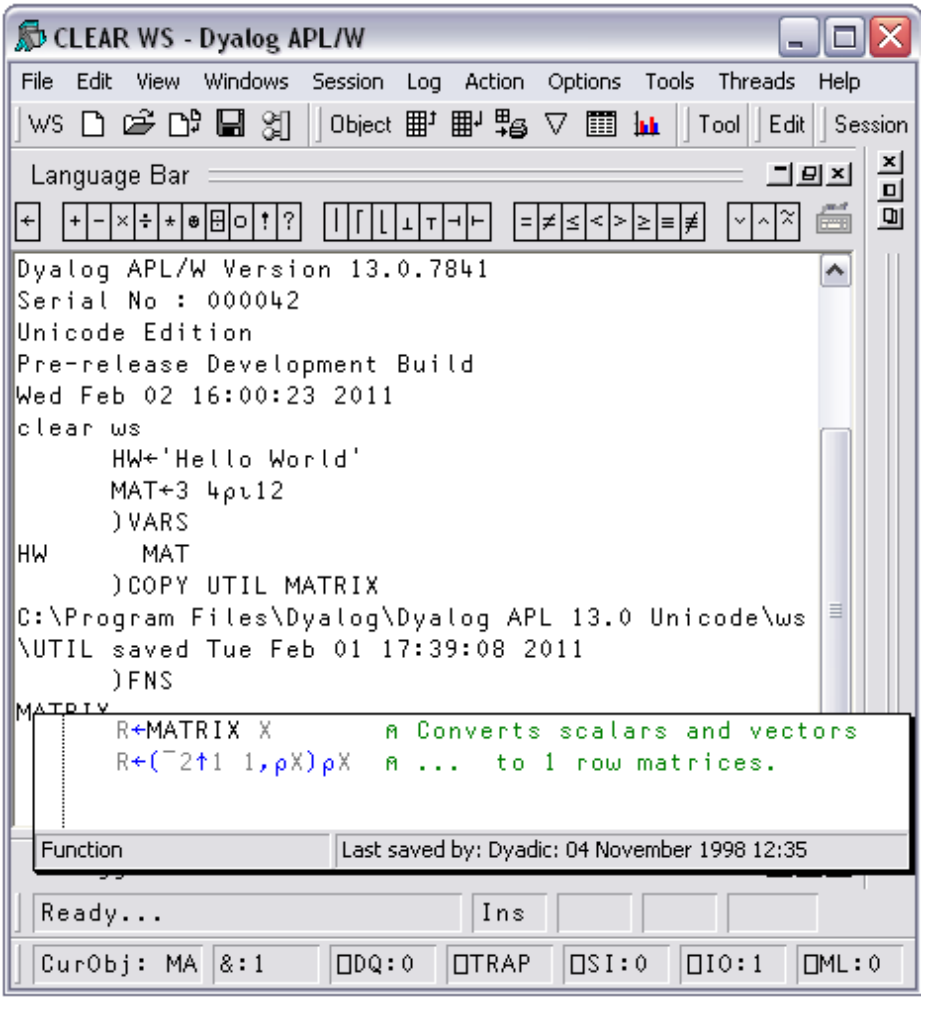
For example, in the following picture the mouse pointer was moved over the name of the variable `HW` in the Session window.



The next picture illustrates the Value Tip displayed when the mouse is hovered over the name of the variable `MAT`.



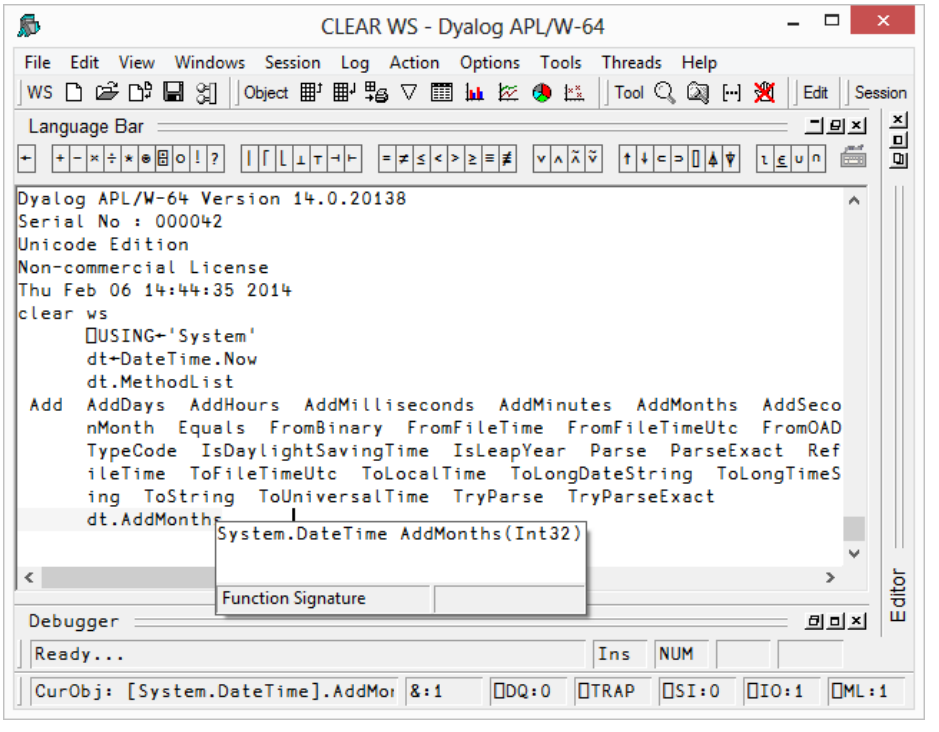
Similarly, if you hover the mouse pointer over the name of a function, the system displays the body of the function as a pop-up, as illustrated below.



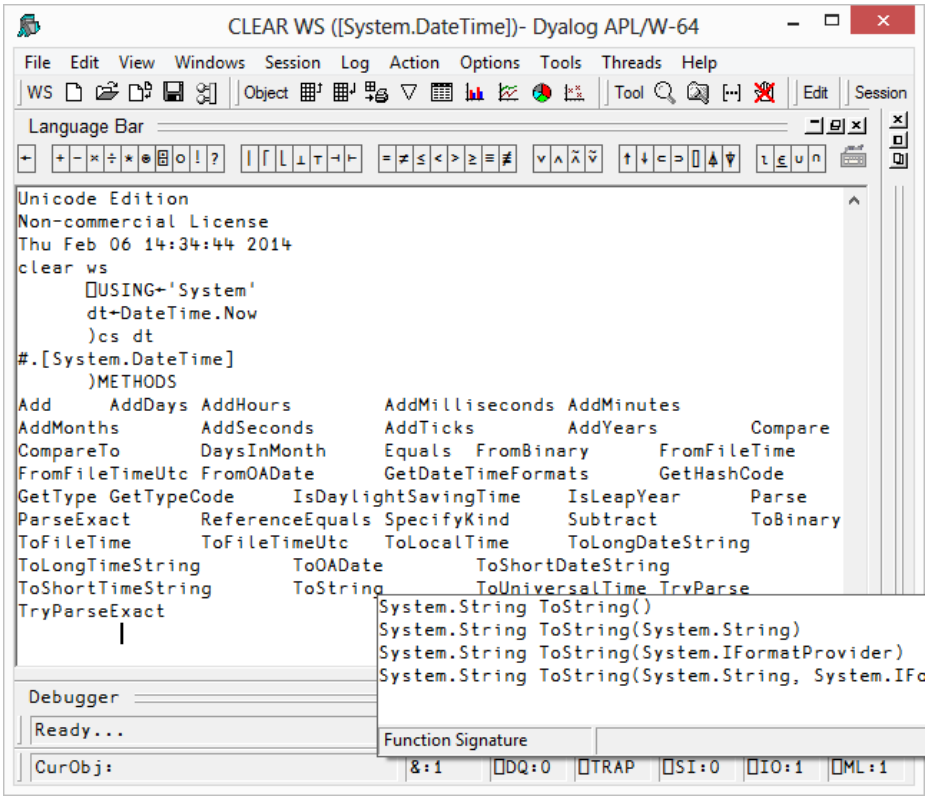
Value Tips for External Functions

Value Tips can also be used to investigate the syntax of external functions. If you hover over the name of an external function, the Value Tip displays its Function Signature.

For example, in the example below, the mouse is hovered over the external function `dt.AddMonths` and shows that it requires a single integer as its argument.



Should the external function provide more than one signature, they are all shown in the Value Tip as illustrated below. Here the function `ToString` has four different overloads.

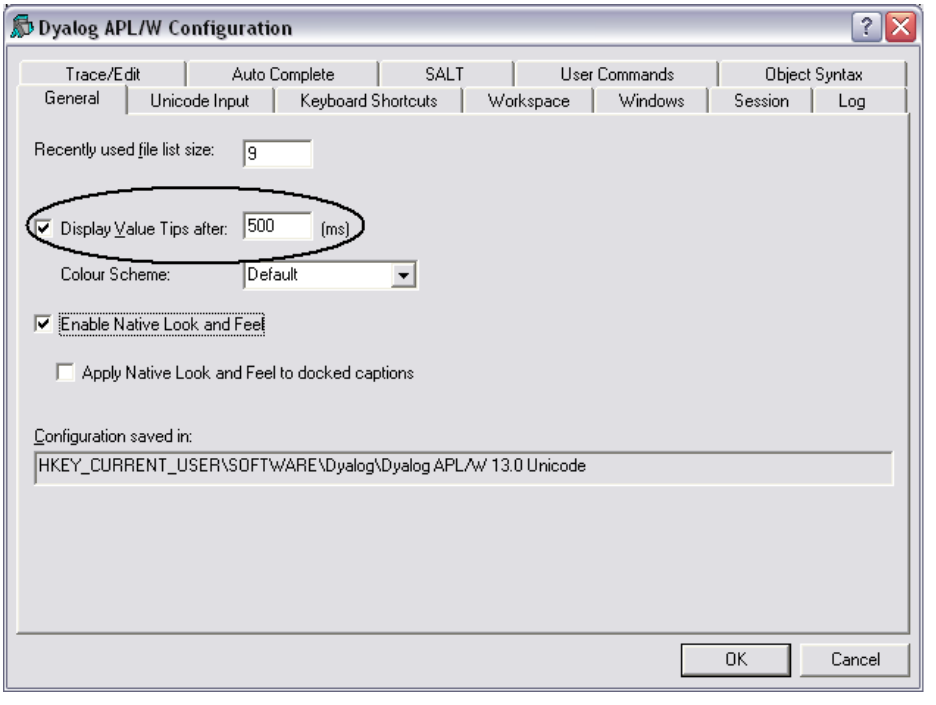


Configuring Value Tips

You may enable/disable Value Tips and select other options from the *General* tab of the *Configuration* dialog box as shown below.

You may experiment by changing the value of the delay before which Value Tips are displayed, until you find a comfortable setting.

Note that the colour scheme used to display the Value Tip for a function need not necessarily be the same colour scheme as you use for the function editor.

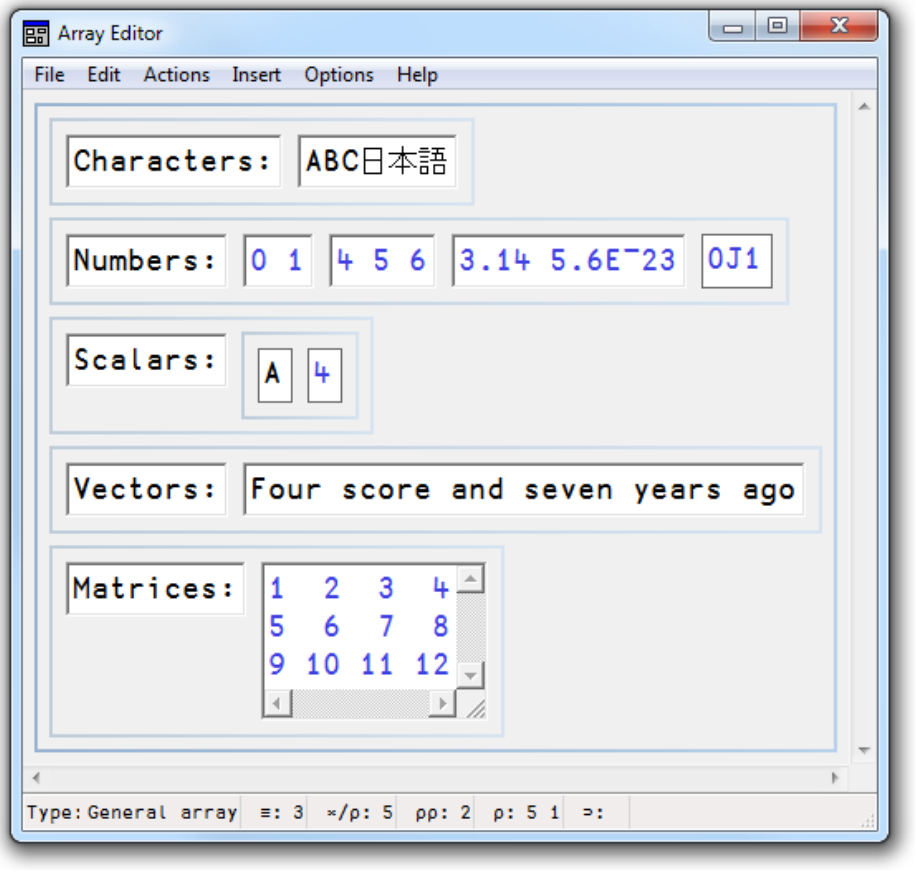


Array Editor

The Array Editor[1] allows you to edit arbitrary arrays. It is invoked by either:

- Clicking the icon in the Session toolbar when the mouse pointer is over the name of a suitable variable.
- Calling the user command `]aedit`, specifying the name of a suitable variable as its argument.
- Calling it directly via `⎕NA`

The Array Editor draws data using a format that is similar to the output of the `DISPLAY` function. For example:



[1]Array Editor Version 1 Release 1 © Copyright davidliebtag.com 2012, 2013

Documentation

Full documentation for the Array Editor, including a list of the keystrokes it uses, is available from the Help menu in the Array Editor's window.

Supported Arrays

The Array Editor supports arrays that consist solely of characters and/or numbers. You may not use it to edit an array that contains an object reference or a ⎕OR.

Reject unsupported data

The way that the Arrays Editor reacts to unsupported arrays is determined by the value of the **Reject unsupported data** option which is accessed by the *Options/Reject unsupported data* menu item on the Array Editor menubar.

If this is set to true (the default), and you try to edit an array containing an object reference, the Array Editor will refuse to start and the system will generate an error message.

```
⎕SE.NumEd.numed: Unexpected error in array editor:
                 DOMAIN ERROR  Argument contained data that is
                 neither simple or nested.
```

If this option is cleared, the Array editor will start but you will not be able to do anything. It is therefore advisable that you leave this option set.

Notes

- The Array Editor is supplied only with Unicode Editions of Dyalog APL/W. Please visit www.davidliebtag.com for details about availability and support for Classic Editions of Dyalog APL/W.
- Namespaces are not supported.
- Internal representations returned by ⎕OR are not supported.
- Only one instance of the Array Editor may be executed at a time.
- All calls to interpreter primitives use a value of 3 for ⎕ML.
- Negative numbers must be represented using high minus signs. For example, ¯3 not -3.

Implementation

The Array Editor is implemented by a DLL named `dlaedit.dll` (32-bit) or `dlaedit64.dll` (64-bit).

The DLL exports two functions: `DyalogEditArray` and `DyalogEditArrayTitle`. The latter is used when you click the the  icon in the Session toolbar (via the APL function `⎕SE.NumEd.numed`) and by the user command `]aedit`

Calling the Array Editor Directly

If you wish to use the Array Editor directly, you may do so as follows using `⎕NA`[1].

For both `DyalogEditArray` and `DyalogEditArrayTitle` the first argument is the array to be edited, while the second argument is a place holder and should always be 0

For `DyalogEditArrayTitle` the 3rd argument is a character vector whose contents are displayed in the caption of the array editor window.

The result is the newly altered array.

Examples

```
⎕NA'dlaedit.dll|DyalogEditArray <pp >pp'              ⍝ 32-bit
⎕NA'dlaedit.dll|DyalogEditArrayTitle <pp >pp <0C2[]'  ⍝ 32-bit

⎕NA'dlaedit64.dll|DyalogEditArray <pp >pp'            ⍝ 64-bit
⎕NA'dlaedit64.dll|DyalogEditArrayTitle <pp >pp <0C2[]'⍝ 64-bit

New←DyalogEditArray Old 0
New←DyalogEditArrayTitle Old 0 Name
```

[1]Note that these are not standard `⎕NA` calls, but rather use an extension to `⎕NA`, called *Direct Workspace Access*. Dyalog does not intend to make this feature generally available at present: if you are interested in this feature please contact sales@dyalog.com.

SharpPlot Graphics Tools

Introduction

Included with Dyalog APL is the SharpPlot graphics library which is part of the RainPro graphics package..

The Session includes a button which calls SharpPlot to generate graphical pictures of the contents of the Current Object (identified by the name under or to the left of the cursor).

For example, if you have a numerical matrix in a variable called `MAT`, you can plot it by first positioning the cursor on the name `MAT` in the Session window, and then clicking the *SharpPlot* button in the Session toolbar.

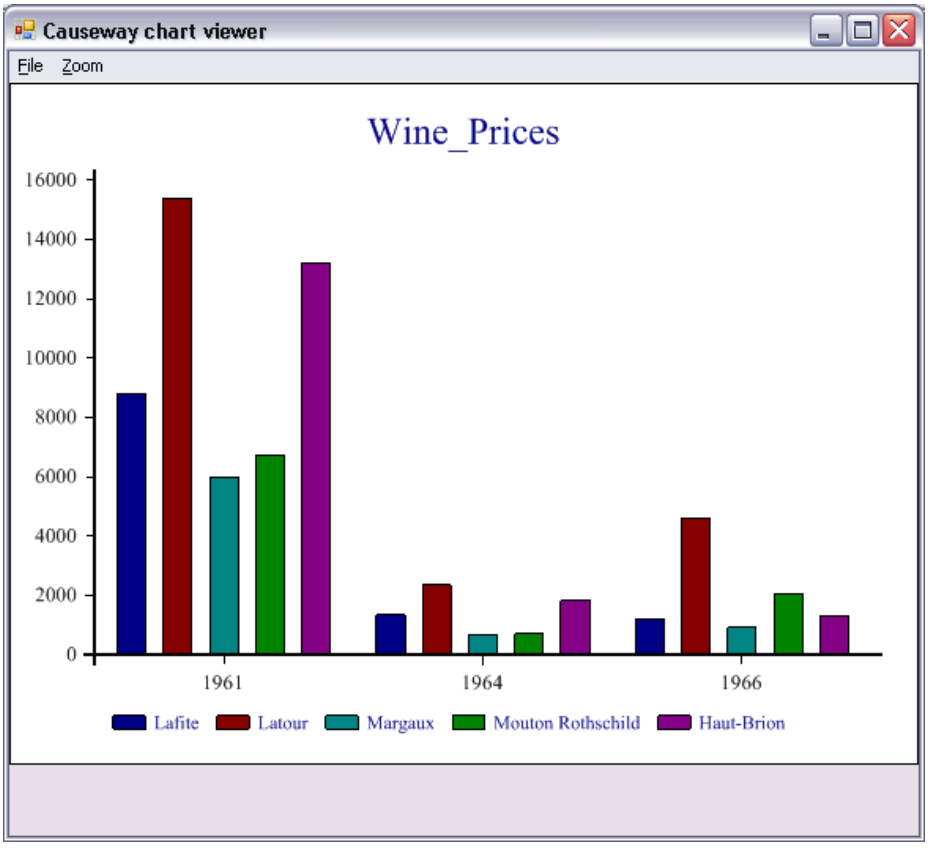
Data Structures

The charting function can plot variables with the following data structures:

- a simple numeric vector
- a vector of simple numeric vectors
- a simple numeric matrix
- a matrix whose first row contains simple character vectors and whose other elements are simple numerics. In bar and line charts, the column headings in row 1 are used as x-axis labels.
- a matrix whose first column contains simple character vectors and whose other elements are simple numerics. In bar and line charts, the row headings in column 1 are used as legends to annotate the different series.
- a matrix whose first row and first column both contain simple character vectors and whose other elements are simple numerics. In bar and line charts, the column headings in row 1 are used as x-axis labels, and the row headings in column 1 are used as legends annotate the different series.

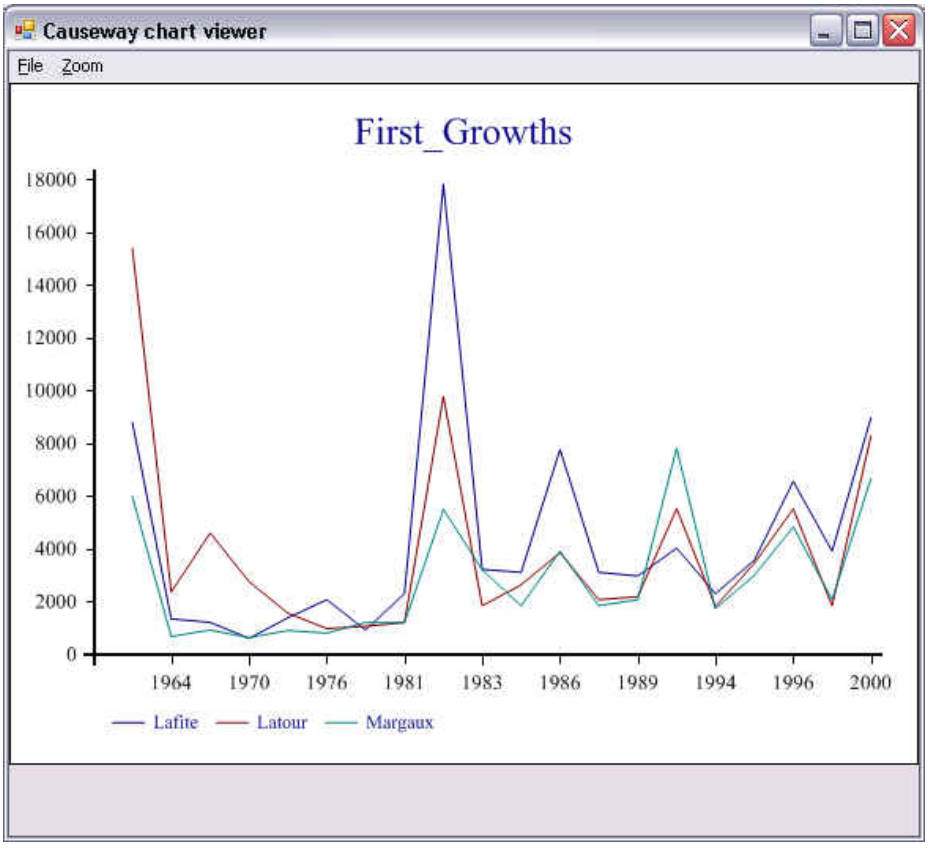
Example: Bar Chart

```
      Wine_Prices
1961 1964 1966
Lafite 8800 1342 1210
Latour 15400 2357.5 4600
Margaux 5980 672.5 920
Mouton Rothschild 6710 713 2070
Haut-Brion 13225 1840 1323
```



Example: Line Chart

```
      First_Growths
1961 1964 1966 1970 1975 1976 1978 ...
Lafite 8800 1342 1210 605 1380 2070 920 ...
Latour 15400 2357.5 4600 2760 1552 978 1058 ...
Margaux 5980 672.5 920 632 900 800 1208 ...
```



Implementation

The SharpPlot Wizard is called by clicking on the SharpPlot button in the Session toolbar. The button has a Select callback which runs the function `⎕SE.Chart.DoChart`. This runs `⎕SE.Chart.Do` which constructs and then runs a function named `⎕SE.Chart.MyChart`.

`⎕SE.Chart.MyChart` uses an instance of the SharpPlot graphics class to produce a chart of your data, which it saves as a temporary file. It then calls the SharpPlot viewer to display the file on your screen. SharpPlot can also be started using the `]chart` user command.

SharpPlot is a library of graphical subroutines, (originally written in APL and machine-translated into C#) which is implemented as a .NET Namespace named Causeway and supplied in the `sharpplot.dll` in the Dyalog program directory.

Notes

Although `⎕SE.Chart.MyChart` is overwritten by successive uses of the graphical button, it is deliberately not erased each time. This allows you to use `MyChart` as a simple template to develop your own custom graphics function.

The image is stored in Microsoft Enhanced Metafile Format in a temporary file whose name and location are generated automatically. The system does not delete the temporary file after use. For further details, See `GetTempFileName` in the Windows documentation..

The default program used to display the EMF file is `SharpView.exe`. You can opt to use a different EMF viewer by setting the `Charts\ViewCMD`registry key to name another program, such as Windows Picture and Fax Viewer.

An attempt to plot the contents of a variable with an unsupported data structure (see above) is handled entirely by error trapping and will result in an error message box and perhaps messages in the Status window.

The Session GUI Hierarchy

As distributed, the Session object `⎕SE` contains two CoolBar objects. The first, named `⎕SE.cbtop` runs along the top of the Session window and contains the toolbars. The second, named `⎕SE.cbbot`, runs along the bottom of the Session windows and contains the statusbars.

The menubar is implemented by a MenuBar object named `⎕SE.mb`.

The toolbars in `⎕SE.cbtop` are implemented by four CoolBand objects, `bandtb1`, `bandtb2`, `bandtb3` and `bandtb4` each containing a ToolControl named `tb`.

The statusbars in `⎕SE.cbbot`, are implemented by two CoolBand objects , `bandtb1` and `bandtb2`, each containing a StatusBar named `sb`.

The Session MenuBar

The Session MenuBar (`⎕SE.mb`) contains a set of menus as follows.

The File Menu

The *File* menu (`⎕SE.mb.file`) provides a means to execute those APL System Commands that are concerned with the active and saved workspaces. The contents of a typical File menu and the operations they perform are illustrated below.

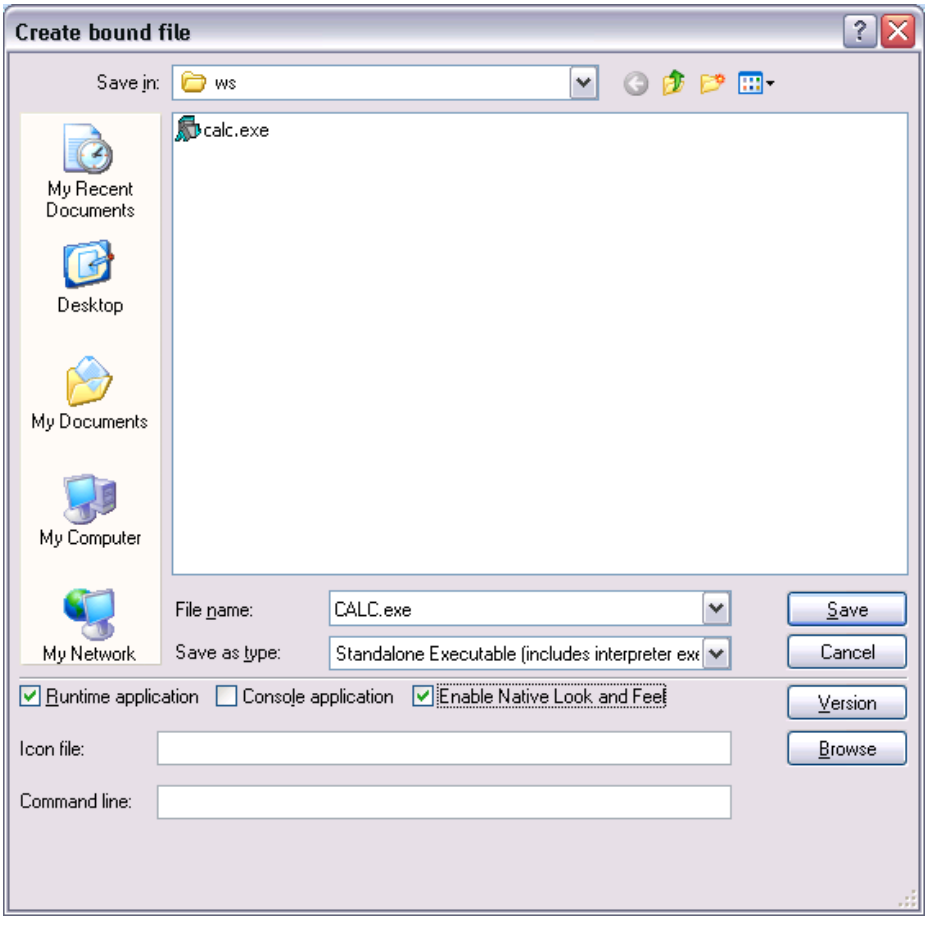
Table 7: File Menu Operations

Item	Action	Description
New	[WSClear]	Prompts for confirmation, then clears the workspace
Open	[WSLoad]	Prompts for a workspace file name, then loads it
Copy	[WSCopy]	Prompts for a workspace file name, then copies it
Save	[WSSave]	Saves the active workspace
Save As	`[WSSaveas]`	Prompts for a workspace file name, then saves it
Export	`[Makeexe]`	Creates a bound executable, an OLE Server, an ActiveX Control, or a .NET Assembly
Export to Memory	`[MakeMemory Assembly]`	Creates an i*n-memory* .NET Assembly
Close AppDomain	`[CloseAppDomain]`	Closes .NET App Domain
Drop	`[WSDrop]`	Prompts for a workspace file name, then erases it
Print	`[PrintFnsInNS]`	Prints functions and operators in current namespace
Print Setup	`[PrintSetup]`	Invokes the print set-up dialog box
Continue	`[Continue]`	Saves the active workspace in CONTINUE.DWS and exits APL
Exit	`[Off]`	Exits APL

Export

The *Export...* menu item allows you to create a bound executable, an OLE Server (in-process or out-of-process), an ActiveX Control or a .NET Assembly.

The dialog box used to create these various different files offers selective options according to the type of file you are making. The system detects which of these types is most appropriate from the objects in your workspace. For example, if your workspace contains an ActiveXControl namespace, it will automatically select the *ActiveX Control* option.

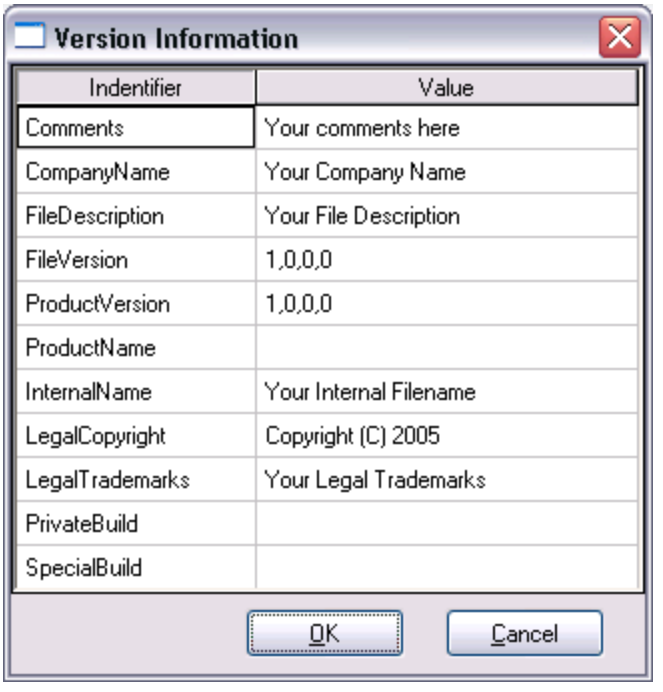


The *Create bound file* dialog box contains the following fields. These will only be present if applicable to the type of bound file you are making.

Item	Description
File name	Allows you to choose the name for your bound file The name defaults to the name of your workspace with the appropriate extension.
Save as type	Allows you to choose the type of file you wish to create.
Runtime application	If this is checked, your application file will be bound with the Run-Time DLL. If not, it will be bound with the Development DLL. The latter should normally only be used to permit debugging.
Console application	Check this box if you want your executable to run as a console application. This is appropriate only if the application has no graphical user interface.
Enable Native Look and Feel	If checked, *Native Look and Feel* will be enabled for your bound file.
Icon file	Allows you to associate an icon with your executable. Type in the pathname, or use the *Browse* button to navigate to an icon file.
Command line	For an out-of-process COM Server, this allows you to specify the command line for the process. For a bound executable, this allows you to specify command-line parameters for the corresponding Dyalog APL DLL.

Pressing the *Version button* brings up the *Version Information* dialog box shown below.

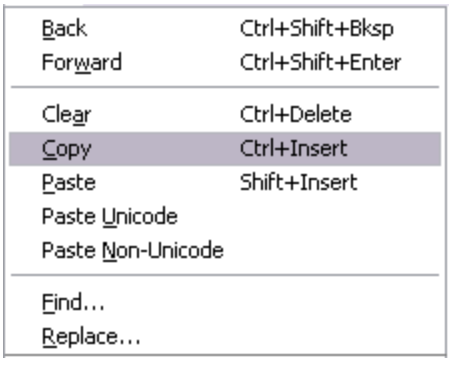
This dialog box allows you to specify versioning information that will be stored in your bound file.



The Edit Menu

The *Edit* menu (`⎕SE.mb.edit`) provides a means to recall previously entered input lines for re-execution and for copying text to and from the clipboard.

Unicode Edition



Classic Edition

Table 8: Edit menu operations

Item	Action	Description
Back	`[Undo]`	Displays the previous input line. Repeated use of this command cycles back through the input history.
Forward	`[Redo]`	Displays the next input line. Repeated use of this command cycles forward through the input history.
Clear	`[Delete]`	Clears the selected text
Copy	`[Copy]`	Copies the selection to the clipboard
Paste	`[Paste]`	Pastes the text contents of the clipboard into the session log at the current location. The new lines are *marked* and may be executed by pressing Enter.
Paste Unicode	`[Pasteunicode]`	Same as *Paste*, but gets the Unicode text from the clipboard and converts to `⎕AV`. **Classic Edition only**.
Paste Non-Unicode	`[PasteAnsi]`	Same as *Paste*, but gets the ANSI text from the clipboard and converts to `⎕AV`. **Classic Edition only**.
Find	`[Find]`	Displays the *Find* dialog box
Replace	`[Replace]`	Displays the *Find/Replace* dialog box

The View Menu

The *View* menu (`⎕SE.mb.view`) toggles the visibility of the Session Toolbar, StatusBar, and Language Bar.

Table 9: View menu operations

Item	Action	Description
Toolbar		Shows/Hides Session toolbars
Statusbar		Shows/Hides Session statusbars
LanguageBar		Shows/Hides Language Bar

The Window Menu

This contains a single action (`⎕SE.mb.windows`) which is to close all of the Edit and Trace windows and the Status window.

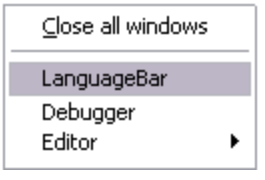


Table 10: Window menu operations

Item	Action	Description
Close all Windows	`[CloseAll]`	Closes all Edit and Trace windows

Note that `[CloseAll]` removes all Trace windows but does *not* reset the State Indicator.

In addition, the *Window* menu will contain options to switch the focus to any subsidiary windows that are docked in the Session as illustrated above.

The Session Menu

The *Session* menu (`⎕SE.mb.session`) provides access to the system operations that allow you to load a session (`⎕SE`) from a session file and to save your current session (`⎕SE`) to a session file. If you use these facilities rarely, you may wish to move them to (say) the *Options* menu or even dispense with them entirely.

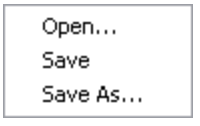


Table 11: Session menu operations

Item	Action	Description
Open	`[SELoad]`	Prompts for a session file name, then loads the session from it, replacing the current one. Sets the File property of `⎕SE` to the name of the file from which the session was loaded.
Save	`[SESave]`	Saves the current session (as defined by `⎕SE`) to the session file specified by the File property of `⎕SE`.
Save As	`[SESaveAs]`	Prompts for a session file name, then saves the current session (as defined by `⎕SE`) in it. Resets the File property of `⎕SE`.

The Log Menu

The *Log* menu (`⎕SE.mb.log`) provides access to the system operations that manipulate Session log files.

Table 12: Log menu operations

Item	Action	Description
New	`[NewLog]`	Prompts for confirmation, then empties the current Session log.
Open	`[OpenLog]`	Prompts for a Session log file, then loads it into memory, replacing the current Session log
Save	`[SaveLog]`	Saves the current Session log in the current log file, replacing its previous contents
Save As	`[SaveLogAs]`	Prompts for a file name, then saves the current Session log in it.
Print	`[PrintLog]`	Prints the contents of the Session log.

The Action Menu

The *Action* menu (`⎕SE.mb.action`) may be used to perform a variety of operations on the *current object* or the *current line*. The current object is the object whose name contains the cursor. The current line is that line that contains the cursor. The *Edit*, *Copy Object*, *Paste Object* and *Print Object* items operate on the current object. For example, if the name `SALES` appears in the session and the cursor is placed somewhere within it, `SALES` is the current object and will be copied to the clipboard by selecting *Copy object* or opened up for editing by selecting *Edit*.

Execute runs the current line; Trace traces it.

Unicode Edition

Classic Edition

Table 13: Action menu operations

Item	Action	Description
Edit	`[Edit]`	Edit the current object
Trace	`[Trace]`	Executes the current line under the control of the Tracer
Execute	`[Execute]`	Executes the current line
Copy Object	`[ObjCopy]`	Copies the contents of the current object to the clipboard.
Paste Object	`[ObjPaste]`	Pastes the contents of the clipboard into the current object, replacing its previous value
Print Object	`[ObjPrint]`	Prints the current object.
Clear Stops	`[ClearTSM]`	Clears all `⎕STOP`, `⎕MONITOR` and `⎕TRACE` settings
Interrupt	`[Interrupt]`	Generates a weak interrupt
Reset	`[Reset]`	Performs `)RESET`

The Options Menu

The *Options* menu (`⎕SE.mb.options`) provides configuration options.

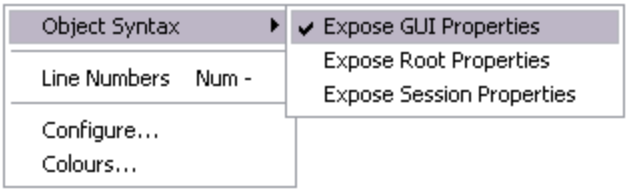


Table 14: Options menu operations

Item	Action	Description
Expose GUI Properties	`[ExposeGUI]`	Exposes the names of properties, methods and events in GUI objects
Expose Root Properties	`[ExposeRoot]`	Exposes the names of the properties, methods and events of the Root object
Expose Session Properties	`[ExposeSession]`	Exposes the names of the properties, methods and events of `⎕SE`
Line Numbers	`[LineNumbers]`	Toggle the display of line numbers in edit and trace windows on/off
Configure	`[Configure]`	Displays the *Configuration* dialog box
Colours	`[ChooseColors]`	Displays the *Colours Selection* dialog box

The values associated with the *Expose GUI*, *Expose Root* and *Expose Session* options reflect the values of these settings in your current workspace and are saved in it.

When you change these values through the *Options* menu, you are changing them in the current workspace only.

The default values of these items are defined by the parameters **default_wx**, **PropertyExposeRoot** and **PropertyExposeSE** which may be set using the *Object Syntax* tab of the *Configuration* dialog.

The Tools Menu

The *Tools* menu (`⎕SE.mb.tools`) provides access to various session tools and dialog boxes.

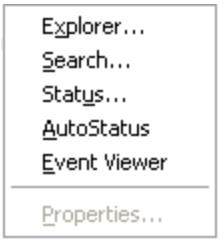


Table 15: Tools Menu Operations

Item	Action	Description
Explorer	`[Explorer]`	Displays the *Workspace Explorer* tool
Search	`[WSSearch]`	Displays the *Workspace Search* tool
Status	`[Status]`	Displays or hides the *Status* window
AutoStatus	`[AutoStatus]`	Toggle; if checked, causes the *Status* window to be displayed when a new message is generated for it
Event Viewer	`[EventViewer]`	Displays or hides the *Event Viewer*
Properties	`[ObjProps]`	Displays a property sheet for the current object

The Threads Menu

The *Threads* menu (`⎕SE.mb.threads`) provides access to various session tools and dialog boxes.

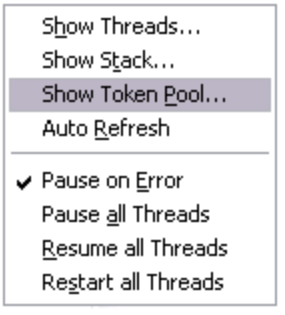


Table 16: Threads Menu Operations

Item	Action	Description
Show Threads	`[Threads]`	Displays the *Threads Tool*
Show Stack	`[Stack]`	Displays the *SI Stack* window
Show Token Pool	`[TokenPool]`	Displays the *Token Pool* window
Auto Refresh	`[ThreadsAutoRefresh]`	Refreshes the *Threads Tool* on every thread switch
Pause on Error	`[ThreadsPauseOnError]`	Pauses all threads on error
Pause all Threads	`[ThreadsPauseAll]`	Pauses all threads
Resume all Threads	`[ThreadsResumeAll]`	Resumes all threads
Restart all Threads	`[ThreadsRestartAll]`	Restarts all threads

The Help Menu

The *Help* menu (⎕SE.mb.help) provides access to the help system which is packaged as a single *Microsoft HTML Help* compiled help file named `help\dyalog.chm`.

Table 17: Help menu operations

Label	Action	Description
Documentation Center	`[DocCenter]`	Opens your web browser on `help\index.html` which displays an index to the on-line PDF documentation and selected internet links.
Latest Enhancements	`[RelNotes]`	Opens `help\dyalog.chm`, starting at the first topic in the Version 14.0 Release Notes section. Note that previous Release Notes are also included for your convenience.
Language Help	`[LangHelp]`	Opens `help\dyalog.chm`, starting at the first topic in the Language Reference section.
Gui Help	`[GuiHelp]`	Opens `help\dyalog.chm`, starting at the first topic in the Object Reference section.
Third Party Licences	`[LicenceHelp]`	Opens `help\dyalog.chm`, starting at the first topic in the Licences for third-party components.
Dyalog Web Site	`[DyalogWeb]`	Opens your web browser on the Dyalog home page.

Label	Action	Description
Email Dyalog	`[DyalogEmail]`	Opens your email client and creates a new message to Dyalog Support, with information about the Version of Dyalog APL you are running.
About Dyalog APL	`[About]`	Displays an *About* dialog box

Session Pop-Up Menu

The Session popup menu (`⎕SE.popup`) is displayed by clicking the right mouse button anywhere in the Session window. If the mouse pointer is over a visible object name, the popup menu allows you to edit, print, delete it or view its properties. Note that the name of the pop-up menu is specified by the Popup property of `⎕SE`.

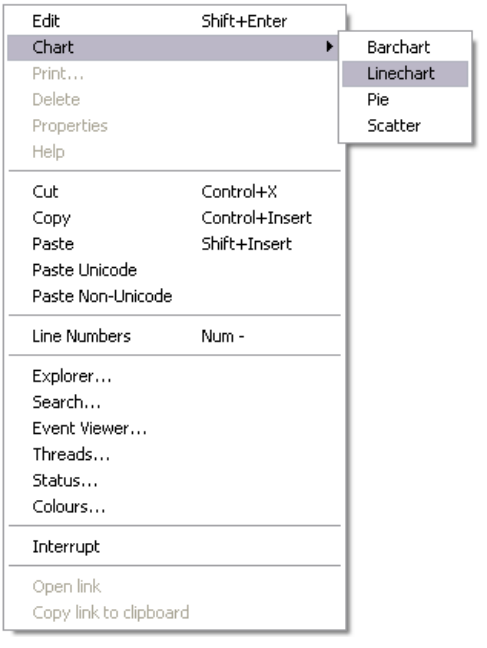


Table 18: Session popup menu operations

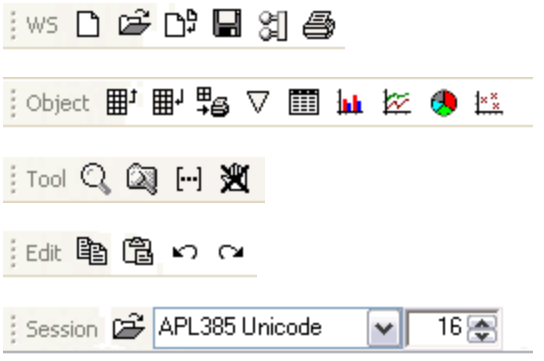
Item	Action	Description
Edit	`[Edit]`	Edits the current object
Chart Wizard	`⎕se.Chart.DoChart`	Opens Chart Wizard on current object
Print	`[ObjPrint]`	Prints the current object
Delete	`[ObjDelete]`	Erases the current object
Properties	`[GUIHelp]`	Displays the *Object Properties* dialog box for the current object
Help	`[Help]`	Displays the help topic associated with the current object or the APL symbol under the cursor
Cut	`[Cut]`	Deletes selected text
Copy	`[Copy]`	Copies the selection to the clipboard
Paste	`[Paste]`	Pastes the text contents of the clipboard into the session log at the current location. The new lines are *marked* and may be executed by pressing Enter.
Paste Unicode	`[PasteUnicode]`	Same as *Paste*, but gets the Unicode text from the clipboard and converts to `⎕AV`
Paste Non-Unicode	`[PasteAnsi]`	Same as *Paste*, but gets the ANSI text from the clipboard and converts to `⎕AV`
Line Numbers	`[LineNumbers]`	Toggles line numbers on/off
Align Comments	`[AlignComments]`	Aligns Comments to current column
Explorer	`[Explorer]`	Displays the *Workspace Explorer*
Search	`[WSSearch]`	Displays the *Find Objects* tool
Event Viewer	`[EventViewer]`	Displays the *Event Viewer*

Item	`Action`	Description
Threads	`[Threads]`	Displays the *Threads Tool*
Status	`[Status]`	Displays the *Status* window
Colours	`[ChooseColors]`	Displays the *Colour Selection* dialog
Interrupt	`[Interrupt]`	Generates a weak interrupt
Open link	`[OpenLink]`	Opens the URL or link using the appropriate program.
Copy link to clipboard	`[CopyLink]`	Copies the URL or link to the Windows Clipboard.

For the last two items, see *Installation & Configuration Guide: Configuration Dialog: General Tab*)

The Session Toolbars

The Session toolbars are contained by four separate CoolBand objects, allowing you to configure their order in whichever way you choose.



The Session tool bars

The bitmaps for the buttons displayed on the session tool bar are implemented by three ImageList objects owned by the CoolBar `⎕SE.cbtop`. These represent the ToolButton images in their normal, highlighted and inactive states and are named `iln`, `ilh` and `ili` respectively.

These images derive from three bitmap resources contained in `dyalog.exe` named `tb_normal`, `tb_hot` and `tb_inactive`. The statements that create these ImageList object in function `BUILD_SESSION` in BUILDSE.DWS are as follows.

```
:With '⎕SE.cbtop'
   'iln'⎕WC'ImageList'('MapCols' 0)('Masked' 1)
   'iln.bm'⎕WC'Bitmap'('' 'tb_normal')('MaskCol'(192 192
192))
   'ilh'⎕WC'ImageList'('MapCols' 0)('Masked' 1)
   'ilh.bm'⎕WC'Bitmap'('' 'tb_hot')('MaskCol'(192 192 19
2))
   'ili'⎕WC'ImageList'('MapCols' 0)('Masked' 1)
   'ili.bm'⎕WC'Bitmap'('' 'tb_inactive')('MaskCol'(192 19
2 192))
:EndWith
```

Workspace (WS) Operations

 Clear Workspace	Executes the system operation `[WSClear]` which asks for confirmation, then clears the workspace.
 Load Workspace	Executes the system operation `[WSLoad]` which displays a file selection dialog box and loads the selected workspace.
 Copy Workspace	Executes the system operation `[WSCopy]` which displays a file selection dialog box and copies the (entire) selected workspace
 Save Workspace	Executes the system operation `[WSSaveas]` which displays a file selection dialog box and saves the workspace in the selected file.
 Re-Export Workspace	Executes the system operation `[ReExport]` which re-exports the workspace using the settings, parameters and options that were previously selected using the *Create Bound File* dialog.
 Print Workspace	Executes the system operation `[PrintFnsInNS]` that prints all the functions and operators in the current namespace.

Object Operations


Copy Object

Executes the system operation `[ObjCopy]` which copies the contents of the current object to the clipboard.


Paste Object

Executes the system operation `[ObjPaste]` which copies the contents of the clipboard into the current object, replacing its previous value.


Print Object

Executes the system operation `[ObjPrint]` that prints the current object.


Edit Object

Executes the system operation `[Edit]` which edits the current object using the standard system editor.


Edit Array

Executes a defined function in `⎕SE` that edits the current object using the Array Editor (Unicode Edition) or a spreadsheet-like interface based upon the Grid object (Classic Edition). See *Array Editor* on page 39.


Barchart

Executes a defined function in `⎕SE` that displays the value of the current object in a Barchart.


Linechart

Executes a defined function in `⎕SE` that that displays the value of the current object in a Linechart.


Piechart

Executes a defined function in `⎕SE` that that displays the value of the current object in a Piechart.


Scatterplot

Executes a defined function in `⎕SE` that that displays the value of the current object in a Scatterplot.

Tools


Explorer

Executes the system operation `[Explorer]` which displays the *Workspace Explorer* tool.


Search

Executes the system operation `[WSSearch]` which displays the *Workspace Search* tool.


Line Numbers

Executes the system operation `[LineNumbers]` which toggles the display of line numbers in edit and trace windows on and off.


Clear all Stops

Executes the system operation `[ClearTSM]` which clears all `⎕STOP`, `⎕MONITOR` and `⎕TRACE` settings

Edit Operations


Copy Selection

Executes the system operation `[Copy]` which copies the selected text to the clipboard.


Paste Selection

Executes the system operation `[Paste]` which pastes the text in the clipboard into the current window at the insertion point.


Recall Last

Executes the system operation `[Undo]` which recalls the previous input line from the input history stack


Recall Next

Executes the system operation `[Redo]` which recalls the next input line from the input history stack.

Session Operations

Load Session

Executes the system operation `[SELoad]` which displays a file selection dialog box and loads the selected Session File.

Select Font

Selects the font to be used in the Session window.

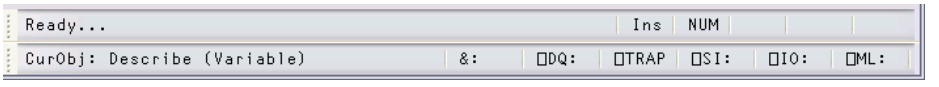
Select Font Size

Selects the size of the font to be used in the Session window.

The Session Status Bar

The session status bar is represented by two CoolBands each of which contains a StatusBar object. There are a number of StatusFields as illustrated below. Your own status bar may be configured differently.

Classic Edition



Unicode Edition

The StatusField objects owned by the session StatusBar may have special values of Style, which are used for operations relevant only to the Session. These styles are summarised in the tables shown below.

Table 19: Session status fields : first row

StatusField	Style	Description
hint	None	Displays hints for the session objects, or "Ready..." when APL is waiting for input
insrep	InsRep	Displays the mode of the Insert key (Ins or Rep)
mode	KeyMode	Displays the keyboard mode. This is applicable only to a multi-mode keyboard. The text displayed is defined by the Mn= string in the Input Table. **Classic Edition Only**.
num	NumLock	Indicates the state of the Num Lock key. Displays "NUM" if Num Lock is on, blank if off.
caps	CapsLock	Indicates the state of the Caps Lock key. Displays "Caps" if Caps Lock is on, blank if off.
pause	Pause	Displays a flashing red "Pause" message when the Pause key is used to halt session output

Table 20: Session status fields : second row

StatusField	Style	Description
curobj	CurObj	Displays the name of the current object (the name last under the input cursor)
tc	ThreadCount	Displays the number of threads currently running (minimum is 1)
dqlen	DQLen	Displays the number of events in the APL event queue
trap	Trap	Turns red if `⎕TRAP` is set
si	SI	Displays the length of `⎕SI`. Turns red if non-zero
io	IO	Displays the value of `⎕IO`. Turns red if `⎕IO` is not equal to the value of the **default_io** parameter
ml	ML	Displays the value of `⎕ML`. Turns red if `⎕ML` is not equal to the value of the **default_ml** parameter

Toggle Status Fields

In the default Session files distributed with this release, the Statusfields used to display the value of `⎕IO`, the state of the Insert key (Ins/Rep) and the current keyboard mode (e.g. Apl/Uni) have callback functions attached to MouseDblClick. This means that you can toggle the state of these fields by double-clicking with the left mouse button.

If you dislike this behaviour, you may set the Event property of the Statusfields to 0 and re-save the Session file. Alternatively, you may modify BUILDSE.DWS and rebuild the Session from scratch.

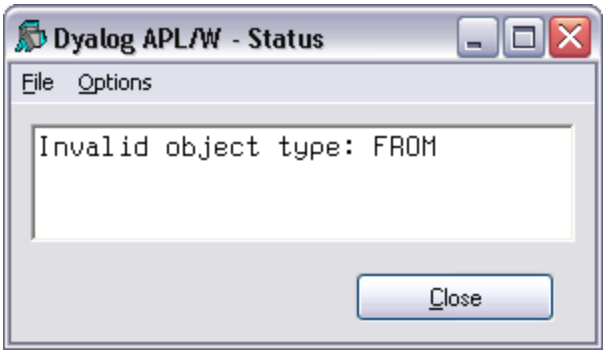
Status Window

The Status window is used to display system messages and supplementary information. These include the operations that take place when you register an OLEServer or ActiveXControl.

The Status window is also used to display supplementary information about errors. For example, if in a ⎕WC statement you misspell the type of an object, you will get a suitable error message in the Status window, in addition to the DOMAIN ERROR message in the Session.

Example

```
      'F'⎕WC'FROM' ⍝ Should be 'FORM'
DOMAIN ERROR
      'F'⎕WC'FROM'
   ^
```



The Status window can be explicitly displayed or hidden using the [Status] system operation which is associated with the *Tools/Status* menu item.

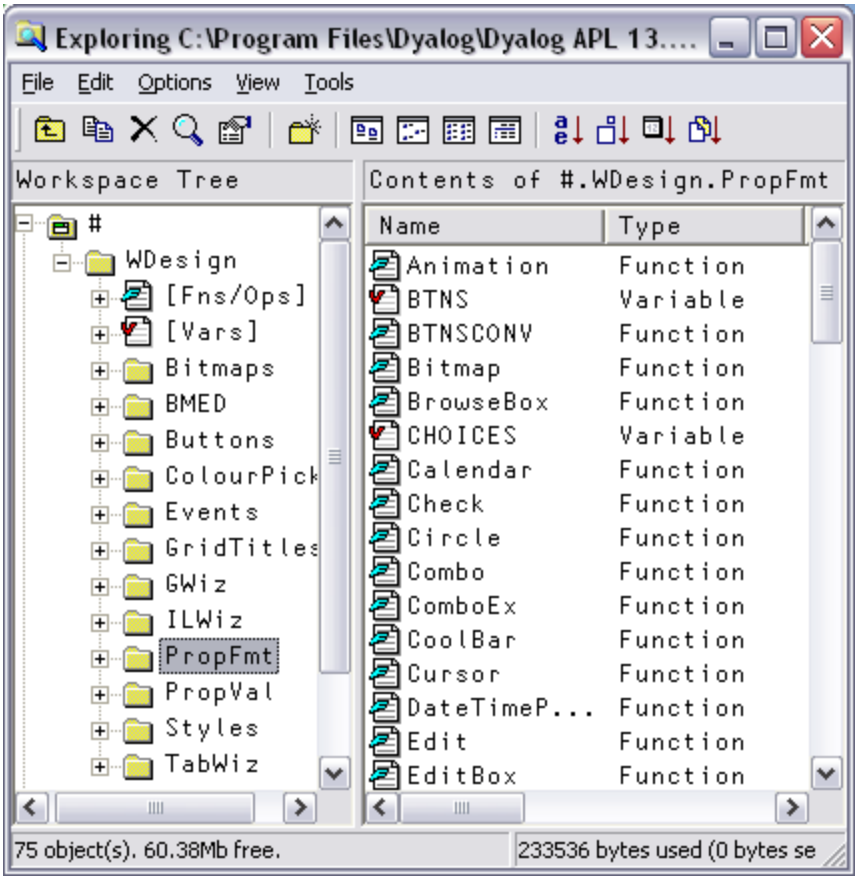
There is also an option to have the Status window appear automatically whenever a new message is written to it. This option is selected using the [AutoStatus] system operation which is associated with the *Tools/AutoStatus* menu item.

Note that when you close the Status window, all the system messages in it are cleared.

The Workspace Explorer Tool

The Explorer tool is a modeless dialog box that may be toggled on and off by the system action `[Explorer]`. In a default Session, this is attached to a MenuItem in the *Tools* menu and a Button on the session toolbar.

The Explorer contains two sub-windows. The one on the left displays the namespace structure of your workspace using a TreeView. The right-hand window is a ListView that displays the contents of the namespace that is selected in the TreeView.



The Explorer is closely modelled on the *Windows Explorer* in Windows and the facilities it provides are very similar. For Windows users, the operation of this tool is probably self-explanatory. However, other users may find the following discussion useful.

Exploring the Workspace

The TreeView displays the structure of your workspace. Initially it shows the root and Session namespaces # and ⎕SE. The icon for # is open indicating that its contents are those that appear in the ListView. You can expand or collapse the TreeView of the workspace structure by clicking on the mini-buttons (labelled + and -) or by double-clicking the icons. A single click on a closed namespace icon opens it and causes its contents to be displayed in the ListView. Another way to open a namespace is to double-click its icon in the ListView. Only one namespace can be open at a time. The icons used in the display are described below.


Class


Namespace (closed)


GUI Namespace (closed)


Namespace (open)


GUI Namespace (open)


Function


Variable


Operator


Indicates an object that has been erased

Viewing and Arranging Objects

The ListView displays the contents of a namespace in one of four different ways namely *Large Icon* view, *Small Icon* view, *List* view or *Details* view. You can switch between views using the *View* menu or the tool buttons that are provided. In the first three views, the system displays the name of the object together with an icon that identifies its type. In *Details* view, the system displays several columns of additional information. You may resize the column widths by dragging or double-clicking the lines in the header. To hide a column, drag its width to the far left. The additional columns are:

Location	This is the namespace containing the object. By definition, this is the same for all of the objects shown in the ListView and is normally hidden
Description	For a function or operator, this is the function header stripped of localised names and comment. For a variable, the description indicates its rank, shape and data type. For a namespace, the description indicates the nature of the namespace; a plain namespace is described as namespace, a GUI Form object is described as Form, and so forth.
Size	The size of the object as reported by `⎕SIZE`.
Modified on	For functions and operators, this is the timestamp when the object was last fixed. For other objects this field is empty.
Modified by	For functions and operators, this is the name of the user who last fixed the object. For other objects this field is empty.

In any view, you may arrange the objects in ascending order of name, size, timestamp or class by clicking the appropriate tool button. In *Details* view, you may sort in ascending or descending order by clicking on the appropriate column heading. The first click sorts in ascending order; the second in descending order.

Moving and Copying Objects

You can move and copy objects from one namespace to another using drag-drop or from the Edit menu.

To *move* one or more objects using drag-and-drop editing:

1. Select the objects you want to move in the ListView.
2. Point to one of the selected objects and then press and hold down the left mouse button. When the drag-and-drop pointer appears, drag the object(s) to another namespace in the TreeView. To indicate which of the namespaces is the current target, its name will be highlighted as you drag the selected object(s) over the TreeView.
3. Release the mouse button to drop the objects into place. The objects will disappear from the ListView because they have been moved to another namespace.

To *copy* one or more objects using drag-and-drop editing, the procedure is the same except that you must press and hold the Ctrl key before you release the mouse button.

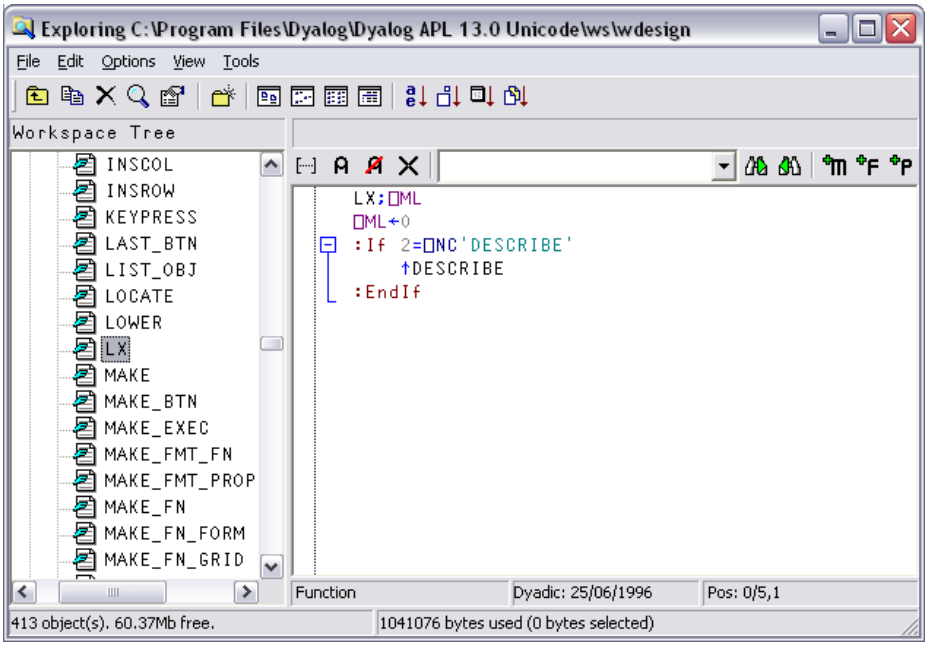
You may also move and copy objects using the *Edit* menu. To do so, select the object (s) and then choose *Move* or *Copy* from the *Edit* menu. You will be prompted for the name of the namespace into which the objects are to be moved or copied. Enter the namespace and click OK.

Editing and Renaming Objects

You can open up an edit window for a function or variable by double-clicking its icon, or by selecting it and choosing *Edit* from the *Edit* menu or from the popup menu. You may rename an object by clicking its name (as opposed to its icon) and then editing this text. You may also select the object and choose *Rename* from the *Edit* menu or from the popup menu. Note that when you rename an object, the original name is discarded. Unlike changing a function name in the editor, this is not a copy operation.

Using the Explorer as an Editor

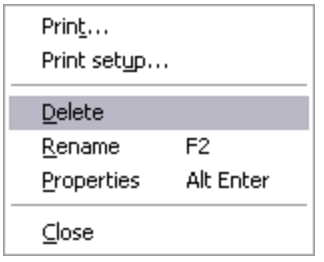
If you open the *Fns/Ops* item, the names of the functions and operators in the namespace are displayed below it alphabetically in the left (tree view) pane. When you select one of these names, the function itself is opened in the right (list view) pane.



You may use this feature to quickly cycle through the functions (or variables) in a namespace, pressing cursor up and cursor down in the left (tree view) pane to move from one to another.

You may also edit the function directly in the right (list view) pane before moving on to another.

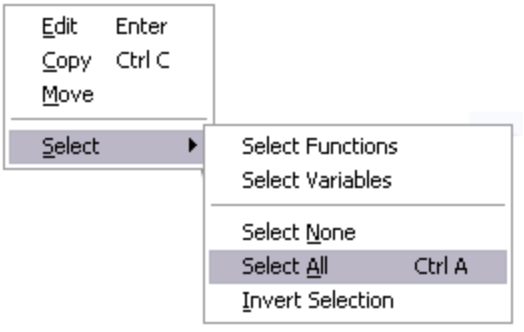
The File Menu



The *File* menu, illustrated above, provides the following actions. All but *Print* setup and *Close* act on the object or objects that are currently selected in the ListView.

Print	Prints the object(s).
Print setup	Displays the Print Configuration dialog box.
Delete	Erases the object(s).
Rename	Renames the object. This option only applies when a single object is selected.
Properties	Displays a property sheet; one for each object that is selected.
Close	Closes the Explorer

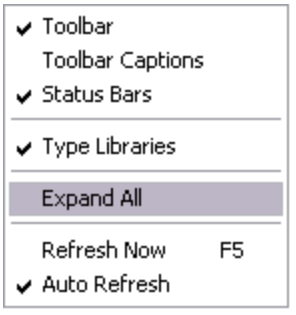
The Edit Menu



The *Edit* menu, illustrated above, provides the following actions. The *Edit*, *Copy* and *Move* operations act on the object or objects that are currently selected in the ListView.

Edit	Opens an edit window for each of the objects selected.
Copy	Prompts for a namespace and copies the object(s) there.
Move	Prompts for a namespace and moves the object(s) there.
Select Functions	Selects all of the functions and operators in the ListView.
Select Variables	Selects all of the variables in the ListView.
Select None	Deselects all of the objects in the ListView.
Select All	Selects all of the objects in the ListView.
Invert Selection	Deselects the selected objects and selects all those that were not selected.

The Options Menu

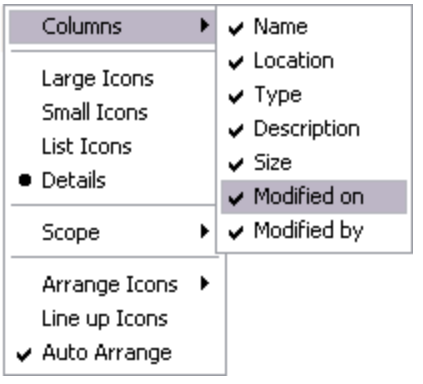


The Options menu, illustrated above, provides the following actions.

Toolbar	Displays or hides the Explorer toolbar.
Toolbar Captions	Displays or hides the button captions on the Explorer toolbar.
StatusBar	Displays or hides the Explorer statusbar.
Type Libraries	Enables/disables the exploring of Type Libraries
Expand All	Expands all namespaces and sub-namespaces in the TreeView, providing a complete view of the workspace structure, including or excluding the Session object ⎕SE.
Refresh Now	Redisplays the TreeView and ListView with the current structure and contents of the workspace. Used if *Auto Refresh* is not enabled.
Auto Refresh	Specifies whether or not the Explorer immediately reflects changes in the active workspace.

If *Auto Refresh* is checked the Explorer is updated every time APL returns to desk-calculator mode. This means that it is always in step with the active workspace. If you have a large number of objects displayed in the Explorer, the update may take a few seconds and you may wish to prevent this by un-checking this menu item If you do so, the Explorer must be explicitly updated by selecting the *Refresh Now* action.

The View Menu



The View menu, illustrated above, provides the following actions.

Columns	Allows you to select which columns you wish to display.
Large Icons	Selects *Large Icon* view in the ListView.
Small Icons	Selects *Small Icon* view in the ListView.
List Icons	Selects *List* view in the ListView.
Details	Selects *Details* view in the ListView.
Scope	Allows you to choose whether the Explorer displays objects in local scope or in global scope.
Arrange Icons	Sorts the items in the ListView by name, type, size or date.
Line up Icons	Rearranges the icons into a regular grid.
Auto Arrange	If checked, the icons are automatically re-arranged when appropriate

.

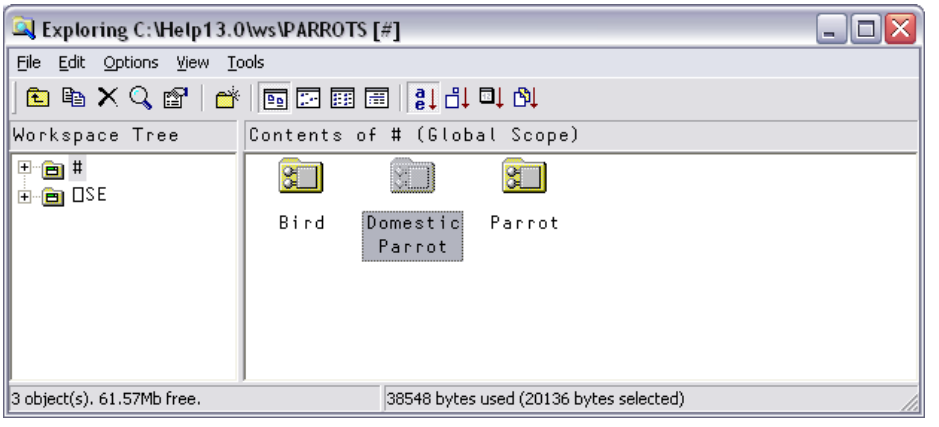
The Tools Menu

The *Tools* menu, illustrated above, provides the following actions.

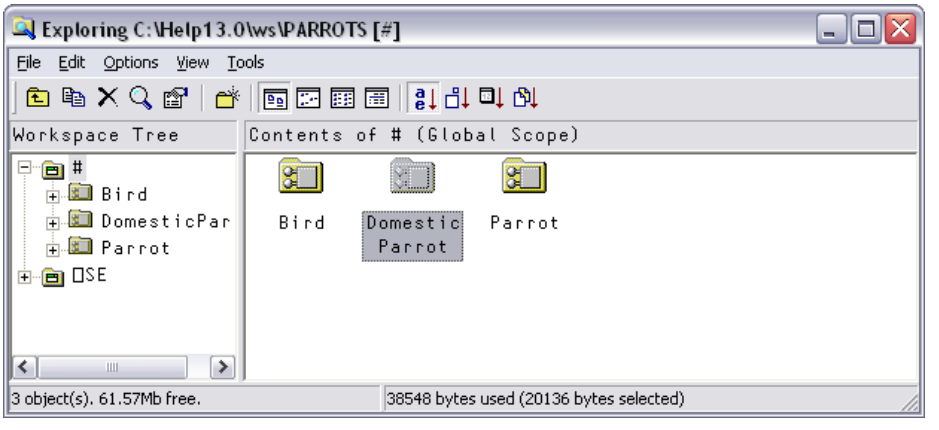
Find	Displays the Find Objects Tool
Go to	Prompts for a namespace and then opens that namespace in the TreeView, displaying its contents in the ListView
Go to Session Space	Opens the namespace in the TreeView control corresponding to the current space in the Session.
Set Session Space	Sets the current space in the Session to be the namespace that is currently open in the TreeView.

Browsing Classes

Classes are represented by  icons. The picture below shows 3 classes: `Bird`, `Parrot` and `DomesticParrot`.

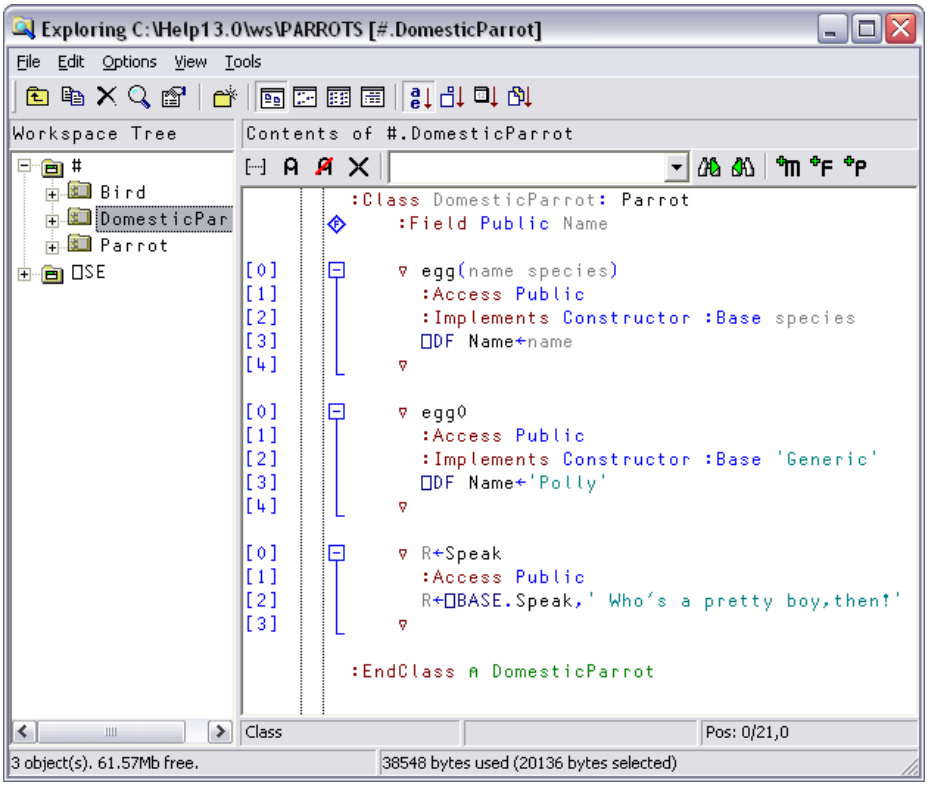


If you open the `#` node in the left-hand pane, you see the contents of `#` as a tree.

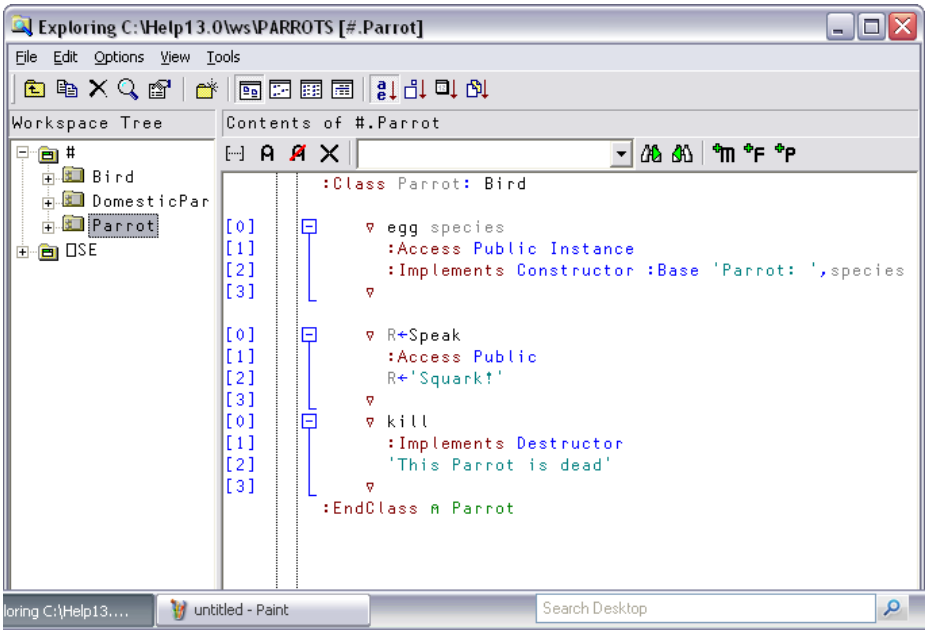


Browsing Class Scripts

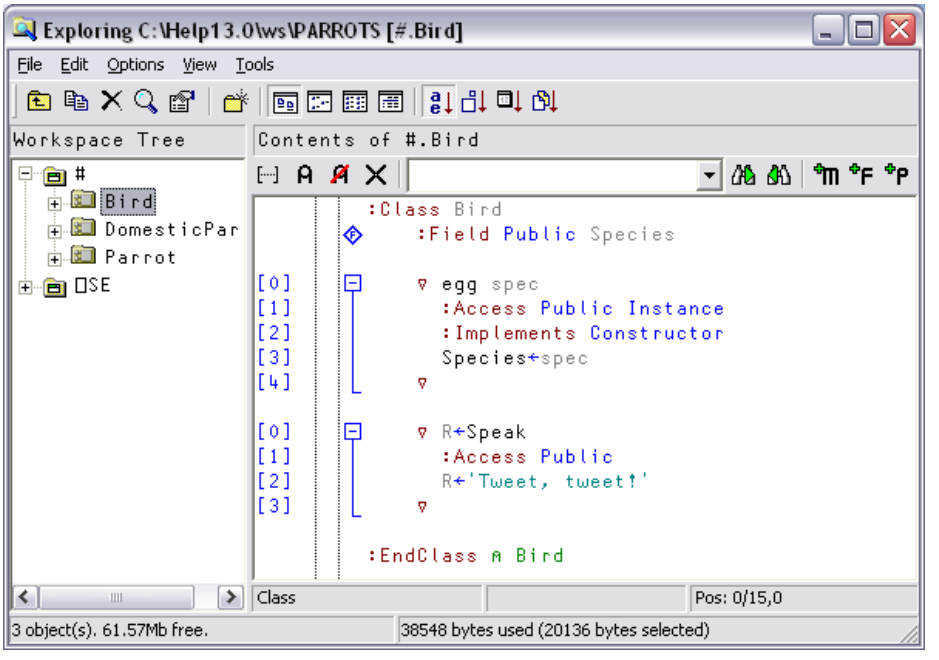
Selecting `DomesticParrot` in the left-hand pane brings up its Class Script in the right-hand pane.



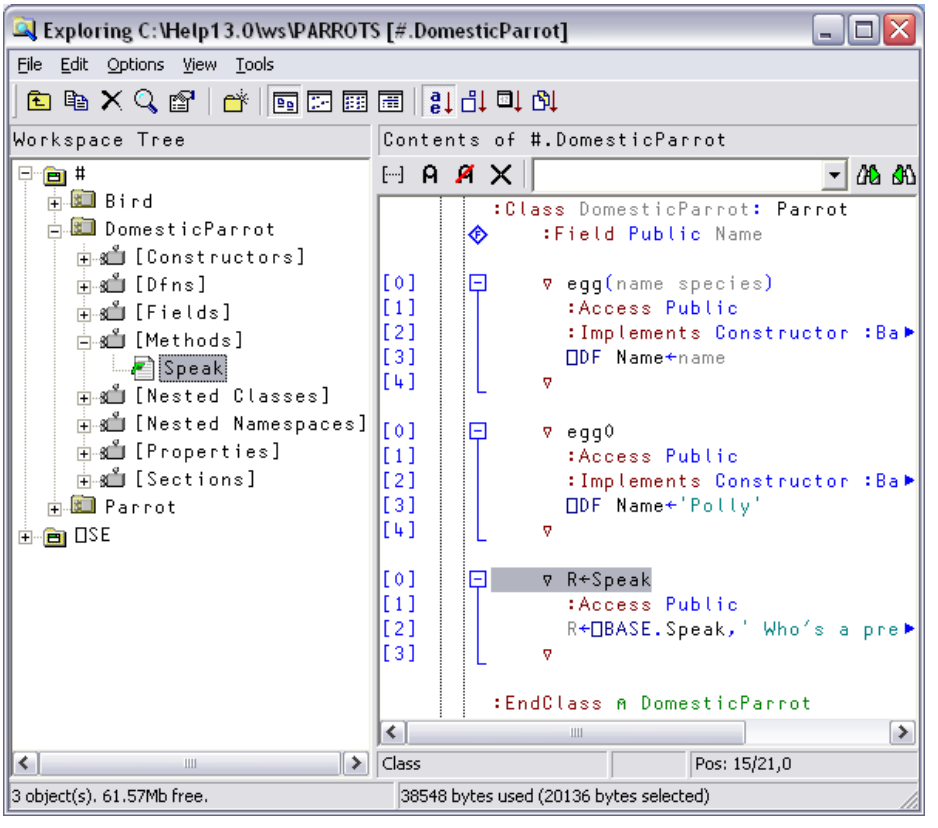
… and selecting `Parrot` in the left-hand pane brings up the Class Script for `Parrot`.



… and finally, selecting `Bird` in the left-hand pane brings up the Class Script for `Bird`.



If you open a Class node, a tree appears to help you to navigate within the Class script. In the picture below, the user has opened the `[Methods]` node and then clicked on `Speak`. The system has responded by scrolling to (if necessary) and highlighting the appropriate section of the script.



Browsing Type Libraries

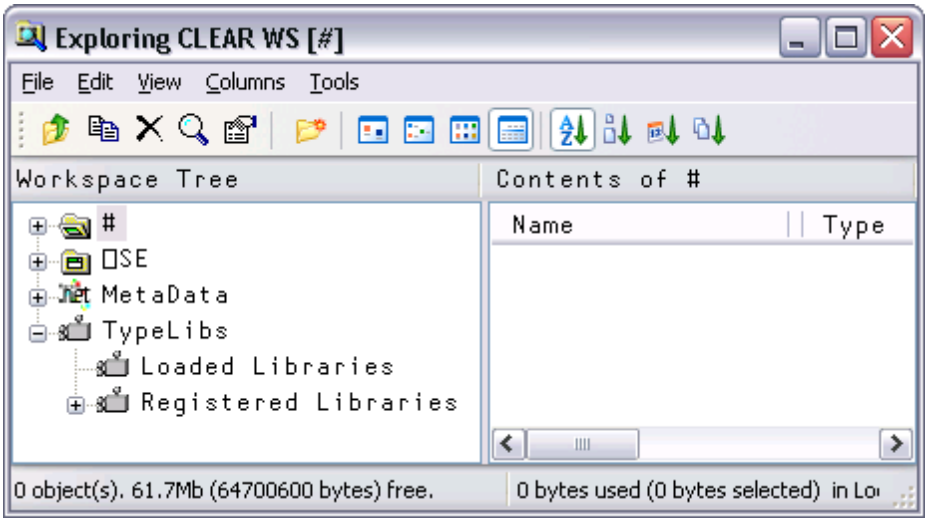
When the *View/Type Libraries* option is enabled, the *Workspace Explorer* allows you to:

- Browse the Type Libraries for all the COM server objects that are installed on your computer, whether or not they are loaded in your workspace.
- Load Type Libraries for COM objects
- Browse the Type Library associated with an OLEClient object that is already instantiated in the workspace.

If the Microsoft .NET Framework is installed, you may in addition:

- Load Metadata for specific .NET classes
- Browse the loaded Metadata, viewing information about classes, methods, properties and so forth.

If the *Type Libraries* option is enabled, the *Workspace Explorer* displays a folder labelled *TypeLibs* which, when opened, displays two others labelled *Loaded Libraries* and *Registered Libraries* as shown below.

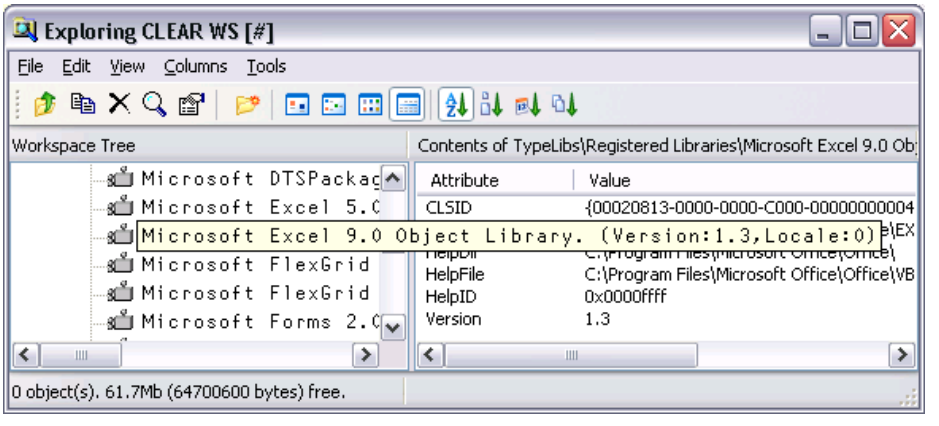


Browsing Registered Libraries

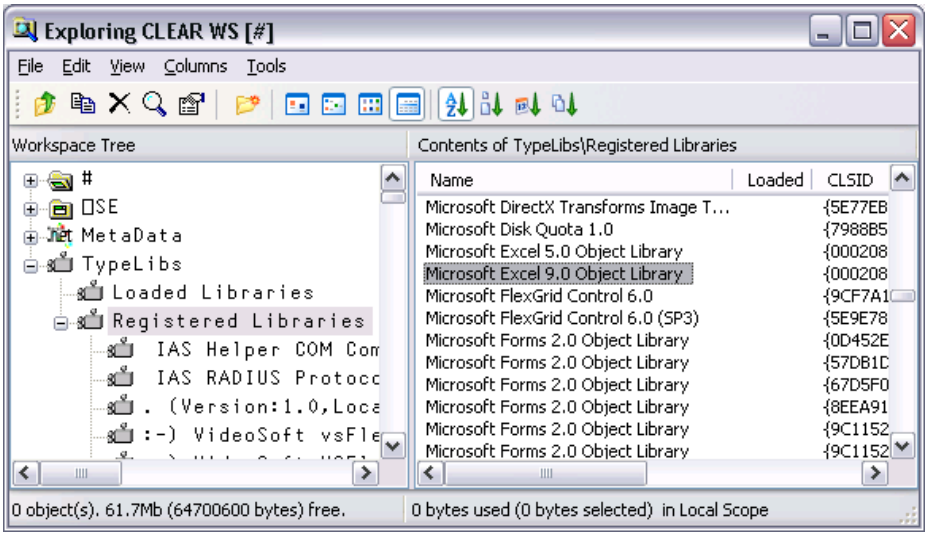
If you open the Registered Libraries folder, the *Workspace Explorer* will display in the tree view pane the names of all the Type Libraries associated with the COM Server objects that are installed on your computer.

If you select one of these Library names, some summary information is displayed in the list view pane.

For example, the result of selecting the Microsoft Excel 9.0 Object Library is illustrated below.



If instead, you select the Registered Libraries folder itself, the list of Registered Type Libraries is displayed in the list view pane



Loading a Type Library

You can load a library shown in the list view pane by double-clicking its name.

Alternatively, you can load a library shown in the tree view pane by selecting *Load* from its context menu.

In either case, a message box will appear asking you to confirm. The operation to load a Type Library may take a few moments to complete.

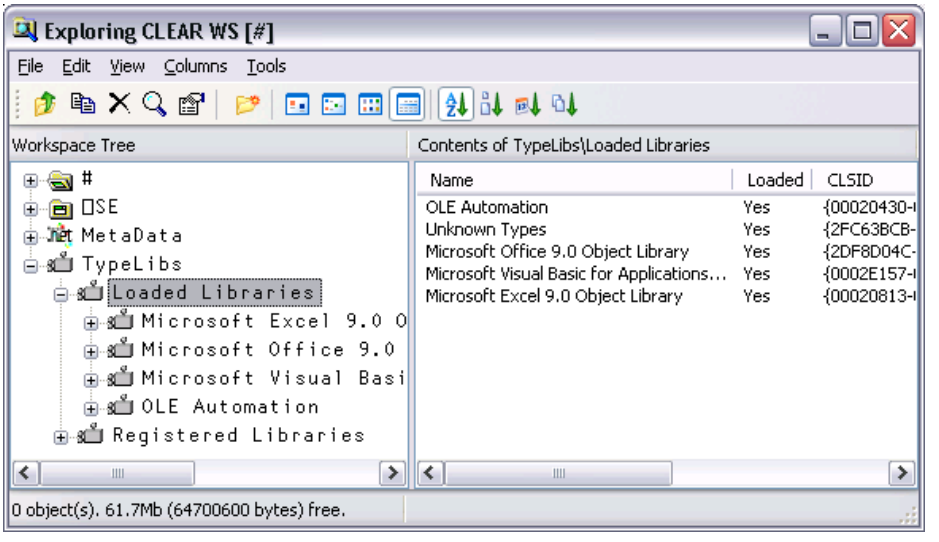
Notice that if the selected Library references any other libraries, they too will be loaded. For example, loading the *Microsoft Excel 9.0 Object Library* brings in the *Microsoft Office 9.0 Object Library* and the *Microsoft Visual Basic for Applications Extensibility 5.3 Library* too. It also contains references to a general library called the *OLE Automation Type Library*, so this is also loaded.

When you `)SAVE` your workspace, all of the Type Libraries that you have loaded will be saved with it. Note that type library information can take up a considerable amount of workspace.

Browsing Loaded Libraries

If you have already loaded any Type Libraries into the workspace, using the Workspace Explorer or as a result of creating one or more OLEClient objects, you can select and open the Loaded Libraries folder.

The picture below illustrates the effect of having loaded the Microsoft Excel 9.0 Object Library.



Notice that any external references to other libraries causes these to be brought in too.

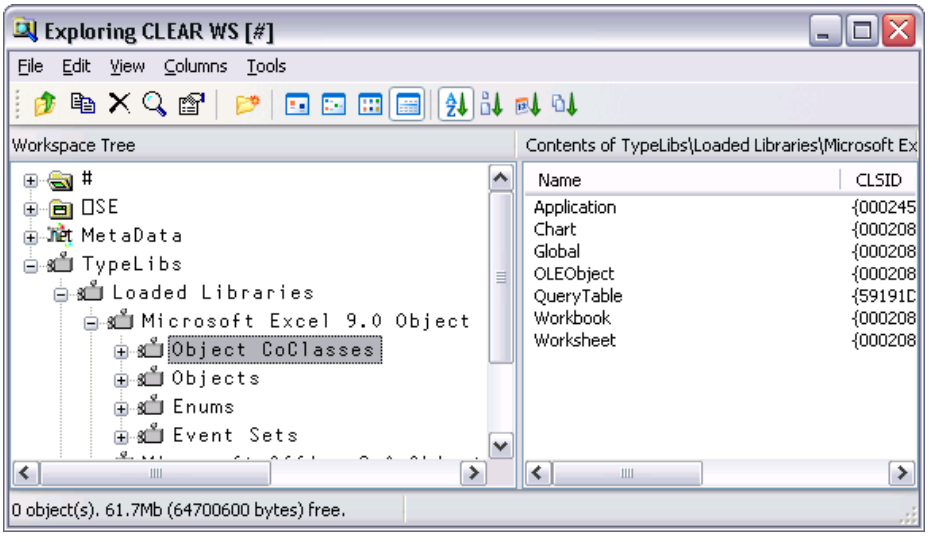
If you select a loaded Type Library, summary information is displayed in the list view pane.

If you open a loaded Type Library, four sub-folders appear named *Object CoClasses*, *Objects*, *Enums* and *Event Sets* respectively.

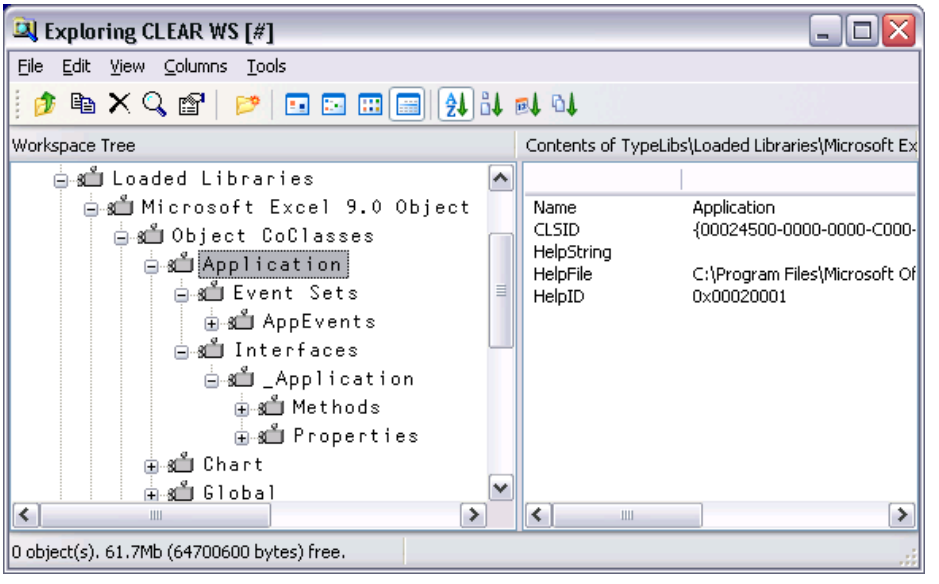
Object CoClasses

A Type Library describes a number of *objects*. Typically, all of the objects have properties and methods, but only some of them, perhaps just a few, generate events. Objects which generate events are represented by *CoClasses*, each of which has a pointer to the object itself and a pointer to an event set.

For example, the Microsoft Excel 9.0 Object Library contains seven CoClasses named *Application*, *Chart*, *Global* etc as shown below.



Opening the Application folder you can see that the *Application* CoClass comprises the *_Application* object coupled with the *AppEvents* event set as shown below.

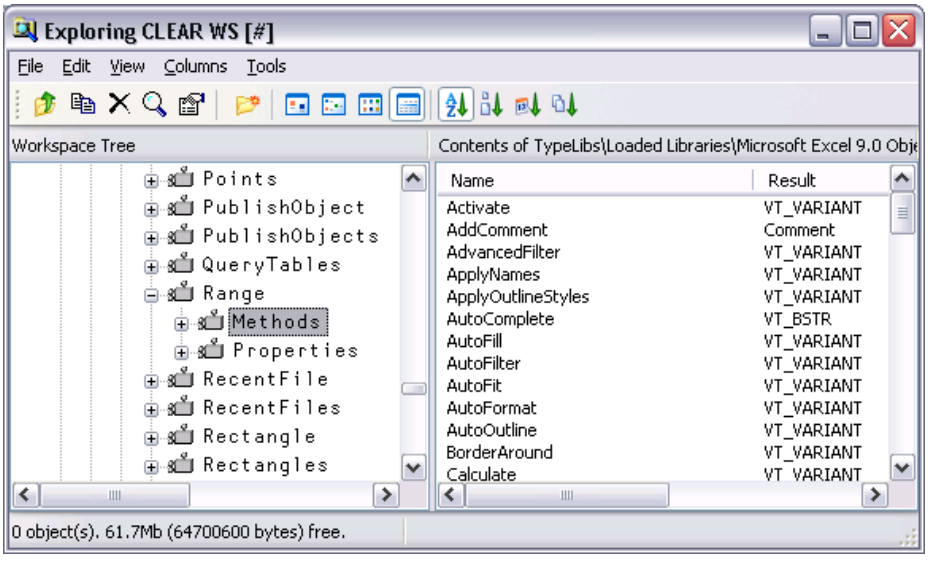


The specific methods, properties and events supported by the CoClass object can be examined by opening the appropriate sub-folder. The same information for these and other objects is also accessible from the *Objects* and *Event Sets* folders as discussed below.

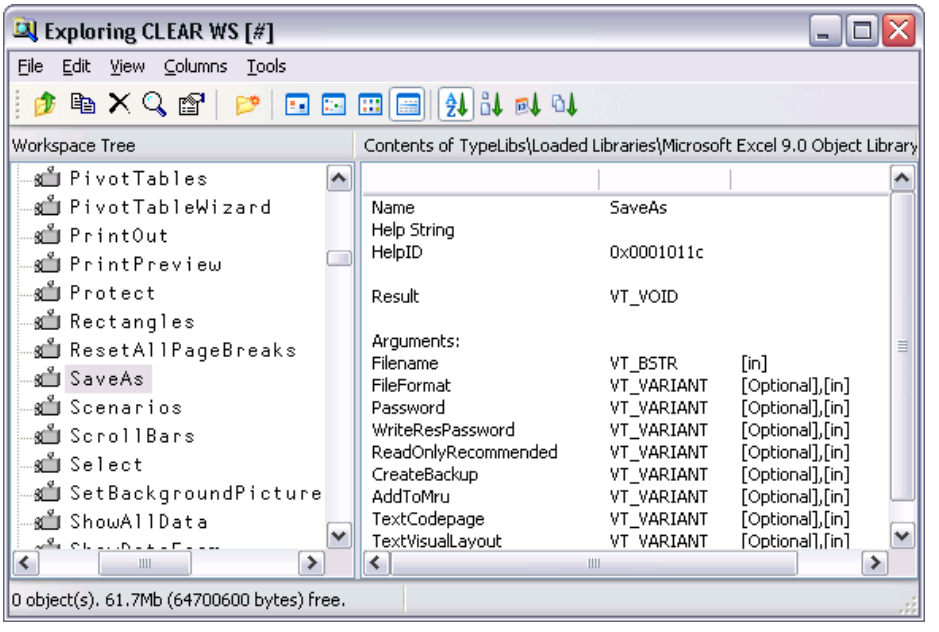
Objects

The *Objects* folder contains several sub-folders each of which represents a named object defined in the library.

Each object folder contains two sub-folders named Methods and Properties. Selecting one of these causes the list of Methods or Properties to be displayed in the list view pane. The picture below shows the Methods exposed by the Microsoft Excel 9.0 Range object.

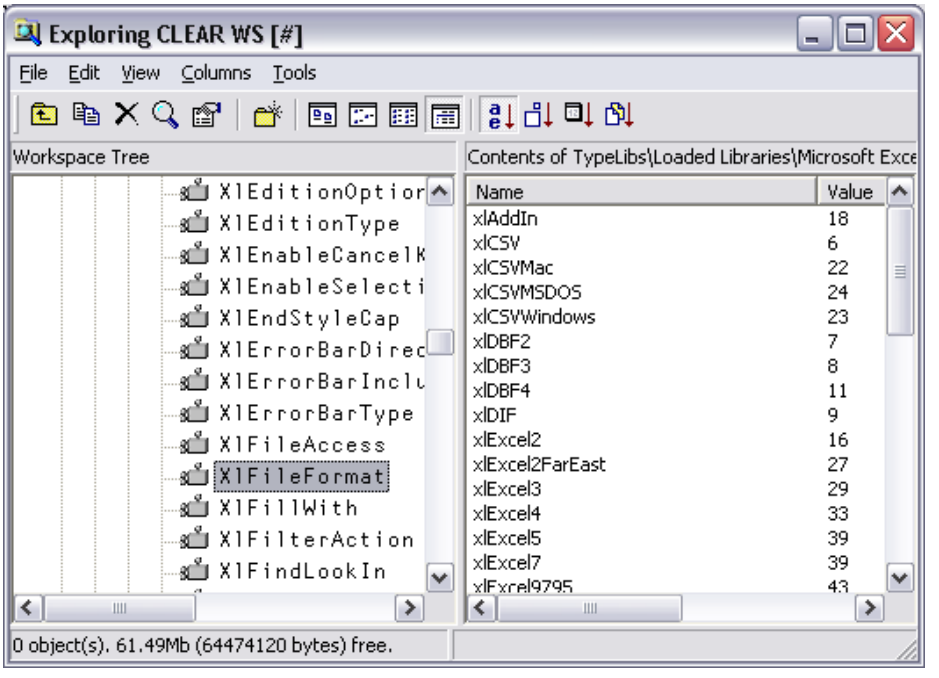


If you open the *Methods* or *Properties* subfolder, you can display more detailed information about individual Methods and Properties. For example, the following picture shows information about the *SaveAs* method exposed by the Microsoft Excel 9.0 Worksheet object.



This tells you that the *SaveAs* method takes up to 9 parameters of which the first, Filename, is mandatory and is of data type VT_BSTR (a character string). Note that [in] indicates that the parameter is an *input* parameter.

Incidentally, the optional Fileformat parameter is an example of a parameter whose value must be one of a list of Enumerated Constants. Even without looking at the documentation, the possible values can be deduced by browsing the *Enums* folder, with the results shown below.



You can therefore deduce that the following expression, executed in the namespace associated with the currently active worksheet, will save the sheet in comma-separated format (CSV) in a file called mysheet.csv:

```
SaveAs 'MYSHEET.CSV' xlCSV
```

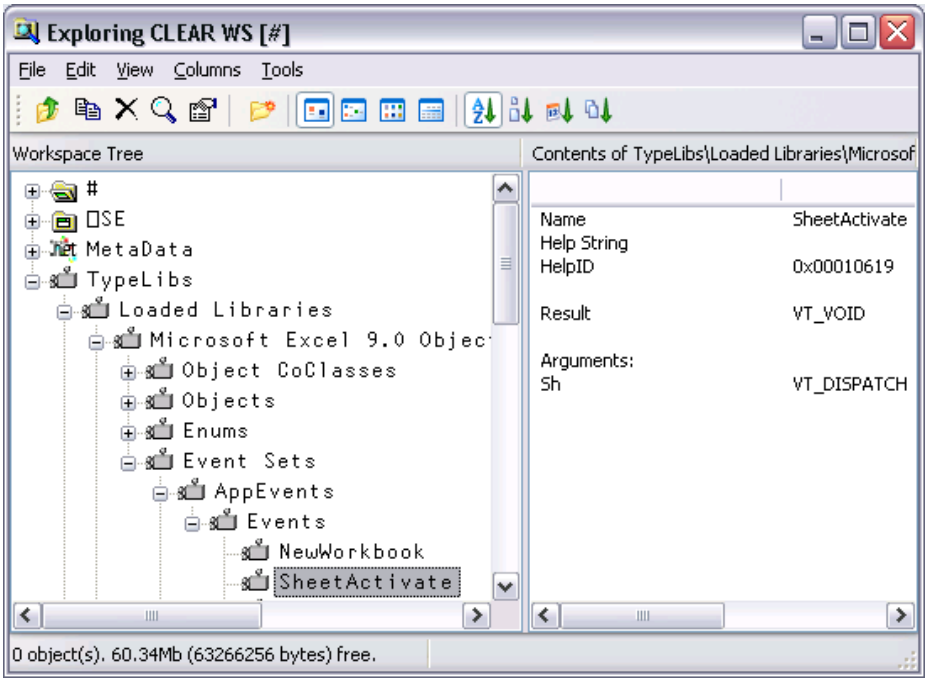
or

```
SaveAs 'MYSHEET.CSV' 6
```

Event Sets

The *Event Sets* folder contains several sub-folders each of which represents a named set of events generated by the objects defined in the library.

If you open one of these event sets, the names of the events it contains are displayed in the tree view pane. If you then select one of the events, its details are displayed in the list view pane as shown below.

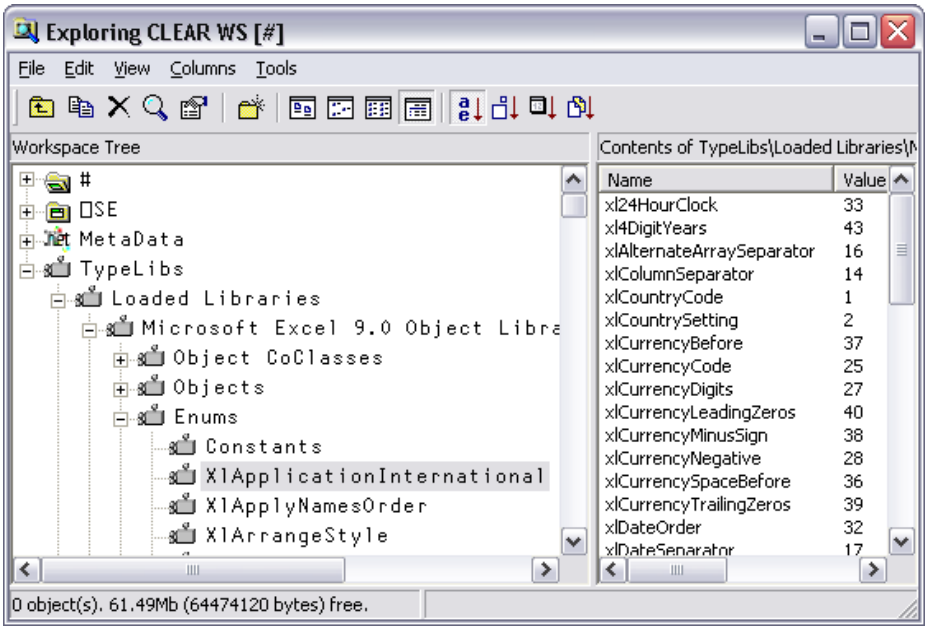


This example shows that when it fires, the SheetActivate event invokes your call-back function with a single argument named *Sh* whose datatype is VT_DISPATCH (in practice, a Worksheet object).

Enums

The *Enums* folder will typically contain several sub-folders each of which represents a named set of enumerated constants.

If you select one of these sets, the names and values of the constants it contains are displayed in the list view pane as shown below.



Browsing .NET Classes

Microsoft supplies a tool for browsing .NET Class libraries called `ILDASM.EXE`[1].

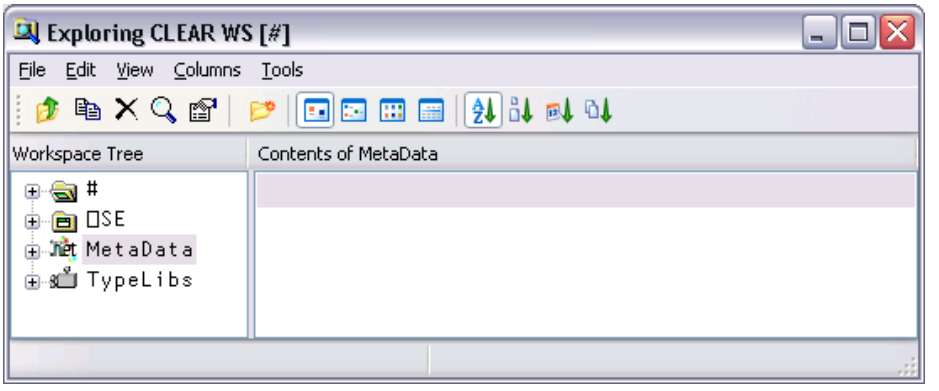
As a convenience, the Dyalog APL Workspace Explorer has been extended to perform a similar task as `ILDASM` so that you can gain access to the information within the context of the APL environment.

The information that describes .NET classes, which is known as its *Metadata*, is part of the definition of the class and is stored with it. This Metadata corresponds to Type Information in COM, which is typically stored in a separate Type Library.

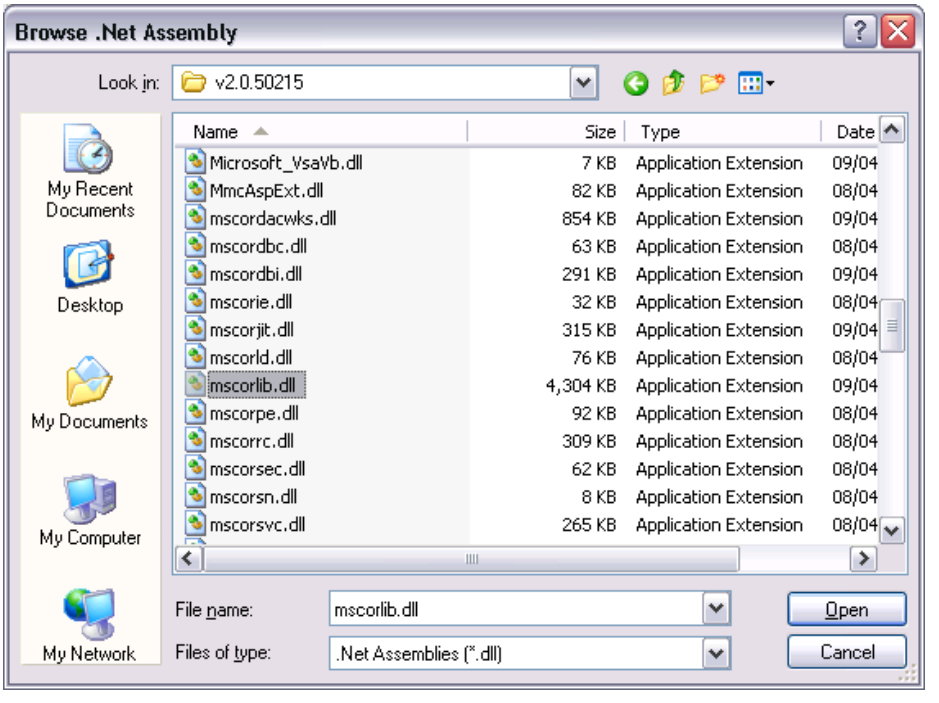
To enable the display of Metadata in the Workspace Explorer, you must have the *Type Libraries* option of the *View* menu checked.

[1] ILDASM.EXE can be found in the .NET SDK and is distributed with Visual Studio

To gain information about one or more .NET Classes, open the Workspace Explorer, right click the *Metadata* folder, and choose *Load*.



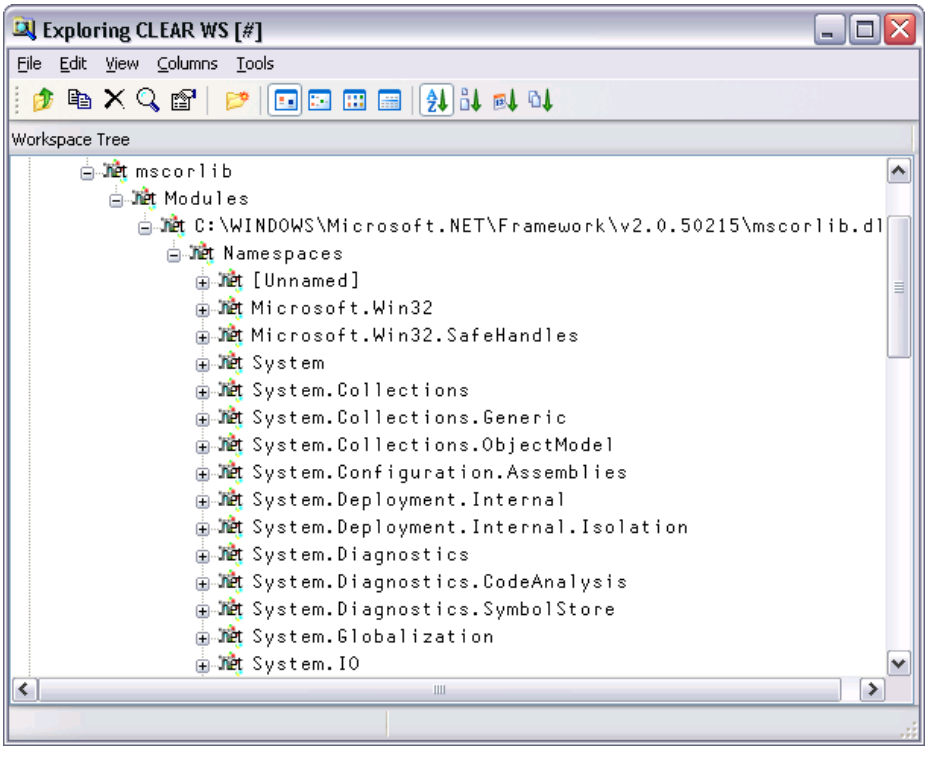
This brings up the *Browse .Net Assembly* dialog box as shown below. Navigate to the .NET assembly of your choice, and click *Open*.



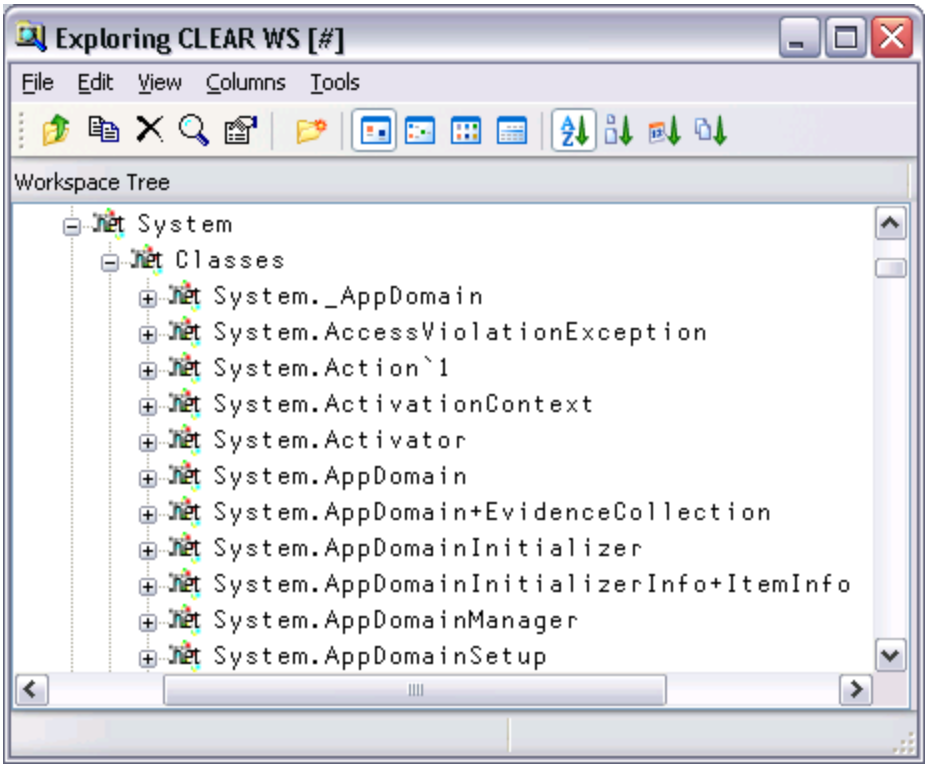
Note that the .NET Classes provided with the .NET Framework are typically located in `C:\WINDOWS\Microsoft.NET\Framework\V2.0.50215`. The last named folder is the Version number.

The most commonly used classes of the .NET Namespace System are stored in this directory in an Assembly named `mscorlib.dll`, along with a number of other fundamental .NET Namespaces.

The result of opening this Assembly is illustrated in the following screen shot. The somewhat complex tree structure that is shown in the Workspace Explorer merely reflects the structure of the Metadata itself.

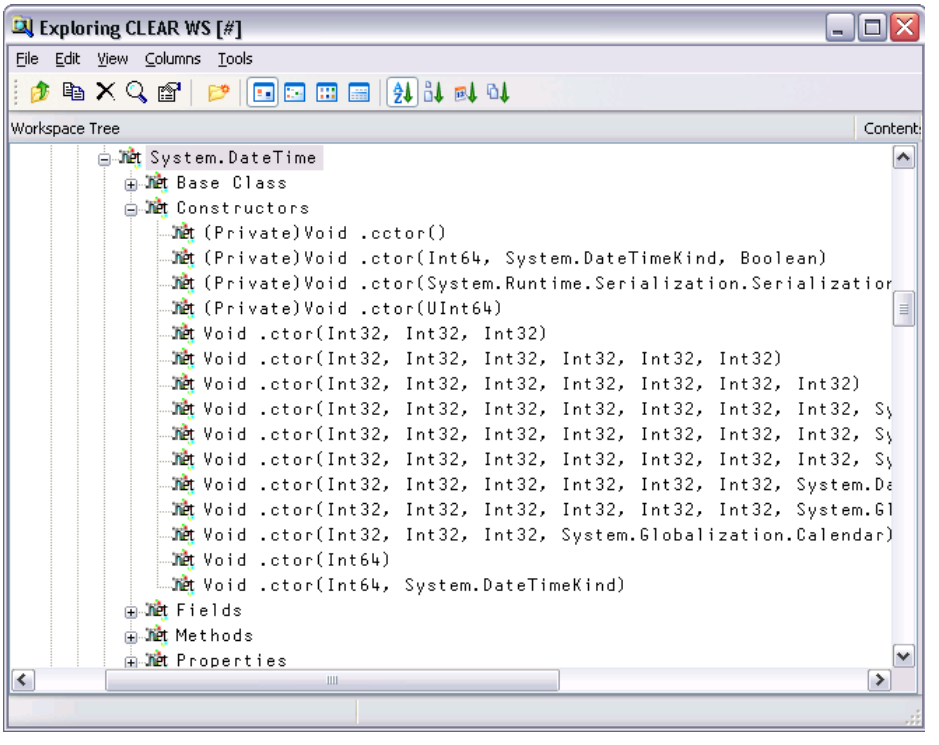


Opening the *System/ Classes* sub-folder causes the Explorer to display the list of classes contained in the .NET Namespace *System* as shown in the picture below.



The *Constructors* folder shows you the list of all of the valid constructors and their parameter sets with which you may create a new instance of the Class by calling `New`. The constructors are those named *.ctor*; you may ignore the one named *.cctor*, (the class constructor) and any labelled as *Private*.

For example, you can deduce that `DateTime.New` may be called with three numeric (`Int32`) parameters, or six numeric (`Int32`) parameters, and so forth. There are in fact seven different ways that you can create an instance of a `DateTime`.

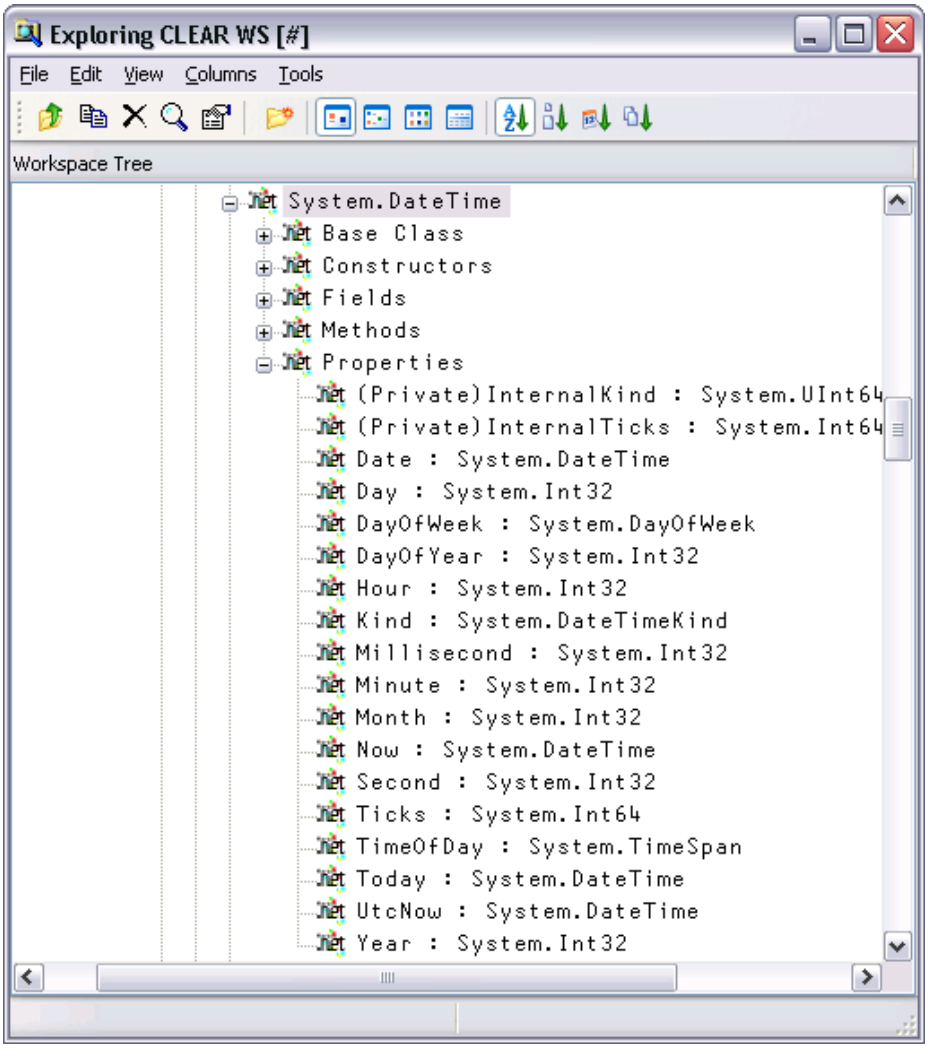


For example, the following statement may be used to create a new instance of `DateTime` (09:30 in the morning on 30th April 2001):

```
      mydt←⎕NEW DateTime (2001 4 30 9 30 0)

      mydt
30/04/2001 09:30:00
```

The *Properties* folder provides a list of the properties supported by the Class. It shows the name of the property followed by its data type. For example, the `DayOfYear` property is defined to be of type `Int32`.



You can query a property by direct reference:

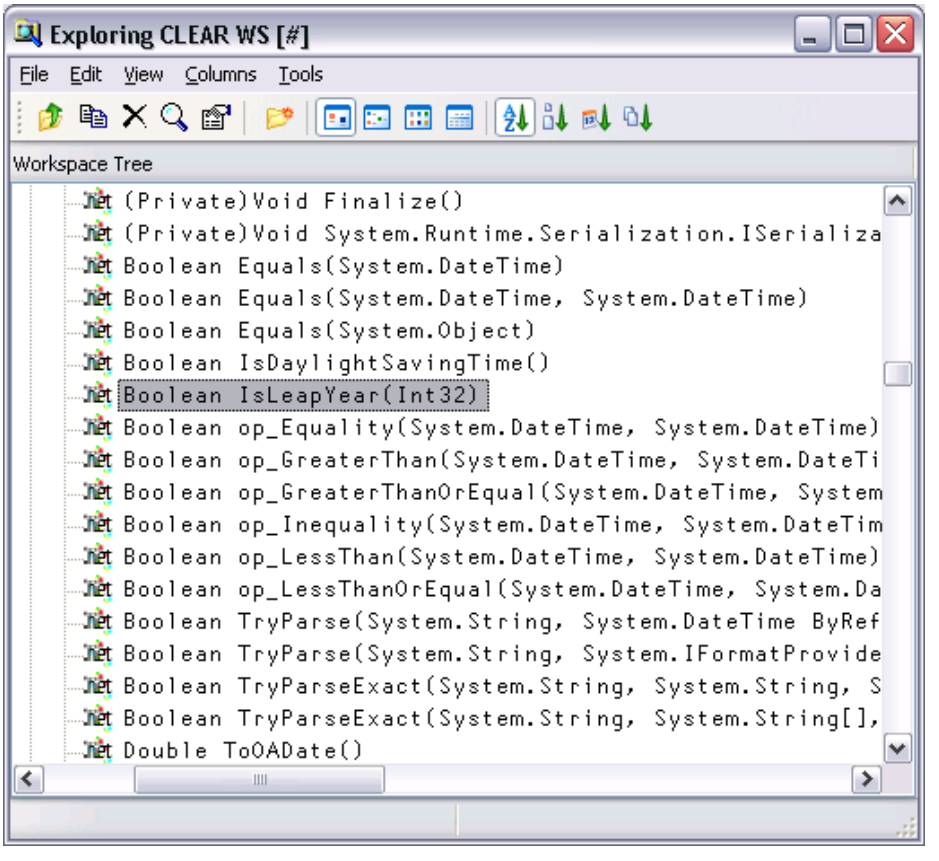
```
      mydt.DayOfWeek
Monday
```

Notice too that the data types of some properties are not simple data types, but Classes in their own right. For example, the data type of the `Now` property is itself `System.DateTime`. This means that when you reference the `Now` property, you get back an object that represents an instance of the `System.DateTime` object:

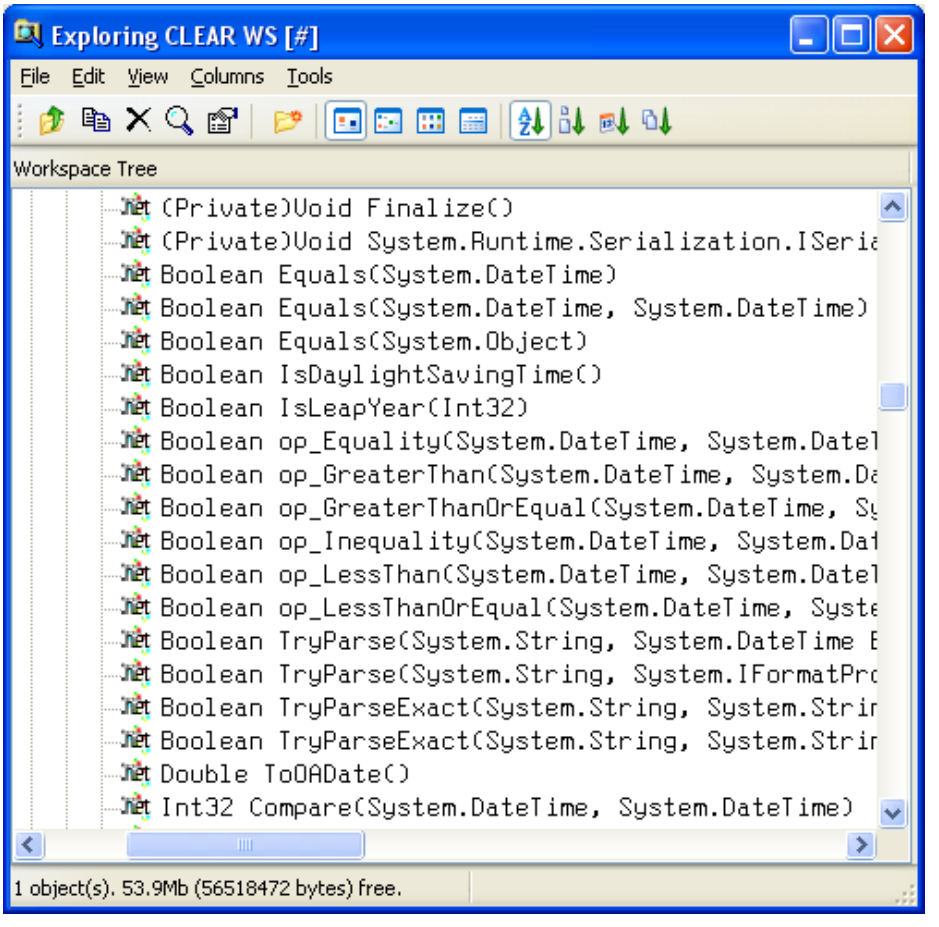
```
      mydt.Now
07/11/2001 11:30:48
      ⎕TS
2001 11 7 11 30 48 0
```

The *Methods* folder lists the methods supported by the Class. The Explorer shows the data type of the result of the method, followed by the name of the method and the types of its arguments. For example, the `IsLeapYear` method takes an `Int32` parameter (year) and returns a `Boolean` result.

```
      mydt.IsLeapYear 2000
1
```



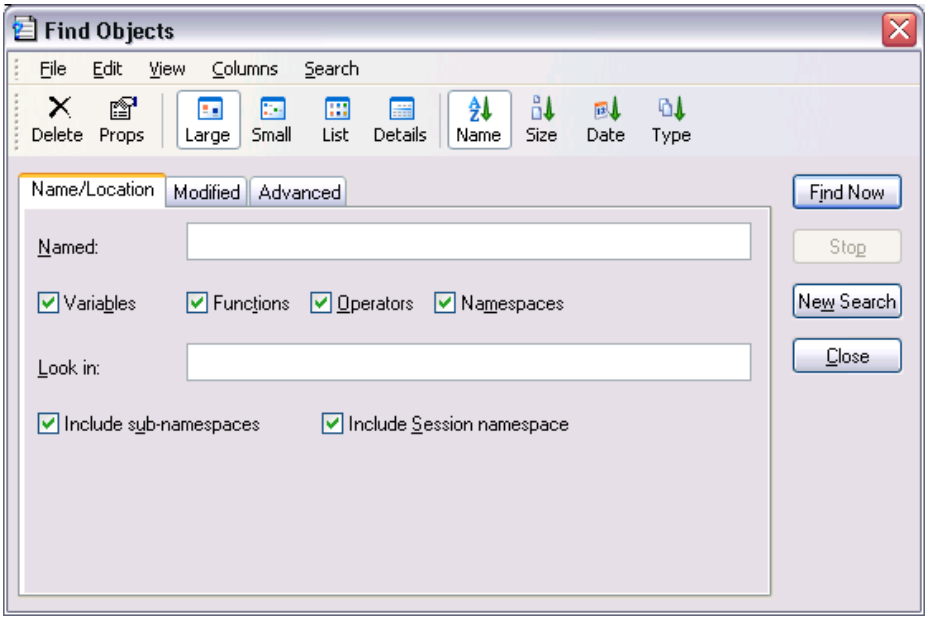
Many of the reported objects are listed as *Private*, which means they are inaccessible to users of the class – you are not able to call them or inspect their value. For more information about classes, see the chapter on *Object Oriented Programming* in the *Dyalog APL Language Reference Manual*.



Find Objects Tool

The *Find Objects* tool is a modeless dialog box that may be toggled on and off by the system action `[WSSearch]`. In a default Session, this is attached to a MenuItem in the Tools menu and a Button on the session toolbar. This tool allows you to search the active workspace for objects that satisfy various criteria.

The first page allows you to specify the name of the object which you wish to find and the namespace(s) in the workspace that are to be searched for it.

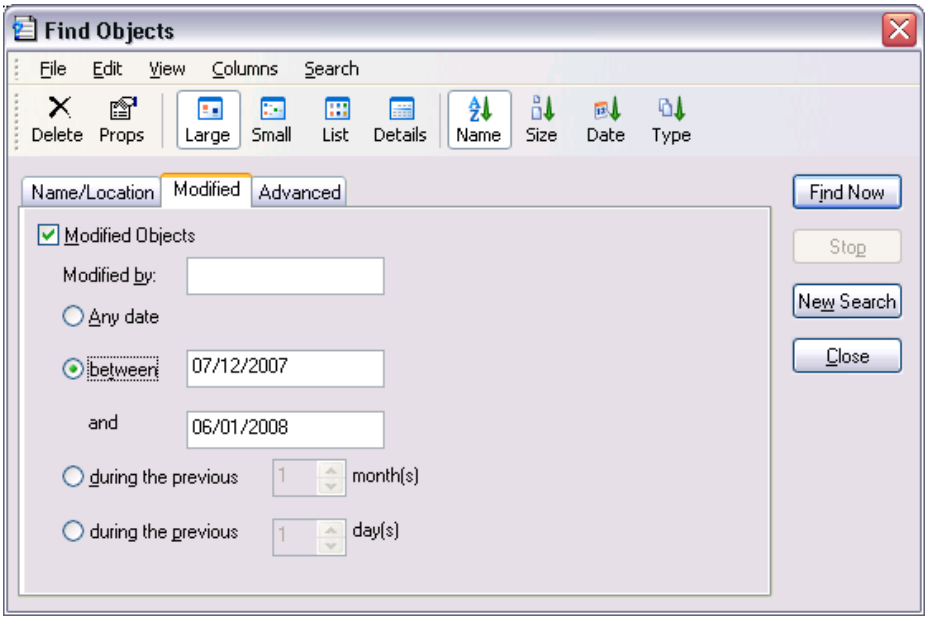


You type the name of the object you wish to find into the field labelled *Named*. To locate all objects beginning with a particular string, enter the string followed by a '`*`' character. For example, if you enter the string `FOO*`, the system will locate all objects whose name begins with `FOO`.

Four check boxes are provided for you to specify the types of objects you wish to locate. For example, if you clear *Variables*, *Operators* and *Namespaces*, the system will only search for functions.

You can restrict the search to a particular namespace by typing its name into the field labelled *Look in*. You can also restrict the search by clearing the *Include sub-namespaces* and *Include Session namespace* check boxes. Clearing the former restricts the search to the root namespace or to the namespace that you have specified in *Look in*, and does not search within any sub-namespaces contained therein. Clearing the latter causes the system to ignore `⎕SE` in its search.

The second page, labelled *Modified*, allows you to search for objects that have been modified by a particular user or at a certain time

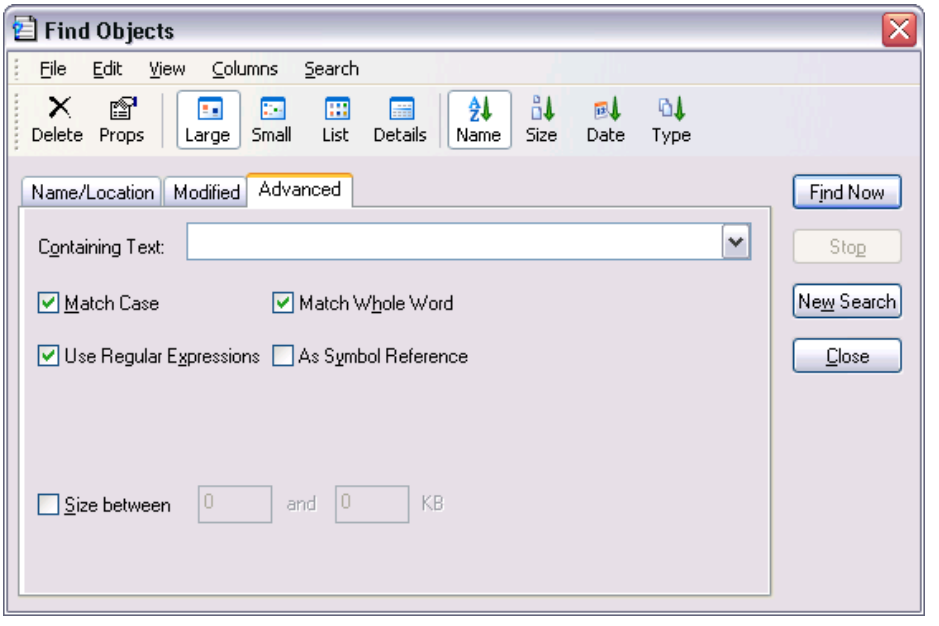


To make the search dependent upon modification, you must check the *Modified Objects* check box.

To locate objects modified by a particular user, enter the user name in the field labelled *Modified by*. Otherwise leave this blank.

To find objects which have been modified at a certain time or within a specified period of time, check the appropriate radio button and enter the appropriate dates or time spans.

The third page, labelled *Advanced*, allows you to search for objects that contain a particular text string.



If you wish to search for objects containing a particular character string, type the string into the field labelled *Containing Text*.

Match Case specifies whether or not the text search is case sensitive.

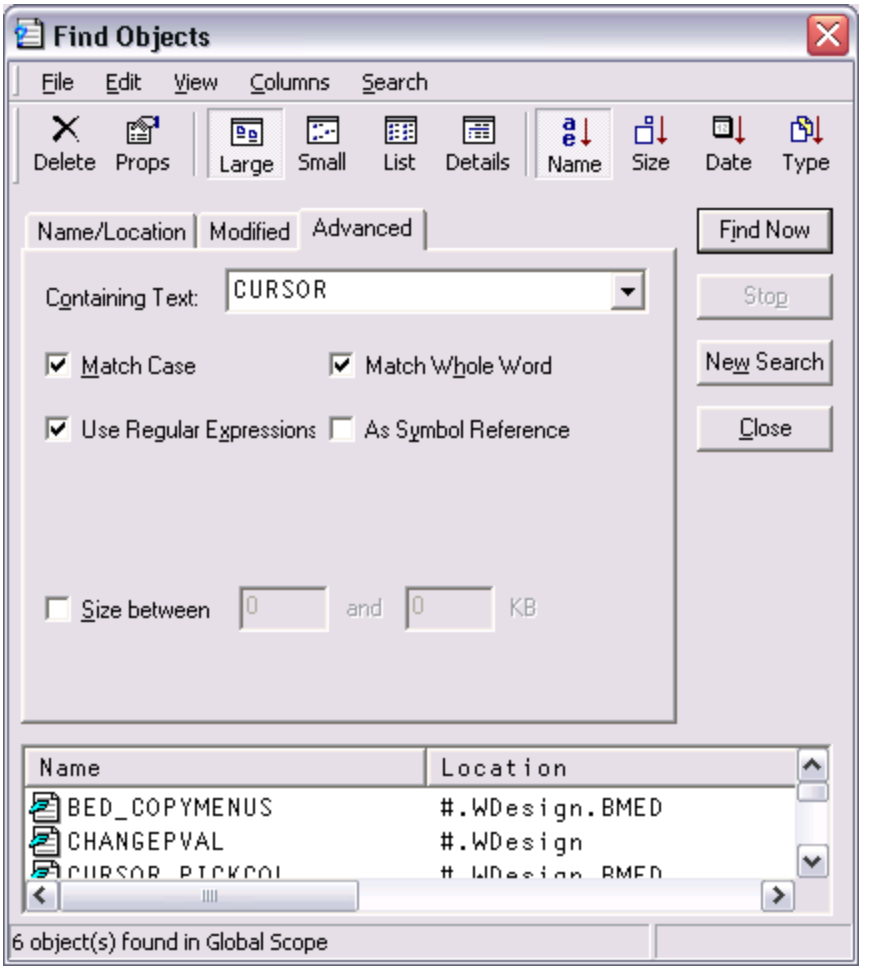
Use Regular Expressions specifies whether or not regular expressions are applicable. For example, if you enter `FOO*` into the field labelled *Containing Text* and check this box, the system will find objects that contain any text string starting with the 3 characters `FOO`. If this box is not checked, the system will find objects that contain the 4 characters `FOO*`.

Match Whole Word specifies whether or not the search is restricted to entire words.

As Symbol Reference specifies whether or not the search is restricted to APL symbols. If so, matching text in comments and other strings is ignored.

If you wish to restrict the search to find only objects whose size is within a given range, check the box labelled *Size* is between and enter values into the fields provided.

When you press the *Find Now* button, the system searches for objects that satisfy all of the criteria that you have specified on all 3 pages of the dialog box and displays them in a ListView. The example below illustrates the result of searching the workspace for all functions containing references to the symbol `CURSOR`.



You may change the way in which the objects are displayed in the ListView using the *View* menu or the tool buttons, in the same manner as for objects displayed in the Workspace Explorer. You may also edit, delete and rename objects in the same way. Furthermore, objects can be copied or moved by dragging from the ListView in the Search tool to the TreeView in the Explorer.

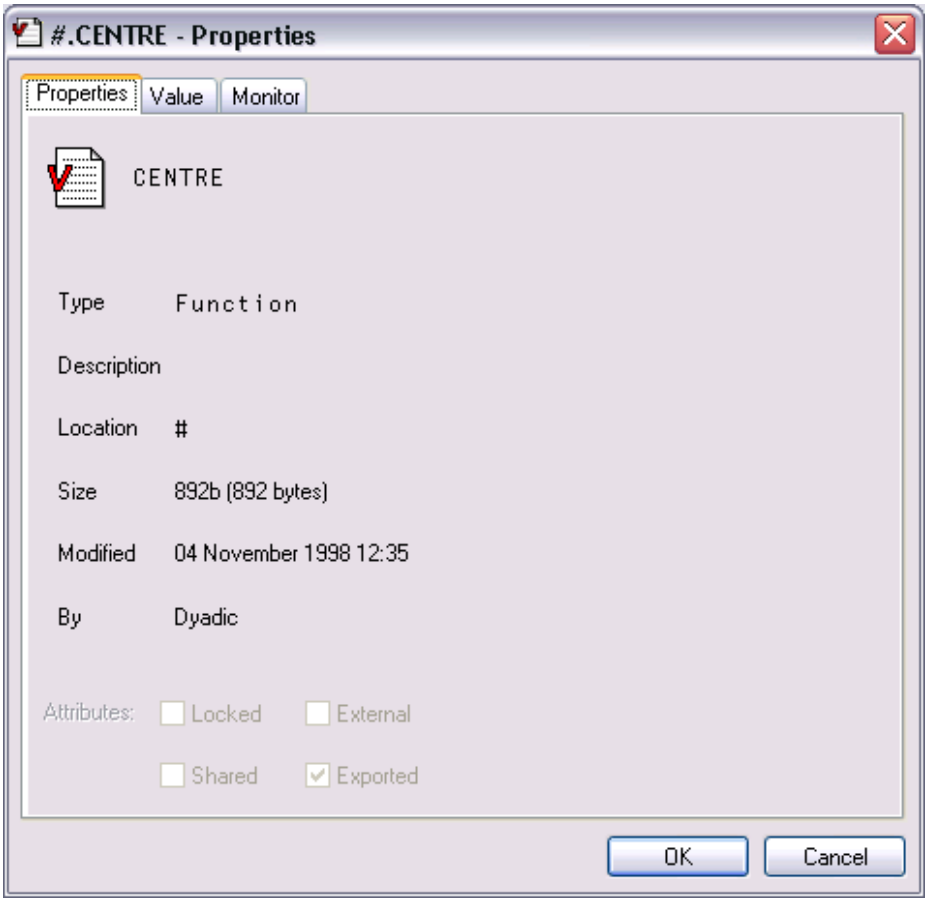
If you wish to specify a completely new set of criteria, press the *New Search* button. This will reset all of the various controls on the 3 pages of the dialog box to their default values.

Object Properties Dialog Box

The Object Properties dialog box displays detailed information for an APL object. It is displayed by executing the system action `[ObjProps]`. In a default Session, this is provided in the *Tools* menu, the Session popup menu and from the Explorer. An example (for a function) is shown below.

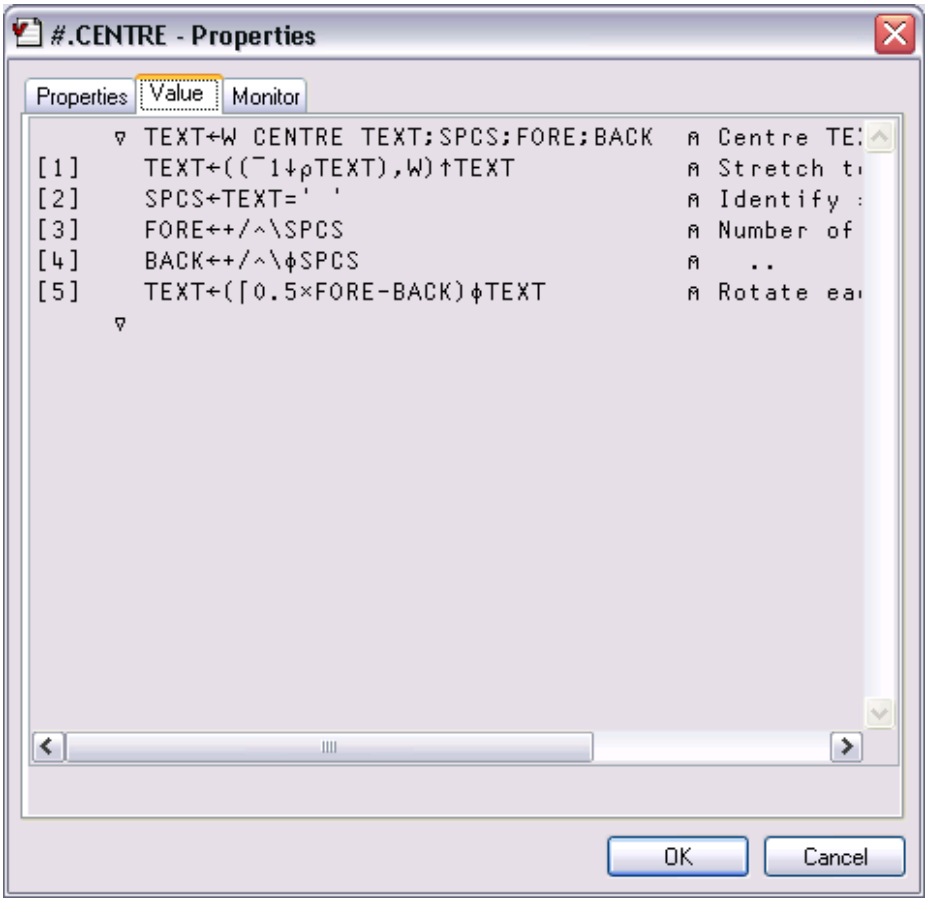
Properties Tab

The *Properties* tab displays general information about the object. For a function, this includes an extract from its header line, when it was last modified, and by whom.



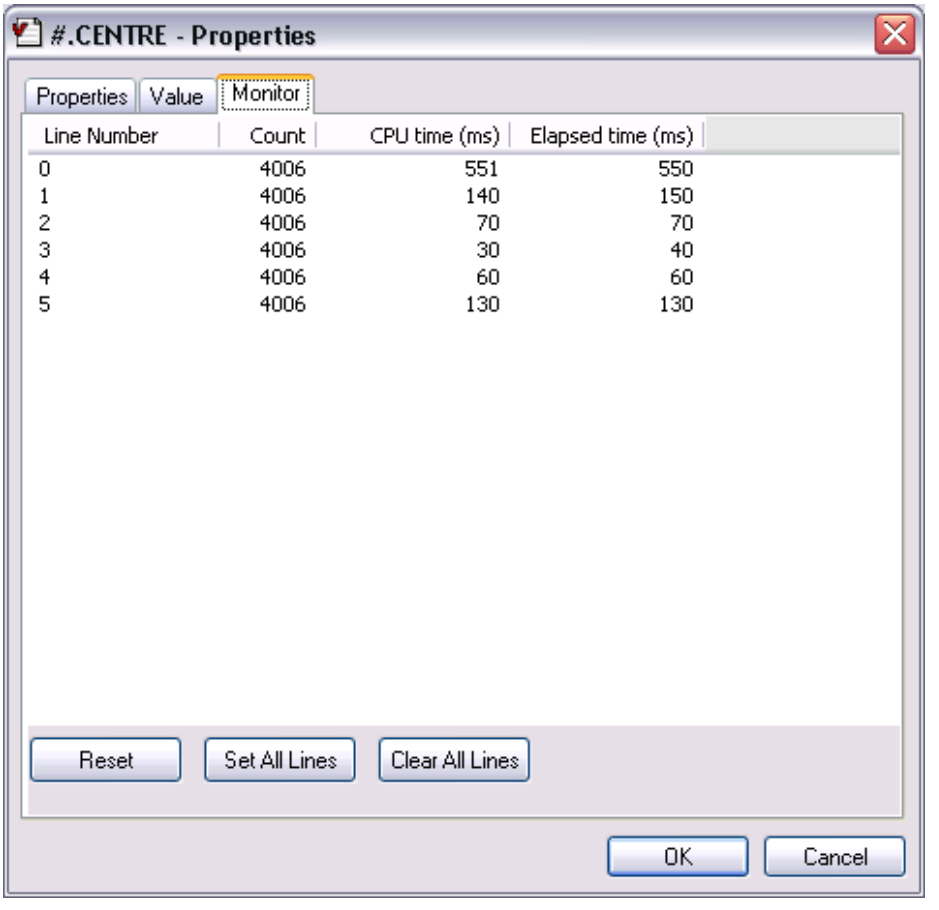
Value Tab

For a variable, the *Values* tab displays the value of the variable. For a function, it displays its canonical representation.



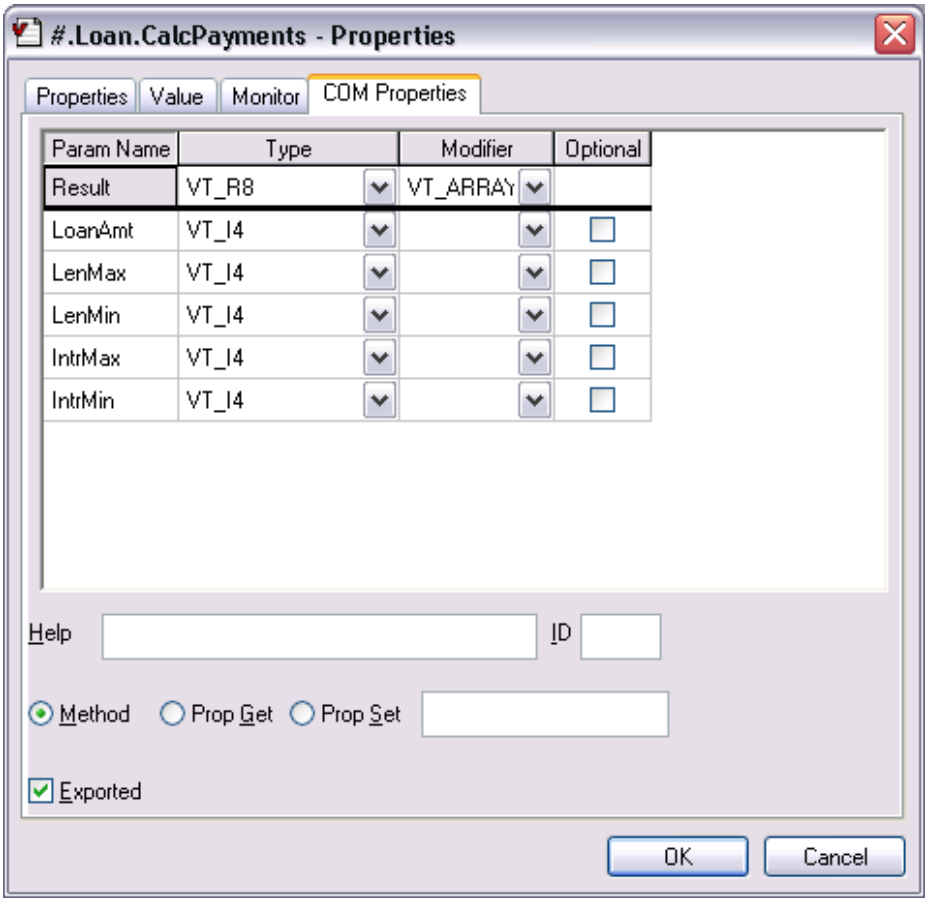
Monitor Tab

The *Monitor* tab applies only to a function and displays the result of `⎕MONITOR`. The *Reset* button resets `⎕MONITOR` for the lines on which it is currently set. The *Set All Lines* button sets `⎕MONITOR` to monitor all the lines in the function. The *Clear All Lines* switches `⎕MONITOR` off.



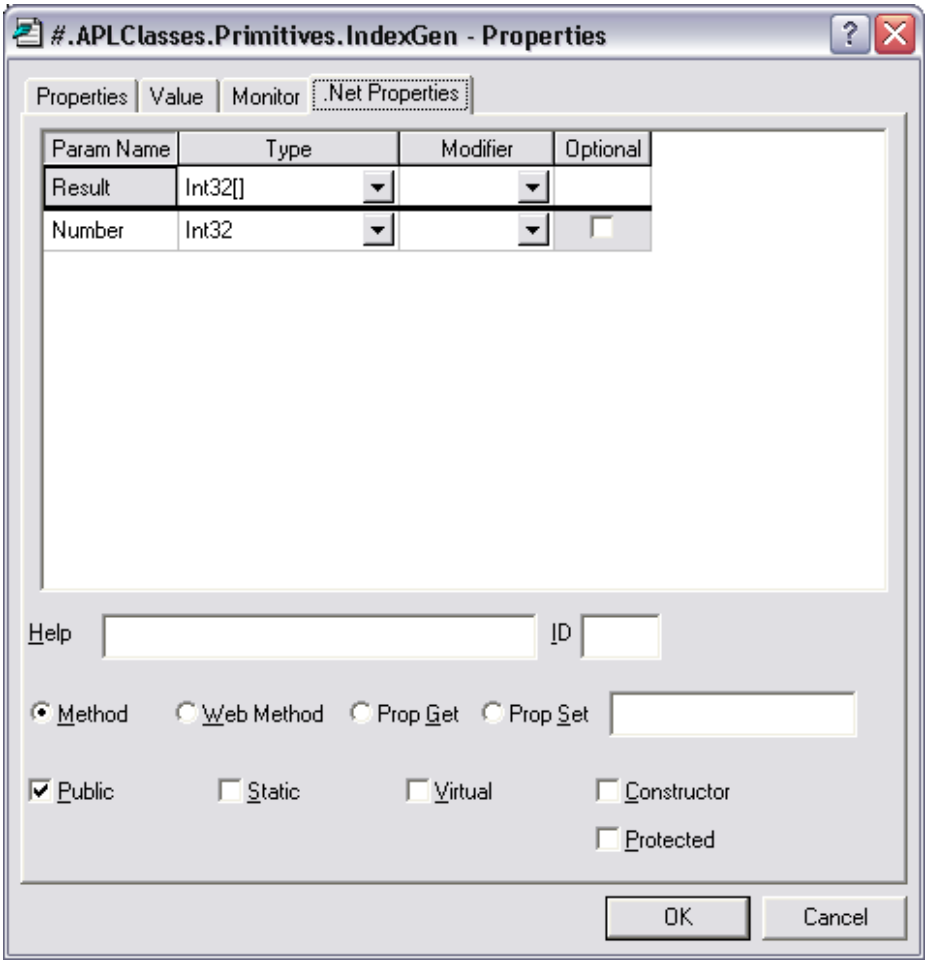
COM Properties Tab

The *COM Properties* tab applies only to a function in an OLEServer or ActiveXControl namespace. The tab is used to define arguments and data types for an exported Method or Property. For further information, see *Interface Guide*.



Net Properties Tab

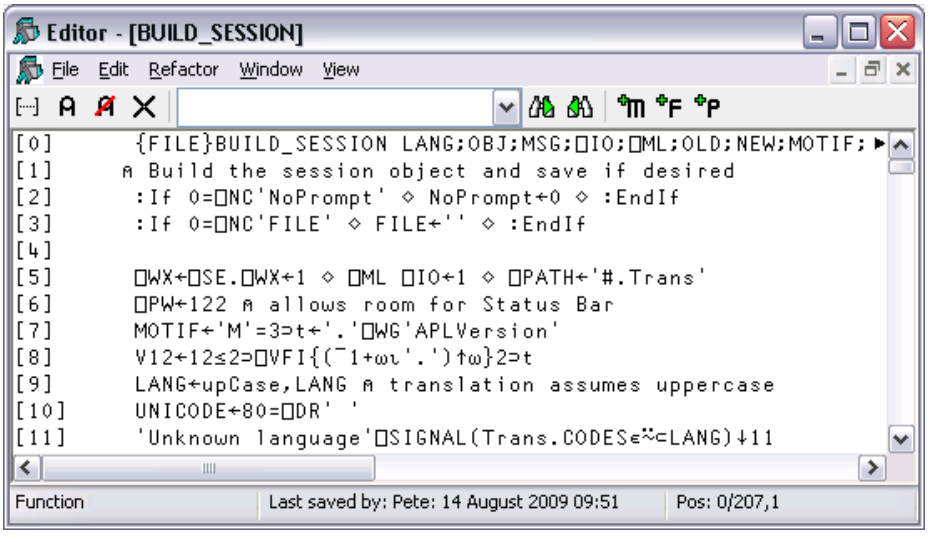
The *Net Properties* tab applies only to a function in a NetType namespace. The tab is used to define arguments and data types for an exported Method or Property. For further information, see *Dyalog .NET Interface Guide*.



The Editor

Invoking the Editor

The editor may be invoked in several ways. From the session, you can use the system command `)ED` or the system function `⎕ED`, specifying the names(s) of the object(s) to be edited. You can also type the name of the object and then press Shift+Enter (ED), click the *Edit* tool on the tool bar, or select *Edit* from the *Action* menu. If you invoke the editor when the cursor is positioned on the empty input line, with a suspended function in the State Indicator, the editor is invoked on the suspended function and the cursor is positioned on the line at which it is suspended. This is termed *naked edit*. These ways of invoking the editor apply only in the session window



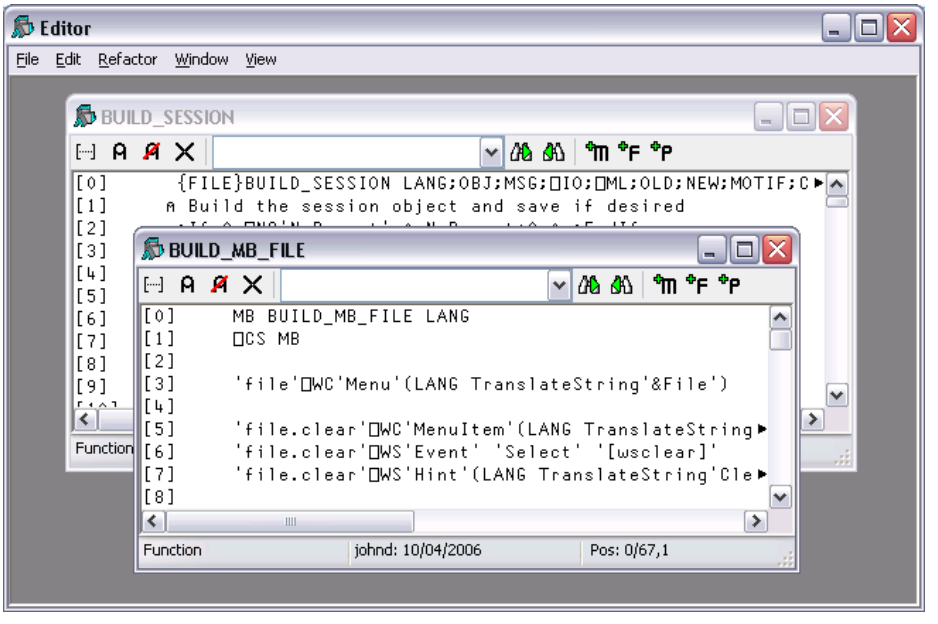
In addition, there is a general *point-and-edit* facility which works in edit and trace windows too. Simply position the input cursor over a name and double-click the left mouse button. Alternatively, you can press Shift+Enter or select *Edit* from the *File* menu. The name can appear in the Session, in an Edit window, or in a Trace window; the effect is the same. Note that, in the Session, typing a name and pressing Shift+Enter is actually a special case of *point-and-edit*. Note also that a *naked edit* can be invoked by double-clicking the left mouse button in the empty input line.

The type of a new object defaults to function/operator unless the object is shadowed, in which case it defaults to a variable (vector of character vectors). You can however specify the type of a new object explicitly using `)ED` or `⎕ED` . For example, typing "`)ED ∈LIST -MAT`" in a `CLEAR WS` would create Edit windows for a vector of character vectors named `LIST` and a character matrix called `MAT`. See `)ED` or `⎕ED` for details.

If the name is not already being edited, it is assigned a new edit window. If you edit a name which is already being edited, the system *focuses* on the existing edit window rather than opening a new one. Edit windows are displayed using the colour combination associated with the type of the object being edited.

Window Management (Standard)

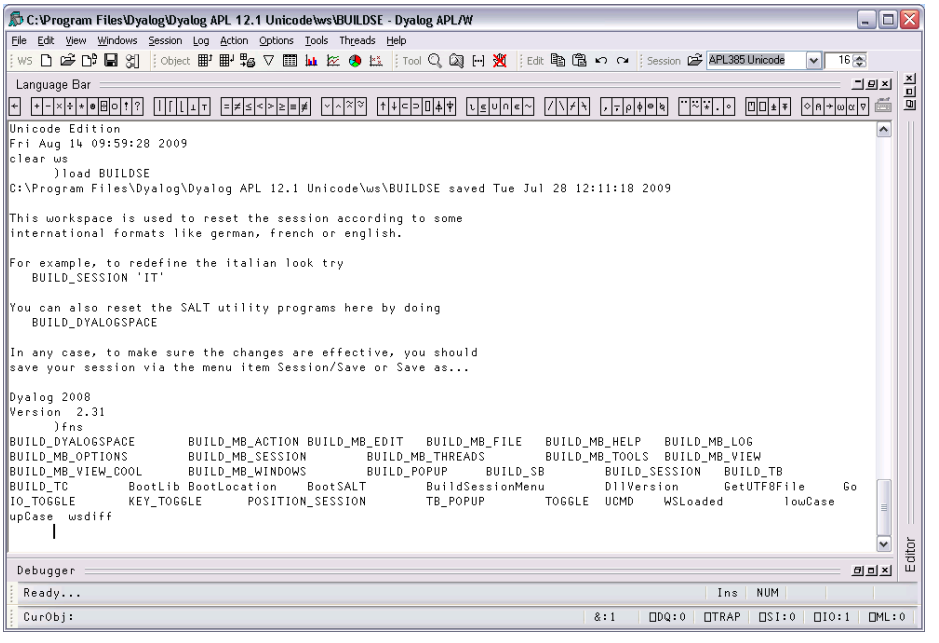
Unless *Classic Dyalog mode* is selected (*Options/Configure/Trace/Edit*), the Editor is a Multiple Document Interface (MDI) window that may be a stand-alone window, or be docked in the Session window. Each of the objects being edited is displayed in a separate sub-window. Individual edit windows are managed using standard MDI facilities.

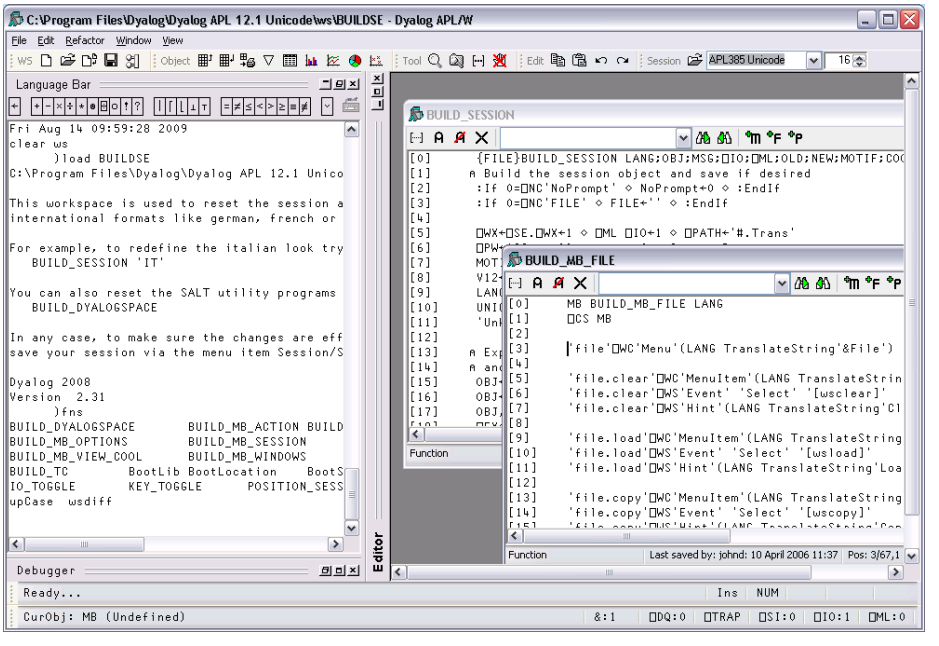


The first edit sub-window window is created at the position specified by the **edit_first_y** and **edit_first_x** parameters which are specified in terms of the size of a character in the current font relative to the top-left corner of the main Editor window. Subsequent ones are staggered according to the values of the **edit_offset_y** and **edit_offset_x** parameters.

The initial size of an edit window is specified by the **edit_rows** and **edit_cols** parameters.

By default, the Session has the Editor docked along the right edge of the Session window. When you edit a function, the Editor window automatically springs into view as illustrated below.

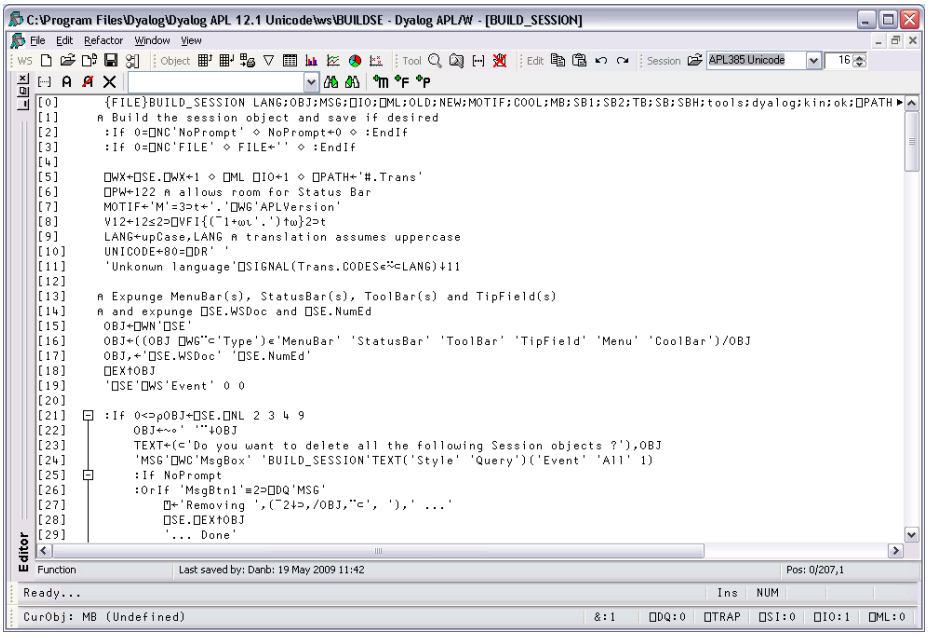




You can resize the Editor pane to view more or less of the Session itself, by dragging its title bar.

Using the buttons in the title bar, you can instantly maximise the Editor pane to allow you to concentrate on editing, or minimise it to reveal the entire Session. In either case, the restore button quickly restores the 2-pane layout.

The picture below shows the effect of maximising the Editor. The `BUILD_SESSION` edit window is itself maximised within the Editor too.



Note that when the Editor has the focus, the Editor menubar is displayed in place of the Session menubar.

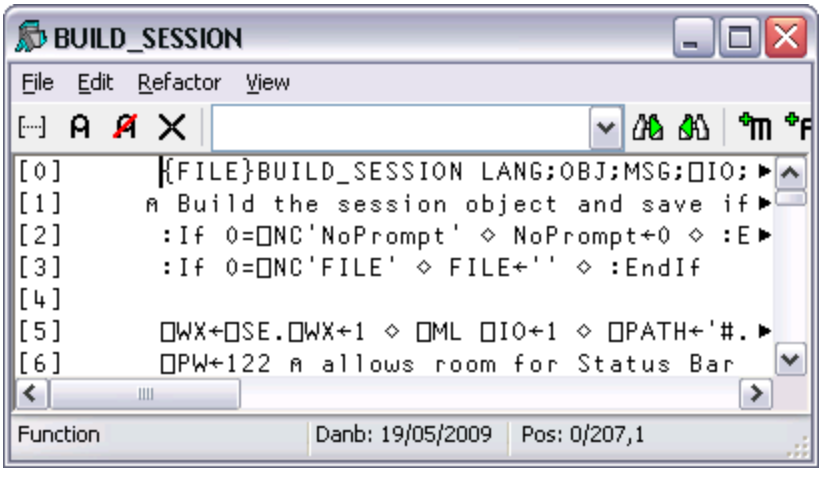
Window Management (Classic Dyalog mode)

If *Classic Dyalog mode* is selected (*Options/Configure/Trace/Edit*) each Edit window is a top-level window created as a child of the Session window. This means that normally Edit windows appear on top of the Session. However, if the **SessionOnTop** parameter is set, the Session window, when given the focus, will appear on top of Edit windows.

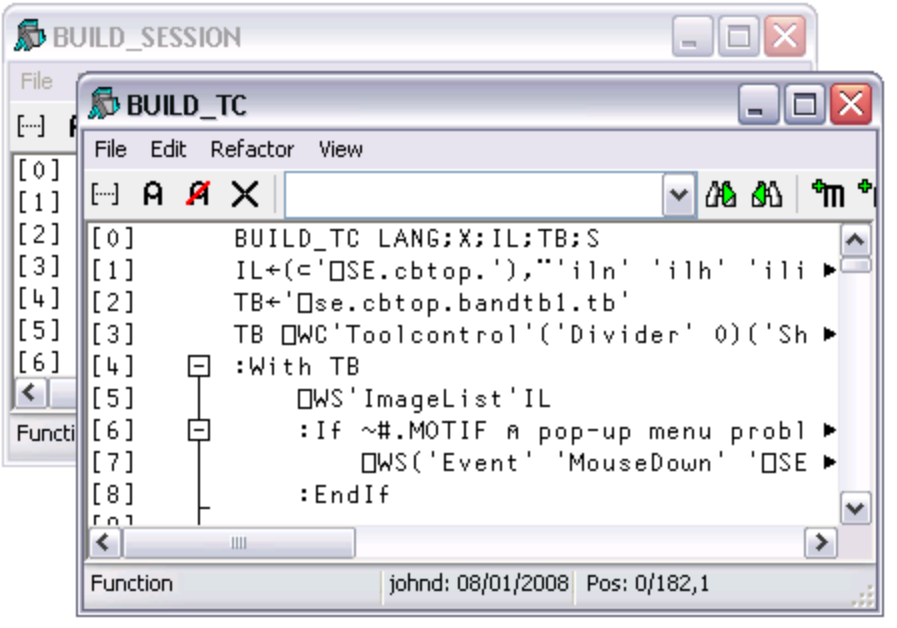
When the first Edit window is opened, its position is determined as follows:

- If the **ClassicModeSavePosition** parameter is set, the first Edit window is displayed at the position that was previously occupied by the most recently saved Edit window.
- If not, the first edit window is created at the position specified by the **edit_first_y** and **edit_first_x** parameters which are specified in terms of the size of a character in the current font relative to the top-left corner of the screen.

The initial size of an edit window is specified by the **edit_rows** and **edit_cols** parameters.



Subsequent ones are staggered according to the values of the **edit_offset_y** and **edit_offset_x** parameters.



Moving around an edit window

You can move around in the edit window using the scrollbar, the cursor keys, and the PgUp and PgDn keys. In addition, Ctrl+Home (UL) moves the cursor to the beginning of the top-line in the object and Ctrl+End moves the cursor to the end of the last line in the object. Home (LL) and End (RL) move the cursor to the beginning and end respectively of the line containing the cursor.

Closing an edit window

Closing an edit window from its System Menu has the same effect as choosing Exit from the *File* Menu; namely that it fixes the object in the workspace and then closes the edit window.

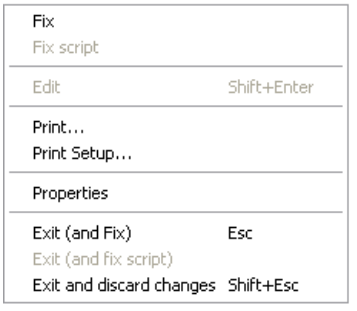
Minimising an edit window

Minimising an edit window causes it to be displayed as a Dyalog APL *Edit* icon, with the name of the object underneath. The edit window can be restored in the normal way, or by an attempt to re-edit the same name.

Editor ToolBar

Toggle line numbers	Toggles Line numbers on/off.
Comment selected text	Adds a comment to the beginning of the current line or all selected lines.
Uncomment selected text	Removes a comment (if present) from the current line or all selected lines.
Save changes and return	Saves changes and closes the current edit window.
Search Box	Enter search text and click one of the following two buttons.
Search for Next Match	Locates the next occurrence of the search text
Search for Previous Match	Locates the previous occurrence of the search.
Search hidden text	Determines whether or not the search examines collapsed blocks.
Match case	Specifies whether or not the search is case-sensitive.
Refactor text as method	Inserts a Method template for the selected name.
Refactor text as field	Inserts a Field template for the selected name.
Refactor text as property	Inserts a Property template for the selected name.

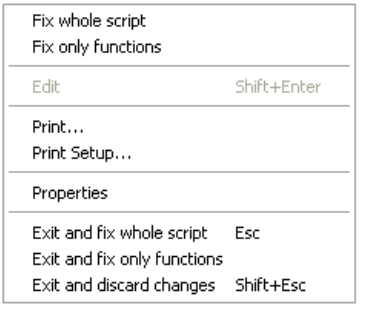
The File Menu



The *File* menu illustrated above is displayed when editing a simple object and provides the following options.

Fix	Fixes the object in the workspace, but leaves the edit window open. Edit history is also preserved. If the data has changed and the **confirm_fix** parameter is set, you will be prompted to confirm.
Fix Script	(Disabled unless editing a script)
Edit	Opens an Edit window on the name under the mouse pointer. (Disabled when not).
Print	Prints the current contents of the edit window.
Print Setup	Displays the *Print Configuration* dialog box.
Properties	Displays the *Object Properties* dialog box for the current object.
Exit (and Fix)	Fixes the object in the workspace and closes the edit window. If the data has changed and the confirm_exit parameter is set, you will be prompted to confirm.
Exit (and fix script)	(Disabled unless editing a script)
Exit and discard changes	Closes the edit window, but does not fix the object in the workspace. If the data has changed and the **confirm_abort** parameter is set, you will be prompted to confirm.

The File Menu (editing a script)



The File menu illustrated above is displayed when editing a script and provides the following different options from those shown on the preceding page.

Fix whole script	Fixes the entire script
Fix only functions	Fixes only the functions in the script.
Exit and fix whole script	Fixes the entire script, and exits the Editor.
Exit and fix only functions	Fixes only the functions in the script and exits the Editor.

Editing Scripts

Suppose that you have a Class that manages a list of items in a shared Field, so somewhere in the script would appear a line such as:-

```
:Field shared public List←0
```

You run your application for a bit, and `List`, which was initially empty, gets updated as new instances of the Class are created. You then edit the Class to add a new function, or fix a bug. In this instance, when you exit the editor you **may not** want `List` to be reset back to the empty vector although you **do want** the new version of the function(s) in the Class to be fixed.

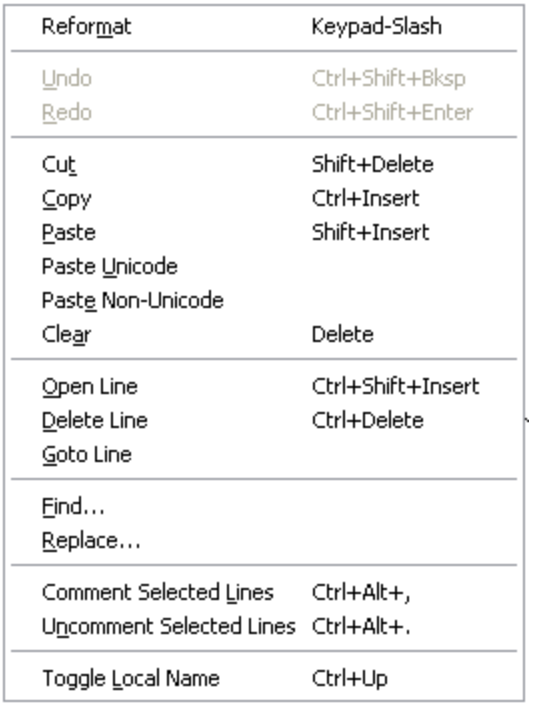
Nevertheless whenever you edit the Class *when it is not suspended*, you probably always want the entire script to be re-fixed, and `List` re-initialised.

The options in the *File* menu shown above provide for these alternatives.

In addition, the Configuration Dialog (see *Installation & Configuration Guide: Configuration Dialog: Trace/Edit Tab*) allows you to define the behaviour of the keystrokes <EP> and <S1> for both the suspended case and the non-suspended case. This association will be displayed against the appropriate action according to the state of the script you are editing.

The Edit Menu

The *Edit* menu provides a means to execute those commands that are concerned with editing text. The Edit menu and the actions it provides are described below.

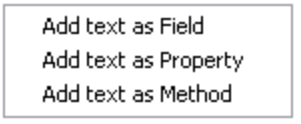


Reformat	Reformats the function body in the edit window, indenting control structures as appropriate.
Undo	Undoes the last change made to the object. Repeated use of this command sequentially undoes each change made since the edit window was opened.
Redo	Re-applies the previous undone change. Repeated use of this command sequentially restores every undone change.
Cut	Copies the selected text to the clipboard and removes it from the object.
Copy	Copies the selected text to the clipboard.
Paste	Copies the text in the clipboard into the object at the current location of the input cursor.
Paste Unicode	Same as *Paste*, but gets the Unicode text from the clipboard and converts to ⎕AV
Paste Non-Unicode	Same as *Paste*, but gets the ANSI text from the clipboard and converts to ⎕AV.
Clear	Deletes the selection or the character under the cursor. Has no effect on the clipboard
Open Line	Inserts a blank line immediately below the current one.
Delete Line	Deletes the current line.
Goto Line	Prompts for a line number, then positions the cursor on that line.
Find	Displays the *Find* dialog box.
Replace	Displays the *Replace* dialog box.
Comment selected lines	Adds a comment symbol to the beginning of all selected lines.
UnComment selected lines	Removes a comment symbol from the beginning of all selected lines.
Toggle Local name	Adds or removes the name under the cursor to/from the function header line.

The *Find* and *Replace* items are used to display the *Find* dialog box and the *Find/Replace* dialog box respectively. These boxes are used to perform search and replace operations and are described later in this Chapter.

Once displayed, each of the two dialog boxes remains on the screen until it is either closed or replaced by the other. This is convenient if the same operations are to be performed over and over again, and/or in several windows. *Find* and *Find/Replace* operations are effective in the window that previously had the focus.

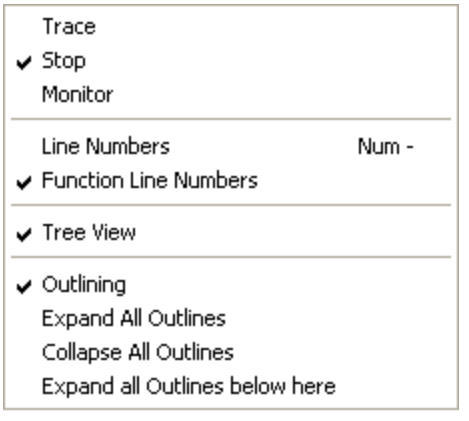
The Refactor Menu



The *Refactor* menu illustrated above applies only when editing a Class and provides the following options. In each case, the user must highlight a name in the Edit window, and then select one of these options to insert the appropriate template for that name into the body of the Class.

Add text as Field	Inserts a Field template for the selected name.
Add text as Property	Inserts a Property template for the selected name.
Add text as Method	Inserts a Method template for the selected text name.

The View Menu



The *View* menu, illustrated above, provides the following actions.

Trace	Displays a column to the left of the function that displays ⎕TRACE settings
Stop	Displays a column to the left of the function that displays ⎕STOP settings
Monitor	Displays a column to the left of the function that displays ⎕MONITOR settings
Line Numbers	Toggles the display of line numbers on/off.
Function Line Numbers	Toggles the display of line numbers on *individual functions* on/off. This option is only enabled when editing a Class, Namespace script or Interface.
Tree View	Toggles the display of the treeview in the left-hand pane.
Outlining	Turns outlining on and off.
Expand All Outlines	Expands all outlines.
Collapse All Outlines	Collapses all outlines
Expand all Outlines below here	Expands all outlines below the level of the current line.

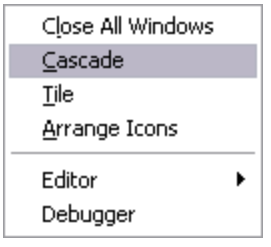
Function Line Numbers

The *Function Line Numbers* option in the Editor menu provides an additional level of line-numbering. If selected, line numbers are displayed *independently* on each individual function (or operator) in the Class. This option is only enabled when you are editing a Class, Namespace script or Interface, and is disabled for all other types of object.

Note that function line-numbering and general line-numbering are independent options and it is possible to have the entire Class numbered (from [0] to the number of lines in the Class) in addition to having line-numbering on each individual function.

The Window Menu

The *Window* menu provides a means to control the display of the various edit windows. The *Window* menu and the actions it provides are described below.



Close All Windows	Closes all the edit windows. If *Confirm on Edit Window Closed* is checked, you will be prompted to confirm for any objects that you have changed.
Cascade	Arranges the edit windows in overlapping fashion.
Tile	Arranges the edit windows in a tiling fashion.
Arrange Icons	Arranges any minimised edit windows.
Editor	Allows you to Select the edit window corresponding to the named object.

Using the Editor

Creating a New Function

Type the name of your function and invoke the editor. To do this you may press Shift+Enter, or select *Edit* from the *Action* menu, or double-click the left button on your mouse, or click the *Edit* tool in the tool bar. A new window will appear on the screen with the name you have chosen displayed in the top border. The name is also inserted in the function header and the cursor positioned to the right. The new window is automatically given the input focus.

Line-Numbers on/off

Try changing the line numbers setting by clicking on the *Line Numbers* option in the *Options* menu. Note that line-numbering on/off is effective for **all** edit windows.

Adding Lines

If the keyboard is in *Insert* mode, pressing Enter at the end of a line opens you a new blank line under the current one and positions the cursor there ready for input. You can also open a new blank line by pressing Ctrl+Shift+Insert (OP).

If the cursor is at the end of the last line in the function, pressing Enter adds another line even if the keyboard is in Replace mode.

Indenting Text

Dyalog APL allows you to insert leading spaces in lines of a function and (unless the **AutoFormat** parameter is set) preserves these spaces between editing sessions. Embedded spaces are however discarded. You can enter spaces using the space bar or the Tab key. Pressing Tab inserts spaces up to the next tab stop corresponding to the value of the **TabStops** parameter. If the **AutoIndent** parameter is set, new lines are automatically indented the same amount as the preceding line.

Reformatting

The RD command (which by default is mapped to Keypad-Slash) reformats a function according to your **AutoFormat** and **TabStops** settings.

Deleting Lines

To delete a block of lines, select them by dragging the mouse or using the keyboard and then press Delete or select *Clear* from the *Edit* menu. A quick way to delete the current line without selecting it first is to press Ctrl+Delete (DK) or select *Delete Line* from the *Edit* menu.

Copying Lines

Select the lines you wish to copy by dragging the mouse or using the keyboard. Then press Ctrl+Insert or select *Copy* from the *Edit* menu. This action copies the selection to the clipboard. Now position the input cursor where you wish to make the copy and press Shift+Insert, or select *Paste* from the *Edit* menu. You can also use this method to duplicate a ragged block of text.

To copy text using drag-and-drop editing:

1. Select the text you want to move.
2. Hold down the Ctrl key, point to the selected text and then press and hold down the left mouse button. When the drag-and-drop pointer appears, drag the cursor to a new location.
3. Release the mouse button to drop the text into place.

Moving Lines

Select the lines you wish to copy by dragging the mouse or using the keyboard. Then press Shift+Delete or select *Cut* from the Edit menu. This action copies the selection to the clipboard and removes it. Now position the input cursor at the new location and press Shift+Insert, or select *Past*e from the Edit menu. You can also use this method to move a *ragged* block of text.

To move text using drag-and-drop editing:

1. Select the text you want to move.
2. Point to the selected text and then press and hold down the left mouse button. When the drag-and-drop pointer appears, drag the cursor to a new location.
3. Release the mouse button to drop the text into place.

Joining and Splitting Lines

To join a line to the previous one: select Insert mode; position the cursor on the first character in the line; press Bksp.

To split a line: select Insert mode; position the cursor at the place you want it split; press Return.

Toggling Localisation

The TL command (which by default is mapped to Ctrl+Up) toggles the localisation of the name under the cursor. If the name is currently global, pressing Ctrl+Up causes the name to be added to the list of locals in the function header. If the name is already localised, pressing Ctrl+Alt+l removes it from the header.

Aligning Comments

When you press the <AC> key, or select Align Comments in the Editor's context menu, the alignment of the comments in every line in the function will be changed so that the left-most comment (Lamp) symbol is in the same column as the cursor, except that:

- Comment symbols that lie between the first column and the first tab stop will remain in or be moved to the first column. For information on setting tab stops, see *Installation & Configuration Guide: Configuration Dialog (Edit/Trace Tab)*.
- Comment symbols will not move further left than the end of the statement.

When a comment is re-aligned, text to the right of the left-most comment symbol (including spaces and other comment symbols) will remain fixed in relation to that symbol.

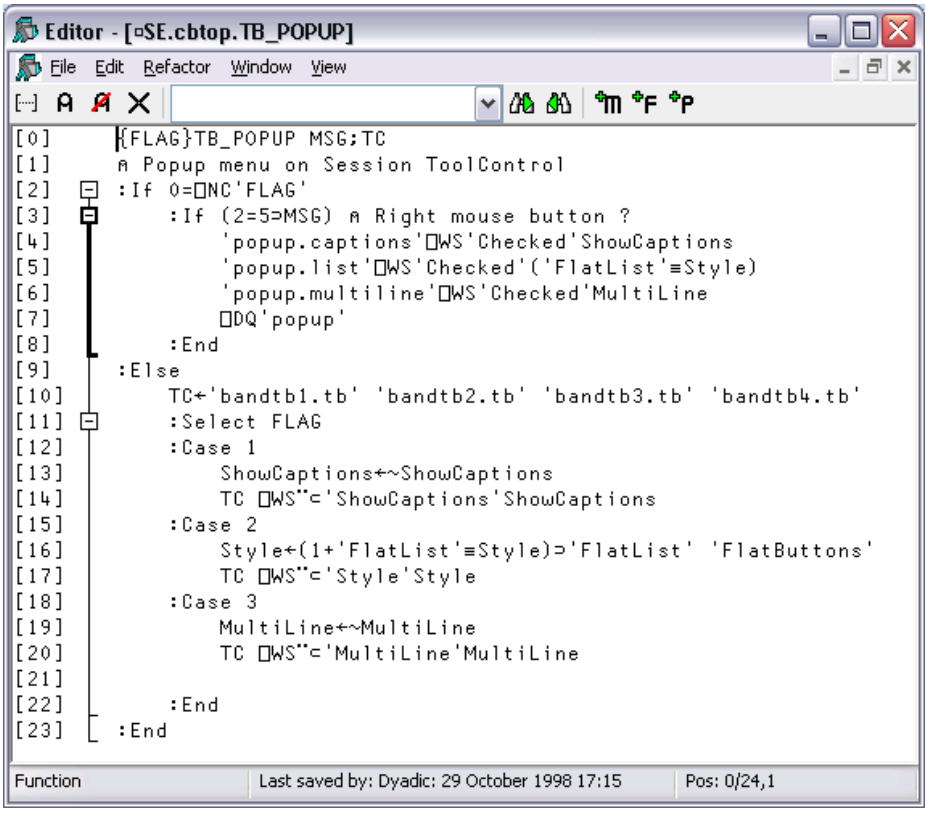
Note that there is no keystroke associated with this command by default; the user must define one. See *Installation & Configuration Guide: Configuration Dialog (Keyboard Shortcuts Tab)*.

Outlining

When you are editing a function, outlining identifies the blocks of code within control structures, and allows you to collapse and expand these blocks so that you can focus your attention on particular parts of the code

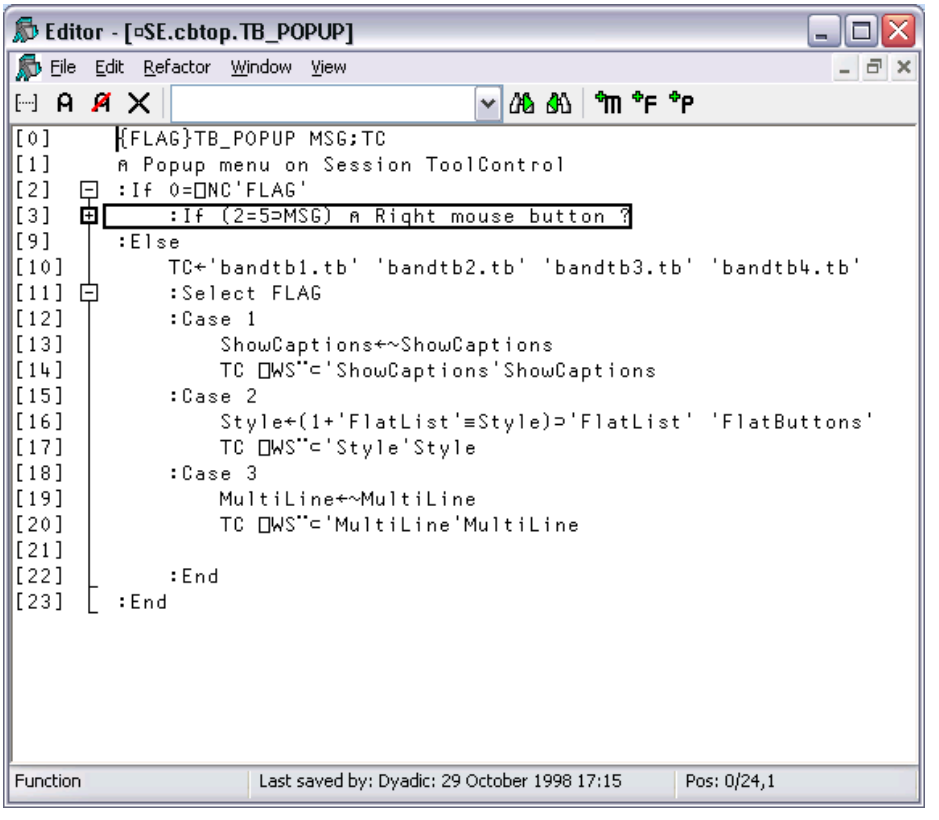
The picture below shows the result of opening the function `⎕SE.cbtop.TB_POPUP`.

```
)ed ⎕SE.cbtop.TB_POPUP
```



Notice that the various control structure blocks are delineated by a treeview diagram.

- When you hover the mouse pointer over one of the boxes that mark the start of a block, the line marking the extent of that block becomes highlighted, as shown above.
- If you click on a ⊟ box, the corresponding section collapses, so that only the first line of the block is displayed, as shown below.
- If you click on a ⊞ box, the corresponding section is expanded.

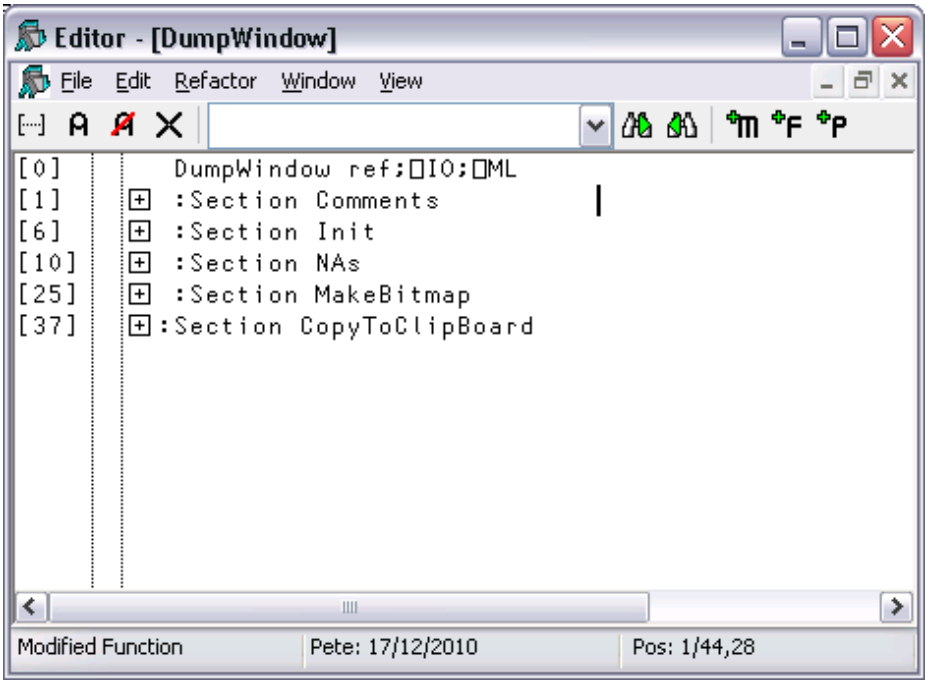


Sections

Functions and scripted objects (classes, namespaces etc.) can be subdivided into Sections with `:Section` and `:EndSection` statements. Both statements may be followed by an optional and arbitrary name or description. The purpose is to split the function up into sections that you can open and close in the Editor, thereby aiding readability and code management. Sections have no effect on the execution of the code, but must follow the nesting rules of other control structures.

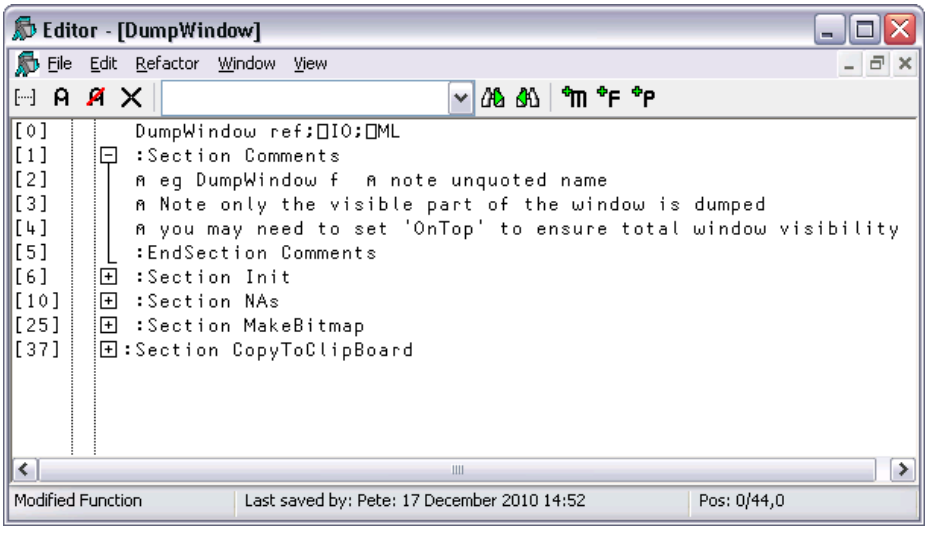
The following picture illustrates the use of sections in a function called `DumpWindow`. The function is divided into 5 sections named `Comments, Init`, `NAs, MakeBitmap` and `CopyToClipBoard`.

The first picture shows the function with all sections closed.

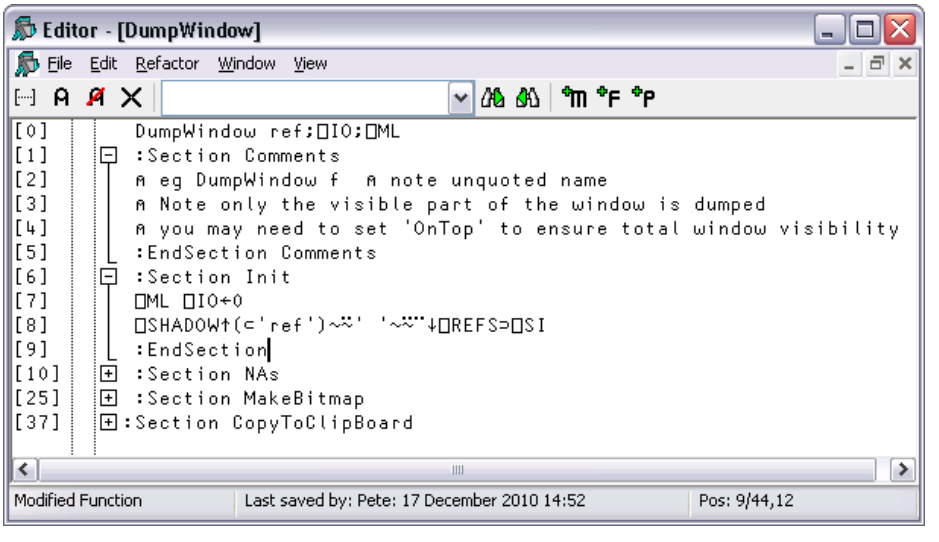


The next picture shows the effect of opening the *Comments* section. Notice how this is delineated by the statements:

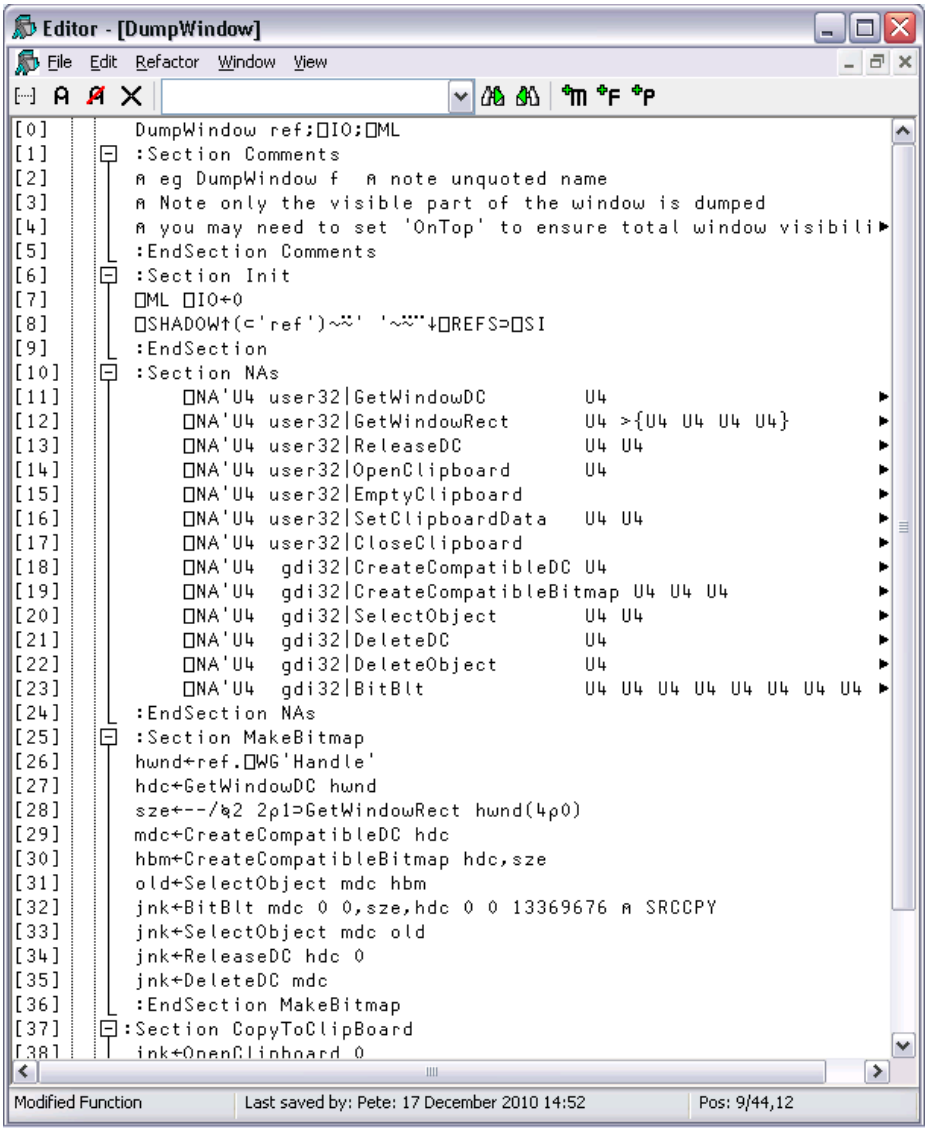
```
:Section Comments
...
:EndSection Comments
```



And with the `Init` section opened too:



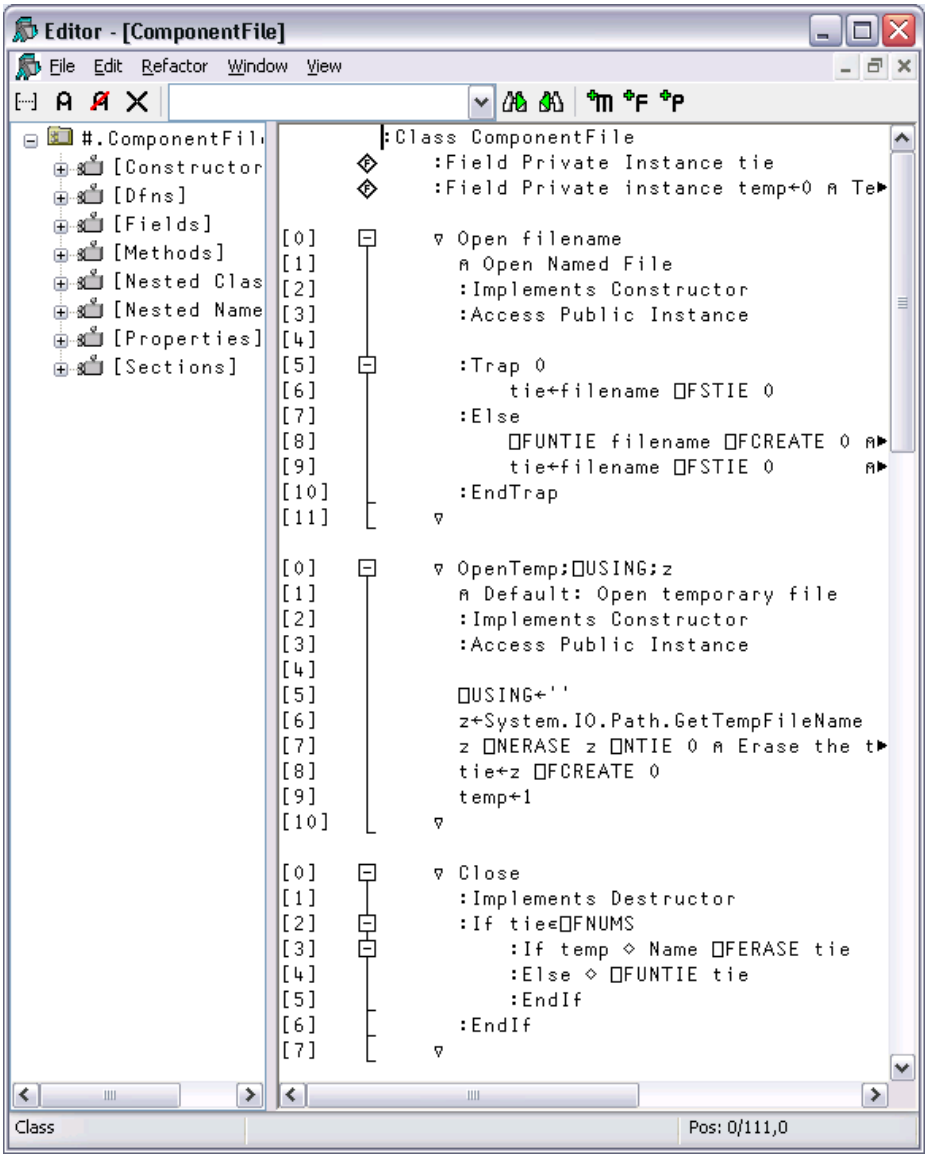
Finally, with all the sections opened:



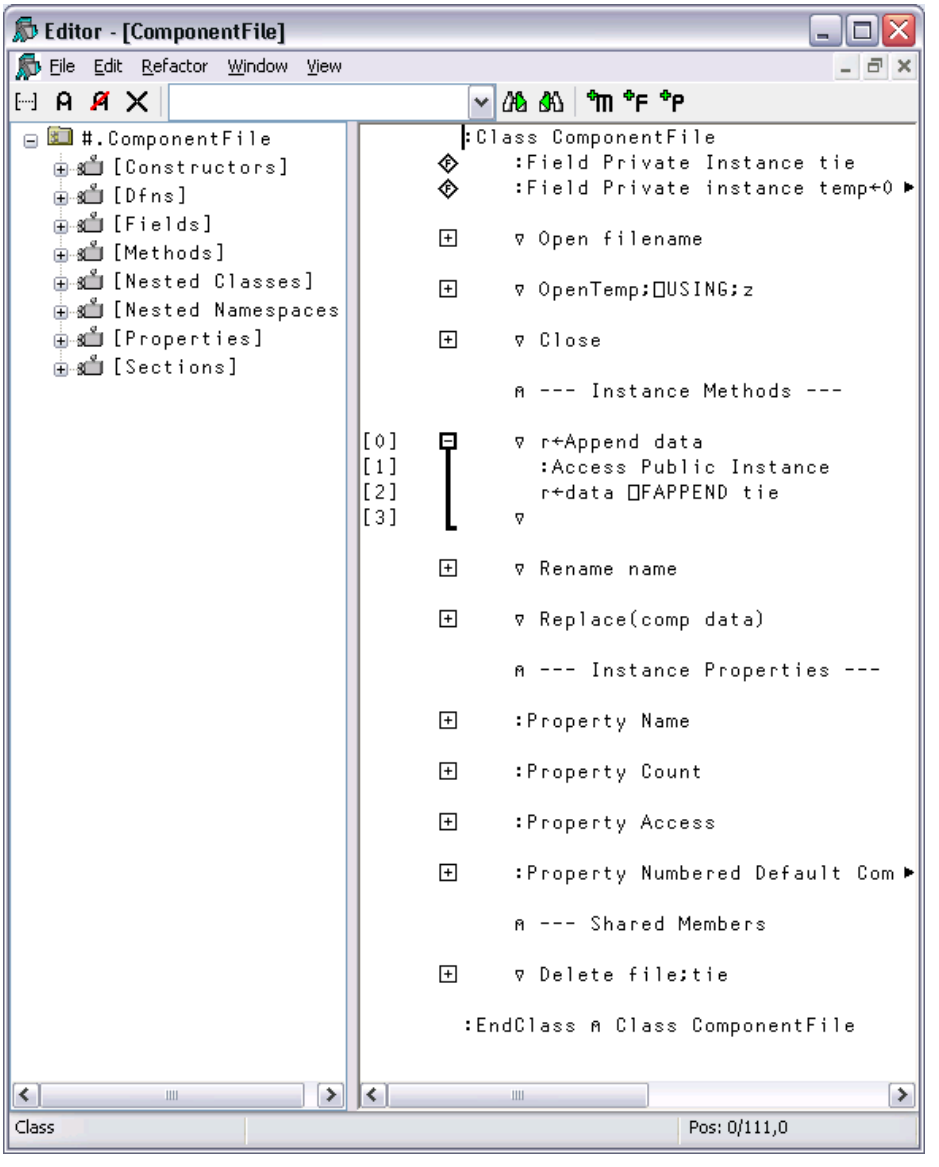
Editing Classes

The picture below shows the result of opening the `ComponentFile` class. Notice how each function is delineated separately and that each function is individually line-numbered.

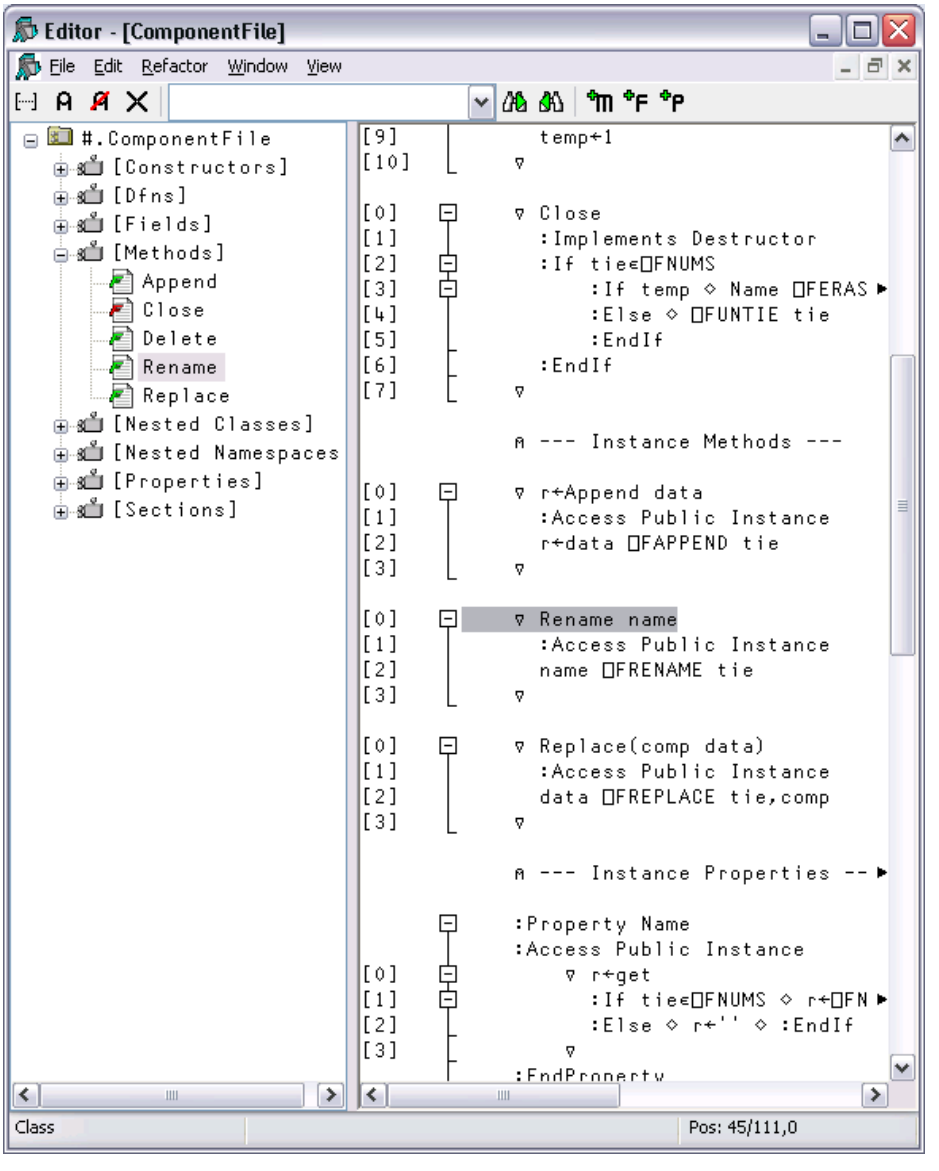
```
)ed ComponentFile
```



The outlining feature really comes into its own when editing classes because you can collapse and expand whole functions. The picture below shows the effect of collapsing all but the `Append` method.



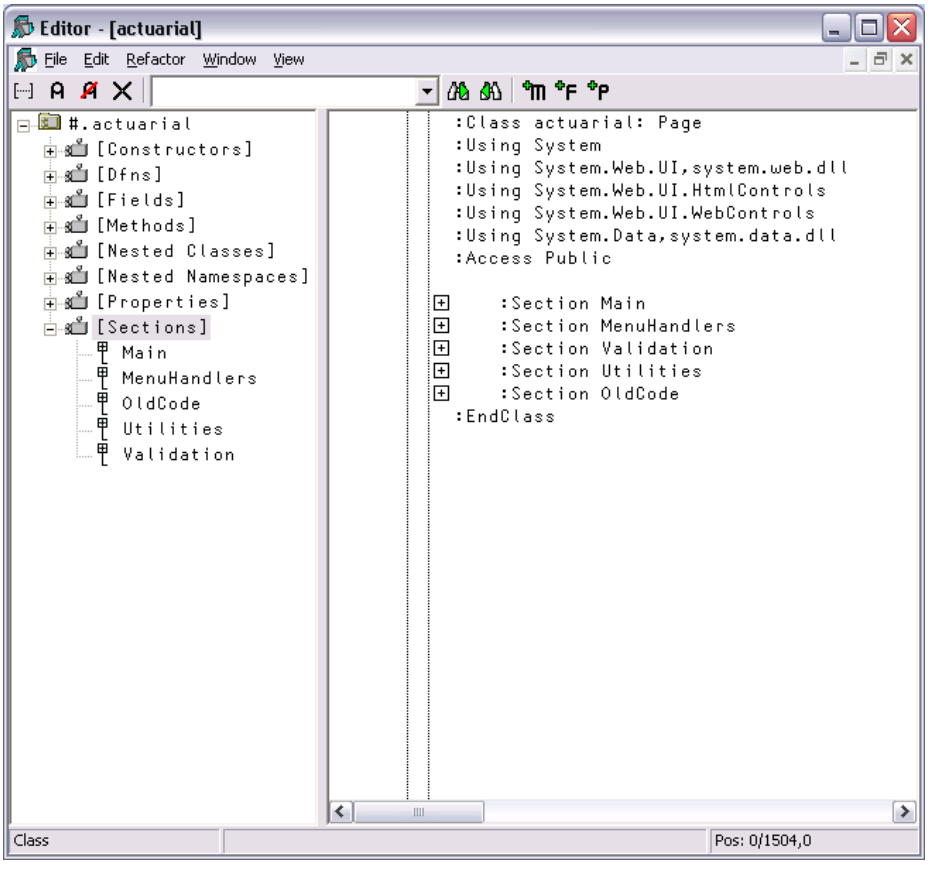
When you edit a class, a separate treeview is optionally displayed in the left pane to make it easy to navigate within the class. When you click on a name in the treeview, the editor automatically scrolls the appropriate section into view (if necessary) and positions the edit cursor at its start. The picture below illustrates the result of opening the `[Methods]` section and then clicking on *Rename*.



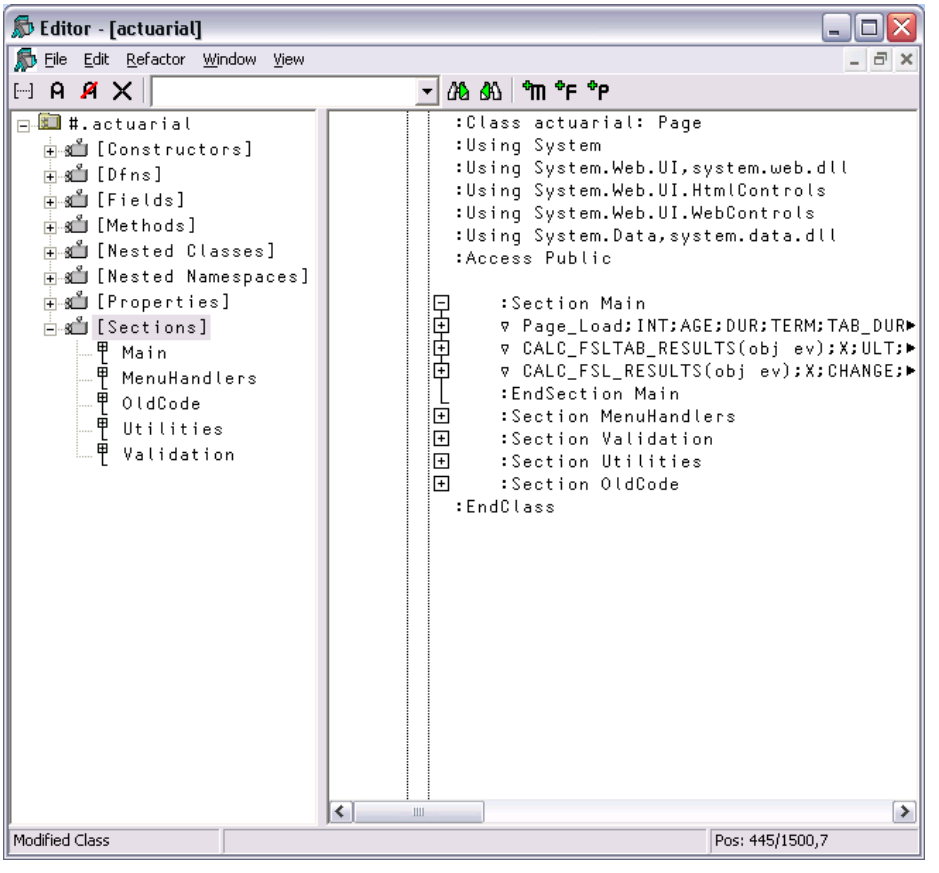
Sections within Scripts

Scripts can also be subdivided into Sections using `:Section` and `:EndSection` statements. As with single functions, the purpose is only to split the script up into sections that you can open and close in the Editor. Sections have no effect on the execution of the code.

The following picture illustrates a Class named `actuarial` which, for editing purposes, has been sub-divided into five separate Sections named `Main`, `MenuHandlers`, `Validation`, `Utilities` and `OldCode`. In this picture, all the Sections are closed.



The next picture shows the effect of opening just the `Main` section.



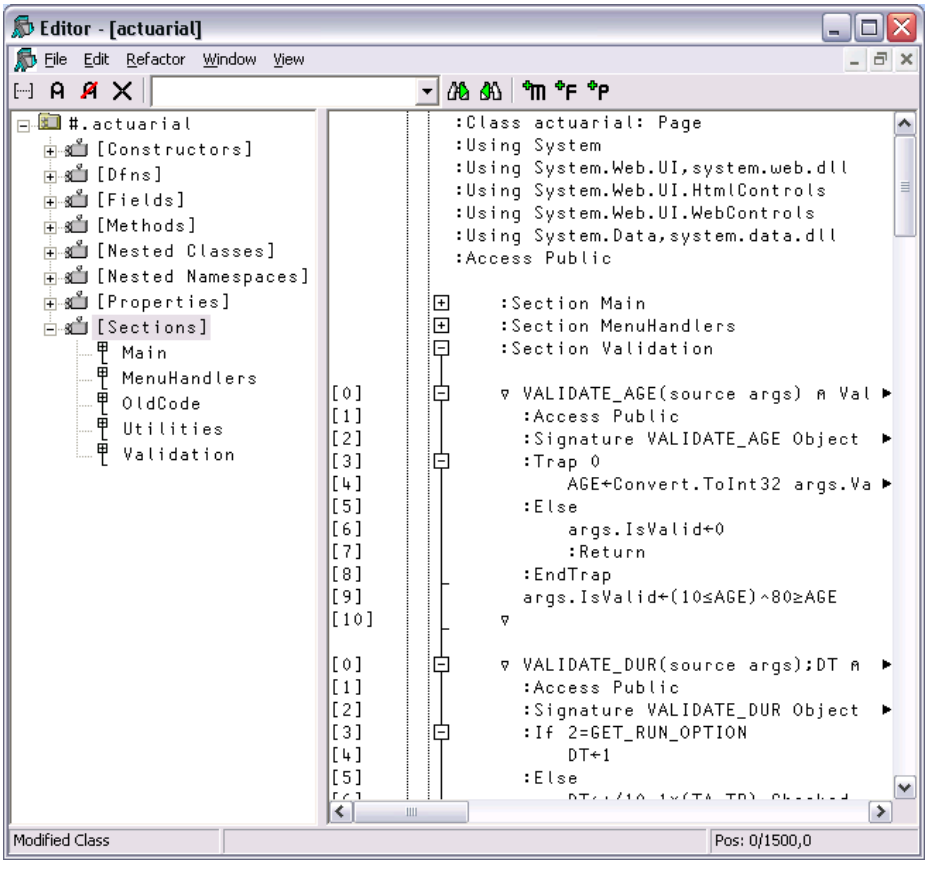
Notice that this section is delimited by the two statements:

```
:Section Main
...
:EndSection Main
```

In this picture the 3 functions within the `Main` section are temporarily closed.

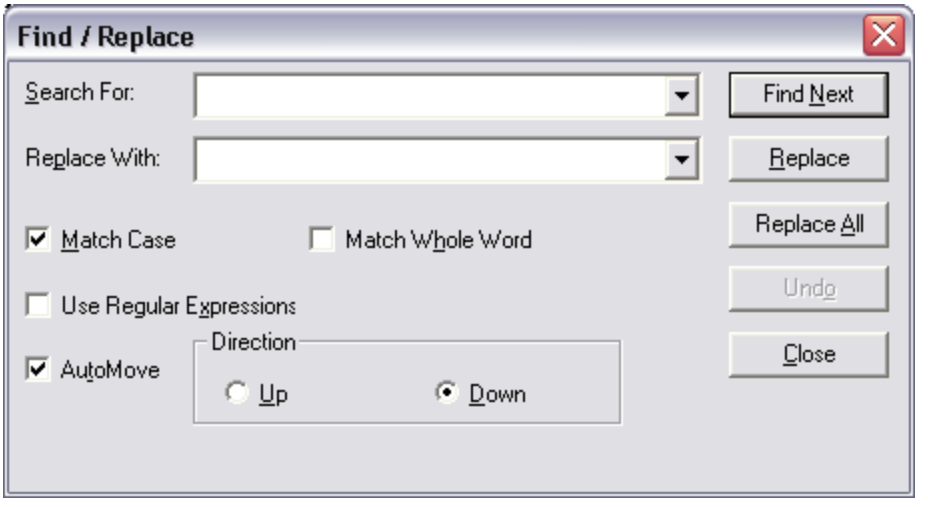
Similarly, the section called `Validation` is delimited by:

```
:Section Validation
...
:EndSection Validation
```


Editor - [actuarial]
File Edit Refactor Window View
#.actuarial
[Constructors]
[Dfns]
[Fields]
[Methods]
[Nested Classes]
[Nested Namespaces]
[Properties]
[Sections]
Main
MenuHandlers
OldCode
Utilities
Validation
:Class actuarial: Page
:Using System
:Using System.Web.UI,system.web.dll
:Using System.Web.UI.HtmlControls
:Using System.Web.UI.WebControls
:Using System.Data,system.data.dll
:Access Public
:Section Main
:Section MenuHandlers
:Section Validation
[0] ∇ VALIDATE_AGE(source args) ⍝ Val
[1] :Access Public
[2] :Signature VALIDATE_AGE Object
[3] :Trap 0
[4] AGE←Convert.ToInt32 args.Va
[5] :Else
[6] args.IsValid←0
[7] :Return
[8] :EndTrap
[9] args.IsValid←(10≤AGE)^80≥AGE
[10] ∇
[0] ∇ VALIDATE_DUR(source args):DT ⍝
[1] :Access Public
[2] :Signature VALIDATE_DUR Object
[3] :If 2=GET_RUN_OPTION
[4] DT←1
[5] :Else
Modified Class
Pos: 0/1500,0

Find and Replace Dialogs

The *Find* and *Find/Replace* dialog boxes are used to locate and modify text in an Edit window.



Search For	Enter the text string that you want to find. Note that the text from the last 10 searches is available from the drop-down list. If appropriate, the search text is copied from the Find Objects tool. This makes it easy to first search for functions containing a particular string, and then to locate the string in the functions.
Replace With	Enter the text string that you want to use as a replacement. Note that the text from the last 10 replacements is available from the drop-down list.
Match Case	Check this box if you want the search to be case-sensitive.
Match Whole Word	Check this box if you want the search to only match whole words.
Use Regular Expressions	Check this box if you want to use various wild card symbols.
AutoMove	If checked, the *Find* or *Find/Replace* dialog box will automatically position itself so as not to obscure a matched search string in the edit window.
Direction	Select *Up* or *Down* to control the direction of search.

Using Find and Replace

Find and Replace work on the concept of a *current search string* and a *current replace string* which are entered using the *Find* and *Find/Replace* Dialog boxes. These boxes also contain buttons for performing search/replace operations.

Suppose that you want to search through a function for references to the string "Adam". It is probably best to work from the start of the function, so first position the cursor there (by pressing Ctrl+Home). Then select *Find* from the *Edit* menu. The *Find* Dialog box will appear on your screen with the input cursor positioned in the edit box awaiting your input. Type "Adam" and click the *Find Next* button (or press Return), and the cursor will locate the first occurrence. Clicking *Find Next* again will locate the second occurrence. You can change the direction of the search by selecting *Up* instead of *Down*. You could search another function for "Adam" by opening a new Edit window for it and clicking *Find Next*. You do not have to redefine the search string.

Now let us suppose that you wish to replace all occurrences of "Adam" with "Amanda". First select *Replace* from the *Edit* menu. This will cause the *Find Dialog* box to be replaced by the *Find/Replace* Dialog box. Enter the string "Amanda" into the box labelled *Replace* With, then click *Replace All*. All occurrences of "Adam" in the current Edit window are changed to "Amanda". To repeat the same global change in another function, simply open an edit window and click *Replace All* again. If instead you only want to change particular instances of "Adam" to "Amanda" you may use *Find Next* to locate the ones you want, and then *Replace* to make each individual alteration.

Saving and Quitting

To save the function and terminate the edit, press Esc (EP) or select Exit from the *File* menu. The new version of the function replaces the previous one (if any) and the edit window is destroyed.

Alternatively, you can select *Fix* from the *File* menu. This fixes the new version of the function in the workspace, but leaves the edit window open. Note that the history is also retained, so you can subsequently undo some changes and fix the function again.

To abandon the edit, press Shift+Esc (QT) or select *Abort* from the *File* menu. This destroys the edit window but does not fix the function. The previous version (if any) is unchanged.

The Tracer

The Tracer is a visual debugging aid that allows you to step through an application line by line. During a Trace you can track the path taken through your code, display variables in edit windows and watch them change, skip forwards and backwards in a function. You can cutback the stack to a calling function and use the Session and Editor to experiment with and correct your code. The Tracer may be invoked in several ways as discussed below.

Tracing an expression

Firstly, you may explicitly trace a function (strictly an expression) by typing an expression then pressing Ctrl+Enter (TC) or by selecting *Trace* from the *Action* menu. This lets you step through the execution of an expression from the beginning.

In the same way as when you execute a statement by pressing Enter, the expression is (if necessary) copied down to the input line and then executed. However, if the expression includes a reference to an unlocked defined function or operator, execution halts at its first line and a Trace window containing the suspended function or operator is displayed on the screen. The cursor is positioned to the left of the first line which is highlighted.

Naked Trace

The second way to invoke the Tracer is when you have a suspended function in the State Indicator and you press Ctrl+Enter (TC) on the empty input line. This is termed *naked trace*. The same thing can be achieved by selecting *Trace* from the *Action* menu on the Session Window or by clicking the *Trace* button in the *Trace Tools*. However, in ALL cases it is essential that the input cursor is on the empty Input line in the Session.

The effect of naked trace is to open the Tracer and to position the cursor on the currently suspended line. It is exactly as if you had Traced to that point from the Input Line expression whose execution caused the suspension.

Automatic Trace

The third way to invoke the Tracer is to have the system do it automatically for you whenever an error occurs. This is achieved by setting the Show trace stack on error option in the *Trace/Edit* tab of the *Configuration* dialog (**Trace_on_error** parameter). When an error occurs, the system will automatically deploy the Tracer. Note that this means that when an error occurs, the Trace window will then receive the input focus and not the Session window.

Tracer Options

From Version 10.1 onwards, the Tracer is designed to be docked in the Session window.

In previous versions of Dyalog APL, the Tracer was implemented as a stack of separate windows (one per function on the calling stack) or as a single, but still separate, window.

You can disable the standard behaviour by selecting *Classic Dyalog mode* from the *Trace/Edit* tab of the *Configuration* dialog box.

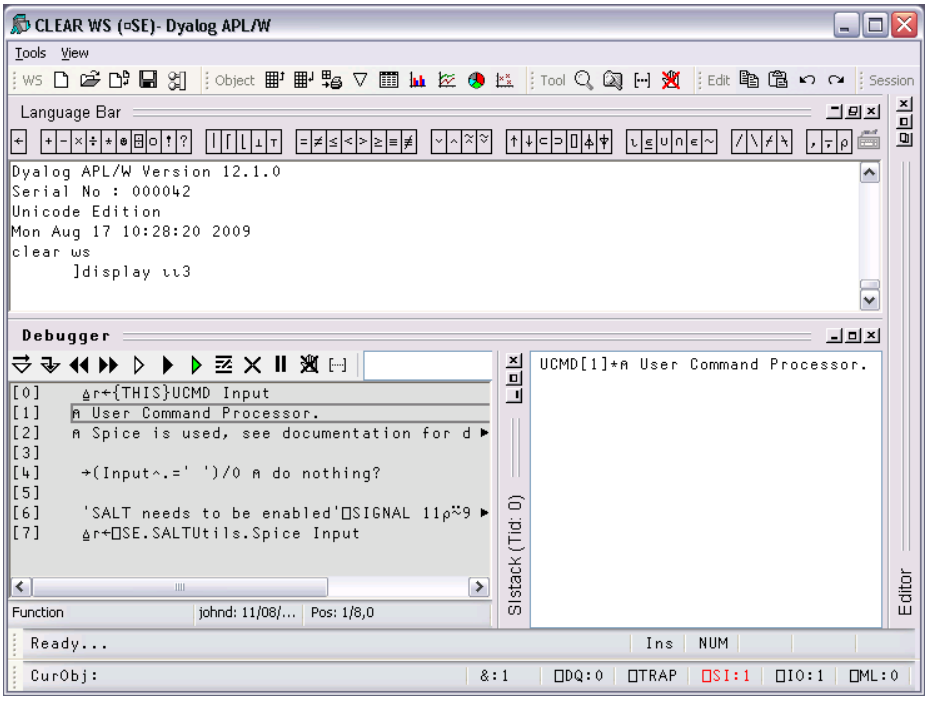
If you do so, you may then choose to have the Tracer operate in multiple windows or in a single window.

These alternatives are discussed later in this Chapter.

The Trace Window

The Tracer is implemented as a single dockable window that displays the function that is currently being executed. There are two subsidiary information windows which are also fully dockable. The first of these (*SIStack*) displays the current function calling stack; the second (*Threads*) displays a list of running threads.

In the default Session files, the Tracer is docked along the bottom edge of the Session window. When you invoke the Tracer, it springs up as illustrated below. In this example, the function being traced is `⎕SE.UCMD`, which is invoked by typing a user-command, in this case `]display`.



In the default layout, the *SIstack* window is displayed alongside the main Tracer window, although this can be hidden or made to appear as a separate floating window, as required.

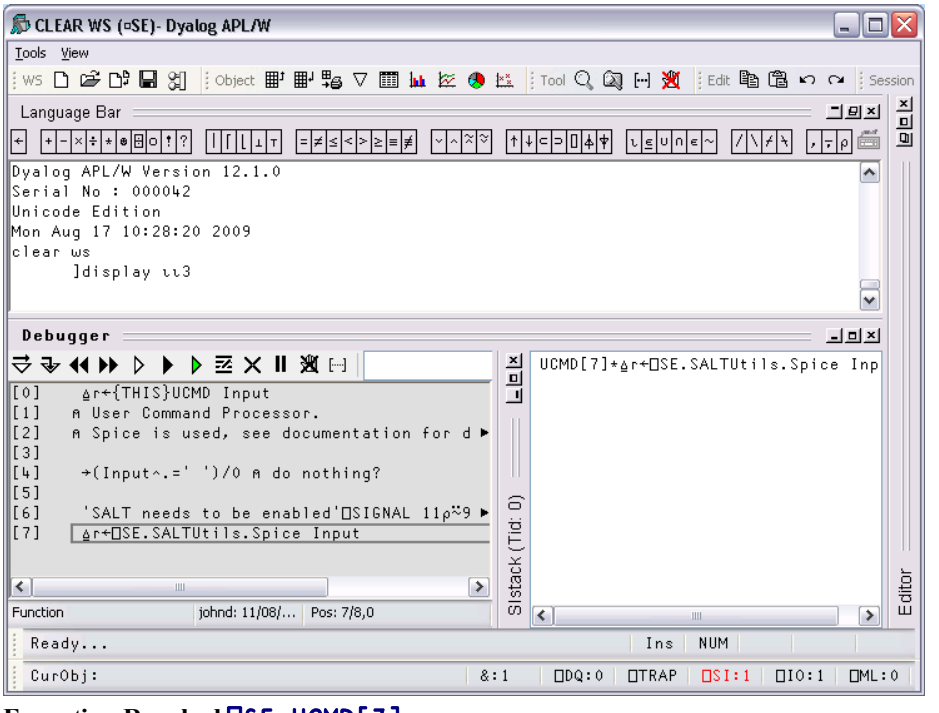
Trace Tools

The Tracer may be controlled from the keyboard, or by using the *Trace Tools* which are arranged along the title bar of the Debugger window. Note that the button names are solely for reference purposes in the description that follows.

Button	Name	Key Code	Keystroke	Description
	Exec	ER	Enter	Executes the current line
	Trace	TC	Ctrl+Enter	Traces execution of the current line
	Back	BK	Ctrl+Shift+Bksp	Skips back one line
	Fwd	FD	Ctrl+Shift+Enter	Skips forward one line
	Restart	RM	→⎕LC	Restarts execution of the current thread, closing all its trace windows
	Restart all			Restarts execution for all threads, closing all trace windows
	Continue	BH		Continues execution of the current thread, leaving Trace windows displayed
	Edit	ED	Shift+Enter	Invokes the Editor
	Exit	EP	Esc	Closes the Trace window, exits the current function
	Intr		Ctrl+Pause	Interrupts execution
	Reset	CB		Clears all break-points (resets ⎕STOP on every function)

Using the Trace Tools, you can **single-step** through the function or operator by clicking the *Exec* and/or *Trace* buttons. If you click *Exec* the current line of the function or operator is executed and the system halts at the next line. If you click *Trace*, the current line is executed but any defined functions or operators referenced on that line are themselves traced. After execution of the line the system again halts at the next one. Using the keyboard, the same effect can be achieved by pressing Enter or Ctrl+Enter.

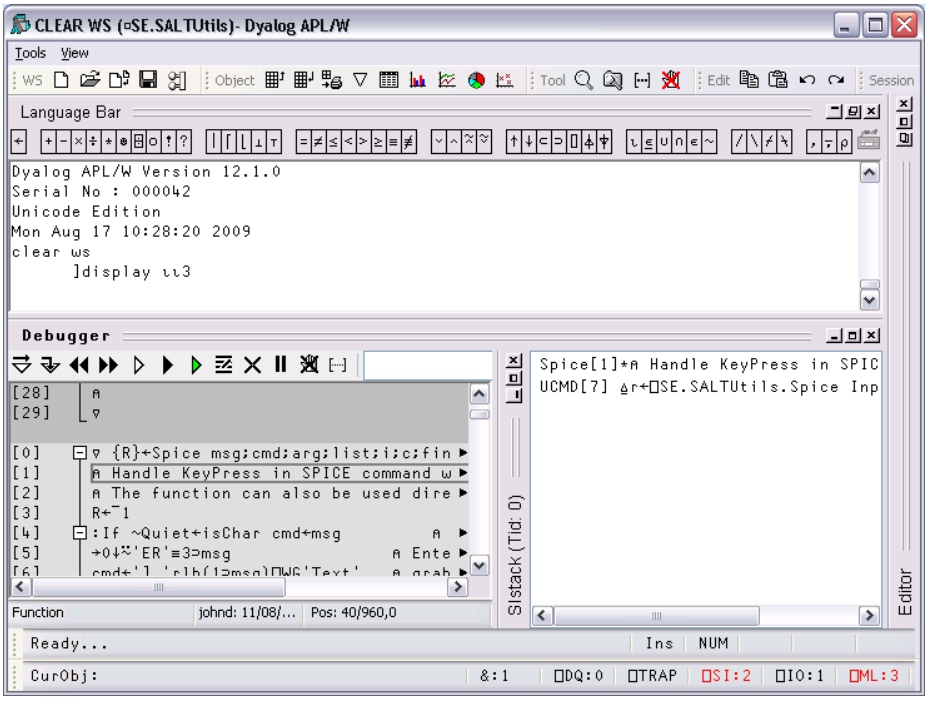
The illustration below shows the state of execution having clicked *Exec* 6 times to reach `⎕SE.UCMD[7]`.



Execution Reached `⎕SE.UCMD[7]`

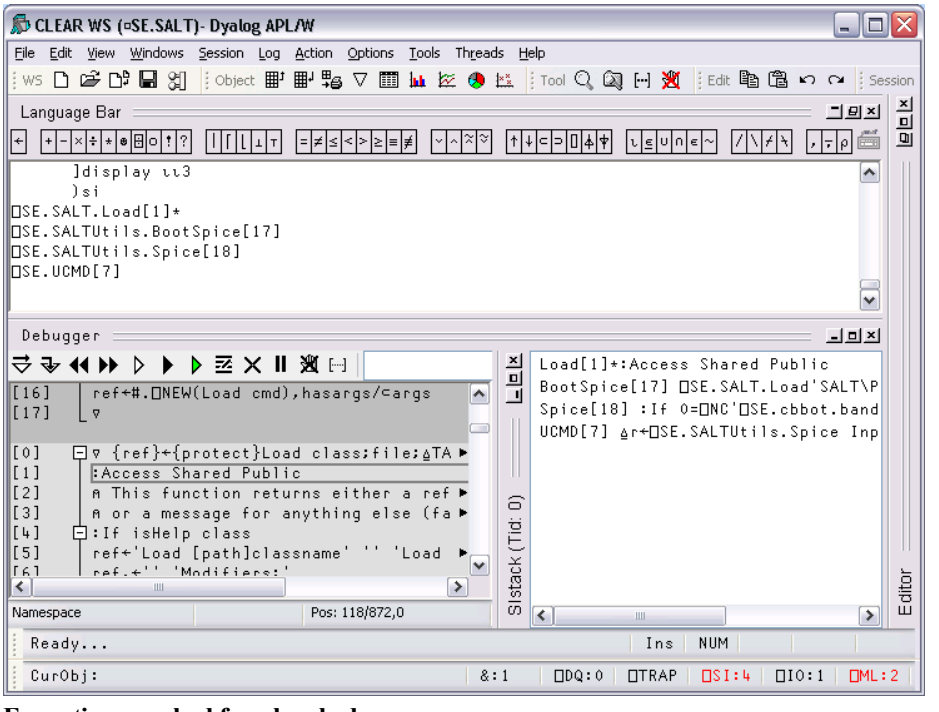
The next illustration shows the result of clicking *Trace* at this point. This caused the system to trace into `⎕SE.SaltUtils.Spice`, the function called from `⎕SE.UCMD[7]`.

Notice how each function call on the stack is represented by an item in the *SIstack* window.



Execution Reached ⎕SE.SALTUtils.Spice [1]

The illustration below shows the state of execution having traced deeper into the system.



Execution reached four levels deep

At this stage, the State Indicator is as follows:

```
      )SI
⎕SE.SALT.Load[1]*
⎕SE.SALTUtils.BootSpice[17]
⎕SE.SALTUtils.Spice[18]
⎕SE.UCMD[7]
```

Controlling Execution

The point of execution may be moved by clicking the *Back* and *Fwd* buttons in the *Trace Tools* window or, using the keyboard, by pressing Ctrl+Shift+Bksp and Ctrl+Shift+Enter. Notice however that these buttons do not themselves change the State Indicator or the display in the *SIStack* window. This happens only when you restart execution from the new point.

You can cut back the stack by clicking the <EP> button in the *Trace Tools* window. This causes execution to be suspended at the start of the line which was previously traced. The same effect can be achieved using the keyboard by pressing Esc. It can also be done by selecting *Exit* from the *File* menu on the Trace Window or by selecting *Close* from its system menu.

The <RM> button removes the Trace window and resumes execution. The same is achieved by the expression →⎕LC. The <BH> button also continues execution, but leaves the Trace window displayed and allows you to watch its progress.

Using the Session and the Editor

Whilst using the Tracer you can skip to the Session or to any Edit window and back again. While it is docked, you may resize the Tracer pane by dragging its title bar, and you may use the buttons provided to maximise, minimise and restore the Tracer pane within the Session window.

Unless you move it, the cursor is positioned to the left of the suspended line in the top Trace window.

Depending where the cursor is in the tracer window, pressing Shift+Enter (ED) or selecting *Edit* from the *File* menu may cause an edit window to open. If the cursor is in the first column of the Trace window, or on whitespace, the Editor is opened on function or operator on top of the stack. If the cursor in on a name, the Editor is opened on the name under the cursor (point-and-edit). With the cursor in any other location, no action is undertaken.

When you finish editing, the window reverts to a trace window with the new definition of the function or operator displayed.

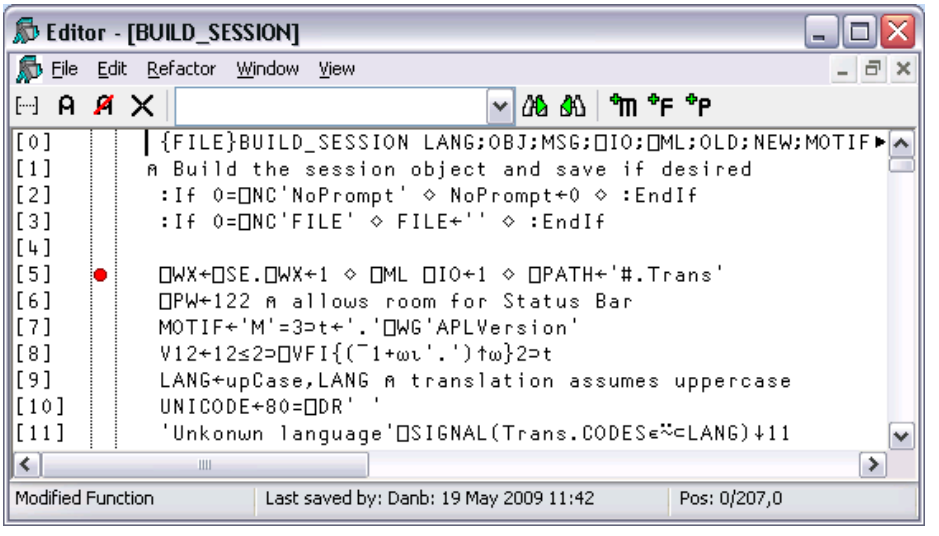
You may also open a new edit window from within the Tracer using point-and-edit.

You can copy text from a trace window to the session for editing and execution or for experimentation.

It is possible to skip from the Tracer to the Session and then re-invoke the Tracer on a different expression.

Setting Break-Points

Break-points are defined by ⎕STOP and may be toggled on and off in an Edit or Trace window by clicking in the appropriate column. The example below illustrates a function with a ⎕STOP break-point set on line [5].



⎕STOP break-points set or cleared in an Edit window are not established until the function is fixed. ⎕STOP break-points set or cleared in a Trace window are established immediately.

Clearing All Break-Points

You can clear all break-points by pressing the above button in the Trace Tools window. This in fact resets ⎕STOP for all functions in the workspace.

The Classic mode Tracer

If you select *Classic Dyalog mode* from the *Trace/Edit* tab in the *Configuration* dialog box, the Tracer behaves in the same way as in Dyalog APL Version 8.2. However, the Tracer is not dockable in the Session.

If you select the Classic mode Tracer, you may choose between multiple trace windows or a single trace window using the *Single Trace Window* option.

Multiple Trace Windows

The following behaviour is obtained by **deselecting** the *Single Trace Window* option.

- Each function on the SI stack is represented by a separate trace window. The top window contains the function that is currently executing, other windows display functions further up the stack, in the order in which they were called.
- When you press Ctrl+Enter or click the *Trace* button on a line that calls another function, a new trace window appears on top of the stack and displays the newly called function.
- When a function exits, its trace window disappears and the focus moves to the previous trace window. When the last function in a traced suspension exits, the last trace window disappears.
- If you click the *Quit this function* button in the *Trace Tools* window, or press Escape, or close the trace window by clicking on its [X] button or typing Alt-F4, the top trace window disappears and the focus moves to the previous trace window
- If you close any of the trace windows further down the stack, the stack will be cut back to the corresponding point, i.e. to the line of code that called the function whose trace window you closed.
- The <RM> button removes all the trace windows and resumes execution. The same is achieved by the expression →⎕LC. The <CS> button also continues execution, but leaves the trace windows displayed and allows you to watch their progress.
- If you minimise any of the trace windows, the entire stack is minimised to a single icon, from which it may be restored.

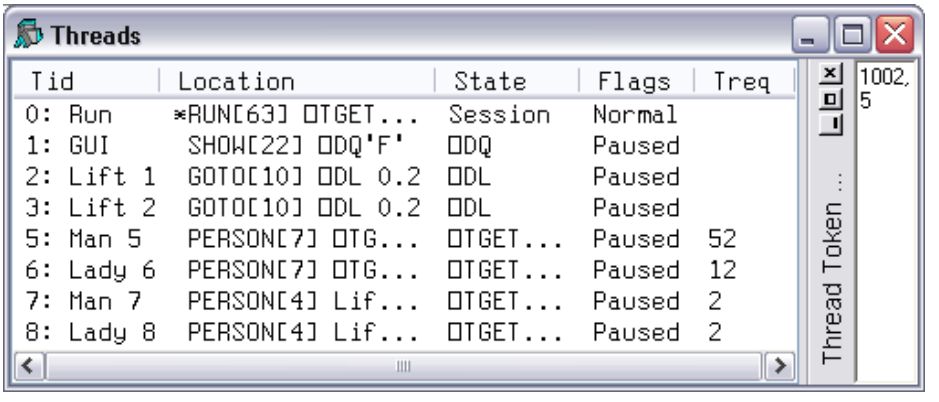
Single Trace Window

The following behaviour is obtained by **selecting** the Single Trace Window option.

- The trace window contains a combo box whose drop-down displays the contents of the SI stack. This box is not provided if there are multiple trace windows.
- The trace window is re-used when tracing into, or returning from, a called function. This means that there is never more than one trace window present.
- When the last function in a traced suspension exits, the trace window disappears.
- If you click the *Quit this function* button in the *Trace Tools* window, or press Escape, the current function is removed from the stack and the trace window reused to display the calling function if there is one.
- Closing the trace window by clicking on its [X] button or typing Alt-F4 removes the window and *clears the current suspension*. It is equivalent to typing naked branch (→) in the session window.
- If you move or resize the trace window, APL remembers its position, so that it reappears in the same position when next used.

The Threads Tool

The Threads Tool is used to monitor and debug multi-threaded applications. To display the Threads Tool, select *Show Threads Tool* from the *Session Threads* menu, or *Threads* from the Session pop-up menu.



The above picture illustrates a situation using the LIFT.DWS workspace after executing the function `RUN`. The *Pause on Error* option was enabled and a Stop was set on `RUN[63]`. When `RUN` suspended at this point, all other threads (1-8) were automatically Paused. Note that all other threads happen to be Paused in the middle of calls to system functions

The columns of the Threads Tool display the following information.

Column	Description
Tid	The Thread ID (`⎕TID`) and name (`⎕TNAME`) if set
Location	The currently executing line of function code
State	Indicates what the thread is doing. (see below)
Flags	Normal or Paused.
Treq	The Thread Requirements (`⎕TREQ`)

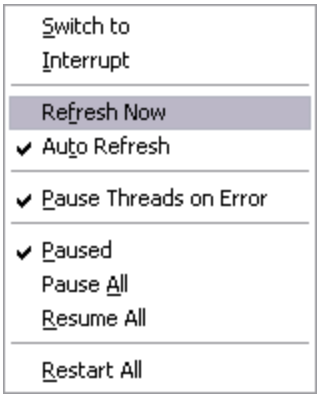
Thread States

State	Description
Pending	Not yet running
Initializing	Not yet running
Defined function	Between lines of a defined function
Dfn	Between lines of a dfn
Suspended	Indicates that the thread is suspended and is able to accept input from the Session window.
Session	Indicates that Session window is connected to this thread.
(no stack)	Indicates that the thread has no SI stack and the Session is connected to another thread. This state can only occur for Thread 0.
Exiting	About to be terminated
`:Hold`	Waiting for a `:Hold` token
`:EndHold`	Waiting for a `:Hold` token
`⎕DL`	Executing `⎕DL`
`⎕DQ`	Executing `⎕DQ`
`⎕NA`	Waiting for a DLL (`⎕NA`) call to return.
`⎕TGET`	Executing `⎕TGET`, waiting for a token
`⎕TGET` (Ready to continue)	Executing `⎕TGET`, having got a token
`⎕TSYNC`	Waiting for another thread to terminate
Awaiting request	Indicates a thread that is associated with a .NET system thread, but is currently unused
Called .NET	Waiting for a call to .NET to return.

Paused/Normal

In addition to the thread state as described above, a thread may be *Paused* or *Normal* as shown in the *Flags* column. A *Paused* thread is one that has temporarily been removed from the list of threads that are being scheduled by the thread scheduler. A *Paused* thread is effectively frozen.

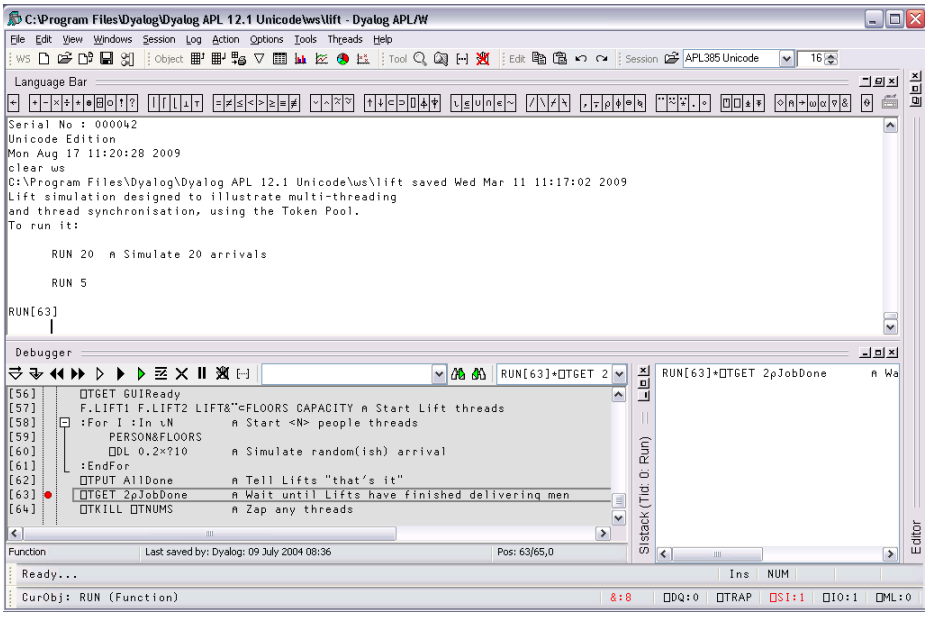
Threads Tool Pop-Up Menu



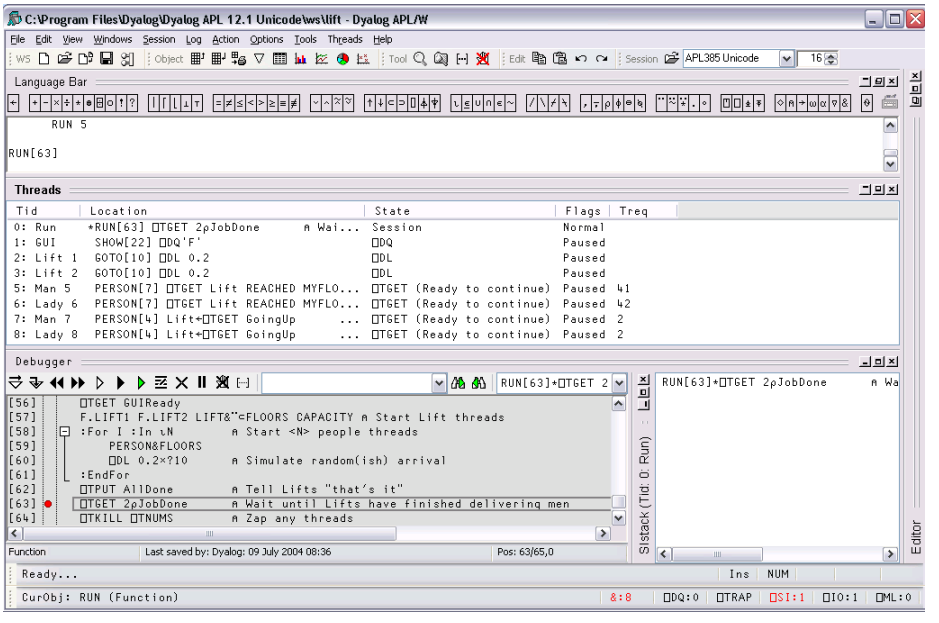
Switch to	Selecting this item causes APL to attempt to suspend (if necessary) and switch to the selected thread, connecting it to the Session and Debugger windows.
Refresh Now	Refreshes the *Threads Tool* display to show the current position and state of each thread.
Auto Refresh	Selecting this item causes the *Threads Tool* to be updated continuously, so that it shows the latest position and state of each thread.
Pause Threads on Error	If this item is checked, APL automatically Pauses all other threads when a thread suspends due to an error or an interrupt.
Paused	This item toggles a thread between being *Paused* and *Normal*. It Pauses a *Normal* thread and resumes a *Paused* thread.
Pause All	This item causes all threads to be *Paused*.
Resume All	This item resumes all threads.
Restart All	This item resumes all *Paused* threads, restarts all suspended threads, and closes the Debugger.

Debugging Threads

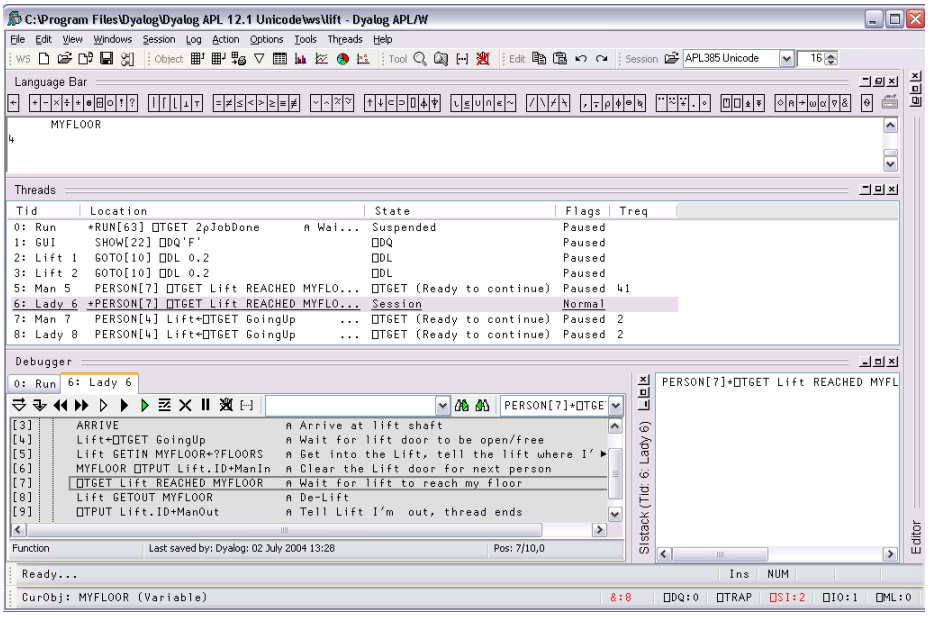
The Debugger provides a tabbed interface that allows you to easily switch between suspended threads for debugging purposes. To keep things simple for non-threaded applications, Tabs are only displayed if there is a thread suspended that is other than Thread 0. The following picture shows the Debugger open on a multi-threaded application (LIFT.DWS) when only Thread 0 is suspended. This has been achieved by setting a stop on `RUN[63]`



In the next picture, the user has chosen to display the *Threads Tool* and then dock it between the Session and Debugger windows. Note that only one thread, thread 0 (Run) is suspended. All the other threads are *Paused* (because *Pause on Error* is enabled).

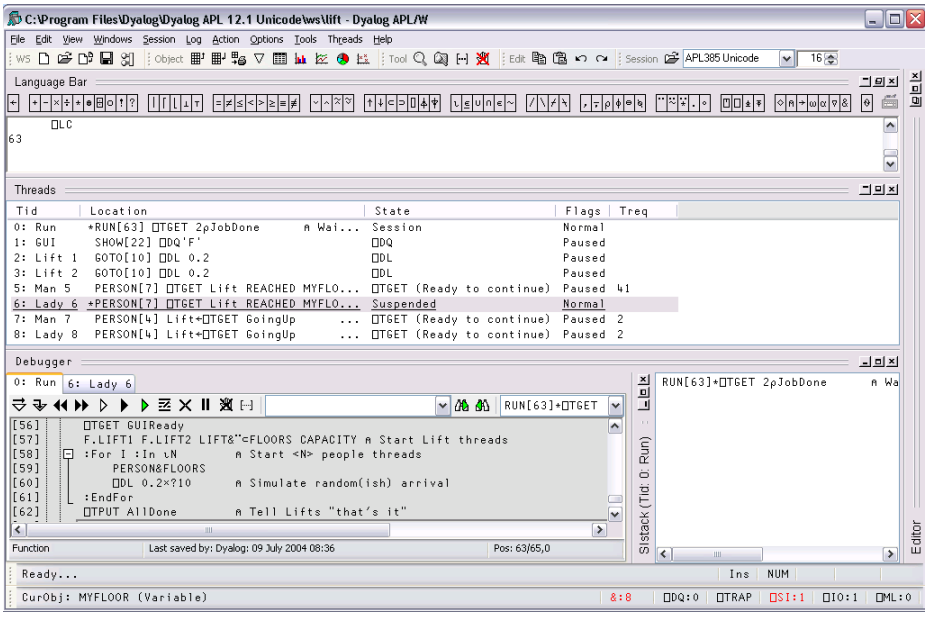


The user then uses the context menu to *Switch To* Thread 6 (whose name is `Lady 6`) which was Paused on `PERSON[7]` in the middle of a `⎕TGET`. The act of switching to this thread caused it to be suspended at the beginning of its current line `PERSON[7]` and the Debugger now displays two Tabs to represent the two suspended threads. Note that both the thread id and the thread name are displayed on the Tabs.



Note also that the Session window is connected to the thread indicated by the selected Tab. In this case, typing `MYFLOOR` into the Session window displays the value of the local variable `MYFLOOR` in Thread 6 (`Lady 6`).

You can use the Tabs to switch between the suspended threads, so clicking the Tab labelled `0:Run` causes the display to change to the picture shown below. The Session is now connected to Thread 0 (`Run`), so the value of `⎕LC` is 63.

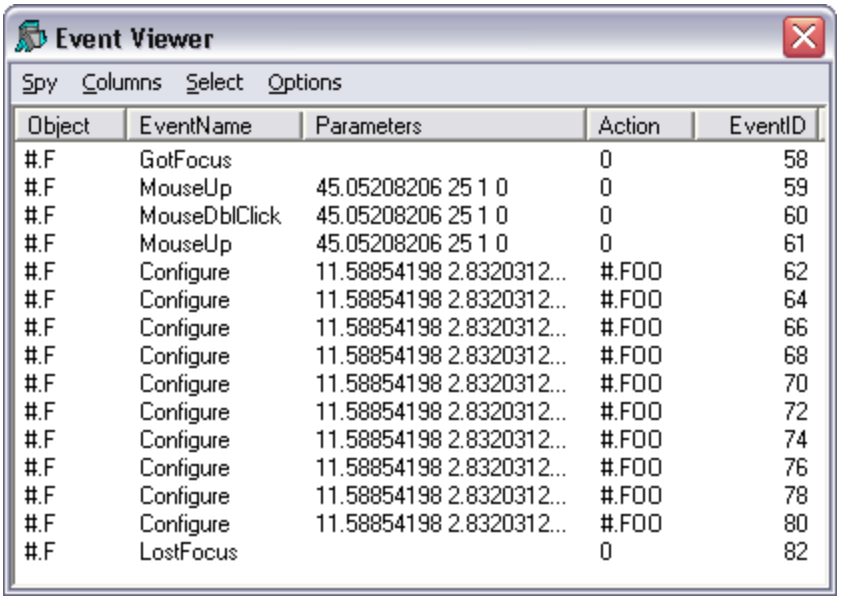


The Event Viewer

The *Event Viewer* can be used to monitor events on Dyalog APL GUI objects. To display the *Event Viewer*, select *Event Viewer* from the *Session Tools* menu.

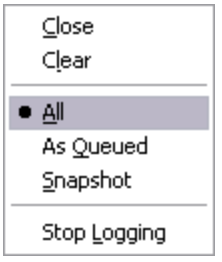
You can choose:

- which types of events you want to monitor
- which objects you want to monitor



In the example illustrated above, the user has chosen to monitor events on a Form `#.F`. Furthermore, the user has chosen to monitor GotFocus, LostFocus, MouseUp, MouseDblClick and Configure events. Notice that there is a callback `#.FOO` attached to the Configure event.

The Spy Menu



The Spy menu, illustrated above, provides the following options and actions.

Close:	Closes the *Event Viewer*
Clear:	Clears all of the event information that is currently displayed in the *Event Viewer*.
All:	In this mode all the events are displayed in the *Event Viewer* as they occur, whether or not there is an action associated with them.
As Queued:	In this mode only events that have associated actions are displayed in the event viewer. Note that KeyPress events are always queued and therefore always appear, even if there is no associated action.
SnapShot:	In this mode the *Event Viewer* displays a snapshot of the internal event queue. Only those events that are currently in the internal APL event queue waiting to be processed are displayed.
Stop Logging:	When checked, this item switches event logging off.

The Columns Menu

The Columns menu allows you to choose which information is displayed for the events you are monitoring.

Object	If checked, this item displays the *name of the object* on which the event occurred.
Event Name	If checked, this item displays the *name* of the event that occurred.
Event Number	If checked, this item displays the *event number* of the event that occurred.
Parameters	If checked, this item displays the *parameters* for the event that occurred. These are the items that would be passed in the argument to a callback function.
Action	If checked, this item displays the *action* associated with the event, for example the name of a callback function, or an expression to be executed.
Thread ID	If checked, this item displays the *thread id* of the thread in which the event occurred
Nqed	If checked, this item displays 0 or 1 according to whether or not the event occurred *naturally* or was generated programmatically by ⎕NQ.
Event ID	If checked, this item displays the *event id* of the event that occurred. This id is used internally.

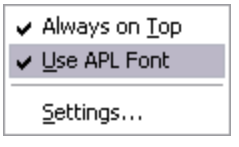
The Select Menu

The *Select* menu allows you to highlight certain events in the *Event Viewer*. For example, if you are monitoring TCP/IP events on a number of TCPSockets, you can highlight just the events for a particular socket.

Select Matching Events	Highlights all the events that have the same Object and Event Name (or Event Number) as the currently selected event.
Select All Events On This Object	Highlights all the events that have the same Object as the currently selected event.
Select All Events Of This Type	Highlights all the events that have the same Event Name (or Event Number) as the currently selected event

These items are also available from the pop-up menu that appears when you press the right mouse button over an event displayed in the *Event Viewer* window.

The Options Menu

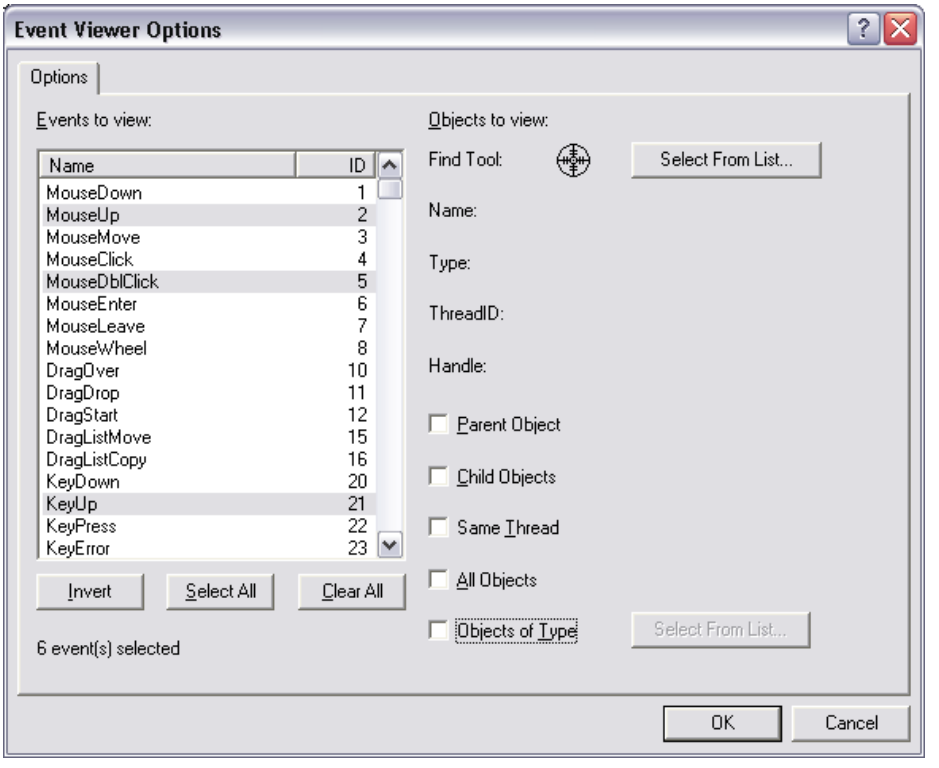


The *Options* menu allows you to choose which information is displayed for the events you are monitoring.

Always on Top	If checked, this item causes the *Event Viewer* window to be displayed above all other windows (including other application windows).
Use APL font	If checked, this item causes the information displayed in the *Event Viewer* window to be displayed using the APL font (the same font as is used in the Session window). If not, the system uses the appropriate Windows font.
Settings...	Displays the *Event Viewer Options* Dialog Box.

Options Dialog Box

The *Event Viewer Options* dialog box allows you to select the objects and events that you wish to monitor.



Events to view

The list box shows all the events that are support by the Dyalog APL GUI and allows you to select which events are to be monitored. Only those events that are selected will be reported. You can sort the events by name or by event number by clicking the appropriate column header.

Objects to view

All Objects	If checked, this item enables event reporting on all Dyalog APL GUI objects.
Objects of Type	If checked, this item activates the adjoining *Select* button and disables all other Object selection mechanisms. Clicking the *Select* button brings up a dialog box that allows you to choose which types of Dyalog APL GUI objects you want to monitor.
Find Tool	This tool allows you to choose a single specific Dyalog APL GUI object that you want to monitor. To use it, drag the *Find Tool* and move it over your Dyalog APL GUI objects. As you drag it, the individual objects are highlighted and their details displayed in the *Name, Type*, *Thread ID* and *Handle* fields. Drop the *Find Tool* on the object of your choice.
Select	Clicking this button brings up a dialog box that displays the entire Dyalog APL GUI structure as a tree view. You can choose a single object by selecting it.

The Session Object

Purpose: The Session object ⎕SE is a special system object that represents the session window and acts as a parent for the session menus, tool bar(s) and status bar.

Children Form, MenuBar, Menu, MsgBox, Font, FileBox, Printer, Bitmap, Icon, Cursor, Clipboard, Locator, Timer, Metafile, ToolBar, StatusBar, TipField, TabBar, ImageList, PropertySheet, OLEClient, TCPSocket, CoolBar, ToolControl, BrowseBox

Properties Type, Caption, Posn, Size, File, Coord, State, Event, FontObj, YRange, XRange, Data, TextSize, Handle, HintObj, TipObj, CurObj, CurPos, CurSpace, Log, Input, Popup, RadiusMode, MethodList, ChildList, EventList, PropList

Methods ChooseFont, FileRead, FileWrite

Events Close, Create, FontOK, FontCancel, WorkspaceLoaded, SessionPrint

Description

There is one (and only one) object of type Session and it is called ⎕SE. You may use ⎕WG, ⎕WS and ⎕WN to perform operations on ⎕SE, but you cannot expunge it with ⎕EX nor can you recreate it using ⎕WC. You may however expunge all its children. This will result in a bare session with no menu bar, tool bar or status bar.

⎕SE is loaded from a session file when APL starts. The name of the session file is specified by the **session_file** parameter. If no session file is defined, ⎕SE will have no children and the session will be devoid of menu bar, tool bar and status bar components.

You may use all of the standard GUI system functions to build or configure the components of the Session to your own requirements. You may also control the Session by changing certain of its properties.

Note that the Session reports a Create event when APL is first started, and a WorkspaceLoaded event when a workspace is loaded or on a clear ws.

The Session also reports a SessionPrint event when certain types of output are about to be displayed. This may be used to alter the normal default display.

Read-Only Properties

The following properties of ⎕SE are read-only and may not be set using ⎕WS:

Type	A character vector containing `'Session'`
Caption	A character vector containing the current caption in the title bar of the Session window.
TextSize	Reports the bounding rectangle for a text string. For a full description, see TextSize in Object Reference.
CurObj	A character vector containing the name of the current object. This is the name under or immediately to the left of the input cursor.
CurPos	A 2-element integer vector containing the position of the input cursor (row and column number) in the session log. This is ⎕IO dependent. If ⎕IO is 1, and the cursor is positioned on the character at the beginning of the first (top) line in the log, CurPos is (1 1). If ⎕IO is 0, its value would be (0 0).
CurSpace	A character vector which identifies the namespace from which the current expression was executed. If the system is not executing code, CurSpace is the current space and is equivalent to the result of `⊃''⎕NS''`.
Handle	The window handle of the Session window.
Log	A vector of character vectors containing the most recent set of lines (input statements and results) that are recorded in the session log. The first element contains the top line in the log.
Input	A vector of character vectors containing the most recent set of input statements (lines that you have executed) contained in the input history buffer.
ChildList	A vector of character vectors containing the types of object that can be created as a child of ⎕SE.
MethodList	A vector of character vectors containing the names of the methods associated with ⎕SE.
EventList	A vector of character vectors containing the names of the events generated by ⎕SE
PropList	A vector of character vectors containing the names of the properties associated with ⎕SE.

Read/Write Properties

The following properties of ⎕SE may be changed using ⎕WS:

Coord	Specifies the co-ordinate system for the session window.
Data	May be used to associate arbitrary data with the session object ⎕SE.
Event	You may use this property to attach an expression or callback function to the Create event or to user-defined events. A callback attached to the Create event can be used to initialise the Session when APL starts.
File	The full pathname of the session file that is associated with the current session. This is the file name used when you save or load the session by invoking the FileRead or FileWrite method.
FontObj	Specifies the APL font. In general, the FontObj property may specify a font in terms of its face name, size, and so forth or it may specify the name of a Font object. For applications, the latter method is recommended as it will result in better management of font resources. However, in the case of the Session object, it is recommended that the former method be used.
HintObj	Specifies the name of the object in which hints are displayed. Unless you specify HintObj individually for session components, this object will be used to display the hints associated with all of the menu items, buttons, and so forth in the session. The object named by this property is also used to display the message "Ready..." when APL is waiting for input.
Popup	A character vector that specifies the name of a popup menu to be displayed when you click the right mouse button in a Session window.
Posn	A 2-element numeric vector containing the position of the top-left corner of the session window relative to the top-left corner of the screen. This is reported and set in units specified by the Coord property.
Size	A 2-element numeric vector containing the height and width of the session window expressed in units specified by the Coord property.

State	An integer that specifies the window state (0=normal, 1=minimised, 2=maximised). You may wish to use this property to minimise and later restore the session under program control. If you save your session with State set to 2, your APL session will start off maximised.
TipObj	Specifies the name of the object in which tips are displayed. Unless you specify TipObj individually for session components, this object will be used to display the tips associated with all of the menu items, buttons, and so forth in the session.
XRange	See *Object Reference*
YRange	See *Object Reference*

Special Events

In addition to the events and methods which are provided by `⎕SE` in common with other GUI objects, the following events are unique to `⎕SE`.

SessionPrint	This event is reported when a value is about to be displayed in the Session window. The default display of the value may be intercepted by a callback function and displayed differently. This event is used by the `]box` and `]rows` user commands.
WorkspaceLoaded	This event is generated when a workspace is loaded or upon `)CLEAR`.

SessionPrint Event 526

Applies To: Session

Description

If enabled, this event is reported when a value is about to be displayed in the Session. It is generated by the display of a variable or the result of a function including system variables and functions. Error messages and output from system commands do not generate this event.

The event message reported as the result of ⎕DQ, or supplied as the right argument to your callback function, is a 2-element vector as follows :

[1]	Object	ref or character vector
[2]	Event	'SessionPrint' or

The attachment of a callback function intercepts and annuls the normal display of any value.

Note that this event may be extended in future; in particular the number of elements in the event message may be increased, and the event may be generated by some system commands. You should therefore allow for such extensions in any code which refers to SessionPrint.

When the event is generated, the left argument of the callback function contains the value which was about to be displayed. The callback function may display this or any other value, using default output or by assignment to ⎕. If so, this output will be processed normally, without generating a subsequent SessionPrint event. If the callback fails to explicitly display anything, nothing will appear in the Session.

Example

```
      ⎕VR'⎕SE.TimeStamp'
     ∇ VAL TimeStamp EV
[1]    ⎕TS VAL
      ∇

      '⎕SE'⎕WS'Event' 'SessionPrint' '⎕SE.TimeStamp'

      2
 2014 9 18 16 20 38 318  2

      ⎕A
 2014 9 18 16 20 44 668  ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The result (if any) of the callback function is ignored.

You may not disable the event (by setting its action to `¯1`), nor generate the event using `⎕NQ`, nor call it as a method.

WorkspaceLoaded Event 525

Applies To: Session

Description

If enabled, this event is reported when a workspace is loaded or on a `clear ws`. You may not nullify or modify the event with a 0-returning callback, nor may you generate the event using `⎕NQ`, or call it as a method.

The event message reported as the result of `⎕DQ`, or supplied as the right argument to your callback function, is a 2-element vector as follows :

`[1]`	Object	ref or character vector
`[2]`	Event	`'WorkspaceLoaded'` or 525

This event is fired immediately after a workspace has been loaded and before the execution of `⎕LX`.

The callback function you attach should be defined in `⎕SE`.

Configuring the Session

As supplied, your default session will have a menu bar, a tool bar and a status bar. There are many ways in which you may configure this set-up, including the following:

You may select a different APL font or character size.

You may alter the appearance of the menus by changing the Caption properties of the various Menu and MenuItem objects. For example, you may prefer the menus to appear in your own language.

You may alter the structure of the menus. For example, you may wish to create a *Search* menu directly on the menu bar rather than having *Find and Replace* as part of the *Edit* menu.

You may add new Menu and MenuItem objects to the menu bar, or new Button objects to the tool bar, that execute APL functions or expressions for you. You can store the code inside the ⎕SE namespace so that it is remains available when you switch from one workspace to another.

You may add other objects to the tool bar to allow you to provide input for your functions or to display output. For example, you may display a Combo object that offers you a selection of names applicable to a particular task.

You may add additional toolbars.

You may remove objects too; for example, you can remove fields from the StatusBar or even delete it entirely. Indeed, you may dispense with the menu bar and/or tool bar as well.

This section illustrates how you can configure your session using worked examples. The examples are by no means exhaustive, but are designed to demonstrate the principles. Please note that the structure and names of the objects used in these examples may not be identical to your default session as supplied. Before you attempt to change your session, please check the structure and the object names using ⎕WN and ⎕WG. The supplied session was created using the function BUILD_SESSION in the workspace BUILDSE. If you wish to make substantial changes to your session, you may find it most convenient to edit the functions in this workspace, re-run BUILD_SESSION, and then save it.

Please note that these examples assume that *Expose Session Properties* is enabled.

Changing the Font

The APL session font is defined by the Font property of ⎕SE. To change the font **permanently**, you should select a different Font and/or size of Font using the combo and spinner boxes on the Session toolbar, and **save your Session**.

Classic Edition is distributed with bitmap fonts suitable for use on your screen, and TrueType fonts for your printer. You *can* use the TrueType font on the screen, but it is less attractive than the bitmap fonts at low resolutions. The bitmap fonts come in two sizes (16 x 8 and 22 x 11) and two weights (normal and bold). You may select other sizes, so long as the height is a multiple of 16 or 22. The scaling is performed automatically by Windows.

Changing Menu Appearance

The name of the Session MenuBar is '⎕SE.mb'. To simplify the specification of object names, we will first change space to the MenuBar itself:

```
      )CS ⎕SE.mb
⎕SE.mb
```

The names of the Menu objects owned by the MenuBar are given by the expression:

```
      'Menu' ⎕WN ''
file  edit  view  windows  session  log  action  options
tools  help
```

The current caption on the file menu is:

```
    file.Caption
&File
```

To change the Caption to Workspace:

```
    file.Caption←'Workspace'
```

To change the colour of the *New* option in the *File* menu to red:

```
    file.clear.FCol←255 0 0
```

Reorganising the Menu Structure

This example shows how you may alter the structure of the session menus by adding a *Search* menu to the menu bar to provide access to the *Find* and *Find/Replace* dialog boxes and removing these options from the *Edit* menu.

To simplify the process, we will first change space into the MenuBar object itself:

```
      )CS ⎕SE.mb
⎕SE.mb
```

Then we can begin by adding the *Search* menu. You can specify where the new menu is to be added using its Posn property. In this case, *Search* will be added at position 3 (after *Edit*).

```
      'search'⎕WC 'Menu' '&Search' 3
```

Next we will remove the Find and Replace MenuItem objects from the *Edit* menu. Their names can be obtained from ⎕WN:

```
      'MenuItem'⎕WN'edit'
edit.prev  edit.next  edit.clear  edit.copy  edit.paste
edit.find  edit.replace
```

It is worth noting that these MenuItems perform their actions because their Event property is set to execute the system operations [Find] and [Replace] respectively when they are selected.

```
      edit.find.Event
  Select  [Find]
      edit.replace.Event
  Select  [Replace]
```

The following statement removes them from the *Edit* menu:

```
      ⎕EX¨'edit.find' 'edit.replace'
```

and the following statements add them to the Search menu:

```
      'search.find' ⎕WC 'MenuItem' '&Find'
                        ('Event' 'Select' '[Find]')
      'search.replace' ⎕WC 'MenuItem' '&Replace'
                        ('Event' 'Select' '[Replace]')
```

Adding your own MenuItem

This example shows how you can add a menu item that executes an APL expression. In this case we will do something very simple; namely add a *Time* option to the Tools menu which will execute ⎕TS. Notice that the statement also defines a Hint. This will be displayed when you select the option, prior to releasing the mouse button to action it.

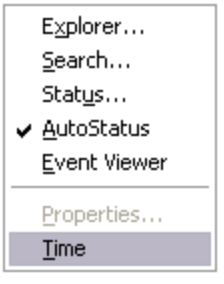
Once again, we will start by changing space into the *Tools* menu itself

```
      )CS ⎕SE.mb.tools
⎕SE.mb.tools
```

Then we will define a new MenuItem to perform the action we require:

```
      'ts'⎕WC'MenuItem' '&Time'
              ('Event' 'Select' '⍎⎕TS')
              ('Hint' 'Display Timestamp')
```

The ⍎ symbol is very important and distinguishes an expression to be executed immediately, as in this case, from a callback function. The resulting *Tools* menu now appears as follows:



A customised Tools menu

Selecting *Time* produces the following output in the session:

```
      2007 12 10 17 10 2 0
```

Adding your own Tool Button

This example shows how you can add a button to the session tool bar that executes an APL function.

The example function we will use is called `XREF`. This function analyses another function, listing the sub-functions that it calls. Instead of returning a result, this example displays the sub-functions in a Form.

```
     ∇ XREF FN;REFS
[1]    :If 0<⍴FN
[2]    :AndIf 3=⎕NC FN
[3]        REFS←⎕REFS FN
[4]        REFS←(3=⎕NC REFS)⌿REFS
[5]        REFS←(↓REFS)~¨' '
[6]        REFS←REFS~⊂FN
[7]        :If 0<⍴REFS
[8]            'F'⎕WC'Form'('Functions called by ',FN)
[9]            F.FontObj←⎕SE.FontObj
[10]           'F.L'⎕WC'List'REFS(0 0)(100 100)
[11]       :EndIf
[12]   :EndIf
     ∇
```

To make this function available from a Session tool button, we need to do a number of things.

Firstly, we must install the function in `⎕SE` so that it is always there, regardless of the current active workspace. This is easily achieved using the Explorer or `⎕NS`.

```
      '⎕SE' ⎕NS 'XREF'
```

Secondly, we need to find another way to specify its argument `FN`. One possibility would be to display a dialog box, asking the user to specify the name of the function to be analysed. A neater solution is to use the CurObj property of `⎕SE` which reports the name under the cursor. Using CurObj, the user can simply place the cursor over the name of the function to be analysed, and then click the `XREF` tool button.

To get `FN` from CurObj, all we need to do is to change the header and lines 1-2 to:

```
[0]   XREF;FN;REFS
[1]   :If 0<⍴FN←⎕SE.CurObj
[2]   :AndIf 3=⎕NC FN←⎕SE.CurSpace,'.',FN
```

Notice that the function name reported by CurObj is prefixed by its pathname which comes from the CurSpace property. This reports the user's current namespace.

Next we will add a new button to the tool bar in the *Tools* CoolBand. Ideally we would use a suitable bitmap, but to simplify the example, we will use a standard text button:

```
      )CS ⎕SE.cbtop.bandtb3.tb
⎕SE.cbtop.bandtb3.tb

      'xref' ⎕WC 'Button' 'XREF'
      'xref' ⎕WS 'Event' 'Select' '⍎⎕SE.XREF'
```

Adding a tool button

User Commands

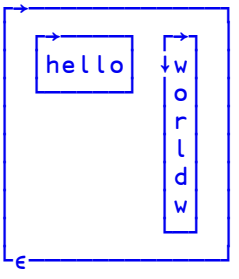
Dyalog APL includes a mechanism to define *User Commands*.

User commands are developer tools, written in APL, which can be executed without having to explicitly copy code into your workspace and/or save it in every workspace in which you want to use it.

A User Command is a name prefixed by a closing square bracket, which may be niladic or take an argument. A User Command executes APL code that is typically stored somewhere outside the current active workspace.

By default, the existing SPICE command processor is hooked up to the user command mechanism, and a number of new SPICE commands have been added. For example:

```
]display 'hello' (⍪'world')
```



The implementation of User Commands is very simple: If a line of input begins with a closing square bracket (`]`), and there exists a function by the name `⎕SE.UCMD`, then the interpreter will call that function, passing the input line (without the bracket) as the right argument.

To add a user command, drop a new Spice command file in the folder SALT\Spice.

Chapter 2:

APL Files

Introduction

Most languages store programs and data separately. APL is unusual in that it allows you to store programs and data together in a workspace.

This can be inefficient if your dataset gets very large; when your workspace is loaded, you are loading ALL of your data, whether you need it or not.

It also makes it difficult for other users to access your data, particularly if you want them to be able to update it.

In these circumstances, you must extract your data from your workspace, and write it to a file on disk, thus separating your data from your program. There are many different kinds of file format. This section is concerned with the APL Component File system which preserves the idea that your data consists of APL objects; hence you can only access this type of file from within APL

The Component File system has a set of system functions through which you access the file. Although this means that you have to learn a whole new set of functions in order to use files, you will find that they provide you with a very powerful mechanism to control access to your data.

Component Files

Overview

A **component file** is a data file maintained by Dyalog APL. It contains a series of APL arrays known as **components** which are accessed by reference to their relative position or **component number** within the file. Component files are just like other data files and there are no special restrictions imposed on names or sizes.

A set of system functions is supplied to perform a range of file operations. These provide facilities to create or delete files, and to read and write components. Facilities are also provided for multi-user access, including the capability to determine who may do what, and file locking for concurrent updates.

Tying and Untying Files

To access an existing component file it must be **tied**, i.e. opened for use. The tie may be **exclusive** (single-user access) or **shared** (multi-user access). A file is **untied**, i.e. closed, using `⎕FUNTIE` or on terminating Dyalog APL. File ties survive `)LOAD`, `⎕LOAD` and `)CLEAR` operations.

Tie Numbers

A file is tied by associating a **file name** with a **tie number**. Tie numbers are integers in the range 1 - 2147483647 and, you can supply one explicitly, or have the interpreter allocate the next available one by specifying 0. The system functions which tie files return the tie number as a 'shy' result.

Creating and Removing Files

A component file is created using `⎕FCREATE` which automatically ties the file for exclusive use. A newly created file is empty, i.e. contains 0 components. A file is removed with `⎕FERASE`, although it must be exclusively tied to do so.

Adding and Removing Components

Components are added to a file using `⎕FAPPEND` and removed using `⎕FDROP`. Component numbers are allocated consecutively starting at 1. Thus a new component added by `⎕FAPPEND` is given a component number which is one greater that that of the last component in the file. Components may be removed from the beginning or end of the file, but not from the middle. Component numbers are therefore contiguous.

Reading and Writing Components

Components are read using ⎕FREAD and overwritten using ⎕FREPLACE. There are no restrictions on the size or type of array which may replace an existing component. Components are accessed by component number, and may be read or overwritten at random.

Component Information

In addition to the data held in a component, the user ID that wrote it and the time at which it was written is also recorded. This control information is useful in providing an audit trail and in facilitating partial backups of components that have changed.

Multi-User Access

⎕FSTIE ties a file for **shared** (i.e. multi-user) access. This kind of access would be appropriate for a multi-user UNIX system, a network of single user PCs, or multiple APL tasks under Microsoft Windows.

⎕FHOLD provides the means for the user to temporarily prevent other co-operating users from accessing one or more files. This is necessary to allow a single logical update involving more than one component, and perhaps more than one file, to be completed without interference from another user. ⎕FHOLD is applicable to External Variables as well as Component Files

File Access Control

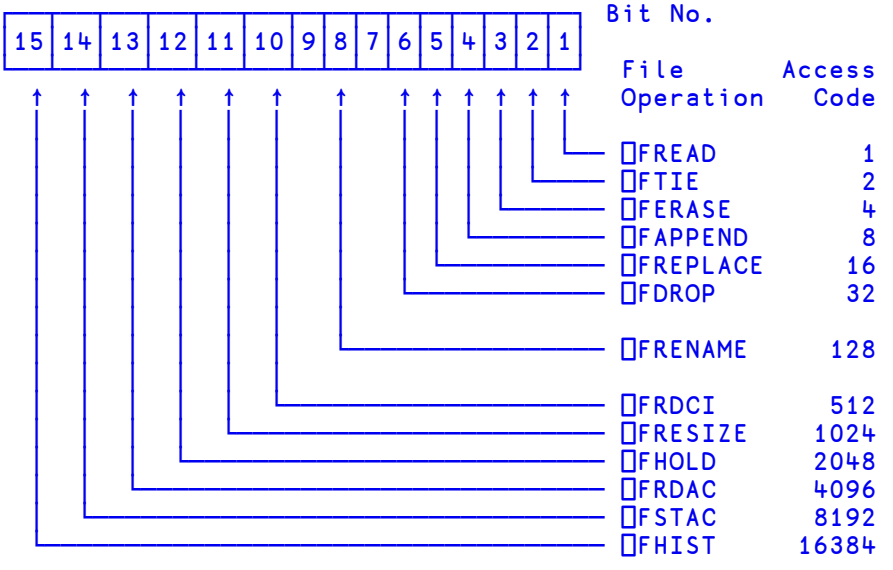
There are two levels of file access control. As a regular data file, the operating system read/write controls for owner and other users apply. In addition, Dyalog APL manages its own access controls using the **access matrix**. This is an integer matrix with 3 columns and any number of rows. Column 1 contains user numbers, column 2 an encoding of permitted file operations, and column 3 passnumbers. Each row specifies which file operations may be performed by which user(s) with which passnumber.

User Number

This is a number which is defined by the **aplnid** parameter. If you intend to use Dyalog APL's **access matrix** to control file access in a multi-user environment, it is desirable to allocate to each user, a distinct **user number**. However, if you intend to rely on under-lying operating system controls, allocating a user number of 0 to everyone is more appropriate. A user number of 0 (which is the installation default), causes APL to circumvent the access matrix mechanism described below.

Permission Code

This is an integer representation of a Boolean mask. Each bit in the mask indicates whether or not a particular file operation is permitted as follows:



For example, if bits 1, 4 and 6 are set and all other relevant bits are zero only ⎕FREAD, ⎕FAPPEND and ⎕FDROP are permitted. A convenient way to set up the mask is to sum the access codes associated with each operation.

For example, the value 41 (1+8+32) authorises ⎕FREAD, ⎕FAPPEND and ⎕FDROP. A value of ¯1 (all bits set) permits all operations. Thus by subtracting the access codes of operations to be forbidden, it is possible to permit all but certain types of access. For example, a value of ¯133 (¯1- 4+128) permits all operations except ⎕FERASE and ⎕FRENAME. Note that the value of unused bits is ignored. Any non-zero permission code allows ⎕FSTIE and ⎕FSIZE. ⎕FCREATE, ⎕FUNTIE, ⎕FLIB, ⎕FNAMES and ⎕FNUMS are not subject to access control. Passnumbers may also be used to establish different levels of access for the same user.

When the user attempts to tie a file using ⎕FTIE or ⎕FSTIE a row of the access matrix is selected to control this and subsequent operations.

If the user is the owner, and the owner's user ID does not appear in the access matrix, the value (⎕AI[1] ¯1 0) is conceptually appended to the access matrix. This ensures that the owner has full access rights unless they are explicitly restricted.

The chosen row is the first row in which the value in column 1 of the access matrix matches the user ID and the value in column 3 matches the supplied passnumber which is taken to be zero if omitted.

If there is no match of user ID and passnumber in the access matrix (including implicitly added rows) then no access is granted and the tie fails with a FILE ACCESS ERROR.

Once the applicable row of the access matrix is selected, it is used to verify all subsequent file operations. The passnumber used to tie the file MUST be used for every subsequent operation. Secondly, the appropriate bit in the permission code corresponding to the file operation in question must be set. If either of these conditions is broken, the operation will fail with `FILE ACCESS ERROR`.

If the access matrix is changed while a user has the file tied, the change takes immediate effect. When the user next attempts to access the file, the applicable row in the access matrix will be reselected subject to the supplied passnumber being the same as that used to tie the file. If access with that password is rescinded the operation will fail with `FILE ACCESS ERROR`.

When a file is created using `⎕FCREATE`, the access matrix is empty. At this stage, the owner has full access with passnumber 0, but no access with a non-zero passnumber. Other users have no access permissions. Thus only the owner may initialise the access matrix.

User 0

If a user has an **aplnid** of 0, the access matrix and supplied passnumbers are ignored. This user is granted full and unrestricted access rights to all component files, subject only to underlying operating system restrictions.

General File Operations

`⎕FLIB` gives a list of **component files** in a given directory. `⎕FNAMES` and `⎕FNUMS` give a list of the names and tie numbers of tied files. These general operations which apply to more than one file are not subject to access controls.

Component File System Functions

See *Language Reference* for full details of the syntax of these system functions.

General	
⎕FAVAIL	Report file system availability
File Operations	
⎕FCREATE	Create a file
⎕FTIE	Tie an existing file (exclusive)
⎕FSTIE	Tie an existing file (shared)
⎕FUNTIE	Untie file(s)
⎕FCOPY	Copy a file
⎕FERASE	Erase a file
⎕FRENAME	Rename a file
File information	
⎕FHIST	Report file events
⎕FNUMS	Report tie numbers of tied files
⎕FNAMES	Report names of tied files
⎕FLIB	Report names of component files
⎕FPROPS	Report file properties
⎕FSIZE	Report size of file
Writing to the file	
⎕FAPPEND	Append a component to the file
⎕FREPLACE	Replace an existing component
Reading from a file	
⎕FREAD	Read one or more components
⎕FRDCI	Read component information

Manipulating a file	
⎕FDROP	Drop a block of components
⎕FRESIZE	Change file size (forces a compaction)
⎕FCHK	Check and repair a file
Access manipulation	
⎕FSTAC	Set file access matrix
⎕FRDAC	Read file access matrix
Control multi-user access	
⎕FHOLD	Hold file(s) - see later section for details

Using the Component File System

Let us suppose that you have written an APL system that builds a personnel database, containing the name, age and place of birth of each employee. Let us assume that you have created a variable DATA, which is a nested vector with each element containing a person's name, age and place of birth:

```
      DISPLAY 2↑DATA
┌→──────────────────────────────────────────────────────┐
│ ┌→──────────────────────┐ ┌→──────────────────────────┐ │
│ │ ┌→──────┐    ┌→────┐ │ │ ┌→──────┐    ┌→────────┐ │ │
│ │ │Jonathan│ 42 │Wales│ │ │ │Pauline│ 21 │Isleworth│ │ │
│ │ └───────┘    └─────┘ │ │ └───────┘    └─────────┘ │ │
│ └∊─────────────────────┘ └∊─────────────────────────┘ │
└∊──────────────────────────────────────────────────────┘
```

Then the following APL expressions can be used to access the database:

Example 1:

Show record 2

```
      DISPLAY 2⊃DATA
┌→─────────────────────────┐
│ ┌→──────┐    ┌→────────┐ │
│ │Pauline│ 21 │Isleworth│ │
│ └───────┘    └─────────┘ │
└∊─────────────────────────┘
```

Example 2:

How many people in the database?

```
      ⍴DATA
3
```

Example 3:

Update Pauline's age

```
(2 2⊃DATA)←16
```

Example 4:

Add a new record to the database

```
DATA ,← ⊂'Maurice' 18 'London'
```

Now let's build a component file to hold our personnel database.

Create a new file, giving the file name, and the number you wish to use to identify it (the file tie number):

```
'COMPFILE' ⎕FCREATE 1
```

If the file already exists, or you have already used this tie number, then APL will respond with the appropriate error message.

Now write the data to the file. We could write a function that loops to do this, but it is neater to take advantage of the fact that our data is a nested vector, and use each (¨).

```
DATA ⎕FAPPEND¨ 1
```

Now we'll try our previous examples using this file.

Example 1:

Show record 2

```
      DISPLAY ⎕FREAD 1 2
.→-------------------------.
| .→------.    .→--------. |
| |Pauline| 21 |Isleworth| |
| '-------'    '---------' |
'←-------------------------'
```

Example 2:

How many people in our database?

```
      ⎕FSIZE 1          ⍝ First component, next
1 125 10324 4294967295  ⍝ component, file size,
                        ⍝ maximum file size

      ¯1+2⊃⎕FSIZE 1     ⍝ Number of data items
```

The fourth element of ⎕FSIZE indicates the file size limit. Dyalog APL does not impose a file size limit, although your operating system may do so, but the concept is retained in order to make this version of Component Files compatible with others.

Example 3:

Update Pauline's age

```
      REC ← ⎕FREAD 1 2        ⍝ Read second component
      REC[2] ← 18             ⍝ Change age
      REC ⎕FREPLACE 1 2       ⍝ And replace component
```

Example 4:

Add a new record

```
      ('Janet' 25 'Basingstoke') ⎕FAPPEND 1
```

Example 5:

Rename our file

```
      'PERSONNEL' ⎕FRENAME 1
```

Example 6:

Tie an existing file; give file name and have the interpreter allocate the next available tie number.

```
      'SALARIES' ⎕FTIE 0
 2
```

Example 7:

Give everyone access to the PERSONNEL file

```
      (1 3⍴0 ¯1 0)⎕FSTAC 1
```

Example 8:

Set different permissions on `SALARIES`.

```
      AM ← 1 3⍴1 ¯1 0     ⍝ Owner ID 1 has full access
      AM⍪← 102 1 0        ⍝ User ID 102 has READ only
      AM⍪← 210 2073 0     ⍝ User ID 210 has
                          ⍝  READ+APPEND+REPLACE+HOLD

      AM ⎕FSTAC 2         ⍝ Store access matrix
```

Example 9:

Report on file names and associated numbers

```
      ⎕FNAMES,⎕FNUMS
PERSONNEL  1
SALARIES   2
```

Example 10:

Untie all files

```
      ⎕FUNTIE ⎕FNUMS
```

Programming Techniques

Controlling Multi-User Access

Obviously, Dyalog APL contains mechanisms that prevent data getting mixed up if two users update a file at the same time. However, it is the programmer's responsibility to control the logic of multi-user updates.

For example, suppose two people are updating our database at the same time. The first checks to see if there is an entry for `'Geoff'`, sees that there isn't so adds a new record. Meanwhile, the second user is checking for the same thing, and so also adds a record for `'Geoff'`. Each user would be running code similar to that shown below:

```
    ∇  UPDATE;DATA;NAMES
[1]    ⍝ Using the component file
[2]    'PERSONNEL' ⎕FSTIE 1
[3]    NAMES←⊃⊖⎕FREAD ¨ 1,¨⍳¯1+2⊃⎕FSIZE 1
[4]    →END×⍳(⊂'Geoff')∈NAMES
[5]    ('Geoff' 41 'Hounslow')⎕FAPPEND 1
[6]    END:⎕FUNTIE 1
    ∇
```

The system function `⎕FHOLD` provides the means for the user to temporarily prevent other co-operating users from accessing one or more files. This is necessary to allow a single logical update, perhaps involving more than one record or more than one file, to be completed without interference from another user.

The code above is replaced by that below:

```
    ∇ UPDATE;DATA;NAMES
[1] ⍝ Using the component file
[2]  'PERSONNEL' ⎕FSTIE 1
[3]  ⎕FHOLD 1
[4]  NAMES←⊃⍪⎕FREAD ¨ 1,¨⍳¯1+2⊃⎕FSIZE 1
[5]  →END×⍳(⊂'Geoff')∈NAMES
[6]  ('Geoff' 41 'Hounslow')⎕FAPPEND 1
[7] END:⎕FUNTIE 1 ⋄ ⎕FHOLD ⍳0
    ∇
```

Successive ⎕FHOLDs on a file executed by different users are queued by Dyalog APL; once the first ⎕FHOLD is released, the next on the queue holds the file. ⎕FHOLDs are released by return to immediate execution, by ⎕FHOLD ⍬, or by erasing the external variable.

It is easy to misunderstand the effect of ⎕FHOLD. It is NOT a file locking mechanism that prevents other users from accessing the file. It only works if the tasks that wish to access the file co-operate by queuing for access by issuing ⎕FHOLDs. It would be very inefficient to issue a ⎕FHOLD on a file then allow the user to interactively edit the data with the hold in operation. What happens if he goes to lunch? Any other user who wants to access the file and cooperates by issuing a ⎕FHOLD would have to wait in the queue for 3 hours until the first user returns, finishes his update and his ⎕FHOLD is released. It is usually more efficient (as well as more friendly) to issue ⎕FHOLDs around a small piece of critical code.

Suppose we had a control file associated with our personnel data base. This control file could be an external variable, or a component file. In both cases, the concept is the same; only the commands needed to access the file are different. In this example, we will use a component file:

```
'CONTROL'⎕FCREATE 1     ⍝ Create control file
(1 3⍴0 ¯1 0) ⎕FSTAC 1   ⍝ Allow everyone access
⍬ ⎕FAPPEND 1            ⍝ Set component 1 to empty
⎕FUNTIE 1               ⍝ And untie it
```

Now we'll allow our man that likes long lunch breaks to edit the file, but will control the hold in a more efficient way:

```
     ∇  EDIT;CMP;CV
[1]   ⍝ Share-tie the control file
[2]    'CONTROL' ⎕FSTIE 1
[3]   ⍝ Share-tie the data file
[4]    'PERSONNEL' ⎕FSTIE 2
[5]   ⍝ Find out which component the user wants to edit
[6]    ASK:CMP←ASK⍙WHICH⍙RECORD
[7]   ⍝ Hold the control file
[8]    ⎕FHOLD 1
[9]   ⍝ Read the control vector
[10]   CV←⎕FREAD 1 1
[11]  ⍝ Make control vector as big as the data file
[12]   CV←(¯1+2⊃⎕FSIZE 2)↑CV
[13]  ⍝ Look at flag for this component
[14]   →(FREE,INUSE)[1+CMP⊃CV]
[15]  ⍝ In use - tell user and release hold
[16]  INUSE:'Record in use' ⋄ ⎕FHOLD 0 ⋄ →ASK
[17]  ⍝ Ok to use - flag in-use and release hold
[18]  FREE:CV[CMP]←1 ⋄ CV ⎕FREPLACE 1 1⋄ ⎕FHOLD 0
[19]  ⍝ Let user edit the record
[20]   EDIT⍙RECORD RECORD
[21]  ⍝ When he's finished, clear the control vector
[22]   ⎕FHOLD 1
[23]  CV←⎕FREAD 1 1 ⋄CV[CMP]←0 ⋄ CV ⎕FREPLACE 1 1
[26]   ⎕FHOLD 0
[27]  ⍝ And repeat
[28]   →ASK
     ∇
```

Component 1 of our CONTROL file acts as a control vector. Its length is set equal to the number of components in the PERSONNEL file, and an element is set to 1 if a user wishes to access the corresponding data component. Only the control file is ever subject to a ⎕FHOLD, and then only for a split-second, with no user inter-action being performed whilst the hold is active.

When the first user runs the function, the relevant entry in the control vector will be set to 1. If a second user accesses the database at the same time, he will have to wait briefly whilst the control vector is updated. If he wants the same component as the first user, he will be told that it is in use, and will be given the opportunity to edit something else.

This simple mechanism allows us to lock the components of our file, rather the than entire file. You can set up more informative control vectors than the one above; for example, you could easily put the user name into the control vector and this would enable you to tell the next user who is editing the component he is interested in.

File Design

Our personnel database could be termed a *record oriented* system. All the information relating to one person is easily obtained, and information relating to a new person is easily added, but if we wish to find the oldest person, we have to read ALL the records in the file.

It is sometimes more useful to have separate components, perhaps stored on separate files, that hold indexes of the data fields that you may wish to search on. For example, suppose we know that we always want to access our personnel database by name. Then it would make sense to hold an index component of names:

```
⍝ Extract name field from each data record
'PERSONNEL' ⎕FSTIE 1
NAMES←⊃∘⎕FREAD¨1,¨⍳¯1+2⊃⎕FSIZE 2

⍝ Create index file, and append NAMES
'INDEX' ⎕FCREATE 2
NAMES ⎕FAPPEND 2
```

Then if we want to find Pauline's data record:

```
NAMES←⎕FREAD 2,1        ⍝ Read index of names
CMP←NAMES⍳⊂'Pauline'    ⍝ Search for Pauline
DATA←⎕FREAD 1,CMP       ⍝ Read relevant record
```

There are many different ways to structure data files; you must design a structure that is the most efficient for your application.

Internal Structure

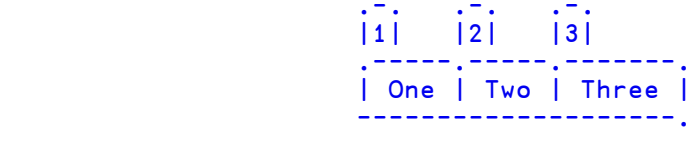
If you are going to make a lot of use of APL files in your systems, it is useful for you to have a rough idea of how Dyalog APL organises and manages the disk area used by such files.

The internal structure of external variables and component files is the same, and the examples given below apply to both.

Consider a component file with 3 components:

```
'TEMP' ⎕FCREATE 1
'One' 'Two' 'Three' ⎕FAPPEND¨1
```

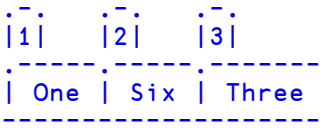
Dyalog APL will write these components onto contiguous areas of disk:



Replace the second component with something the same size:

```
      'Six' ⎕FREPLACE 1 2
```

This will fit into the area currently used by component 2.

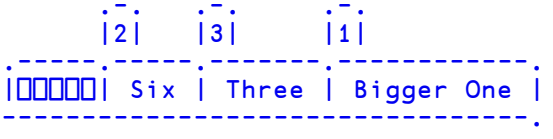


If your system uses fixed length records, then the size of your components never change, and the internal structure of the file remains static.

However, suppose we start replacing larger data objects:

```
      'Bigger One' ⎕FREPLACE 1 1
```

This will not fit into the area currently assigned to component 1, so it is appended to the end of the file. Dyalog APL maintains internal tables which contain the location of each component; hence, even though the components may not be physically stored in order, they can always be accessed in order.



The area that was occupied by component 1 now becomes free.

Now we'll replace component 3 with something bigger:

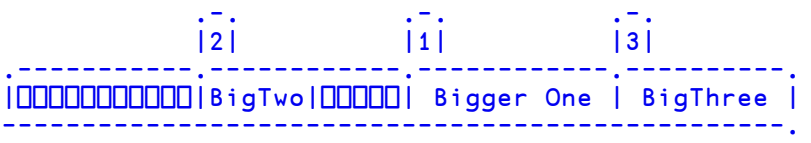
```
      'BigThree' ⎕FREPLACE 1 3
```

Component 3 is appended to the end of the file, and the area that was used before becomes free:

Dyalog APL keeps tables of the size and location of the free areas, as well as the actual location of your data. Now we'll replace component 2 with something bigger:

```
      'BigTwo' ⎕FREPLACE 1 2
```

Free areas are used whenever possible, and contiguous holes are amalgamated.



You can see that if you are continually updating your file with larger data objects, then the file structure can become fragmented. At any one time, the disk area occupied by your file will be greater than the area necessary to hold your data. However, free areas are constantly being reused, so that the amount of unused space in the file will seldom exceed 30%.

Whenever you issue a monadic `⎕FRESIZE` command on a component file, Dyalog APL COMPACTS the file; that is, it restructures it by reordering the components and by amalgamating the free areas at the end of the file. It then truncates the file and releases the disk space back to the operating system (note that some versions of UNIX do not allow the space to be released). For a large file with many components, this process may take a significant time.

Error Conditions

`FILE SYSTEM NOT AVAILABLE`

A `FILE SYSTEM NOT AVAILABLE` (Error code 28) error will be generated if the operating system returns an unexpected error when attempting to get a lock on a componenet file.

`FILE SYSTEM TIES USED UP`

A `FILE SYSTEM TIES USED UP` (Error code 30) error will be generated when a attempt is made to open more component files than is possible.

`FILE TIED`

A `FILE TIED` error is reported if you attempt to tie a file which another user has exclusively tied.

Limitations

File Tie Quota

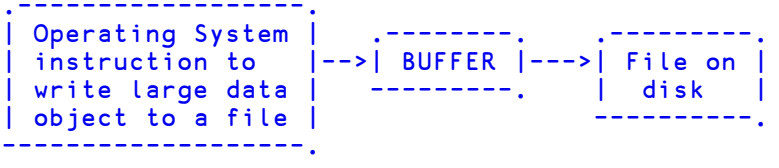
The File Tie Quota is the maximum number of files that a user may tie concurrently. Dyalog APL itself allows a maximum of 1024 under UNIX and 512 under Windows, although in either case your installation may impose a lower limit. When an attempt is made to exceed this limit, the report `FILE TIE QUOTA` (Error code 31) is given. This error will also be generated if an attempt is made to exceed the maximum number of open files that is imposed by the operating system.

File Name Quota

Dyalog APL records the names of each user's tied files in a buffer of 40960 bytes. When this buffer is full, the report `FILE NAME QUOTA USED UP` (Error code 32) will be given. This is only likely to occur if long pathnames are used to identify files.

The Effect of Buffering

Disk drives are fairly slow devices, so most operating systems take advantage of a facility called buffering. This is shown in simple terms below:



When you issue a write to a disk area, the data is not necessarily sent straight to the disk. Sometimes it is written to an internal buffer (or cache), which is usually held in (fast) main memory. When the buffer is full, the contents are passed to the disk. This means that at any one time, you could have data in the buffer, as well as on the disk. If you machine goes down whilst in this state, you could have a partially updated file on the disk. In these circumstances, the operating system generally recovers your file automatically.

If this facility is exploited, it offers very fast file updating. For systems that are I/O bound, this is a very important consideration. However, the disadvantage is that whilst it may appear that a write operation has completed successfully, part of the data may still be residing in the buffer, waiting to be flushed out to the disk. It is usually possible to force the buffer to empty; see your operating system manuals for details (UNIX automatically invokes the `sync()` command every few seconds to flush its internal buffers).

Dyalog APL exploits this facility, employing buffers internal to APL as well as making use of the system buffers. Of course, these techniques cannot be used when the file is shared with other users; obviously, the updates must be written immediately to the disk. However, if the file is exclusively tied, then several layers of buffers are employed to ensure that file access is as fast as possible.

You can ensure that the contents of all internal buffers are flushed to disk by issuing `⎕FUNTIE 0` at any time.

Integrity and Security

The structure of component files, the asynchronous nature of the buffering performed by APL, by the Operating System, and by the external device sub-system, introduces the potential danger that a component file might become damaged. To prevent this happening, the component file system includes optional journaling and check-sum features. These are optional because the additional security these features provide comes at the cost of reduced performance. You can choose the level of security that is appropriate for your application.

When journaling is enabled (see `⎕FPROPS`), files are updated using a journal which effectively prevents system or network failures from causing file damage.

Additional security is provided by the check sum facility which enables component files to be repaired using the system function `⎕FCHK`. See *Language Reference: File Check and Repair*.

Level 1 journaling protects a component file from damage caused by an abnormal termination of the APL process. This could occur if the process is deliberately or accidentally terminated by the user or by the Operating System, or by an error in Dyalog APL.

Level 2 journaling provides protection not just against the possibility that the APL process terminates abnormally, but that the Operating System itself fails. However, a damaged component file must be explicitly repaired using the system function `⎕FCHK` which will repair any damaged components by rolling them back to their previous states.

Level 3 provides the same level of protection as Level 2, but following the abnormal termination of either APL or the Operating System, the rollback of an incomplete update will be automatic and no explicit repair will be needed.

Higher levels of Journaling inevitably reduce the performance of component file updates.

For further information, see ⎕FPROPS and ⎕FCHK.

Operating System Commands

APL files are treated as normal data files by the operating system, and may be manipulated by any of the standard operating system commands.

Do not use operating system commands to copy, erase or move component files that are tied and in use by an APL session.

Chapter 3:

Error Trapping

Error Trapping Concepts

The purpose of this section is to show some of the ways in which the ideas of error trapping can be used to great effect to change the flow of control in a system.

Most APLs have error trapping facilities in one form or another, but this section discusses the facilities available to a Dyalog APL programmer.

First, we must have an idea of what is meant by error trapping. We are all used to entering some duff APL code, and seeing a (sometimes) rather obscure, esoteric error message echoed back:

```
      10÷0
DOMAIN ERROR
      10÷0
     ^
```

This message is ideal for the APL programmer, but not so for the end user. We need a way to bypass the default action of APL, so that we can take an action of our own, thereby offering the end user a more meaningful message.

Every error message reported by Dyalog APL has a corresponding error number (for a list of error codes and message, see ⎕TRAP, Language Reference). Many of these error numbers plus messages are common across all versions of APL. We can see that the code for `DOMAIN ERROR` is 11, whilst `LENGTH ERROR` has code 5.

Dyalog APL provides two distinct but related mechanisms for the trapping and control of errors. The first is based on the control structure `:Trap ... :EndTrap`, and the second, on the system variable ⎕TRAP. The control structure is easier to administer and so is recommended for normal use, while the system variable provides slightly finer control and may be necessary for specialist applications.

Last Error number and Diagnostic Message

Dyalog APL keeps a note of the last error that occurred, and provides this information through system functions: ⎕EN, ⎕EM and ⎕DM.

```
      10÷0
DOMAIN ERROR
      10÷0
     ^
```

Error Number for last occurring error:

```
      ⎕EN
11
```

Error Message associated with code 11:

```
      ⎕EM 11
DOMAIN ERROR
```

⎕DM (Diagnostic Message) is a 3 element nested vector containing error message, expression and caret:

```
      ⎕DM
 DOMAIN ERROR          10÷0          ^
```

Use function DISPLAY to show structure:

```
      DISPLAY ⎕DM
┌→──────────────────────────────────────────────────┐
│ ┌→──────────┐ ┌→──────┐ ┌→──────┐                 │
│ │DOMAIN ERROR│ │  10÷0 │ │     ^ │                 │
│ └───────────┘ └───────┘ └───────┘                 │
└∊──────────────────────────────────────────────────┘
```

Mix (↑) of this vector produces a matrix that displays the same as the error message produced by APL:

```
      ↑⎕DM
DOMAIN ERROR
      10÷0
     ^
```

Error Trapping Control Structure

You can embed a number of lines of code in a `:Trap` control structure within a defined function.

```
[1]    ...
[2]    :Trap 0
[3]        ...
[4]        ...
[5]    :EndTrap
[6]    ...
```

Now, whenever *any* error occurs in one of the enclosed lines, or in a function called from one of the lines, processing stops immediately and control is transferred to the line following the `:EndTrap`. The 0 argument to `:Trap`, in this case represents any error. To trap only specific errors, you could use a vector of error numbers:

```
[2]    :Trap 11 2 3
```

Notice that in this case, no extra lines are executed after an error. Control is passed to line `[6]` either when an error has occurred, *or* if all the lines have been executed without error. If you want to execute some code *only* after an error, you could re-code the example like this:

```
[1]    ...
[2]    :Trap 0
[3]        ...
[4]        ...
[5]    :Else
[6]        ...
[7]        ...
[8]    :EndTrap
[9]    ...
```

Now, if an error occurs in lines `[3-4]`, (or in a function called from those lines), control will be passed immediately to the line following the `:Else` statement. On the other hand, if all the lines between `:Trap` and `:Else` complete successfully, control will pass out of the control structure to (in this case) line `[9]`.

The final refinement is that specific error cases can be accommodated using `:Case [List]` constructs in the same manner as the `:Select` control structure.

```
[1]   :Trap 17+⍳21            ⍝ Component file errors.
[2]       tie←name ⎕ftie 0    ⍝ Try to tie file
[3]       'OK'
[4]   :Case 22
[5]       'Can''t find ',name
[6]   :CaseList 25+⍳13
[7]       'Resource Problem'
[8]   :Else
[9]       'Unexpected Problem'
[10]  :EndTrap
```

Note that `:Trap` can be used in conjunction with `⎕SIGNAL` described below.

Traps can be nested. In the following example, code in the inner trap structure attempts to tie a component file, and if unsuccessful, tries to create one. In either case, the tie number is then passed to function `ProcessFile`. If an error other than 22 (`FILE NAME ERROR`) occurs in the inner trap structure, or an error occurs in function `ProcessFile` (or any of its called function), control passes to line immediately to line `[9]`.

```
[1]   :Trap 0
[2]       :Trap 22
[3]           tie←name ⎕ftie 0
[4]       :Else
[5]           tie←name ⎕fcreate 0
[6]       :EndTrap
[7]       ProcessFile tie
[8]   :Else
[9]       'Unexpected Error'
[10]  :EndTrap
```

Trap System Variable: ⎕TRAP

The second way of trapping errors is to use the system variable: ⎕TRAP.

⎕TRAP, can be assigned a nested vector of **trap specifications**. Each trap specification is itself a nested vector, of length 3, with each element defined as:

list of error numbers	The error numbers we are interested in.
action code	Either 'E' (Execute) or 'C' (Cut Back). There are others, but they are seldom used.
action to be taken	APL expression, usually a branch statement or a call to an APL function.

So a single trap specification may be set up as:

```
⎕TRAP←5 'E' 'ACTION1'
```

and a multiple trap specification as:

```
⎕TRAP←(5 'E' 'ACTION1')((1 2 3) 'C' 'ACTION2')
```

The action code E tells APL that you want your action to be taken in the function in which the error occurred, whereas the code C indicates that you want your action to be taken in the function where the ⎕TRAP was *localised*. If necessary, APL must first travel back up the execution stack (cut-back) until it reaches that function.

Example Traps

These action codes are best illustrated by example.

Dividing by Zero

Let's try setting a ⎕TRAP on DOMAIN ERROR:

```
MSG←'''Please give a non-zero right arg'''
⎕TRAP←11 'E' MSG
```

When we enter:

```
10÷0
```

APL executes the expression, and notes that it causes an error number 11. Before issuing the standard error, it scans its ⎕TRAP table, to see if you were interested enough in that error to set a trap; you were, so APL executes the action specified by you:

```
      10÷0
Please give non-zero right arg
```

Let's reset our ⎕TRAP:

```
      ⎕TRAP←0⍴⎕TRAP          ⍝ No traps now set
```

and write a defined function to take the place of the primitive function ÷:

```
    ∇ R←A DIV B
[1]   R←A÷B
[2] ∇
```

Then run it:

```
      10 DIV 0
DOMAIN ERROR
      DIV[1] R←A÷B
             ^
```

Let's edit our function, and include a localised ⎕TRAP:

```
    ∇ R←A DIV B;⎕TRAP
[1] ⍝ Set the trap
[2]   ⎕TRAP←11 'E' '→ERR1'
[3] ⍝ Do the work; if it results in error 11,
[4] ⍝ execute the trap
[5]   R←A÷B
[6] ⍝ All OK if we got to here, so exit
[7]   →0
[8] ⍝ Will get here only if error 11 occurred
[9] ERR1:'Please give a non-zero right arg'
    ∇
```

Running the function with good and bad arguments has the desired effect:

```
      10 DIV 2
5

      10 DIV 0
Please give a non-zero right arg
```

⎕TRAP is a variable like any other, and since it is localised in DIV, it is only effective in DIV and any other functions that may be called by DIV. So....

```
     10÷0
DOMAIN ERROR
     10÷0
    ^
```

still gives an error, since there is no trap set in the global environment.

Other Errors

What happens to our function if we run it with other duff arguments:

```
      1 2 3 DIV 4 5
LENGTH ERROR
DIV [4] R←A÷B
        ^
```

Here is an error that we have taken no account of.

Change `DIV` to take this new error into account:

```
     ∇ R←A DIV B;⎕TRAP
[1]  ⍝ Set the trap
[2]    ⎕TRAP←(11 'E' '→ERR1')(5 'E' '→ERR2')
[3]  ⍝ Do the work; if it results in error 11,
[4]  ⍝ execute the trap
[5]    R←A ÷ B
[6]  ⍝ All OK if we got to here, so exit
[7]    →0
[8]  ⍝ Will get here only if error 11 occurred
[9]  ERR1:'Please give a non-zero right arg'◇→0
[10] ⍝ Will get here only if error 5 occurred
[11] ERR2:'Arguments must be same length'
     ∇

      )RESET

      1 2 3 DIV 4 5
Arguments must be the same length
```

But here's yet another problem that we didn't think of:

```
     (2 3⍴⍳6) DIV (2 3 4⍴⍳24)
RANK ERROR
DIV [4] R←A÷B
        ^
```

Global Traps

Often when we are writing a system, we can't think of everything that may go wrong ahead of time; so we need a way of catching "everything else that I may not have thought of". The error number used for "everything else" is zero:

```
      )RESET
```

Set a global trap:

```
      ⎕TRAP ← 0 'E' ' ''Invalid arguments'' '
```

And run the function:

```
      (2 3⍴⍳6) DIV (2 3 4⍴⍳24)
Invalid arguments
```

In this case, when APL executed line 4 of our function DIV, it encountered an error number 4 (RANK ERROR). It searched the local trap table, found nothing relating to error 4, so searched further up the stack to see if the error was mentioned anywhere else. It found an entry with an associated Execute code, so executed the appropriate action AT THE POINT THAT THE ERROR OCCURRED. Let's see what's in the stack:

```
      )SI
DIV[4]*

      ↑⎕DM
RANK ERROR
DIV[4] R←A÷B
         ^
```

So although our action has been taken, execution has stopped where it normally would after a RANK ERROR.

Dangers

We must be careful when we set global traps; let's call the non-existent function BUG whenever we get an unexpected error:

```
      )RESET
      ⎕TRAP ← 0 'E' 'BUG'
      (2 3⍴⍳6) DIV (2 3 4⍴⍳24)
```

Nothing happens, since APL traps a RANK ERROR on line 4 of DIV, so executes the trap statement, which causes a VALUE ERROR, which activates the trap action, which causes a VALUE ERROR, which .... etc. etc. If we had also chosen to trap on 1000 (ALL INTERRUPTS), then we'd be in trouble!

Let's define a function BUG:

```
    ∇ BUG
[1] ⍝ Called whenever there is an unexpected error
[2]   '*** UNEXPECTED ERROR OCCURRED IN: ',⊃1↓⎕SI
[3]   '*** PLEASE CALL YOUR SYSTEM ADMINISTRATOR'
[4]   '*** WORKSPACE SAVED AS BUG.',⊃1↓⎕SI
[5]   ⍝ Tidy up ... reset ⎕LX, untie files ... etc
[6]   ⎕SAVE 'BUG.',⊃1↓⎕SI
[7]   '*** LOGGING YOU OFF THE SYSTEM'
[8]   ⎕OFF
    ∇
```

Now, whenever we run our system and an unexpected error occurs, our BUG function will be called.

```
      10 DIV 0
Please give non-zero right arg

      (2 3⍴⍳6) DIV (2 3 4⍴⍳12)

*** UNEXPECTED ERROR OCCURRED IN: DIV
*** PLEASE CALL YOUR SYSTEM ADMINISTRATOR'
*** WORKSPACE SAVED AS BUG.DIV
*** LOGGING YOU OFF THE SYSTEM'
```

The system administrator can then load BUG.DIV, look at the SI stack, discover the problem, and fix it.

Looking out for Specific Problems

In many cases, you can of course achieve the same effect of a trap by using APL code to detect the problem before it happens. Consider the function TIE∆FILE, which checks to see if a file already exists before it tries to access it:

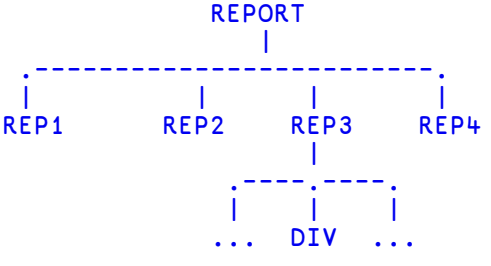
```
    ∇ R←TIE∆FILE FILE;FILES
[1]  ⍝ Tie file FILE with next available tie number
[2]  ⍝
[3]  ⍝ All files in my directory
[4]    FILES←⎕FLIB 'mydir'
[5]  ⍝ Remove trailing blanks
[6]    FILES←dbr¨↓FILES
[7]  ⍝ Required file in list?
[8]    →ERR×⍳~(⊂FILE)∈FILES
[9]  ⍝ Tie file with next number
[10]   FILE ⎕FTIE R←1+⌈/0,⎕FNUMS
[11] ⍝ ... and exit
[12]    →0
[13] ⍝ Error message
[14]  ERR:R←'File does not exist'
    ∇
```

This function executes the same code whether the file name is right or wrong, and it could take a while to get all the file names in your directory. It would be neater, and more efficient to take action ONLY when the file name is wrong:

```
      ∇ R←TIE∆FILE FILE;⎕TRAP
[1]   ⍝ Tie file FILE with next available tie number
[2]   ⍝
[3]   ⍝ Set trap
[4]     ⎕TRAP←22 'E' '→ERR'
[5]   ⍝ Tie file with next number
[6]     FILE ⎕FTIE R←1+⌈/0,⎕FNUMS
[7]   ⍝ ... and exit if OK
[8]     →0
[9]   ⍝ Error message
[10]    ERR:R←'File does not exist'
```

Cut-Back versus Execute

Let us consider the effect of using Cut-Back instead of Execute. Consider the system illustrated below, in which the function `REPORT` gives the user the option of 4 reports to be generated:




```
      ∇  REPORT;OPTIONS;OPTION;⎕TRAP
[1]    ⍝ Driver functions for report sub-system. If an
[2]    ⍝ unexpected error occurs, take action in the
[3]    ⍝ function where the error occurred
[4]    ⍝
[5]    ⍝ Set global trap
[6]     ⎕TRAP←0 'E' 'BUG'
[7]    ⍝ Available options
[8]     OPTIONS←'REP1' 'REP2' 'REP3' 'REP4'
[9]    ⍝ Ask user to choose
[10]   LOOP:→END×⍳0=⍴OPTION←MENU OPTIONS
[11]   ⍝ Execute relevant function
[12]    ⍎OPTION
[13]   ⍝ Repeat until EXIT
[14]    →LOOP
[15]   ⍝ Now end
[16]   END:
```

Suppose the user chooses REP3, and an unexpected error occurs in DIV. The good news is that the System Administrator gets a snapshot copy of the workspace that he can play about with:

```
      )LOAD BUG.DIV  ⍝ Load workspace
saved ......

      )SI            ⍝ Where did error occur?
DIV[4]*
REP3[6]
⍎
REPORT[7]


      ⍞⎕DM           ⍝ What happened?
RANK ERROR
DIV[4] R←A÷B
         ^
      ∇              ⍝ Edit function on top of stack
[0]R←A DIV B
.........
```

The bad news is, our user is locked out of the whole system, even though it may only be REP3 that has a problem. We can get around this by making use of the CUT-BACK action code.

```
    ∇ REPORT;OPTIONS;OPTION;⎕TRAP
[1] ⍝ Driver functions for report sub-system. If an
[2] ⍝ unexpected error occurs, cut the stack back
[3] ⍝ to this function, then take action
[4] ⍝
[5] ⍝ Set global trap
[6]  ⎕TRAP←0 'C' '→ERR'
[7] ⍝ Available options
[8]  OPTIONS←'REP1' 'REP2' 'REP3' 'REP4'
[9] ⍝ Ask user to choose
[10] LOOP:→END×⍳0=⍴OPTION←MENU OPTIONS
[11] ⍝ Execute relevant function
[12]  ⍎OPTION
[13] ⍝ Repeat until EXIT
[14]  →LOOP
[15] ⍝ Tell user ...
[16] ERR:MESSAGE'Unexpected error in',OPTION
[17] ⍝ ... what's happening
[18]  MESSAGE'Removing from list'
[19] ⍝ Remove option from list
[20]  OPTIONS←OPTIONS~⊂OPTION
[21] ⍝ And repeat
[22]  →LOOP
[23] ⍝ End
[24] END:
```

Suppose the user runs this version of REPORT and chooses REP3. When the unexpected error occurs in DIV, APL will check its trap specifications, and see that the relevant trap was set in REPORT with a cut-back code. APL therefore **cuts back the stack to the function in which the trap was localised, THEN takes the specified action**. Looking at the SI stack above, we can see that APL must jump out of DIV, then REP3, then ⍎, to return to line 7 of REPORT; THEN it takes the specified action.

Signalling Events

It would be useful to be able to employ the idea of cutting back the stack and taking an alternative route through the code, when a condition other than an APL error occurs. To achieve this, we must be able to trap on errors other than APL errors, and we must be able to define these errors to APL. We do the former by using error codes in the range 500 to 999, and the latter by using ⎕SIGNAL.

Consider our system; ideally, when an unexpected error occurs, we want to save a snapshot copy of our workspace (execute BUG in place), then immediately jump back to REPORT and reduce our options. We can achieve this by changing our functions a little, and using ⎕SIGNAL:

```
     ∇ REPORT;OPTIONS;OPTION;⎕TRAP
 [1] ⍝ Driver functions for report sub-system. If an
 [2] ⍝ unexpected error occurs, make a snapshot copy
 [3] ⍝ of the workspace, then cutback the stack to
 [4] ⍝ this function, reduce the option list & resume
 [5] ⍝ Set global trap
 [6]  ⎕TRAP←(500 'C' '→ERR')(0 'E' 'BUG')
 [7] ⍝ Available options
 [8]  OPTIONS←'REP1' 'REP2' 'REP3' 'REP4'
 [9] ⍝ Ask user to choose
[10] LOOP:→END×⍳0=⍴OPTION←MENU OPTIONS
[11] ⍝ Execute relevant function
[12]  ⍎OPTION
[13] ⍝ Repeat until EXIT
[14]  →LOOP
[15] ⍝ Tell user ...
[16] ERR:MESSAGE'Unexpected error in',OPTION
[17] ⍝ ... what's happening
[18]  MESSAGE'Removing from list'
[19] ⍝ Remove option from list
[20]  OPTIONS←OPTIONS~⊂OPTION
[21] ⍝ And repeat
[22]  →LOOP
[23] ⍝ End
[24] END:
```

```
    ∇ BUG
[1] ⍝ Called whenever there is an unexpected error
[2]   '*** UNEXPECTED ERROR OCCURRED IN: ',⍕1↓⎕SI
[3]   '*** PLEASE CALL YOUR SYSTEM ADMINISTRATOR'
[4]   '*** WORKSPACE SAVED AS BUG.',⍕1↓⎕SI
[5]   ⍝ Tidy up ... reset ⎕LX, untie files ... etc
[6]   ⎕SAVE 'BUG.',⍕1↓⎕SI
[7]   '*** RETURNING TO DRIVER FOR RESELECTION'
[8]   ⎕SIGNAL 500
    ∇
```

Now when the unexpected error occurs, the first trap specification catches it, and the `BUG` function is executed in place. Instead of logging the user off as before, an `error 500` is signalled to APL. APL checks its trap specifications, sees that 500 has been set in `REPORT` as a cut-back, so cuts back to `REPORT` before branching to `ERR`.

Flow Control

Error handling, which employs a combination of all the system functions and variables described, allows us to dynamically alter the flow of control through our system, as well as allow us to handle errors gracefully. It is a very powerful facility, which is simple to use, but is often neglected.

Handling Unexpected Application Errors in Windows

When running an APL application, it is possible that an unexpected error will occur.

It is advisable to set a trap at the top level of the application which traps all possible errors; in this way the programmer can cater for any errors that are not already explicitly trapped by, for example, writing information to a file, or saving the workspace. On UNIX in particular it may also be useful to call `⎕OFF` with a positive integer to the right of the `⎕OFF` - this is used as the exit code to APL.

It is also possible to generate an error which it is not possible to trap in APL code; examples include attempting to access the session in a runtime APL, or generating an error which causes APL to crash (for example, by the incorrect use of a shared library function).

By default in such cases, APL will pop up a message box, and cannot continue until the user selects the OK button.

It is possible to override this behaviour by setting the configuration parameter `DYALOG_NOPOPUPS` to 1. This will cause system popups to be suppressed; it does not suppress application popups generated by APL code.

With `DYALOG_NOPOPUPS=1` APL will terminate silently, except that an aplcore file will be generated. The location of the aplcore file can be controlled by the configuration parameter `APLCoreName`. It may be more useful to ask the operating system to handle the unexpected termination of the APL process, for example, by bringing up a debugger, or Dr Watson. This can be achieved by setting the configuration parameter `PassExceptionsToOpSys` to 1. In most cases it is useful to set `DYALOG_NOPOPUPS=1` too.

It is also possible to log such events to the Windows Event Log. Setting the configuration parameter `DYALOG_EVENTLOGGINGLEVEL` to a value greater than 0 will cause this to happen. If the configuration parameter `DYALOG_EVENTLOGNAME` is not set, then an event log called Dyalog will be created which can be viewed from the Windows Event Viewer. The first time that such an event occurs the following entries will be added to the Windows registry:

The key *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Dyalog APL* with values

Value Name	Value
Sources	Dyalog APL
MaxSize	150000000

The key *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Dyalog APL\Dyalog APL* with values

Value Name	Value
EventMessageFile	DYALOG\dyalog.exe
CategoryMessageFile	DYALOG\dyalog.exe
Category Count	5
TypesSupported	7

where DYALOG is the directory where Dyalog APL is installed.

If `DYALOG_EVENTLOGNAME` is set, it should contain the name of the log to which events will be logged. For example

```
DYALOG_EVENTLOGNAME="MyApp Event Log"
```

When set, no registry entries are added by Dyalog, but if the above registry entries have been manually created, the events will be logged to an event log which has the name "MyApp Event Log". If the registry entries described above have not been created, the events will instead be logged into the Application Log, and the Event Viewer will display text similar to the following when events are viewed:

The description for Event ID (1) in Source (MyApp Event Log) cannot be found. The local computer may not have the necessary registry information or message DLL files to display messages from a remote computer. You may be able to use the /AUXSOURCE= flag to retrieve this description; see Help and Support for details. The following information is part of the event: Syserror: 995 code: 2 Aplcore "aplcore1" has been created.

Index