DYALOG APL

The tool of thought for expert programming

User Guide

Version 12.1

Version 12.1.0 produced on 2009/11/06

First Edition October 2009

Contents

CHAPTER 1

Installation and Configuration

Files and Directories

File Naming Conventions

The following file naming conventions have been adopted for the various files distributed with and used by Dyalog APL/W.

Extension	Description
.DWS	Dyalog APL Workspace
.DSE	Dyalog APL Session
.DCF	Dyalog APL Component File
.DXV	Dyalog APL External Variable
.DIN	Dyalog APL Input Table
.DOT	Dyalog APL Output Table
.DFT	Dyalog APL Format File
.DXF	Dyalog APL Transfer File
.DLF	Dyalog APL Session Log File
.dyalog	Dyalog APL SALT file
.dyapp	Dyalog APL SALT application file

Classic and Unicode Editions

The defining feature of Version 12.0 was support for *Unicode* character data. This necessarily entailed a change in the internal format of character arrays stored in the workspace and on component files and in external variables. This in turn means that the adoption of Unicode *may* require **code changes and data conversions in applications**.

For this reason, Version 12.0 and Version 12.1 are available in two separate editions; *Unicode* and *Classic*.

- The *Unicode* edition is intended for users who need to develop Unicode applications now, and are prepared to make the necessary (usually small) changes to existing applications in order to support new Unicode character types.
- The *Classic* edition is intended for customers who want to take advantage of other product enhancements, but do not wish to use Unicode at this time.

The two different editions are maintained from the same source code, and every effort will be made to ensure that they are identical except for the handling of character arrays, and the transfer of data into and out of the workspace.

APL Fonts

Unicode Edition

The default font for the Unicode Edition is APL385 Unicode[1] which is a TrueType font and is installed as part of Dyalog APL. APL385 Unicode is the font used to print APL characters in this manual. In principle, you may use any other Unicode font that includes the APL symbols, such as Arial Unicode MS (available from Microsoft).

Classic Edition

In the Classic Edition, there are two types of APL font provided; bitmap (screen) and TrueType. There are also two different layouts, which referred to as *Std* and *Alt*.

The bitmap fonts are designed for the screen alone and are named *Dyalog Std* and *Dyalog Alt*. The TrueType fonts have a traditional 2741-style italic appearance and are named *Dyalog Std TT* and *Dyalog Alt TT*.[1]

The *Std* layout, which was the standard layout for Versions of Dyalog APL up to Version 10.1 contains the APL underscored alphabet A–Z. **The underscored alphabet is a deprecated feature and is only supported in this Version of Dyalog APL for backwards compatibility.**

The *Alt* layout, which replaces the *Std* layout as the standard layout for Version 12.0 Classic Edition onwards, does not have the underscored alphabet, but contains additional National Language characters in their place. Note that the extra National Language symbols share the same ⎕AV positions with the underscored alphabet. If, for example, you switch from the *Std* font layout to the alternative one, you will see the symbol Á (A-acute) instead of the symbol –.

You may use either a bitmap font or a TrueType font in your APL session (see *Chapter 2* for details). You MUST use a TrueType font for printing APL functions.

[1] The Dyalog Std TT, Dyalog Alt TT, and APL385 Unicode fonts are the copyright of Adrian Smith.

Integrated APL Keyboard (Unicode Edition Only)

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.

Version 12.1 is supplied with two sets of such keyboards (one using Ctrl and one using AltGr) for a range of different languages. These keyboards were created using the Microsoft Keyboard Layout Creator (MSKLC) and you may use the same tool to customise one of the supplied keyboards or to create a new one.

During the installation of Dyalog Version 12.1 Unicode Edition, setup installs a Ctrl and (if available) an AltGr APL keyboard layout onto your system. These keyboard layouts are installed as additional services for your default Input Language. For further details, see Unicode Edition Keyboard on page 66.

On-Screen Keyboard

Included with Dyalog APL Version 12.1 Unicode Edition is the Comfort On-Screen Keyboard 2.1 which has been specially extended for use with Dyalog APL and is distributed under a licence agreement with Comfort Software. The On-Screen keyboard is a really useful tool that works with any Windows application and replaces Kibitzer in the Unicode Edition. Kibitzer remains part of the Classic Edition.

Interoperability and Compatibility

Introduction

Workspaces and component files are stored on disk in a binary format (illegible to text editors). This format differs between machine architectures and among versions of Dyalog. For example a file component written by a PC will almost certainly have an internal format that is different from one written by a UNIX machine. Similarly, a workspace saved from Dyalog Version 12.1 will differ internally from one saved by a previous version of Dyalog APL.

It is convenient for versions of Dyalog APL running on different platforms to be able to *interoperate* by sharing workspaces and component files. From Version 11.0, component files and workspaces can generally be shared between Dyalog interpreters running on different platforms. However, this is not always possible. For example, component files created by Version 10.1 can often not be shared across platforms, even when used by later versions (the system function `⎕FCOPY` can be used to make a logically identical copy of an old file, which is fully inter-operable).

The following sections describe other limitations in inter-operability:

Code

Code which is saved in workspaces, or embedded within `⎕OR`s stored in component files, can generally only be read by the version which saved them and later versions of the interpreter. In the case of workspaces, a load (or copy) from an older version would fail with the message:

```
this WS requires a later version of the interpreter.
```

In the case of `⎕OR`, unpredictable behaviour may result if an older version reads a `⎕OR` saved by a later version of the system. Thus, `⎕OR` is ***not recommended*** as a mechanism for sharing code or objects between different versions of APL.

“Ordinary” Arrays

With the exception of the Unicode restrictions described in the following paragraphs, Dyalog APL provides complete inter-operability for arrays which only contain (nested) character and numeric data. Such arrays can be stored in component files - or transmitted using `TCPSocket` objects and Conga connections, and shared between all versions and across all platforms.

As mentioned in the introduction, full cross-platform interoperability of component files is only available for large component files (see the following section), and for small component files created by Version 11.0 or later.

32 vs 64-bit Component Files

Large (64-bit-addressing) component files are inaccessible to versions of the interpreter that pre-dated their introduction (versions earlier than 10.1).

The second item in the right argument of ⎕FCREATE determines the addressing type of the file.

```
'small'⎕fcreate 1 32    ⍝ create small file.
'large'⎕fcreate 1 64    ⍝ create large file.
```

If the second item is missing, the file type defaults to 64-bit-addressing. In versions prior to 12.0, the default was 32-bit addressing.

Note that *small* (32-bit-addressing) cannot contain Unicode data. Unicode editions of Dyalog APL can only write character data which would be readable by a Classic edition (consisting of elements of ⎕AV).

External Variables

External variables are implemented as small (32-bit -adressing) component files, and subject to the same restrictions as these files. External variables are unlikely to be developed further; Dyalog recommends that applications which use them should switch to using mapped files or traditional component files. Please contact Dyalog if you need further advice on this topic.

32 vs. 64-bit Interpreters

From Dyalog APL Version 11.0 onwards, there are two separate versions of programs for 32-bit and 64-bit machine architectures (in general, the 32-bit versions will also run on 64-bit machines running 64-bit operating systems). There is complete inter-operability between 32- and 64-bit interpreters.

Unicode vs. Classic Editions

From Version 12.0 onwards, a Unicode edition is available, which is able to work with the entire Unicode character set. Classic editions (a term which includes versions prior to 12.0) are limited to the 256 characters defined in the atomic vector, ⎕AV). Large (64-bit-addressing) component files have a Unicode property; when this is enabled, Unicode data may be stored in the file. The Unicode property is always off for small (32-bit addressing) files, which may not contain Unicode data. When a Unicode edition writes to a component file which may not contain Unicode data, character data is mapped to ⎕AV, and can therefore be read without problems by Classic editions.

A TRANSLATION ERROR will occur if a Unicode edition writes to a non-Unicode component file, if the data being written contains characters which are not in ⎕AV (see ⎕AVU for more details). Likewise, a Classic edition (Version 12.0 or later) will issue a TRANSLATION ERROR if it reads Unicode data from a component file, and is unable to map it to ⎕AV. Version 10.1 cannot read components containing Unicode data.

A TRANSLATION ERROR will also issued when a Classic edition)LOADs or)COPYs a workspace containing Unicode data which cannot be mapped to ⎕AV.

TCPSocket objects have an APL property which corresponds to the Unicode property of a file, if this is set to Classic (the default) the data in the socket will be restricted to ⎕AV, if Unicode it will contain Unicode character data. As a result, TRANSLATION ERRORs can occur on transmission or reception in the same way as when updating or reading a file component.

File Journaling

Version 12.0 introduces File Journaling (level 1), and 12.1 adds levels 2 and 3. Versions earlier than 12.0 cannot tie files which have any form of journaling enabled. Version 12.0 cannot tie files with journaling levels other than 1. Files can be shared with earlier versions by using ⎕FPROPS to switch journaling off.

Auxiliary Processors

A Dyalog APL process is restricted to starting an AP of exactly the same architecture. In other words, the AP must share the same word-width and byte-ordering as its interpreter process.

Session Files

Session (.dse) files may only be used on the platform on which they were created and saved.

The APL Command Line

The command line for Dyalog APL is as follows:

dyalog [options] [debug] [file] [param] [param] [param]...

where:

[options]

-x	No ⎕LX execution on workspace loads.
-a	Start in USER mode.
-b	Suppress the banner in the Session..
-Fxx	Default to creating xx-bit files (where xx is 32 or 64).
-s	Disable the Session.
-q	Don't quit APL on error (used when piping input into APL).
-c	Signifies a command-line comment. All characters to the right are ignored.

[debug]

-Dc	Check workspace integrity after every callback function.
-Dw	Check workspace integrity on return to session input.
-DW	Check workspace integrity after every line of APL (application will run slowly as a result)
-DK	Log session keystrokes in (binary) file **APLLOG**.

[file] The name of a Dyalog APL workspace to be loaded. Unless specified, the file extension .DWS is assumed.

[param] A parameter name followed by an equals sign (=) and a value. Note that the parameter name may be one of the standard APL parameters described below, or a name and value of your own choosing (see Object Reference, GetEnvironment method).

Examples:

```
c:\program files\…\dyalog.exe myapp maxws=64000
c:\program files\…\dyalog.exe session_file=special.dse
c:\program files\…\dyalog.exe myapp aplt=mytrans.dot myparam=42
```

APL Exit Codes

When APL or a bound .EXE terminates, it returns an exit code to the calling environment. If APL is started from a desktop icon, the return code is ignored. However, if APL is started from a script (UNIX) or a command processor, the exit code is available and may be used to determine whether or not to continue with other processing tasks. The return codes are:

0 successful `⎕OFF,)OFF, )CONTINUE`, graphical exit from GUI

1 APL never got started. This will occur if there was a failure to read a translate file, there is insufficient memory, or a critical parameter is incorrectly specified or missing.

2 APL was terminated by SIGHUP or SIGTERM (UNIX) or in response to a QUIT WINDOWS request. APL has done a clean exit.

3 APL issued a syserror

Note that if APL terminates with a core dump, SIGSEGV etc (UNIX), the return code is determined by the Operating System.

It is also possible for an application to return a custom exit code as the optional argument to `⎕OFF`.

Configuration Parameters

Introduction

Dyalog APL/W is customised using a set of configuration parameters which are defined in a registry folder.

In addition, parameters may be specified as environment variables or may be specified on the APL command line.

Furthermore, you are not limited to the set of parameters employed by APL itself as you may add parameters of your own choosing.

Setting Parameter Values

You can change the parameters in 4 ways:

1. Using the Configuration dialog box that is obtained by selecting *Configure* from the *Options* menu on the Dyalog APL/W session. See *Chapter 2* for details.
2. By directly editing the Windows Registry using REGEDIT.EXE or REGEDIT32.EXE.
3. By defining the parameters as DOS environment variables.
4. By defining the parameters on the APL command line.

This scheme provides a great deal of flexibility, and a system whereby you can override one setting with another. For example, you can define your normal workspace size (***maxws***) in your .INI file or Registry, but override it with a new value specified on the APL command line. The way this is done is described in the following section.

How APL Obtains Parameter Values

When Dyalog APL/W requires the value of a parameter, it uses the following rules.

1. If the parameter is defined on the APL command line, this value is used.
2. Otherwise, APL looks for an environment variable of the same name and uses this value.
3. Otherwise, if the parameter in question is **inifile**, the default value `of Software\Dyalog\Dyalog APL/W 12.1 Unicode` (Unicode Edition) or `of Software\Dyalog\Dyalog APL/W 12.1 Unicode` (Classic Edition) is assumed.
4. Otherwise, if the parameter in question is **dyalog**, the name of the directory from which the Dyalog APL program was loaded is assumed.
5. The value of any other parameter is obtained from the registry folder defined by the value of **inifile**.

Note that the value of a parameter obtained by the GetEnvironment method (see Object Reference) uses exactly the same set of rules.

The following section details those parameters that are implemented by Registry Values in the top-level folder identified by **inifile**. Values that are implemented in sub-folders are *mainly* internal and are not described in detail here. However, any Value that is maintained via a configuration dialog box will be named and described in the documentation for that dialog box in Chapter 2.

AplCoreName

This parameter specifies the directory and name of the file in which the aplcore should be saved. The optional wild-card character (*) is replaced by a unique string when the file is written. For example:

```
APLCORENAME=C:\mycores\aplcore*.dat
```

APLFormatBias

This parameter specifies which algorithm is used in formatting floating-point numbers using the

If `APLFormatBias` is 0 (the default if it is not specified), floating point numbers are formatted as accurately as possible using high-precision mathematical routines.

If you set `APLFormatBias` to 1, there is a significant performance benefit but numbers whose decimal part (the part after the decimal point) ends in 5 may be rounded up rather than down.

For further information, see Language Reference.

aplfscb

This parameter specifies the location of the File System Control Block (FSCB) and is applicable only if **File_Control** is set to 1. The FSCB is a file which is used to control and synchronise access to shared component files and external variables. See Chapter 3 for further details.

aplk Classic Edition Only

This parameter specifies the name of your Input Translate Table, which defines your keyboard layout. The keyboard combo in the *Configure* dialog box displays all the files with the .DIN extension in the APLKEYS sub-directory. You may choose any one of the supplied tables, and you may add your own to the directory. Note that the FILE.DIN table is intended for input from **file**, and should not normally be chosen as a keyboard table. Classic Edition only

aplkeys Classic Edition Only

This parameter specifies a search path for the Input Translate Table and is useful for configuring a run-time application. It consists of a string of directories separated by the semicolon (;) character. Its default value is the APLKEYS sub-directory of the directory in which Dyalog APL/W is installed (defined by **dyalog**)

aplnid

This parameter specifies the *user number* that is used by the component file system to control file sharing and security. If you wish to share component files and/or external variables in a network, *and you choose to use other than the default file control mechanism* (`File_Control=2`, see below), it is essential that each user has a unique **aplnid** parameter. It may be any integer in the range 0 to 65535. Note that an **aplnid** value of 0 causes the user to bypass APL's access control matrix mechanism.

aplt

This parameter specifies the name of the Output Translate Table. The default is WIN.DOT and there is rarely a need to alter it.

apltrans

This parameter specifies a search path for the Output Translate Table and is useful for configuring a run-time application. It consists of a string of directories separated by the semicolon (;) character. Its default value is the sub-directory APLTRANS in the directory in which Dyalog APL/W is installed.

auto_pw

This parameter specifies whether or not the value of ⎕PW is derived automatically from the current width of the Session Window. If auto_pw is 1, the value of ⎕PW changes whenever the Session Window is resized and reflects the number of characters that can be displayed on a single line. If auto_pw is 0 (the default) ⎕PW is independent of the Session Window size.

AutoFormat

This parameter specifies whether or not you want automatic formatting of Control Structures in functions. The default value is 0. If this parameter is set to 1, formatting is done automatically for you when a function is opened for editing or converted to text by ⎕CR, ⎕NR and ⎕VR. Automatic formatting first discards all leading spaces in the function body. It then prefixes all lines with a single space except those beginning with a label or a comment symbol (this has the effect of making labels and comments stand out). The third step is to indent Control Structures. The size of the indent depends upon the **TabStops** parameter.

AutoIndent

This parameter specifies whether or not you want semi-automatic indenting during editing. The default value is 1. This means that when you enter a new line in a function, it is automatically indented by the same amount as the previous line. This option simplifies the entry of indented Control Structures.

ClassicMode

This parameter specifies whether or not the Session operates in *Dyalog Classic mode*. The default is 0. If this parameter is set to 1, the Editor and Tracer behave in a manner that is consistent with previous versions of Dyalog APL.

confirm_abort

This parameter specifies whether or not you will be prompted for confirmation when you attempt to abort an edit session after making changes to the object being edited. Its value is either 1 (confirmation is required) or 0. The default is 0.

confirm_close

This parameter specifies whether or not you will be prompted for confirmation when you close an edit window after making changes to the object being edited. Its value is either 1 (confirmation is required) or 0. The default is 0.

confirm_fix

This parameter specifies whether or not you will be prompted for confirmation when you attempt to fix an object in the workspace after making changes in the editor. Its value is either 1 (confirmation is required) or 0. The default is 0.

confirm_session_delete

This parameter specifies whether or not you will be prompted for confirmation when you attempt to delete lines from the Session Log. Its value is either 1 (confirmation is required) or 0. The default is 1.

CreateAplCoreOnSyserror

This parameter specifies whether or not an aplcore file is generated when APL exits with a system error.

default_div

This parameter specifies the value of `⎕DIV` in a clear workspace. Its default value is 0.

DefaultHelpCollection

Dyalog attempts to use the Microsoft Document Explorer and online help, for example from Visual Studio (if installed), to display help for external objects, such as .Net Types. In most cases the default settings of "ms-help://ms.mscc.v80" will be sufficient. On some configurations it may be necessary to change this.

default_io

This parameter specifies the value of `⎕IO` in a clear workspace. Its default value is 1.

default_ml

This parameter specifies the value of ⎕ML in a clear workspace. Its default value is 0.

default_pp

This parameter specifies the value of ⎕PP in a clear workspace. Its default value is 10.

default_pw

This parameter specifies the value of ⎕PW in a clear workspace. Its default value is 76. Note that ⎕PW is a property of the Session and the value of default_pw is overridden when a Session file is loaded.

default_rl

This parameter specifies the value of ⎕RL in a clear workspace. Its default value is 16807.

default_rtl

This parameter specifies the value of ⎕RTL in a clear workspace. Its default value is 0.

default_wx

This parameter specifies the value of ⎕WX in a clear workspace. This in turn determines whether or not the names of properties, methods and events of GUI objects are exposed. If set (⎕WX is 1), you may query/set properties and invoke methods directly as if they were variables and functions respectively. As a consequence, these names may not be used for global variables in GUI objects.

DockableEditWindows

This parameter specifies whether or not individual edit windows can be undocked from (and docked back into) the (MDI) Editor window. Its default value is 0. This parameter does not apply if **ClassicMode** is set to 1.

DoubleClickEdit

This parameter specifies whether or not double-clicking over a name invokes the editor. Its default is 1. If DoubleClickEdit is set to 0, double-clicking selects a word and triple-clicking selects the entire line.

dyalog

This parameter specifies the name of the directory in which Dyalog APL/W is installed.

DyalogEmailAddress

This parameter specifies the contact email address for Dyalog Limited.

DyalogHelpDir

This parameter specifies the full pathname of the directory that contains the Dyalog APL help file (dyalog.chm).

DyalogInstallDir

This parameter specifies the full pathname of the directory in which Dyalog APL is installed.

DyalogWebSite

This parameter specifies the URL for the Dyalog web site.

edit_cols, edit_rows

These parameters specify the initial size of an edit window in character units.

edit_first_x, edit_first_y

These parameters specify the initial position on the screen of the *first* edit window in character units. Subsequent edit windows will be staggered. These parameters only apply if **ClassicMode** is 1.

edit_offset_x, edit_offset_y

These parameters specify the amount by which an edit window is staggered from the previous one.

ErrorOnExternalException

This is a Boolean parameter that specifies the behaviour when a System Exception occurs in an external DLL. If this parameter is set to 1, and an exception occurs in a call on an external DLL. APL generates an `EXTERNAL DLL EXCEPTION` error (91), instead of terminating with a System Error. This error may be trapped.

EditorState

This is an internal parameter that remembers the state of the last edit window (normal or maximised). This is used to create the next edit window in the appropriate state.

File_Control

This parameter specifies the Component File System Control mechanism. It is an integer with the value 0, 1 or 2:

0. Access to Component Files is controlled in **memory**. This is the fastest control mechanism but is applicable *only* to a stand-alone situation. If you are sharing component files with other users or between two APL sessions, you must not use this option.
1. Access to Component Files is controlled by a **File System Control Block**. This is a separate file shared by all APL users that records the current state of all file ties and locks. This mechanism is provided primarily for compatibility with previous versions of Dyalog APL/W.
2. Access to Component Files is controlled by standard **Operating System** facilities. This is the preferred control mechanism for shared component files and is the default.

greet_bitmap

This parameter specifies the filename of a bitmap to be displayed during initialisation of the Dyalog APL application. It is used typically to display a product logo from a runtime application. The bitmap will remain until either an error occurs, or it is removed using the GreetBitmap method of the Root object.

```
greet_bitmap=c:\myapp\logo.bmp
```

history_size

This parameter specifies the size of the buffer used to store previously entered (input) lines in the Session.

IndependentTrace

This parameter specifies whether or not the Trace windows are children of the Session window. The default is 0 (Trace windows **are** children of the Session). This applies only if **ClassicMode** is 1.

inifile

This parameter specifies the name of the Windows Registry folder that contains the configuration parameters described in this section. For example,

```
INIFILE=Software\Dyalog\mysettings
```

If the parameter is not defined, **inifile** defaults to the current directory.

InitialKeyboardLayout Unicode Edition Only

This parameter specifies the name of the keyboard to be selected on startup. When you start an APL session, this layout will automatically be selected as the current keyboard layout if the value of InitialKeyboardLayoutInUse is 1.

InitialKeyboardLayoutInUse Unicode Edition Only

This Boolean parameter specifies whether or not the keyboard specified by InitialKeyboardLayout is selected as the current keyboard layout when you start an APL session.

InitialKeyboardLayoutShowAll Unicode Edition Only

This Boolean parameter specifies whether or not all installed keyboards are listed in the choice of keyboards in the Configuration dialog box (Unicode Input tab) .

input_size

This parameter specifies the size of the buffer used to store marked lines (lines awaiting execution) in the Session.

lines_on_functions

This parameter specifies whether or not line numbers are displayed in edit and trace windows. It is either 0 (the default) or 1.

Note that this parameter determines your overall preference for line numbering, and this setting persists between APL sessions. You can however still toggle line numbering on and off dynamically as required by clicking *Line Numbers* in the *Options* menu on the Session Window. These temporary settings are not saved between APL sessions.

localdyalogdir

This parameter specifies the name of the directory in which Dyalog APL/W is installed on the client, in a client/server installation

.

log_file

This parameter specifies the full pathname of the Session log file.

log_file_inuse

This parameter specifies whether or not the Session log is saved in a session log file.

log_size

This parameter specifies the size of the Session log buffer in Kb.

mapchars — Classic Edition Only

In previous versions of Dyalog APL, certain pairs of characters in ⎕AV were mapped to a single font glyph through the output translate table. For example, the ASCII pipe ¦ and the APL style | were both mapped to the APL style |. From Version 7.0 onwards, it has been a requirement that the mapping between ⎕AV and the font is strictly one-to-one (this is a consequence of the new native file system). Originally, the mapping of the ASCII pipe and the APL style, the APL and ASCII quotes, and the ASCII ^ and the APL ^ were hard-coded. The mapping is defined by the **mapchars** parameter.

mapchars is a string containing pairs of hexadecimal values which refer to 0-origin indices in ⎕AV. The first character in each pair is mapped to the second on output. The default value of **mapchars** is `DB0DEBA7EEC00BE0` which defines the following mappings.

From			To		
Hex	**Decimal**	**Symbol**	**Hex**	**Decimal**	**Symbol**
DB	219	‘	0D	13	'
EB	235	^	A7	167	^
EE	238	⍬	C0	192	¦
0B	11	.	E0	224	.

To clear all mappings, set MAPCHARS=0000

maxws

This parameter determines your workspace size in kilobytes and is the amount of Windows memory allocated to the workspace at APL start-up. `MAXWS` is specified as an integer number followed optionally by the letter k, m or g (in upper or lower case) to indicate kilobytes, megabytes or gigabytes. If omitted, the default is kilobytes.

The default value is 16384 (16 Mb). If you want a larger (or smaller) workspace you must change this value. For example, to get a 64 MB workspace:

```
MAXWS=64m
```

Dyalog APL places no implicit restriction on workspace size, and the virtual memory capability of MS-Windows allows you to access more memory than you have physically installed. However if you use a workspace that **greatly** exceeds your physical memory you will encounter excessive *paging* and your APL programs will run slowly.

Note that the memory used for the workspace must be *contiguous* memory, and, under Windows, this is typically limited to a maximum of 1.6GB. This is a Windows restriction, and not one that is imposed by Dyalog APL.

PassExceptionsToOpSys

This is a Boolean parameter that specifies the default state of the *Pass Exception* check box in the System Error dialog box.

pfkey_size

This parameter specifies the size of the buffer that is used to store programmable function key definitions (⎕PFKEY).

ProgramFolder

This parameter specifies the name of the folder in which the Dyalog APL program icons are installed.

PropertyExposeRoot

This parameter specifies whether or the names of properties, methods and events of the Root object are exposed. If set, you may query/set the properties of Root and invoke the Root methods directly as if they were variables and functions respectively. As a consequence, these names may not be used for global variables in your workspace.

PropertyExposeSE

This parameter specifies whether or the names of properties, methods and events of the Session object are exposed. If set, you may query/set the properties of ⎕SE and invoke ⎕SE methods directly as if they were variables and functions respectively. As a consequence, these names may not be used for global variables in the ⎕SE namespace.

qcmd_timeout

This parameter specifies the length of time in milliseconds that APL will wait for the execution of a DOS command to start. Its default value is 5000 milliseconds.

RunAsService

When RunAsService is set to 1 (the default is 0) Dyalog APL will not prompt for confirmation when the user logs off, and the interpreter will continue to run across the logoff logon process

SaveContinueOnExit

Specifies whether or not your current workspace is saved as CONTINUE.DWS before APL terminates.

SaveLogOnExit

Specifies whether or not your Session log is saved before APL terminates.

SaveSessionOnExit

Specifies whether or not your current Session is saved in your Session file before APL terminates.

Serial

Specifies your Dyalog APL/W Serial Number.

session_file

This parameter specifies the name of the file from which the APL session (⎕SE) is to be loaded when APL starts. If not specified, a .DSE extension is assumed. This session file contains the ⎕SE object that was last saved in it. This object defines the appearance and behaviour of the Session menu bar, tool bar(s) and status bar, together with any functions and variables stored in the ⎕SE namespace.

ShowStatusOnError

Specifies whether or not the Status window is automatically displayed (if required) when APL attempts to write output to it.

SingleTrace

Specifies whether there is a single Trace window, or one Trace window per function. This applies only if **ClassicMode** is 1.

StatusOnEdit

Specifies whether or not a status bar is displayed at the bottom of an Edit window.

sm_cols, sm_rows

These parameters specify the size of the window used to display SM when it is used *stand-alone.* They are **not** used if the window is specified using the SM object.

TabStops

This parameter specifies the number of spaces inserted by pressing the Tab key in the editor. Its default value is 4.

trace_cols, trace_rows

These parameters specify the initial size of a trace window in character units.

trace_first_x, trace_first_y

These parameters specify the initial position on the screen of the *first* trace window in character units. Subsequent trace windows will be staggered. This applies only if **ClassicMode** is 1.

trace_offset_x, trace_offset_y

These parameters specify the amount by which a trace window is staggered from the previous one. These apply only if **ClassicMode** is 1 and **SingleTrace** is 0.

Trace_level_warn

This parameter specifies the maximum number of Trace windows that will be displayed when an error occurs and **Trace_on_error** is set to 1. If there are a large number of functions in the state indicator, the display of their Trace windows may take several seconds. This parameter allows you to restrict the potential delay to a reasonable value and its default is 16. If the number of Trace windows would exceed this number, the system instead displays a warning message box. This parameter is ignored if you invoke the Tracer explicitly. This parameter applies only if **ClassicMode** is 1 and **SingleTrace** is 0.

Trace_on_error

This parameter is either 0 (the default) or 1. If set to 1, **Trace_on_error** specifies that the Tracer is automatically deployed when execution of a defined function halts with an error. A stack of Trace windows is immediately displayed, with the top Trace window receiving the input focus.

TraceStopMonitor

This parameter specifies which of the ⎕TRACE (1), ⎕STOP (2) and ⎕MONITOR (4) columns are displayed in Trace and Edit windows. Its value is the sum of the corresponding values.

UnicodeToClipboard

This parameter specifies whether or not text that is transferred to and from the Windows clipboard is treated as Unicode text. If UnicodeToClipboard is 0 (the default), the symbols in ⎕AV are mapped to ASCII text (0-255). In particular, the APL symbols are mapped to ASCII symbols according to their positions in the Dyalog APL font. If UnicodeToClipboard is 1, the symbols in ⎕AV are mapped to Unicode text and the APL symbols are mapped to their genuine Unicode equivalent values.

wspath

This parameter defines the workspace path. This is a list of directories that are searched in the order specified when you)LOAD or)COPY a workspace and when you start an Auxiliary Processor. The directory paths are specified using Operating System specific conventions and separated by ";" (Windows) or ":" (UNIX).

The following Windows example causes)COPY,)LOAD and)LIB to look first in the current directory, then in D:\MYWS, and then in the (normal) *installation workspace* directory.

```
wspath=.;D:\MYWS;
       C:\Program Files\Dyalog\Dyalog APL 12.1 Unicode\ws
```

XPLookAndFeel

This parameter is not used directly. See page 135.

XPLookAndFeelDocker

This parameter specifies whether or not the title bars in docked windows honour Native (XP) Look and Feel, if this is enabled at the Windows level. If unspecified, the default is 0.

yy_window

This parameter defines how Dyalog APL is to interpret a 2-digit year number. Dyalog APL is millennium-compliant. However it is possible that the applications you have written are not.

This is because Dyalog allows a choice of input date formats for ⎕SM and GUI edit fields. If you have chosen a 2-digit year format such as MM/DD/YY, then an input of 02/01/00 will by default be interpreted as 1st February 1900 - not 1st February 2000.

If your application uses a 4-digit year format such as YYYY-MM-DD, the problem will not arise.

You can use the **yy_window** parameter to cause your application to interpret 2-digit dates in as required without changing any APL code.

Sliding versus Fixed Window

Two schemes are in common use within the industry: Sliding or Fixed date windows.

Use a Fixed window if there is a *specific year*, for example 1970, before which, dates are meaningless to your application. Note that with a fixed window, this date (say 1970) will still be the limit if your application is running in a hundred years time.

Use a Sliding window if there is a *time period*, for example 30 years, before which dates are considered too old for your application. With a sliding window, you will always be able to enter dates up to (say) 30 years old, but after a while, specific years in the past (for example 1970) will become inaccessible.

Setting a Fixed Window

To make a fixed window, set environment variable **yy_window** to the 4-DIGIT year which is the earliest acceptable date. For example:

```
YY_WINDOW=1970
```

This will cause the interpreter to convert any 2-digit input date into a year in the range 1970, 1971 ... 2069

Setting a Sliding Window

To make a sliding window, set environment variable **yy_window** to the 1- or 2-DIGIT year which determines the oldest acceptable date. This will typically be negative.

```
YY_WINDOW=-30
```

Conversion of dates now depends on the current year:

If the current year is 1999, the earliest accepted date is 1999-30 = 1969.

This will cause the interpreter to convert any 2-digit input date into a year in the range 1969, 1970 ... 2068.

However if your application is still running in the year 2010, the earliest accepted date then will be 2010-30 = 1980. So in the year 2010, a 2-digit year will be interpreted in the range 1980, 1981 ... 2079.

Advanced Settings

You can further restrict date windows by setting an upper as well as lower year limit.

```
YY_WINDOW=1970,1999
```

This causes 2-digit years to be converted only into the range 1970, 1971 ... 1999. Any 2-digit year (for example, 54) not convertible to a year in this range will cause a DOMAIN ERROR.

The sliding window equivalent is:

```
YY_WINDOW=-10,10
```

This would establish a valid date window, ten years either side of the current year. For example, if the current year is 1998, the valid range would be (1998-10) – (1998+10), in other words: 1988, 1989, → 2008.

One way of looking at the **yy_window** variable is that it specifies a 2-element vector. If you supply only the first element, the second one defaults to the first element + 99.

Note that the system uses only the number of digits in the year specification to determine whether it refers to a fixed (4-digits) or sliding (1-, or 2-digits) window. In fact you can have a fixed lower limit and a sliding upper limit, or vice versa.

```
YY_WINDOW=1990,10
```

Allows dates as early as 1990, but not more than 10 years hence.

```
YY_WINDOW=0,1999
```

Allows dates from the current year to the end of the century.

If the second date is before, or more that 99 years after the first date, then any date conversion will result in a `DOMAIN ERROR`. This might be useful in an application where the end-user has control over the input date format and you want to disallow any 2-digit date input.

```
YY_WINDOW=1,0
```

Registry Sub-Folders

A large amount of configuration information is maintained in the Windows Registry in sub-folders of the main folder identified by **inifile**.

Many of these values are dynamic, for example the position of the various Session windows, is maintained in a Registry sub-folder so that their appearance is maintained from one invocation of APL to the next. These type of Registry values are considered to be internal and are therefore not described herein.

However, and Registry Value that is maintained via a configuration dialog box will be named and described in the documentation for that dialog box in Chapter 2.

AutoComplete

This contains registry entries that describe your personal AutoComplete options. See Auto Complete Tab on page 149.

Charts

This contains entries that control the way charts are produced and displayed when you click one of the chart buttons. See Object Operations on page 127.

Colours

This contains entries that describe the colour schemes you have and your personal preferences. See Colour Selection Dialog on page 156.

Event Viewer

This contains entries that describe your settings for the Event Viewer. See page 252.

Explorer

This contains entries that describe your settings for the Workspace Explorer. See page 167.

files

This contains the size of your recently used file list (see page 134) and the list of your most recently loaded workspaces.

KeyboardShortcuts

This contains the definitions of your Keyboard Shortcuts (Unicode Edition only). See page 139.

LanguageBar

This contains the definitions of the symbols, tips, and help for the symbols in the LanguageBar.

Printing

This contains the entries for your Printer Setup options. See page 159.

SALT

This contains entries for SALT. See page 151.

Search

This contains dynamic entries for the Find Objects Tool. See page 199.

Threads

This contains entries to remember your preferences for Threads. See The Threads Menu on page 121.

ValueTips

This contains entries for your Value Tips preferences. See page 134.

WindowRects

This contains entries to maintain the position of various Session tool windows so that they remain consistent between successive invocations of APL.

Workspace Management

Workspace Size and Compaction

The *maximum* amount of memory allocated to a Dyalog APL workspace is defined by the **maxws** parameter.

Upon `)LOAD` and `)CLEAR`, APL allocates an amount of memory corresponding to the size of the workspace being loaded (which is zero for a clear ws) plus the *workspace delta*.

The workspace delta is $1/16^{th}$ of **maxws**, except if there is less than $1/16^{th}$ of **maxws** in use, delta is $1/64^{th}$ of **maxws**. This may also be expressed as follows:

```
delta←maxws{⌈⍺÷⍵>⍺÷16)⌽64 16}ws
```

where `maxws` is the value of the **maxws** parameter and `ws` is the currently allocated amount of workspace. If **maxws** is 16384KB, the workspace delta is either 256KB or 1024 KB, and when you start with a `clear ws` the workspace occupies 256KB.

When you erase objects or release symbols, areas of memory become free. APL manages these free areas, and tries to reuse them for new objects. If an operation requires a contiguous amount of workspace larger than any of the available free areas, APL reorganises the workspace and amalgamates all the free areas into one contiguous block as follows:

1. Any un-referenced memory is discarded. This process, known as *garbage collection*, is required because whole cycles of refs can become un-referenced.
2. Numeric arrays are *demoted* to their tightest form. For example, a simple numeric array that happens to contain only values 0 or 1, is demoted or *squeezed* to have a `⎕DR` type of 11 (Boolean).
3. All remaining used memory blocks are copied to the low-address end of the workspace, leaving a single free block at the high-address end. This process is known as *compaction*.
4. In addition to any extra memory required to satisfy the original request, an additional amount of memory, equal to the workspace delta, is allocated. This will always cause the process size to increase (up to the **maxws** limit) but means that an application will typically achieve its working process size with at most 4+15 memory reorganisations.
5. However, if after compaction, the amount of used workspace is less than 1/16 of the Maximum workspace size (MAXWS), the amount reserved for working memory is reduced to 1/64th MAXWS. This means that workspaces that are operating within 1/16th of MAXWS will be more frugal with memory

Note that if you try to create an object which is larger than free space, APL reports `WS FULL`.

The following system function and commands force a workspace reorganisation as described above:

```
⎕WA, )RESET, )SAVE, )LOAD, )CLEAR
```

However, in contrast to the above, **any spare workspace above the workspace delta is returned to the Operating System**. On a Windows system, you can see the process size changing by using Task Manager.

The system function `⎕WA` may therefore be used judiciously (workspace reorganisation takes time) to reduce the process size after a particularly memory-hungry operation.

Note that in Dyalog APL, the SYMBOL TABLE is entirely dynamic and grows and shrinks in size automatically. There is no `SYMBOL TABLE FULL` condition.

Interface with Windows

Windows Command Processor commands may be executed directly from APL using the system command `)CMD` or the system function `⎕CMD`. This system function is also used to start other Windows programs. For further details, see the appropriate sections in *Language Reference*.

Auxiliary Processors

Introduction

Auxiliary Processors (APs) are non-APL programs which provide Dyalog APL users with additional facilities. They run under the control of Dyalog APL.

Typically, APs are used where speed of execution is critical, for utility libraries, or as interfaces to other products. APs may be written in any compiled language, although C is preferred and is directly supported.

Starting an AP

An Auxiliary Processor is invoked using the dyadic form of ⎕CMD. The left argument to ⎕CMD is the name of the program to be executed; the value of the **wspath** parameter is used to find the named file. In Dyalog APL/W, the right argument to ⎕CMD is ignored.

```
'xutils' ⎕CMD ''
```

On locating the specified program, Dyalog APL starts the AP and initialises a memory segment for communication between the workspace and the AP. This communication segment allows data to be passed from the workspace to the other process, and for results to be passed back. The AP then sends APL some information about its external functions (names, code numbers and calling syntax), which APL enters in the symbol table. APL then continues processing while the AP waits for instructions.

Using the AP

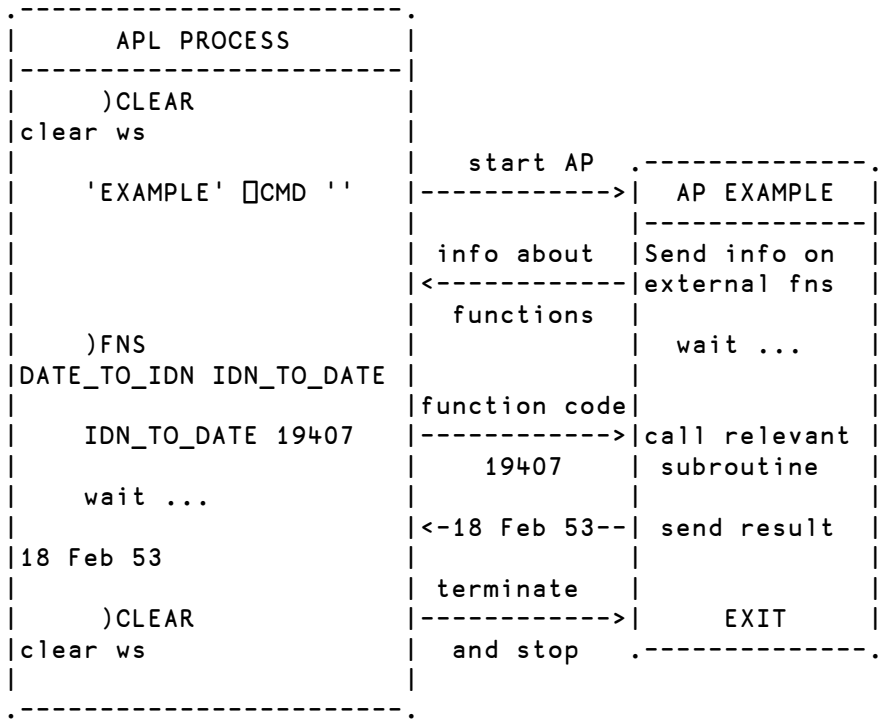
Once established, an AP is used by making a reference to one of its external functions. An external function behaves as if it was a locked defined function, but it is in effect an entry point to the AP. When an external function is referenced, APL transmits a code number to the AP, followed by any arguments. The AP then takes over and performs the desired processing before posting the result back.

Terminating the AP

An AP is terminated when all of its external functions are expunged from the active workspace. This could occur with the use of)CLEAR,)LOAD,)ERASE, ⎕EX,)OFF,)CONTINUE or ⎕OFF.

Example:

Start an Auxiliary Processor called **EXAMPLE**. This fixes two external functions called `DATE_TO_IDN` and `IDN_TO_DATE` which deal with the conversion of International Day Numbers to Julian Dates.



Access Control for External Variables

External variables may be EXCLUSIVE or SHARED. An exclusive variable can only be accessed by the owner of the file. If you are on a Local Area Network (LAN) a shared external variable may be accessed (concurrently) by other users. The exclusive or shared status of an external variable is set by the `XVAR` function in the UTIL workspace.

Access to an external variable is faster if it has exclusive status than if it is shared. This is because if several users are accessing the file data must always be read and written directly to disk. If it has exclusive status, the system uses buffering and avoids disk accesses where possible.

ODBC Configuration (SQAPL.INI)

SQAPL uses default parameters which are adequate for most purposes. They are:

```
MaxRows=50
MaxCursors=25
DefaultType=<C80
```

Should you wish to change any of these parameters, you must create an SQAPL.INI file. This file must be located in the directory specified by your **sqaplpath** parameter which is defined in the Software\Insight\SQApl section in the Windows Registry. This is inserted during installation and is normally the directory in which Dyalog APL/W is installed.

SQAPL.INI should contain a section for each of the connection service you wish to use, corresponding to the sections in your ODBC.INI.

Example:

```
[dBase_sdk20]
DatabaseType=ODBC
DefaultType=<C80
MaxCursors=30
MaxRows=100
```

The section name must be the same as the corresponding section name in the ODBC configuration file ODBC.INI. The **DatabaseType** parameter should always have the value ODBC, other versions of SQAPL also support SQLNK for a SequeLink service. **DefaultType** specifies the default data type to be used, and we recommend the value *<C80*, to make the default an 80-element character bind variable (see the section on Bind Variable Data Types for details). It is recommended that you use the defaults for the two parameters mentioned above.

MaxCursors specifies the maximum number of cursors which may be opened for this driver. **MaxRows** gives the default block size for Fetch operations with this driver. The APL programmer can set **MaxRows** for each cursor at run-time, but the value in SQAPL.INI file is used as the default.

Creating Executables

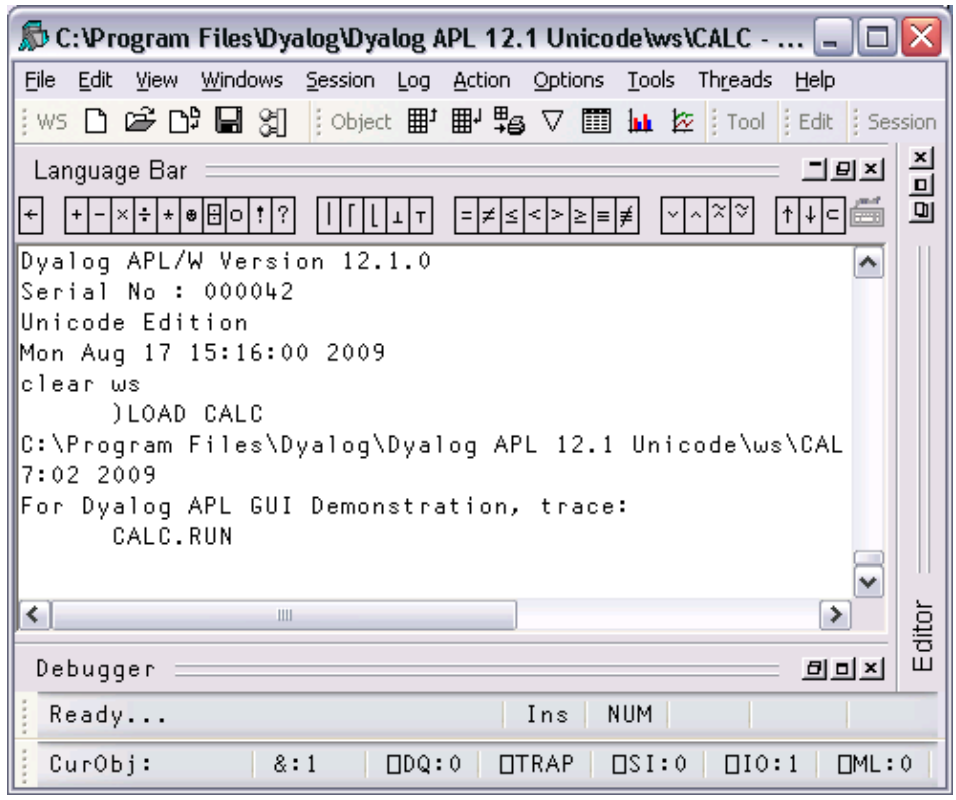
Dyalog APL provides the facility to package an APL workspace as a Windows executable (EXE). This may be done by selecting *Export ...* from the *File* menu of the APL Session window.

The system provides the following options:

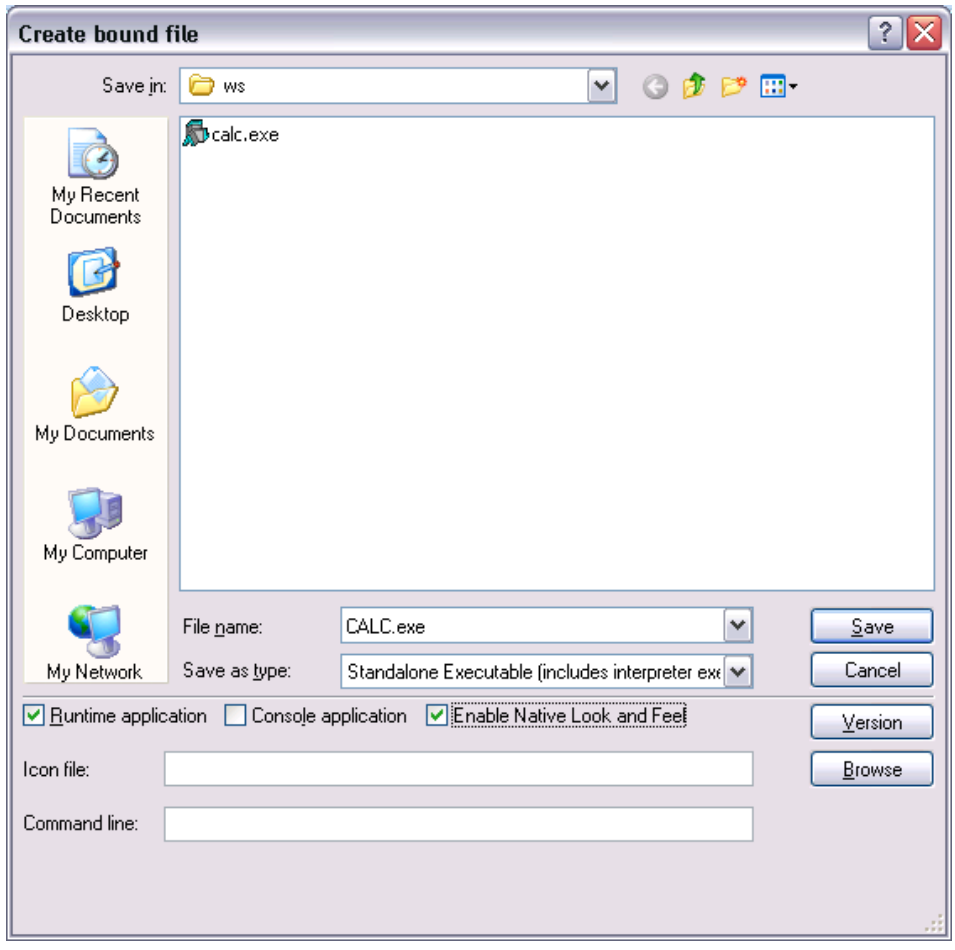
- You may bind your EXE as a Dyalog APL run-time application, or as a Dyalog APL developer application. The second option will allow you to debug the application should it encounter an APL error.
- You may bind your EXE as a console-mode application. A console application does not have a graphical user interface, but runs as a background task using files or TCP/IP to perform input and output.
- You may specify whether or not your .EXE will honour XP Look and Feel if this is enabled at the Windows level.

You can package the workspace as a stand-alone executable or as a .EXE file that must be accompanied by the Dyalog APL Dynamic Link Library (`dyalog120.dll` or `dyalog120rt.dll`), in which case the DLL should be installed in the same directory (as the EXE) or in the Windows System directory.

The following example illustrates how you can package the supplied workspace `calc.dws` as an executable. Before making the executable, it is essential to set up the latent expression to run the program using `⎕LX` as shown. Notice that in this case it is not necessary to execute `⎕OFF`; the `calc.exe` program will terminate normally when the user closes the calculator window and the system returns to Session input.



Then, when you select *Export...* from the *File* menu, the following dialog box is displayed.



In the example shown, the program is to be saved in `ws`, the (*supplied workspaces*) directory from which the workspace was loaded (the default).

The *Save as Type* option has been set to *Standalone Executable (includes interpreter exe)* which means that a single .EXE will be created containing the Dyalog APL executable and the CALC workspace.

The *Runtime application* checkbox is checked, indicating that `calc.exe` is to incorporate the runtime version of Dyalog APL..

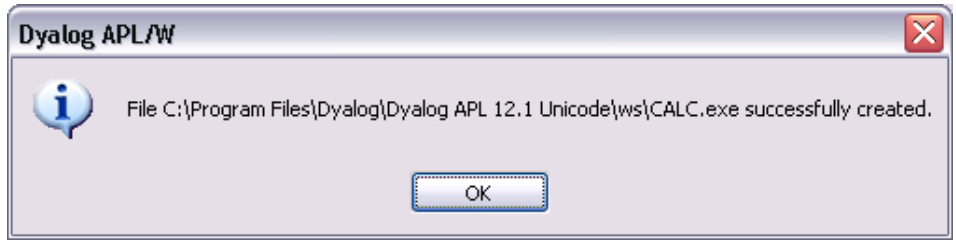
As this is a GUI application, the *Console application* checkbox is left unset.

The *Enable Native Look and Feel* checkbox has been set so that `calc.exe` will honour XP Look and Feel if it is enabled at the Windows level.

Note that if you enter the name of a file containing an icon (use the *Browse* button to browse for it) that icon will be bound with your executable and be use instead of the standard Dyalog APL icon.

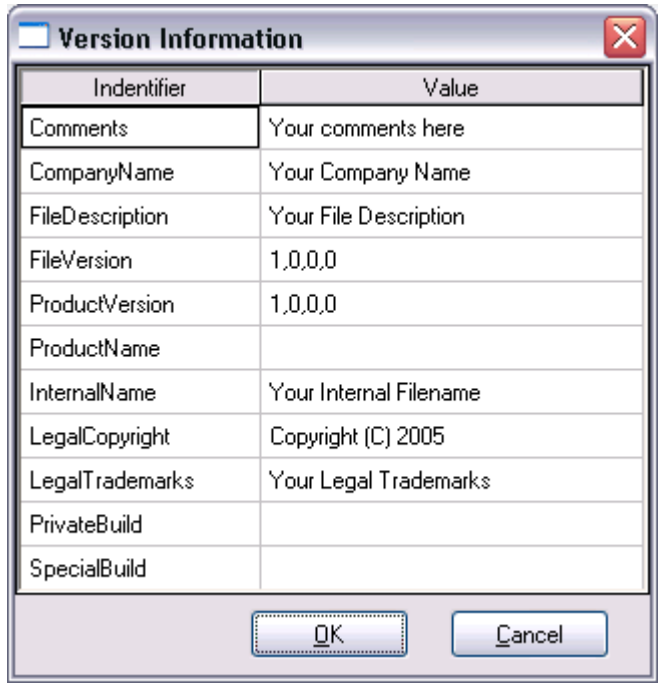
The Command Line box allows you to enter parameters and values that are to be passed to your executable when it is invoked.

On clicking *Save*, the following message box is displayed to confirm success.



Version Information

You may embed version information into your .exe by clicking the *Version* button and then completing the *Version Information* dialog box that is illustrated below.



Run-Time Applications and Components

Using Dyalog APL you may create different types of run-time applications and components. Note that the distribution of run-time applications and components requires a Dyalog APL Run-Time Agreement. Please contact Dyalog or your distributor, or see the Dyalog web page for more information.

The following table shows a list of distributable components for the two Editions. These are referred to in the remainder of this Chapter by the name shown in the first column of the table. It is essential that you distribute the components that are appropriate for the Edition you are using.

Name	Folder	File Name
Unicode Edition		
Run-Time EXE	`Dyalog APL 12.1 Unicode`	`dyalogrt.exe`
Run-Time DLL	`Dyalog APL 12.1 Unicode\bin`	`dyalog121rt_unicode.dll`
Bridge DLL	`Dyalog APL 12.1\bin`	`bridge121_unicode.dll`
DYALOG32	`Dyalog APL 12.1 Unicode\bin`	`dyalog32.dll`
DOS_32	`Dyalog APL 12.1 Unicode`	`dos_32.dll`
Classic Edition		
Run-Time EXE	`Dyalog APL 12.1 Classic`	`dyalogrt.exe`
Run-Time DLL	`Dyalog APL 12.1 Classic\bin`	`dyalog121rt.dll`
Bridge dll	`Dyalog APL 12.1\bin`	`bridge121.dll`
DYALOG32	`Dyalog APL 12.1 Classic\bin`	`dyalog32.dll`
DOS_32	`Dyalog APL 12.1 Classic`	`dos_32.dll`
Both Editions		
DyalogNet DLL	`Dyalog APL 12.1\bin`	`dyalognet.dll`

Stand-alone run-time

This is the simplest type of run-time to install. Using the *File/Export* menu item on the Session window, you can create a standard Windows executable program file (EXE) which contains your workspace and the Run-Time version of the Dyalog APL interpreter. To distribute your application, you need to supply and install:

1. Your bound executable (EXE)
2. whatever additional files that may be required by your application

The command-line for your application should simply invoke your EXE, with whatever start-up parameters it may require. Note that your application icon and any start-up parameters for the Run-Time Interpreter are specified and bound with the EXE when you make it.

If your application uses any component of the Microsoft .Net Framework, you must also distribute the Bridge DLL and DyalogNet DLL which must both be installed in the *global assembly cache* (GAC) using the `gacutil.exe` utility program. In addition, the Bridge DLL must either be on the system path or placed in the same directory as your EXE.

Bound run-time

This option requires the separate installation of the Run-Time DLL, but compared with the *stand-alone executable* option, may save disk space and memory if your customer installs and runs several different Dyalog applications. Using the *File/Export* menu item on the Session window, you can create a standard Windows executable program file (EXE) which contains your workspace bound to the Run-Time DLL. To distribute your application, you need to supply and install:

1. Your bound executable (EXE)
2. The Run-Time DLL
3. whatever additional files that may be required by your application

The command-line for your application should simply invoke your EXE, with whatever start-up parameters it may require. Note that your application icon and any start-up parameters for the Run-Time DLL are specified and bound with the EXE when you make it.

If your application uses any component of the Microsoft .Net Framework, you must also distribute the Bridge DLL and DyalogNet DLL which must both be installed in the *global assembly cache* (GAC) using the `gacutil.exe` utility program. In addition, the Bridge DLL must either be on the system path or placed in the same directory as your EXE.

Workspace based run-time

A workspace based run-time application consists of the Dyalog APL Run-Time Program (Run-Time EXE) and a separate workspace. To distribute your application, you need to supply and install:

1. Your workspace
2. The Run-Time EXE
3. whatever additional files that may be required by your application

The command-line for your application invokes the Run-Time EXE, passing it start-up parameters required for the Run-Time EXE itself (such as MAXWS) and any start-up parameters that may be required by your application. You will need to associate your own icon with your application during its installation.

If your application uses any component of the Microsoft .Net Framework, you must also distribute the Bridge DLL and DyalogNet DLL which must both be installed in the *global assembly cache* (GAC) using the `gacutil.exe` utility program. In addition, the Bridge DLL must either be on the system path or placed in the same directory as your EXE.

Out-of-process COM Server

To make an out-of-process COM Server, you must:

1. Establish one or more OLEServer namespaces in your workspace, populated with functions and variables that you wish to export as methods, properties and events.
2. Use the *File/Export ...* menu item on the Session window to register the COM Server on your computer so that it is ready for use.

The command-line for your COM Server invokes the Run-Time EXE, passing it start-up parameters required for the Run-Time EXE itself (such as MAXWS) and any start-up parameters that may be required by your application.

To distribute an out-of-process COM Server, you need to supply and install the following files:

1. Your workspace
2. The associated Type Library (.tlb) file (created by *File/Export*)
3. The Run-Time EXE
4. whatever additional files that may be required by your application

To install an out-of-process COM Server you must set up the appropriate Windows registry entries. See Interface Guide for details.

In-process COM Server

To make an in-process COM Server, you must:

1. Establish one or more OLEServer namespaces in your workspace, populated with functions and variables that you wish to export as methods, properties and events.
2. Use the *File/Export ...* menu item on the Session window to create an in-process COM Server (DLL) which contains your workspace bound to the Run-Time DLL. This operation also registers the COM Server on your computer so that it is ready for use.

To distribute your component, you need to supply and install

1. Your COM Server file (DLL)
2. The Run-Time DLL
3. Whatever additional files that may be required by your COM Server.

Note that you must register your COM Server on the target computer using the `regsvr32.exe` utility.

ActiveX Control

To make an ActiveX Control, you must:

1. Establish an ActiveXControl namespaces in your workspace, populated with functions and variables that you wish to export as methods, properties and events.
2. Use the *File/Export* → menu item on the Session window to create an ActiveX Control file (OCX) which contains your workspace bound to the Dyalog APL Run-Time Dynamic Link Library (`dyalog121rt.dll`). This operation also registers the ActiveX Control on your computer so that it is ready for use.

To distribute your component, you need to supply and install

1. Your ActiveX Control file (OCX)
2. The Run-Time DLL
3. Whatever additional files that may be required by your ActiveX Control.

Note that you must register your ActiveX Control on the target computer using the `regsvr32.exe` utility.

Microsoft .Net Assembly

A Microsoft .Net Assembly contains one or more .Net Classes. To make a Microsoft .Net Assembly, you must:

1. Establish one or more NetType namespaces in your workspace, populated with functions and variables that you wish to export as methods, properties and events.
2. Use the *File/Export ...* menu item on the Session window to create a Microsoft .Net Assembly (DLL) which contains your workspace bound to the Run-Time DLL.

To distribute your .Net Classes, you need to supply and install

1. Your Assembly file (DLL)
2. The Run-Time DLL
3. The Bridge DLL
4. The DyalogNet DLL
5. Whatever additional files that may be required by your .Net Assembly.

The Bridge DLL and DyalogNet DLL must be installed in the *global assembly cache* (GAC) using the `gacutil.exe` utility program. In addition, the Bridge DLL must be on the system path.

Additional Files for SQAPL

If your application uses the *SQAPL/EL ODBC* interface, you must distribute and install four additional files, according to the Edition you are using, as shown in the table below.

Name	Folder	File Name
Unicode Edition		
SQAPL INI	`Dyalog APL 12.1 Unicode`	`sqapl.ini`
SQAPL ERR	`Dyalog APL 12.1 Unicode`	`sqapl.err`
SQAPL DLL	`Dyalog APL 12.1 Unicode\bin`	`cndya60Uni.dll`
APLUNICD INI	`Dyalog APL 12.1 Unicode\bin`	`aplunicd.ini`
Classic Edition		
SQAPL INI	`Dyalog APL 12.1 Classic`	`sqapl.ini`
SQAPL ERR	`Dyalog APL 12.1 Classic`	`sqapl.err`
SQAPL DLL	`Dyalog APL 12.1 Classic\bin`	`cndya60.dll`
APLUNICD INI	`Dyalog APL 12.1 Classic\bin`	`aplunicd.ini`

The SQAPL DLL must be installed in the user's Windows directory or be on the user's path.

You must also create the following registry entries (for each and every user) in a folder named `HKEY_CURRENT_USER/Software/Insight/SQAPL`. You cannot specify these parameters any other way.

APL_UNICODE This specifies the full path name of the APLUNICD INI, including the file name and its extension.

SQAPLPATH This specifies the full path name of the directory in which the SQAPL INI and SQAPL ERR files are installed

Miscellaneous Other Files

AUXILIARY PROCESSORS

If you use any of the Auxiliary Processors (APs) included in the sub-directory `XUTILS`, you must include these with your application. Note that, like workspaces, Dyalog APL searches for APs using the `wspath` parameter. If your application uses APs, you must ensure that you specify `wspath` or that the default `wspath` is adequate for your application..

DYALOG32

This DLL is used by some of the functions provided in the `QUADNA.DWS` workspace. If you include any of these in your application this DLL must be installed in the user's Windows directory or be on the user's path.

DOS_U32

This DLL is used by the functions provided in the `DOSUTILS.DWS` workspace. If you include any of these in your application this DLL must be installed in the user's Windows directory or be on the user's path.

Registry Entries for Run-Time Applications

The Run-Time DLL does not obtain any parameter values from the Windows registry. If you need to specify any Dyalog APL parameter values, they must be defined in the command line when you create an EXE.

The Run-Time EXE *does* obtain parameter values for the Windows registry, but does not require them to be present. If the default values of certain parameters are inappropriate, you may specify their values on the command line. There is normally no requirement to install registry entries for a run-time application that uses the Run-Time EXE.

For example, your application may requires a greater or lesser `MAXWS` parameter (workspace size) than the default value. This may be done by adding the phrase `MAXWS=nnnn` (where `nnnn` is the required workspace size **in kilobytes**) after the name of your application workspace on the command line, for example:

```
dyalogrt.exe MYAPP.DWS MAXWS=8096
```

Note that the default value of the `DYALOG` parameter (which specifies where it looks for various other files and sub-directories) is the directory from which the application (`dyalogrt.exe`) is loaded.

Nevertheless, registry entries will be required in the following circumstances.

1. If your Classic Edition run-time application requires that the user inputs APL characters, you will need to specify input/output tables (parameters `APLK`, `APLT`, `APLKEYS` and `APLTRANS`).
2. If your application uses the `NFILES` Auxiliary Processor (now superseded by the `⎕Nxxx` system functions), you must specify a registry entry for the `APLKEYS` parameter. This is required so that `NFILES` can find any translate tables you may use. Note that `NFILES` cannot see the values of parameters specified on the APL command line, so you must specify `APLKEYS` in the registry.

Installing Registry Entries

To specify parameters using the Registry, you must install a suitable registry folder for each user of your application. By default, Version 12.1 will use the registry folder:

```
HKEY_CURRENT_USER\Software\Dyalog\Dyalog APL/W 12.1 Unicode
```

or

```
HKEY_CURRENT_USER\Software\Dyalog\Dyalog APL/W 12.1
```

You may choose a different name for your registry folder if you wish. If so, you must tell Dyalog APL the name of this folder by specifying the `INIFILE` parameter on the command line. For example:

```
dyalogrt.exe MYAPP.DWS INIFILE=Software\MyCo\MyApplication
```

You may install entries into the registry folder in one of two ways:

1. Using a proprietary installation program such as InstallShield

2. Using the `REGEDIT` utility. This utility program installs registry entries defined in a text file that is specified as the argument to the program. For example, if your file is called `APLAPP.REG`, you would install it on your user's system by executing the command:

   ```
   REGEDIT APLAPP.REG
   ```

   An example 5-line file that specifies the `APLNID` and `MAXWS` parameters might be as follows:

   ```
   Windows Registry Editor Version 5.00

   [HKEY_CURRENT_USER\Software\Dyalog\Dyalog APL/W 12.1]
   "aplnid"="42"
   "maxws"="8096"
   ```

COM Objects and the Dyalog APL DLL

Introduction

In each Edition, there are two versions of the Dyalog APL Dynamic Link Library, named `dyalog121_unicode.dll` and `dyalog121rt_unicode.dll` (Unicode Edition) and `dyalog121.dll` and `dyalog121rt.dll` (Classic Edition).

`dyalog121_unicode.dll` and `dyalog121.dll` are complete Dyalog APL development systems packaged as Dynamic Link Libraries.

`dyalog121_unicode.dll` and `dyalog121rt_unicode.dll` and `dyalog121rt.dll` are the run-time versions of `dyalog121.dll`.

In the remainder of this section, the term *the Dyalog APL DLL* is used to refer to any one of these DLLs. The term COM object is used to refer to a Dyalog APL in-process OLE Server (OLEServer object) or a Dyalog APL ActiveX Control (ActiveXControl object).

The Dyalog APL DLL is used to host COM objects and .Net objects written in Dyalog APL. Although this section describes how it operates with COM objects, much of this also applies when it hosts .Net objects. Further information is provided in the *.Net Interface Guide*.

Classes, Instances and NameSpace Cloning

A COM object, whether written in Dyalog APL or not, represents a *class*. When a host application loads a COM object, it actually creates an *instance* of that class.

When a host application creates an instance of a Dyalog APL COM object, the corresponding OLEServer or ActiveXControl namespace is *cloned*. If the host creates a second instance, the original namespace is cloned a second time.

Cloned OLEServer and ActiveXControl namespaces are created in almost exactly the same way as those that you can make yourself using ⎕OR and ⎕WC except that they do not have separate names. In fact, each clone believes itself to be the one and only original OLEServer or ActiveXControl namespace, with the same name, and is completely unaware of the existence of other clones.

Notice that cloning does not initially replicate all the objects within the OLEServer or ActiveXControl namespace. Instead, the objects inside the cloned namespaces are actually represented by pointers to the original objects in the original namespace. Only when an object is changed does any information get replicated. Typically, the only objects likely to differ from one instance to another are variables, so only one copy of the functions will exist in the workspace. This design enables many instances of a Dyalog APL COM object to exist without overloading the workspace.

Workspace Management

The Dyalog APL DLL does not use a fixed maximum workspace size, but automatically increases the size of its active workspace as required. If you write a run-away COM object, or if there is insufficient computer memory available to load a new control, it is left to the host application or to Windows itself to deal with the situation.

When an application loads its first Dyalog APL COM object, it starts the Dyalog APL DLL which initialises a `CLEAR WS.` It then copies the namespace tree for the appropriate OLEServer or ActiveXControl object into its active workspace.

This namespace tree comprises the OLEServer or ActiveXControl namespace itself, together with all its parent namespaces *with the exception of* the root workspace itself. Note that for an ActiveXControl, there is at least one parent namespace that represents a Form.

For example, if an ActiveXControl namespace is called `#.F.Dual`, the Dyalog APL DLL will copy the contents of `#.F` into its active workspace when the first instance of the control is loaded by the host application.

If the same host application creates a *second instance* of the *same* OLEServer or ActiveXControl, the original namespace is cloned as described above and there is no further impact on the workspace

If the same host application creates an instance of a *different* Dyalog APL COM object, the namespace tree for this second object is copied from its DLL or OCX file into the active workspace. For example, if the second control was named `X.Y.MyControl`, the entire namespace `X` would be copied. This design raises a number of points:

1. Unless you are in total control of the user environment, you should design a Dyalog APL COM object so that it can operate in the same workspace as another Dyalog APL COM object supplied by another author. You cannot make any assumptions about file ties or other resources that are properties of the workspace itself.
2. If you write an ActiveXControl whose ultimate parent namespace is called `F`, a host application could not use your control at the same time as another ActiveXControl (perhaps supplied by a different author) whose ultimate parent namespace is also called `F`.
3. Dyalog APL COM objects must not rely on variables or utility functions that were present in the root workspace when they were saved. These functions and variables will *not* be there when the object is run by the Dyalog APL DLL.
4. A Dyalog APL COM object may *create* and subsequently *use* functions and variables in the root workspace, but if two different COM objects were to adopt the same policy, there is a danger that they would interfere with one another. The same is true for `⎕SE`.

Multiple COM Objects in a Single Workspace

If your workspace contains several OLEServer or ActiveXControl objects which have the same ultimate parent namespace, the Dyalog APL DLL will copy them all into the active workspace at the time when the first one is instanced. If the host application requests a second COM object that is already in the workspace, the namespace tree is not copied again.

If the workspace contains several OLEServer or ActiveXControl objects which have different ultimate parents, their namespace trees will be copied in separately.

Parameters

The Dyalog APL DLL does not read parameters such as aplnid or wspath from the registry, command-line or environment variables. This means that all such parameters will have their default values.

System Errors

Introduction

Dyalog APL will display a System Error Dialog and (normally) terminate in one of two circumstances:

1. As a result of the failure of a workspace integrity check
2. As a result of a System Exception

Workspace Integrity

When you `)SAVE` your workspace, Dyalog APL first performs a workspace integrity check. If it detects any discrepancy or violation in the internal structure of your workspace, APL does not overwrite your existing workspace on disk. Instead, it displays the System Error dialog box and saves the workspace, together with diagnostic information, in an **aplcore** file before terminating.

A System Error code is displayed in the dialog box and should be reported to Dyalog for diagnosis.

Note that the internal error that caused the discrepancy could have occurred at any time prior to the execution of `)SAVE` and it may not be possible for Dyalog to identify the cause from this **aplcore** file.

If APL is started in debug mode with the **–Dc**, **-Dw** or **–DW** flags, the Workspace Integrity check is performed more frequently, and it is more likely that the resulting aplcore file will contain information that will allow the problem to be identified and corrected.

System Exceptions

Non-specific System Errors are the result of Operating System exceptions that can occur due to a fault in Dyalog APL itself, an error in a Windows or other DLL, or even as a result of a hardware fault. The following system exceptions are separately identified.

Code	Description	Suggested Action
900	A Paging Fault has occurred	As the most likely cause is a temporary network fault, recommended course of action is to restart your program.
990 & 991	An exception has occurred in the Development or Run-Time DLL.	
995	An exception has occurred in a DLL function called via ⎕NA	Carefully check your ⎕NA statement and the arguments that you have passed to the DLL function
996	An exception has occurred in a DLL function called via a *threaded* ⎕NA call	As above
997	An exception has occurred while processing an incoming OLE call	
999	An exception has been caused by Dyalog APL or by the Operating System	

Recovering Data from aplcore files

Objects may often (but not always) be recovered from **aplcore** using `)COPY`. Note that because (by default) the aplcore file has no extension, it is necessary to explicitly add a dot, or APL will attempt to find the non-existent file aplcore.DWS, i.e.

```
)COPY aplcore.
```

Reporting Errors to Dyalog

If APL crashes and saves an aplcore file, please email the following information to support@dyalog.com:

- a brief description of the circumstances surrounding the error
- your Dyalog APL Version number and Build ID (see Help/About)
- the aplcore file itself

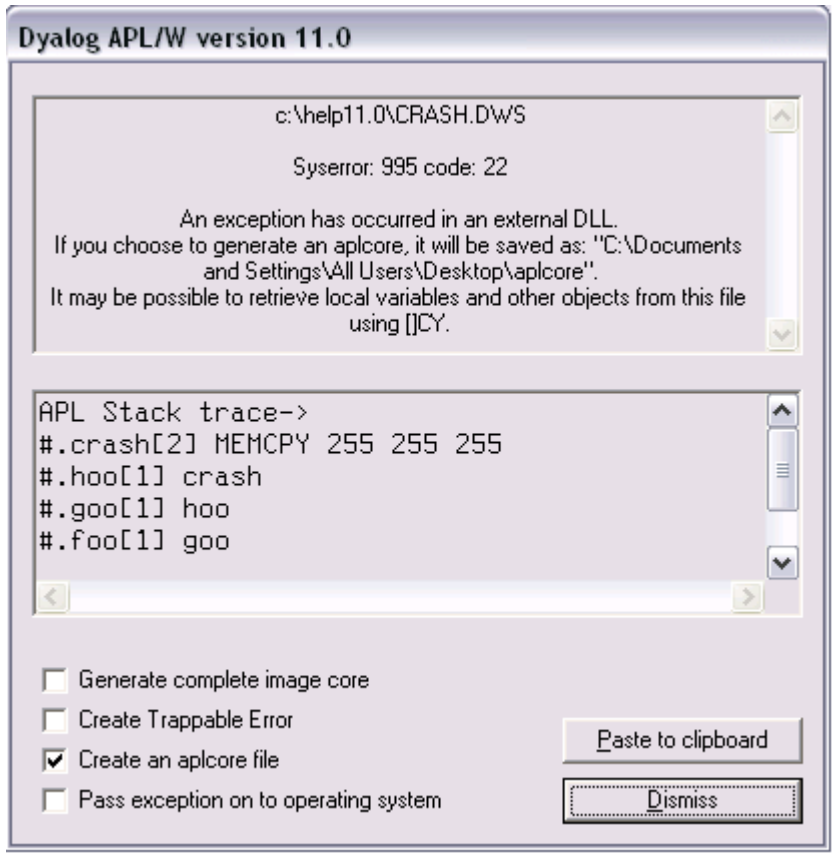
If the problem is reproducible, i.e. can be easily repeated, please also send the appropriate description, workspace, and other files required to do so.

System Error Dialog Box

The System Error Dialog illustrated below was produced by deliberately inducing a system exception in the Windows DLL function `memcpy()`. The functions used were:

```
     ∇ foo
[1]    goo
     ∇
     ∇ goo
[1]    hoo
     ∇
     ∇ hoo
[1]    crash
     ∇

     ∇ crash
[1]    ⎕NA'dyalog32|MEMCPY u u u'
[2]    MEMCPY 255 255 255
     ∇
```


Dyalog APL/W version 11.0
c:\help11.0\CRASH.DWS
Syserror: 995 code: 22
An exception has occurred in an external DLL.
If you choose to generate an aplcore, it will be saved as: "C:\Documents and Settings\All Users\Desktop\aplcore".
It may be possible to retrieve local variables and other objects from this file using []CY.
APL Stack trace->
#.crash[2] MEMCPY 255 255 255
#.hoo[1] crash
#.goo[1] hoo
#.foo[1] goo
Generate complete image core
Create Trappable Error
Create an aplcore file
Pass exception on to operating system
Paste to clipboard
Dismiss

Options

Item	Parameter	Description
Generate complete image core	CreateAplCoreonSyserror	Dumps a complete core image with the *User Mode Process Dumper* (a Microsoft tool) - see below.
Create Trappable Error		If you check this box (only enabled on System Error codes 995 and 996), APL will not terminate but will instead generate an error 91 (`EXTERNAL DLL EXCEPTION`) when you press *Dismiss*.
Create an aplcore file	CreateAplCoreonSyserror	If this box is checked, an aplcore file will be created.
Pass exception on to operating system	PassExceptionsToOpSys	If this box is checked, the exception will be passed on to your current debugging tool (e.g. Visual Studio).
Paste to clipboard		Copies the contents of the APL stack trace window to the Clipboard.

Generate complete image core

The *Generate complete image core* option attempts to execute `[SYSDIR]\userdump.exe`, where `[SYSDIR]` is the windows system directory (typically `c:\windows\system32`, and `userdump.exe` is the User Mode Process Dumper, a Microsoft tool that can be downloaded from the following url (which you may copy from Winhelp and paste into a browser):

http://www.microsoft.com/downloads/details.aspx?FamilyID=e23cd741-d222-48df-9cd8-28796f414256&DisplayLang=en

The process creates a file called dyalog.core in the current directory. This file contains much more debug information than a normal aplcore (and is much larger than an aplcore) and can be sent to Dyalog Limited (zip it first please). Alternatively the file can be loaded into Visual Studio .Net to do your own debugging.

Debugging your own DLLs

If you are using Visual Studio on Microsoft Windows XP (or similar), the following procedure should be used to debug your own DLLs when an appropriate Dyalog APL System Error occurs.

Ensure that the *Pass Exception* box is checked, then click on *Dismiss* to close the System Error dialog box.

The system exception dialog box appears. Click on *Debug* to start the process in the Visual Studio debugger.

After debugging, the system exception dialog box appears again. Click on *Don't send* to terminate Microsoft Windows XP's exception handling.

ErrorOnExternalException Parameter

This parameter allows you to prevent APL from displaying the System Error dialog box (and terminating) when an exception caused by an external DLL occurs. The following example illustrates what happens when the functions above are run, but with ErrorOnExternalException set to 1.

```
     □←2 □NQ'.' 'GetEnvironment'
'ErrorOnExternalException'
1
     foo
EXTERNAL DLL EXCEPTION
crash[2] MEMCPY 255 255 255
         ^
     □EN
91
     )SI
crash[2]*
hoo[1]
goo[1]
foo[1]
```

CHAPTER 2

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

Unicode Edition

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.

Unicode Edition is supplied with a two sets of such keyboards (one using Ctrl and one using AltGr) for a range of different languages. These keyboards were created using the Microsoft Keyboard Layout Creator (MSKLC) and you may use the same tool to customise one of the supplied keyboards or to create a new one.

Classic Edition

Classic Edition uses a proprietary mechanism for the input of APL symbols and also provides a fully customisable keyboard.

The layout is defined by an Input Translate Table whose name is specified by the **aplk** parameter. This is a character file with a .DIN extension that (normally) resides in the APLKEYS sub-directory. The Input Translate Table provides two kinds of mapping.

Firstly, it specifies the mapping between a keystroke and a character in ⎕AV. For example (in unified mode) it specifies that Ctrl+r means ⎕AV[174] (⍴).

Secondly, it specifies the mapping between keystrokes and special *actions* or *commands*. For example, that Shift+Delete means *cut*. In non-GUI implementations of Dyalog APL, all commands must be issued through the keyboard. In Dyalog APL/W, most commands may also be given using menus and buttons or with the mouse. Commands are mapped to particular keystrokes through the Input Translate Table for your keyboard. The keystrokes used have been carefully chosen so as to be compatible with Common User Access (CUA) conventions. If you do not like this standard mapping, you can change it by editing this file.

Dyalog APL IME

The Dyalog APL IME, which is provided with both Editions, is an *Input Method Editor*, a Microsoft Windows technology designed for entering characters that are not present on a normal keyboard. It is provided so that you can enter APL symbols in other applications, including text editing and word-processing programs such as NotePad, WordPad, and Microsoft Word. In particular, it allows you to use such applications to create and edit APL scripts.

Although the Unicode APL keyboards may also be used with other applications, they suffer from conflicts with application dependent hot keys. For example, if you use the Dyalog Ctrl keyboard with Microsoft Word, you will find that Ctrl+A will select all the text in the document, rather than produce the symbol ⍺. The IME is different from "normal" keyboards in that any keystroke which is defined as producing a symbol is invisible to the application which is using the IME. The application only "sees" the character that is generated. Thus, if you use the Dyalog IME with Microsoft Word, Ctrl+A will not select all the text in your document, but always produce the symbol ⍺.

Of course, this means that you can no longer use hotkeys based on Ctrl in Word when the IME is active.

The Dyalog APL IME keyboard layout is identical to that provided by the Classic Edition in Unified mode, as shown on page 75.

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 the section on the Editor later in this Chapter. See also DoubleClickEdit parameter.

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.

Version 12.1 is supplied with two sets of such keyboards (one using Ctrl and one using AltGr) for a range of different languages as listed below. These keyboards were created using the Microsoft Keyboard Layout Creator (MSKLC) and you may use the same tool to customise one of the supplied keyboards or to create a new one.

Installation

During the Installation of Dyalog Version 12.1 Unicode Edition, setup installs a Ctrl and (if available) an AltGr APL keyboard layout onto your system. These keyboard layouts are installed as additional services for your default Input Language.

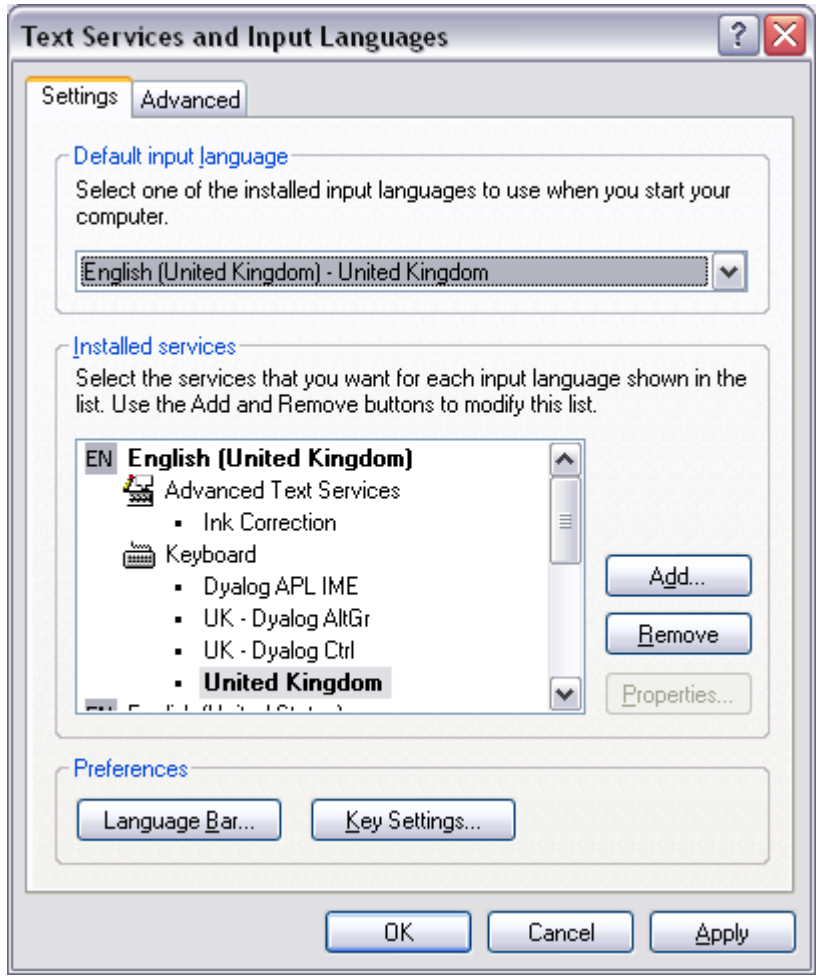
The following table lists the APL keyboards included with Dyalog APL Version 12.1 Unicode Edition at the time of publication. Other keyboards will be included as they are developed.

Ctrl Keyboards	**AltGr Keyboards**
Danish - Dyalog Ctrl	Danish - Dyalog AltGr
Finnish - Dyalog Ctrl	Finnish - Dyalog AltGr
French - Dyalog Ctrl	French - Dyalog AltGr
German - Dyalog Ctrl	German Dyalog AltGr
Icelandic - Dyalog Ctrl	
Italian - Dyalog Ctrl	Italian - Dyalog AltGr
Norwegian - Dyalog Ctrl	
Russian - Dyalog Ctrl	
Swedish - Dyalog Ctrl	
UK - Dyalog Ctrl	UK - Dyalog AltGr
US - Dyalog Ctrl	US - Dyalog AltGr

Setup automatically installs only those keyboards that correspond to your default Input Language, as specified via *Control Panel/Regional and Language Options*.

Note that if your default input language is not one of those listed in the table, Setup will not install any APL keyboards. However, you may create your own layout (or adapt one of the existing ones) using MSKLC).

The following picture illustrates the *Text Services and Input Languages* configuration pane after installing Unicode Edition onto a Windows XP system on which the default Input Language is English (United Kingdom).



Which Keyboard Should I Use?

The two keyboard layouts (Dyalog Ctrl, Dyalog AltGr) and the Dyalog APL IME have the following advantages and disadvantages, depending on which application you use them with:

Ctrl keyboards are familiar and likely to be the first choice for anyone with experience in using Dyalog APL. The keyboards work well in the APL session, but if you use them with other common Windows applications, many of the APL symbols cannot be entered because applications use key combinations like Ctrl+A to select all text or Ctrl+S to save the current document.

AltGr keyboards move the APL symbols onto keys which are less frequently used as shortcuts. However, there is generally only one AltGr key on a keyboard, so typing is a little less convenient. Also, some applications are starting to use AltGr-based shortcuts. For example, Google Desktop uses AltGr+G as a global hotkey.

The **IME** is required to APL scripts using non-APL text editors and/or word processing software in situations where the **Ctrl** keyboard suffers from conflicts with application dependent hot-keys.

Recommended Strategies

Dyalog recommends one of the following two strategies:

AltGr: If you are comfortable with the **AltGr** keyboard and you don't have many situations where you need to type APL characters into an application in which AltGr is used for hotkeys, set your *locale - Dyalog AltGr* keyboard up as the default keyboard on your machine, and it should work well in all applications. You can activate the IME using the Windows Language Bar if you occasionally have a problem with application hotkeys making it hard to enter APL symbols.

Ctrl+IME: If you find that the **Ctrl** layout is more attractive, you can use the **Ctrl** keyboard most of the time, and switch to the IME if you need to enter APL symbols into an application which uses Ctrl for hotkeys. If you need to switch frequently, we recommend that you use Control Panel to set up a hotkey to toggle between keyboards (and possibly remove the AltGr keyboard to reduce the number of keyboards that you will be toggling between).

Configuring your APL Keyboard for Use

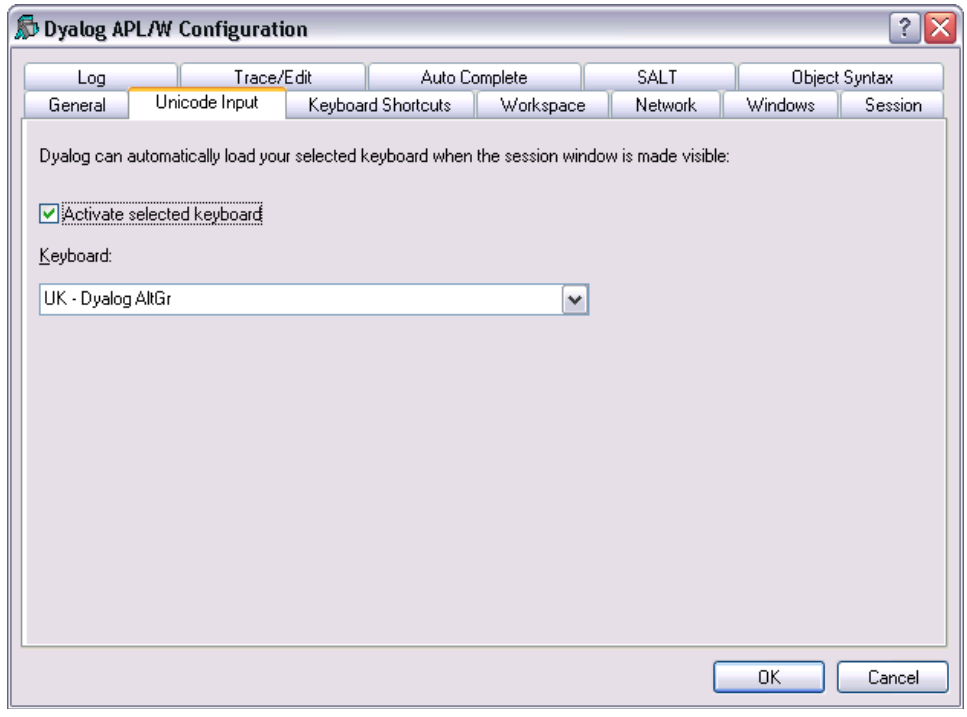
There are 3 different ways to use your APL keyboard:

1. Configure APL to automatically select your APL keyboard on start-up
2. Manually select your APL keyboard for use with your APL session window every time you start APL.
3. Make the APL keyboard your default Windows keyboard (for all applications)

Automatic Keyboard Selection

Unicode Edition can optionally select your APL keyboard each time you start APL. To achieve this, open the Unicode Input configuration pane (Options/Configure/Unicode Input) then:

In the *Keyboard* drop-drown, select one of your installed APL keyboards.
Enable the *Activate selected keyboard* checkbox
Click *OK*



The value of the checkbox and the name of your chosen keyboard are saved in registry keys named `InitialKeyboardLayoutInUse` and `InitialKeyboardLayout`.

The choices shown in the above picture will be reflected by the following values:

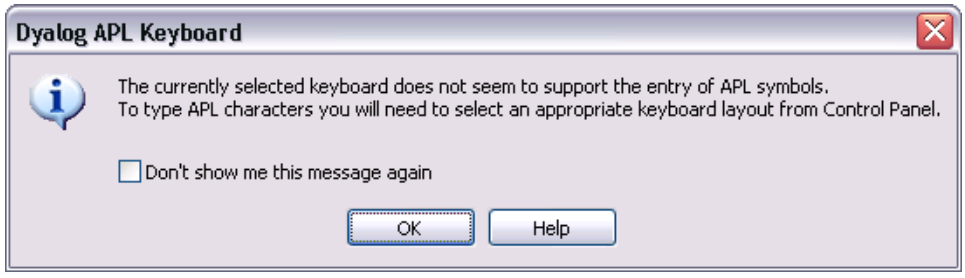
```
InitialKeyboardLayoutInUse = 1

InitialKeyboardLayout = " UK - Dyalog AltGr"
```

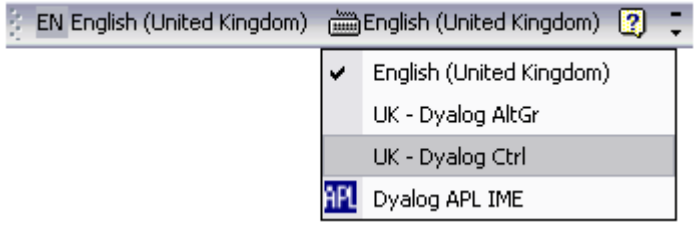
Manual Keyboard Selection

Each time you start APL, the Session window will be associated with your current Windows keyboard layout. This will be either your default keyboard, or the one you most recently selected from the Language Bar.

On start-up, Unicode Edition tests your current keyboard to see if it includes any definitions that will generate an APL symbol. If the current keyboard is incapable of generating APL symbols, the system will display the following message box.



You can switch to an APL keyboard using the Language Bar, as illustrated in the following picture:

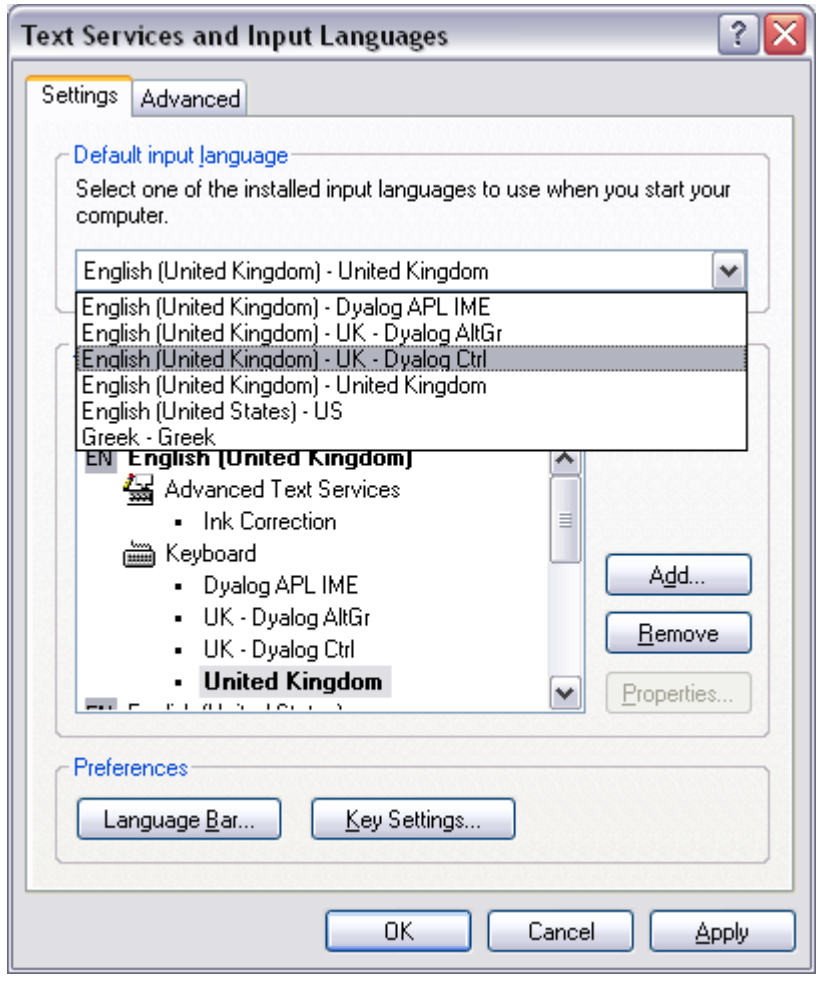


Making your APL Keyboard the default Windows keyboard

Both the Dyalog AltGr and Dyalog Ctrl keyboard layouts are designed to be fully compatible with your standard keyboard and you may adopt one of these as your default Windows keyboard. To do this, simply make it the Default Input Language as illustrated by the next 2 pictures. Note that the default keyboard layout is shown in bold.

To change your default keyboard (Windows XP), open *Control Panel/Regional and Languages*, select the *Languages* tab and click *Details*. This brings up the *Text Services and Input Languages* dialog box shown below.

Select your choice of APL keyboard from the drop-down list as illustrated.



If you wish to, you can select a keystroke to enable you to select it quickly from the keyboard.

On-Screen Keyboard

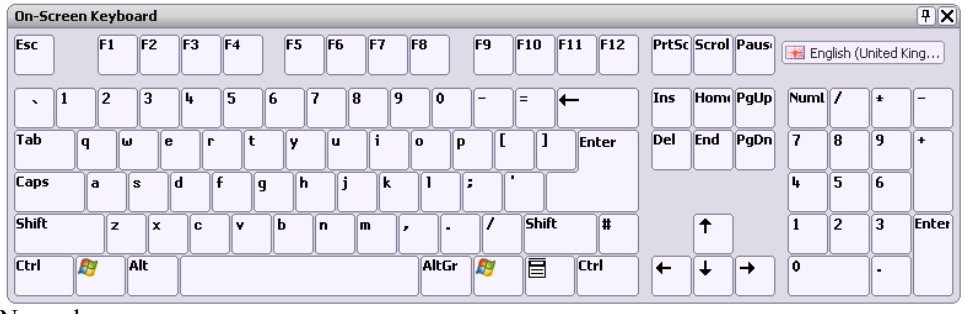
Included with Dyalog APL Version 12.1 is the Comfort On-Screen Keyboard 2.1 which is specially configured for use with Dyalog APL (Unicode Edition) and distributed under a licence agreement with Comfort Software.

The On-Screen keyboard is highly configurable and supports a wide range of visual effects including different colour schemes and transparency options.

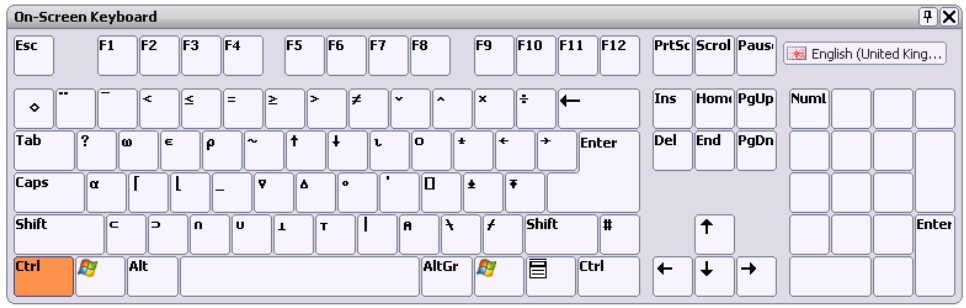
Not only does it support a large number of standard physical keyboards, but it includes a tool to design your own layout corresponding to the actual keyboard attached to your computer.

You may choose to have the On-Screen keyboard permanently shown or have it pop-up on a specific keystroke or when you press and hold Shift, Ctrl or Alt, and there is a corresponding variety of ways to have it disappear.

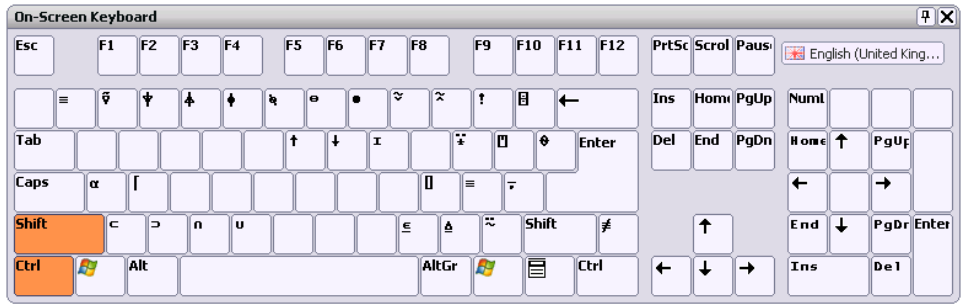
The following pictures illustrate the appearance of a UK - Dyalog Ctrl keyboard, in Normal, Ctrl and Ctrl+Shift modes.



Normal



Ctrl Mode



Ctrl+Shift

Classic Edition Keyboard

The standard Classic Edition Version 12.1 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`. The keyboard tables supplied with previous versions of Dyalog APL are distributed in the `old\aplkeys` sub-directory and may be used instead. Please refer to previous versions of this document.

Note that from Version 11.0 onwards, the standard tables do not support the entry of APL underscored characters ⍙ABCDEFGHIJKLMNOPQRSTUVWXYZ.

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.

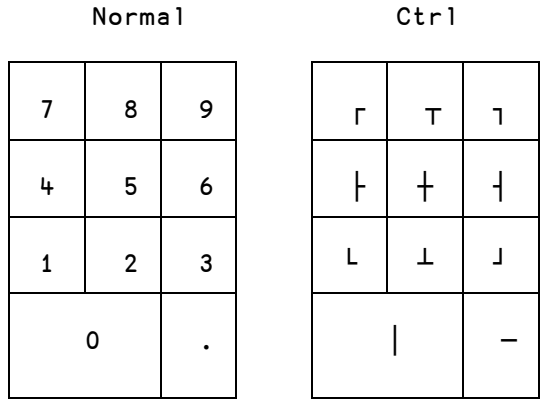
Traditional Layout

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**.



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.

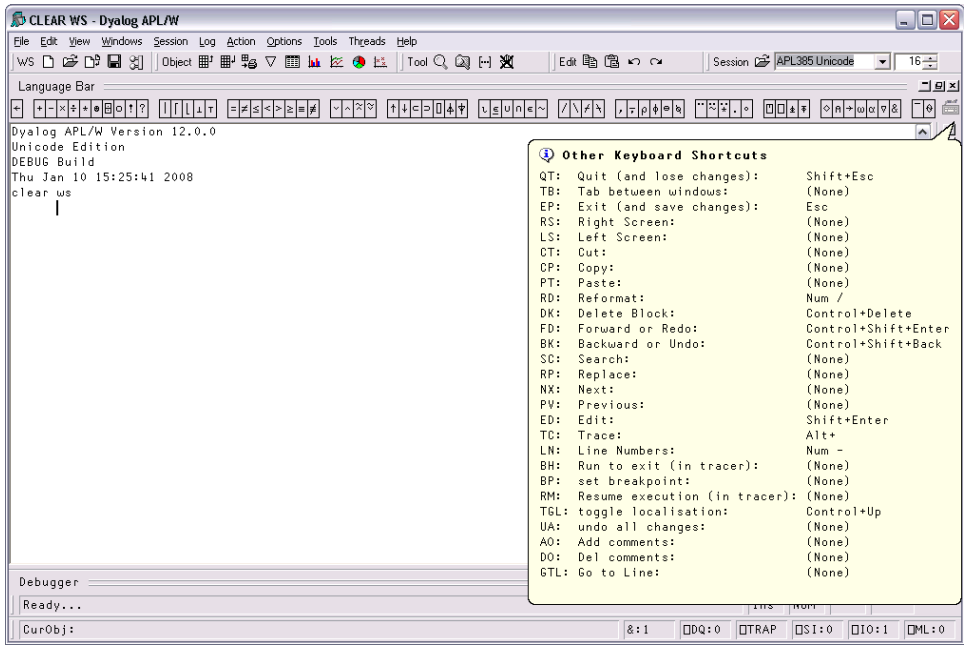
Unicode Edition

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.

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.

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

Cursor movement 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

Selection Commands

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

Editing Directives

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

Special Operations

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.

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

Default Colour Scheme - Session

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

Default Colour Scheme Edit windows

Syntax Colouring in the Session

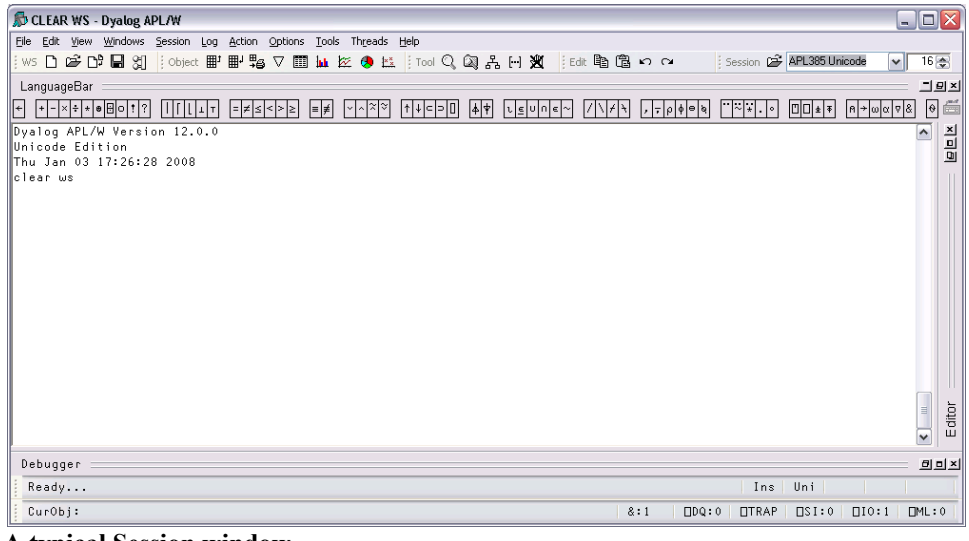
As an adjunct to the overall Session Colour Scheme, you may choose to apply a *syntax colouring scheme* to the current Session Input line(s). You may also extend syntax colouring to previously entered input lines, although this only applies to input lines in the current session; syntax colouring information 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 *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.

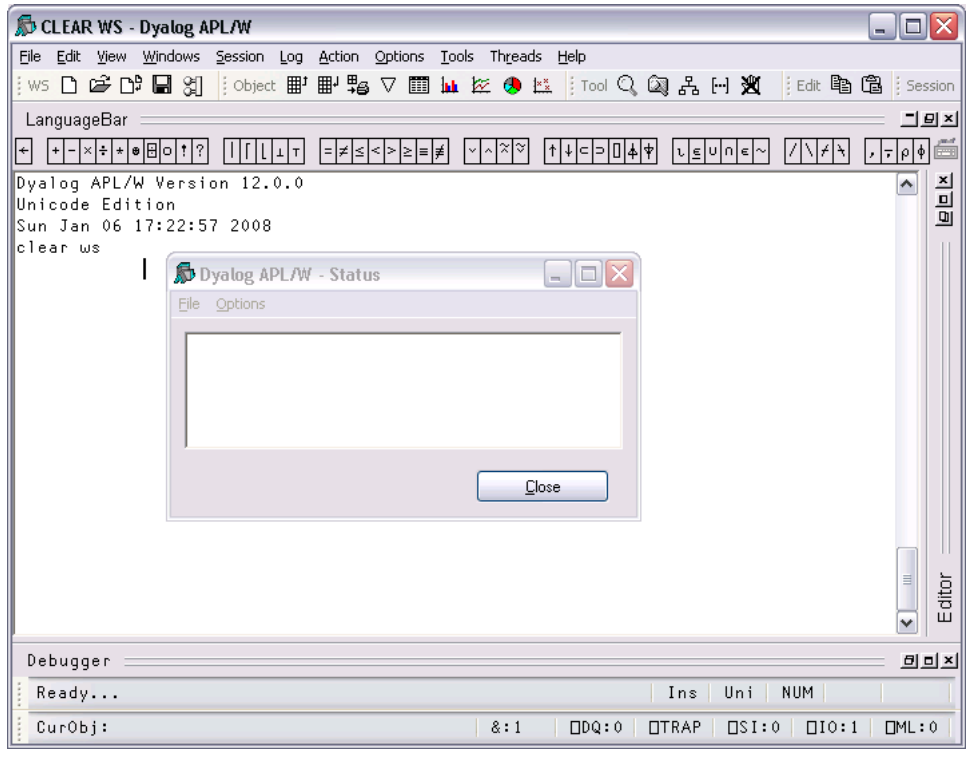
In addition to the Session Window itself, there are various subsidiary windows which are described later in the Chapter. In general, these subsidiary windows may be docked inside the Session window, or may be stand-alone floating windows. You may dock and undock these windows as required. The standard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

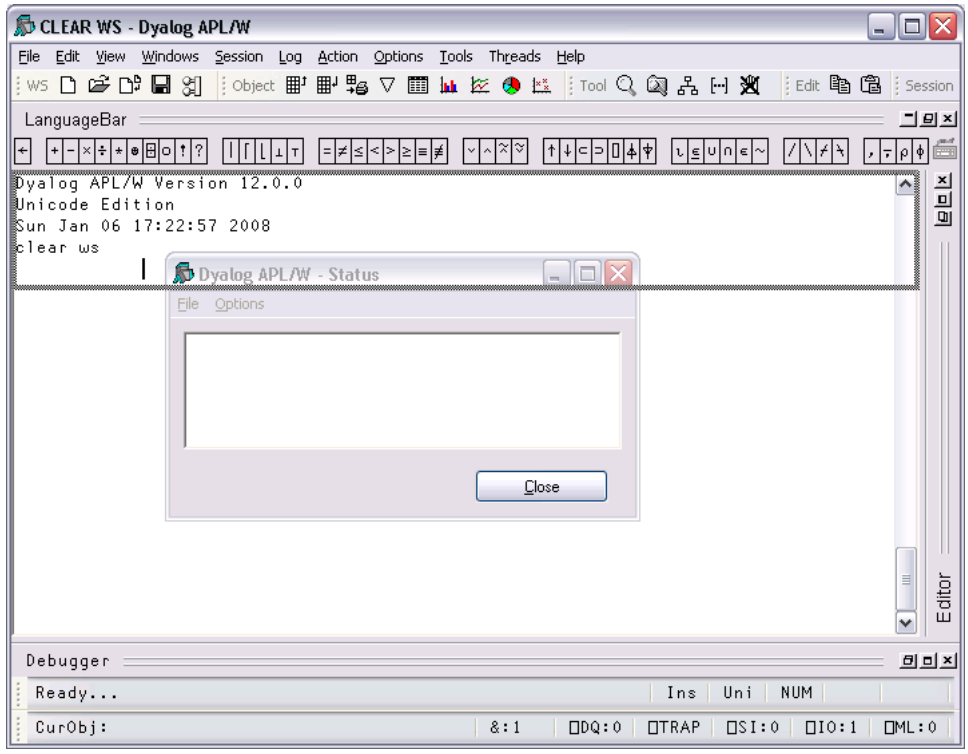
Docking

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.

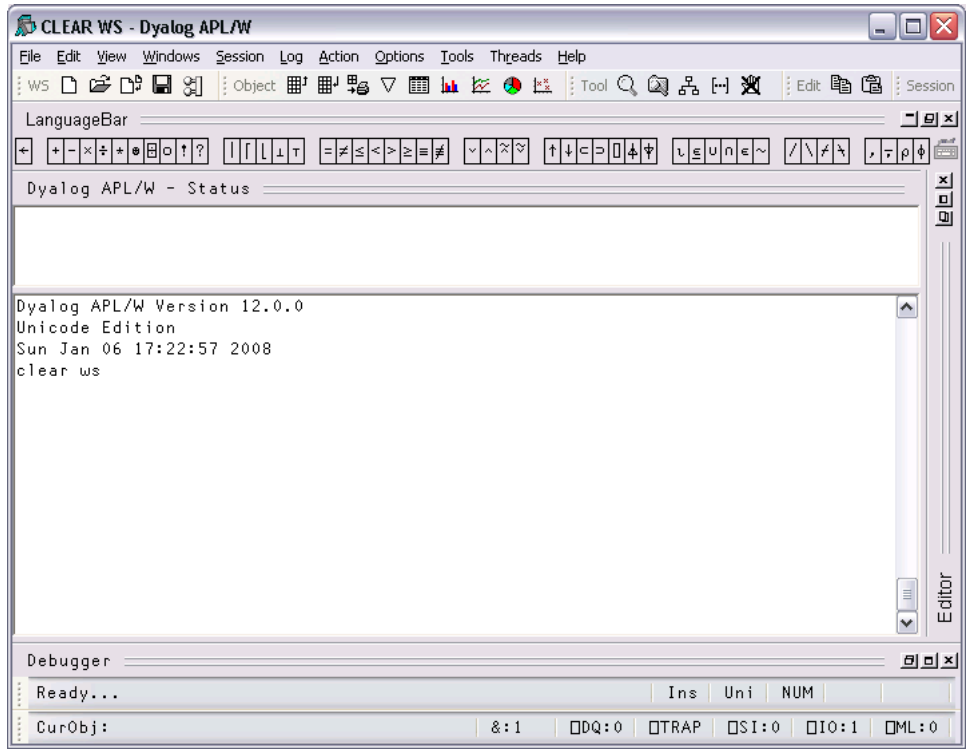


If you press the left mouse button down over the Status window title bar, and drag it, you will find that when the mouse pointer is close to an edge of the Session window, the drag rectangle indicates a docking zone as shown below. This indicates the space that the window will occupy if you now release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

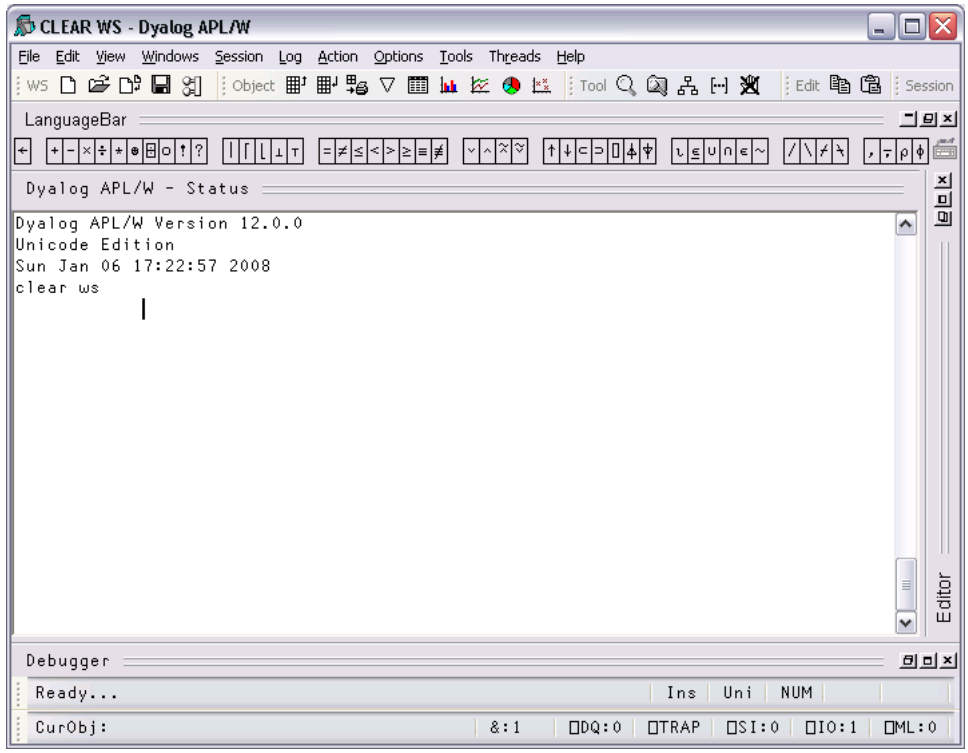


The next picture shows the result of the docking operation. The Session window is now split into 2 panes, with the Status window in the upper pane and the Session log window in the lower pane. You can resize the panes by dragging with the mouse.

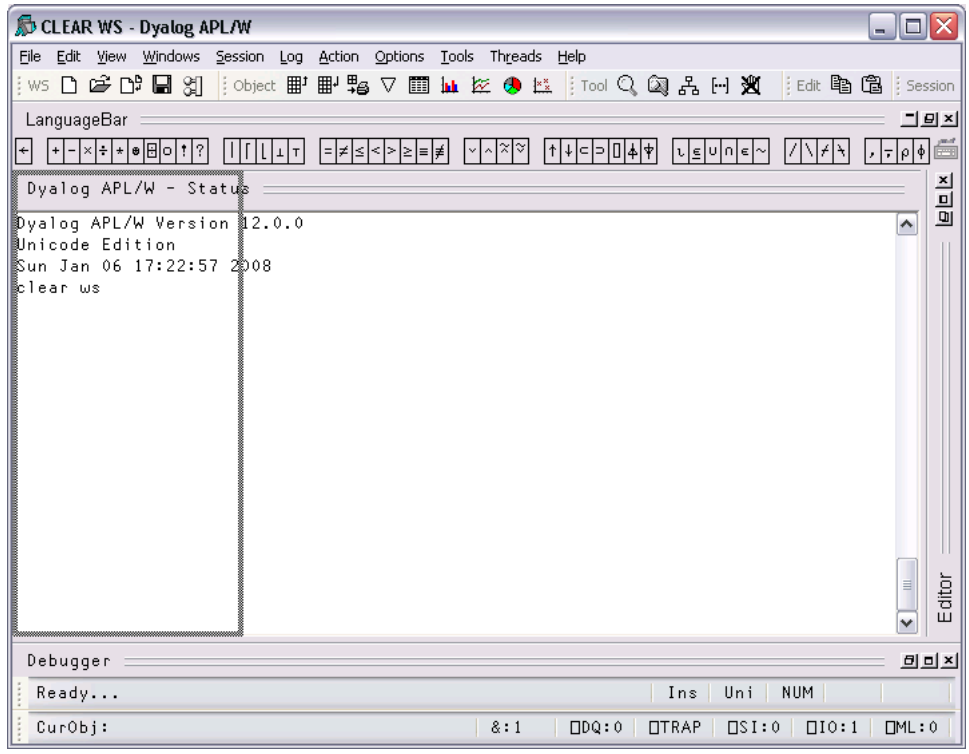
You will notice that a docked window has a title bar (in this case, the caption is *Status*) and 3 buttons which are used to *Minimise*, *Maximise* and *Close* the docked window.



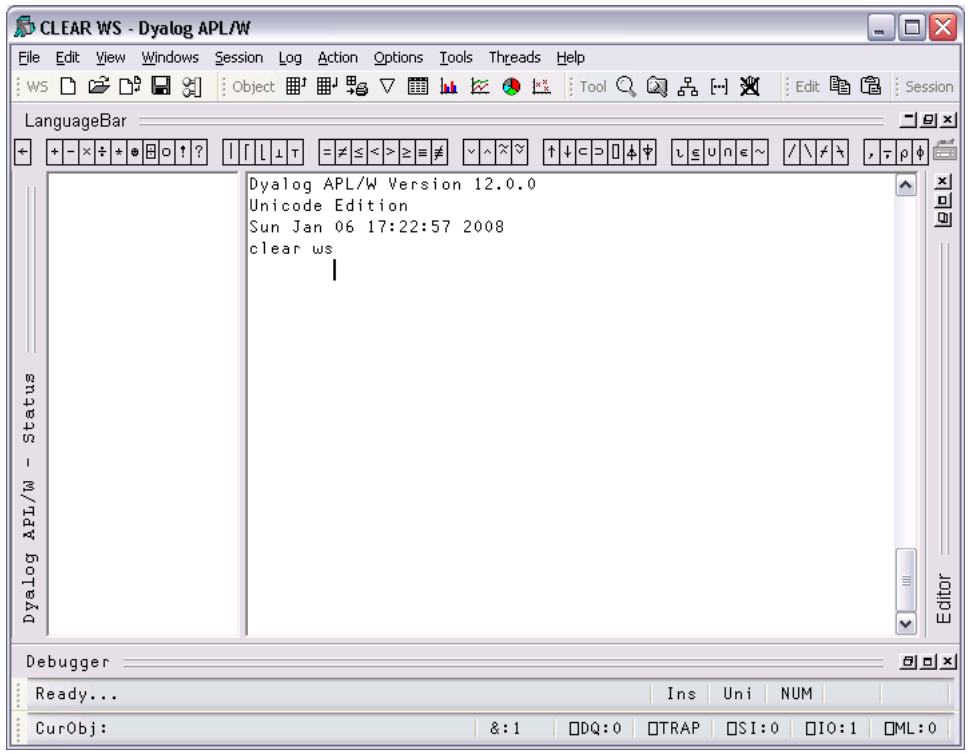
The next picture shows the result of minimising the Status window pane. All that remains of it is its title bar. The Minimise button has changed to a Restore button, which is used to restore the pane to its original size.



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.



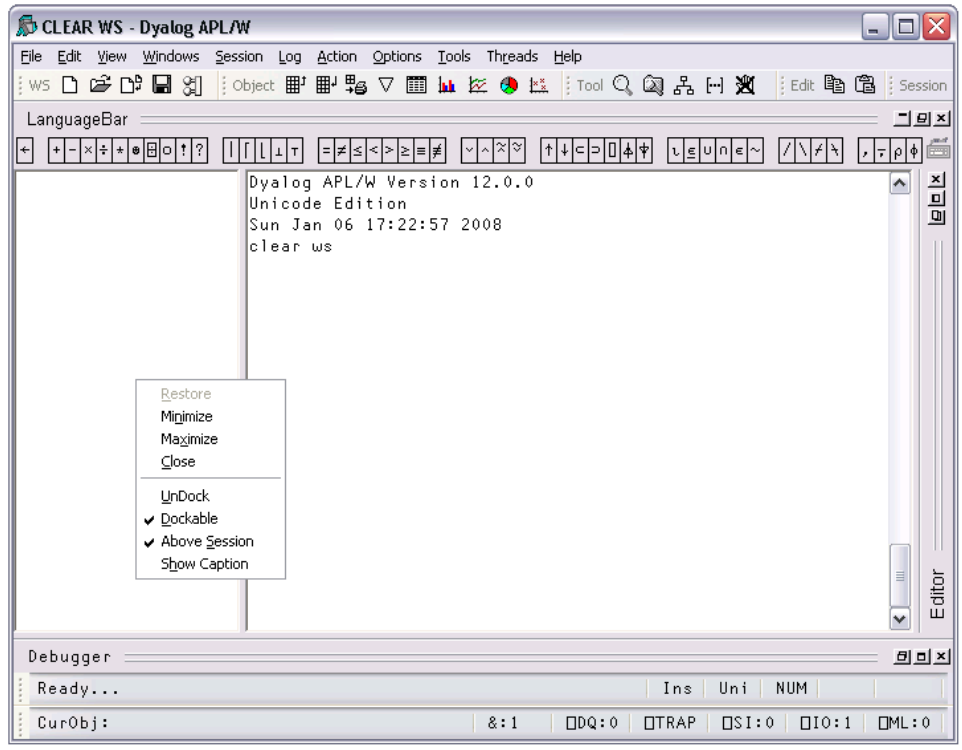
If you click the right mouse button over any window, its context menu is displayed. If the window is dockable, the context menu contains the following options:

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 file. When you close your APL session, the Session log is written back out to the log file, replacing its previous contents.

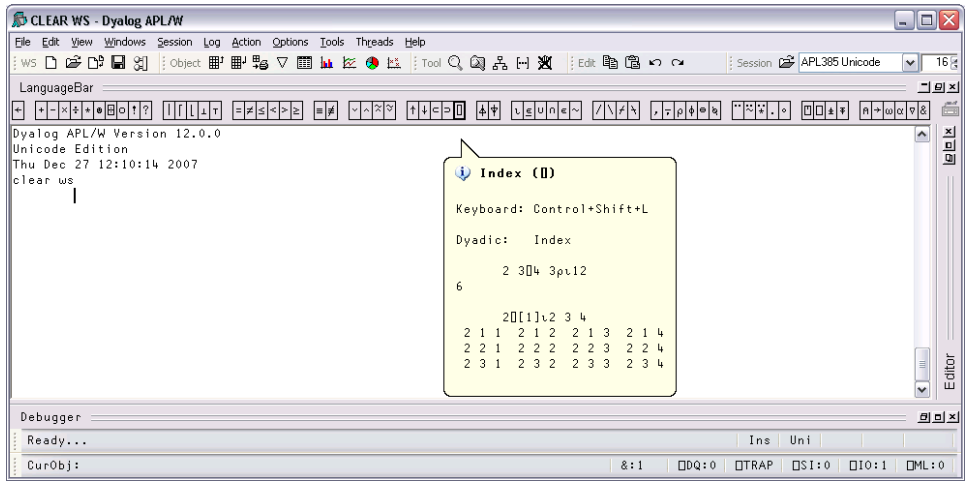
In general you type an expression into the input line, then press Enter (ER) to run it. After execution, the expression and any displayed results become part of the session log.

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:

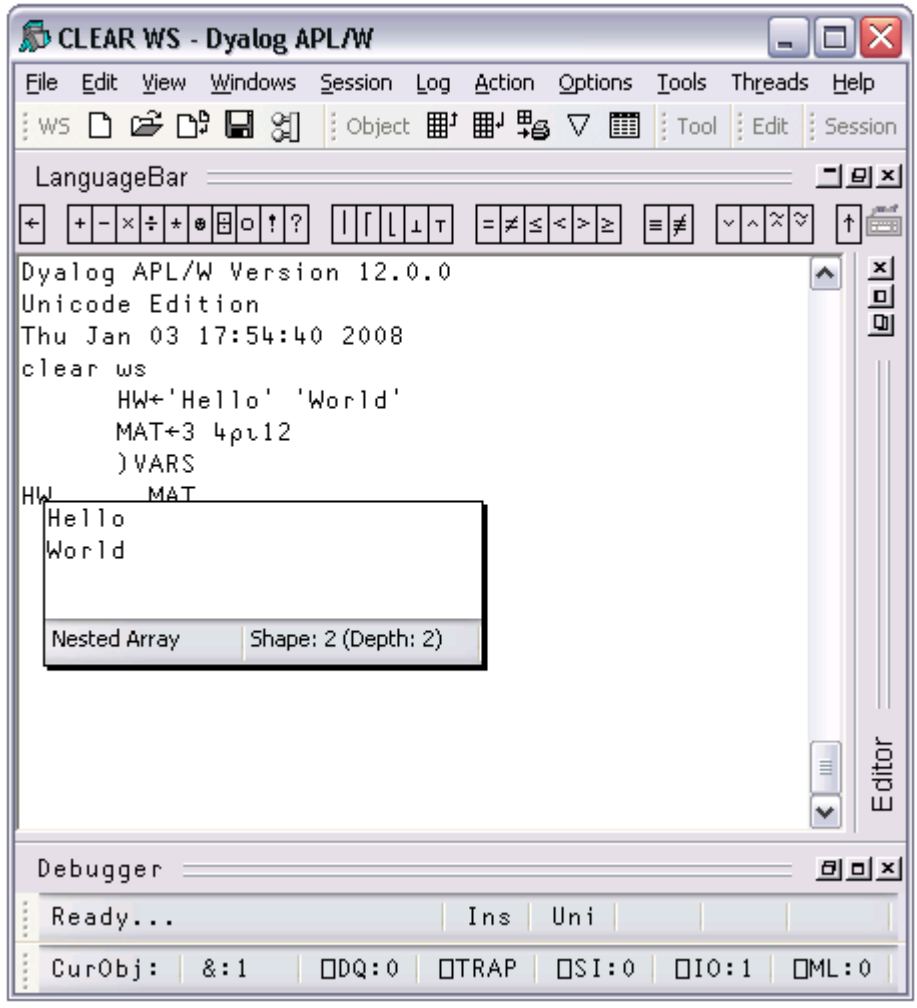
Enter `SALES` in the *Find* box, and `NEWSALES` in the *Replace* box. Now click the *Replace All* button. You will see all occurrences of `SALES` change to `NEWSALES`. Furthermore, each changed line in the session becomes marked (Red on White). Now click on the session and press Enter (or select *Execute*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

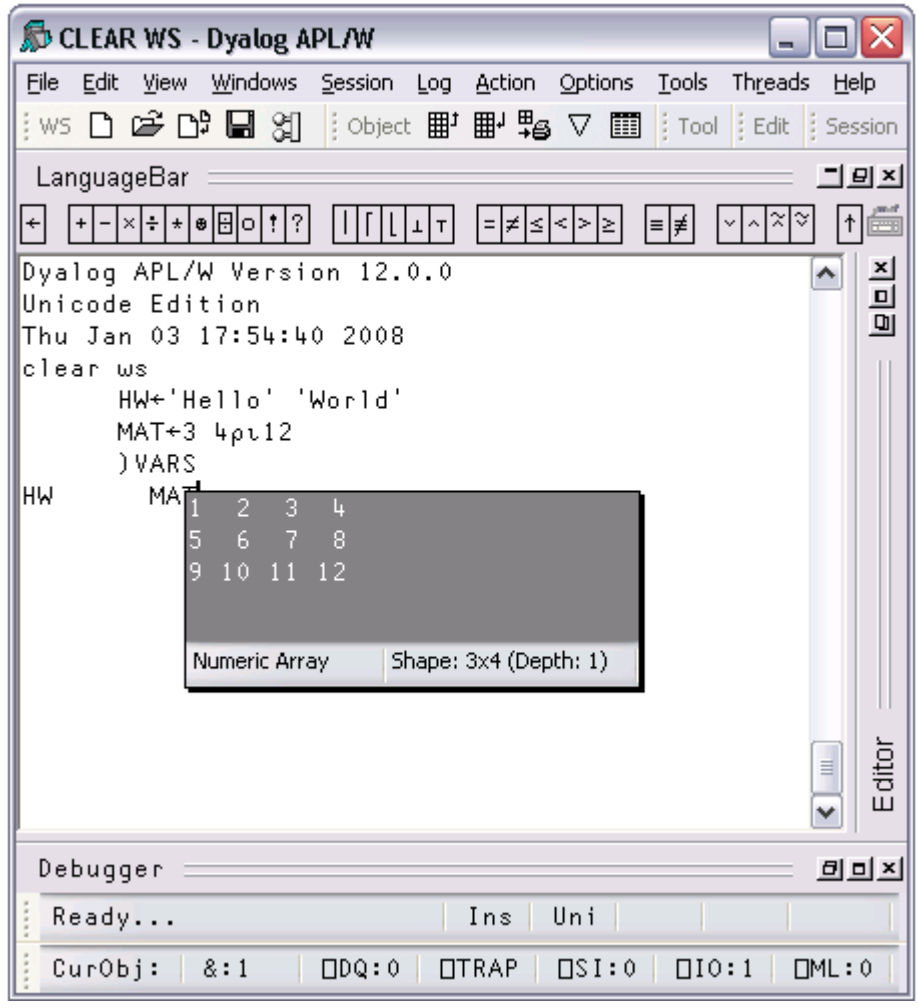
Value Tips

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.

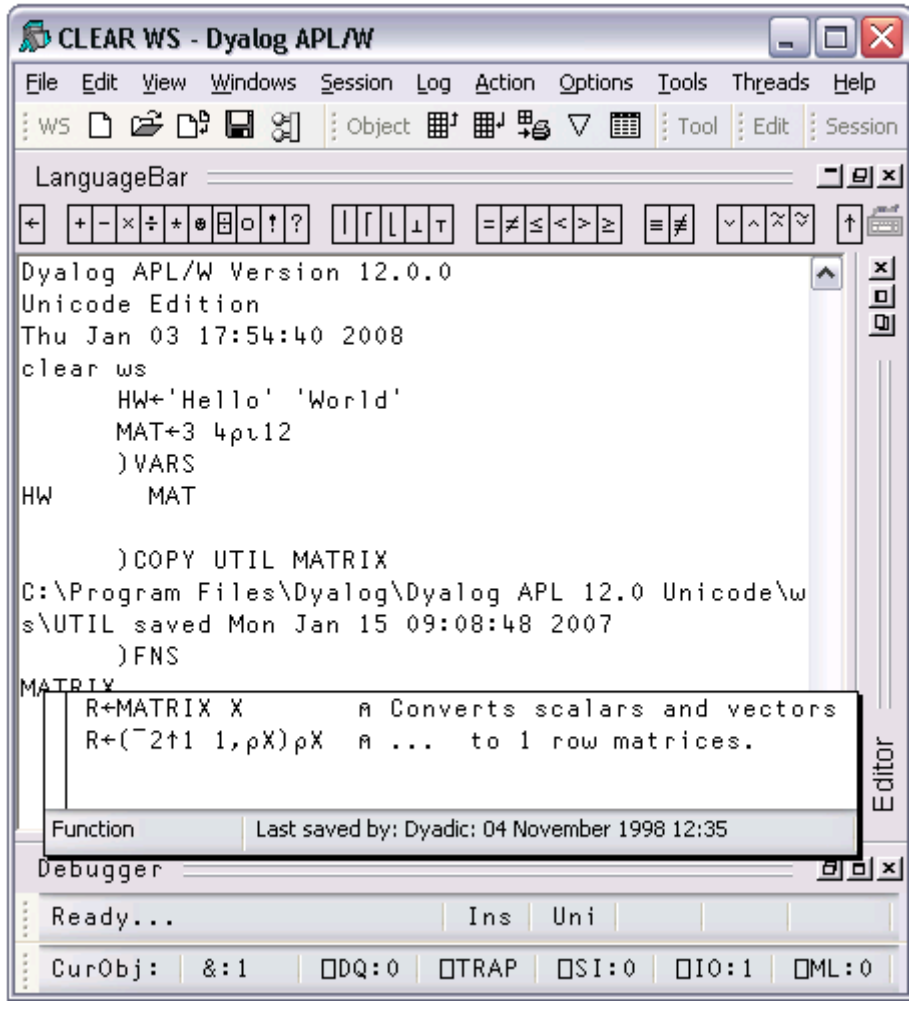
For example, in the following picture the mouse pointer was moved over the name of the variable `HW` in the Session window.



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.

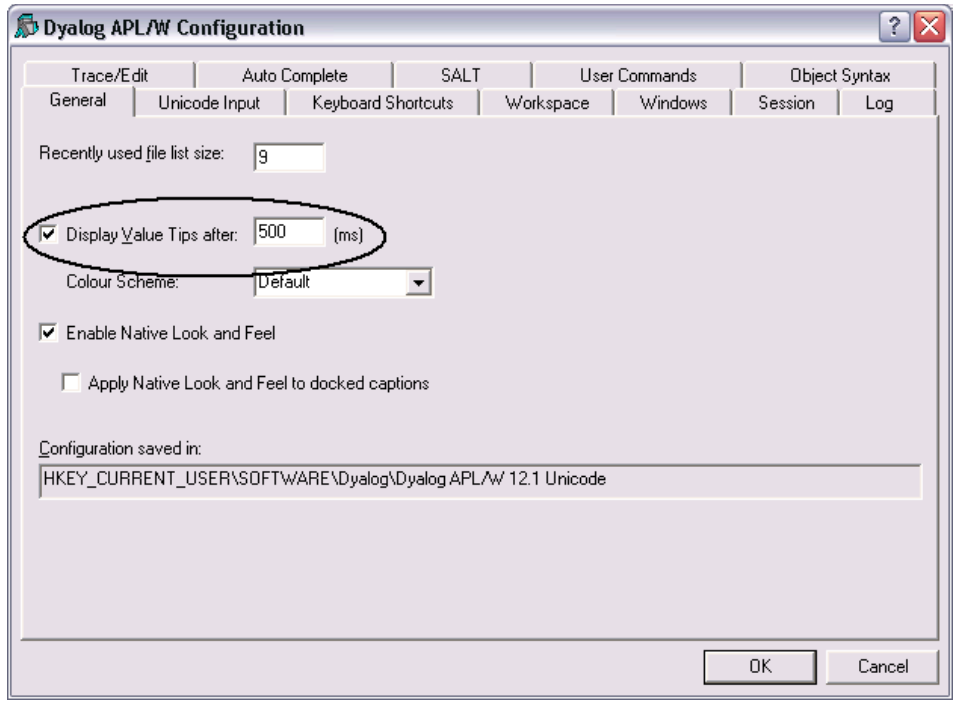


Configuring Value Tips

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.



SharpPlot Graphics

Introduction

Included with Version 12 (32-bit Windows versions only with the Microsoft .Net Framework Version 2.0 or later installed) is the SharpPlot graphics library which is part of the RainPro graphics package.

The Version 12.1 Session includes 4 buttons which use SharpPlot to generate simple 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 one of the 4 graphical buttons in the Session toolbar.

Data Structures

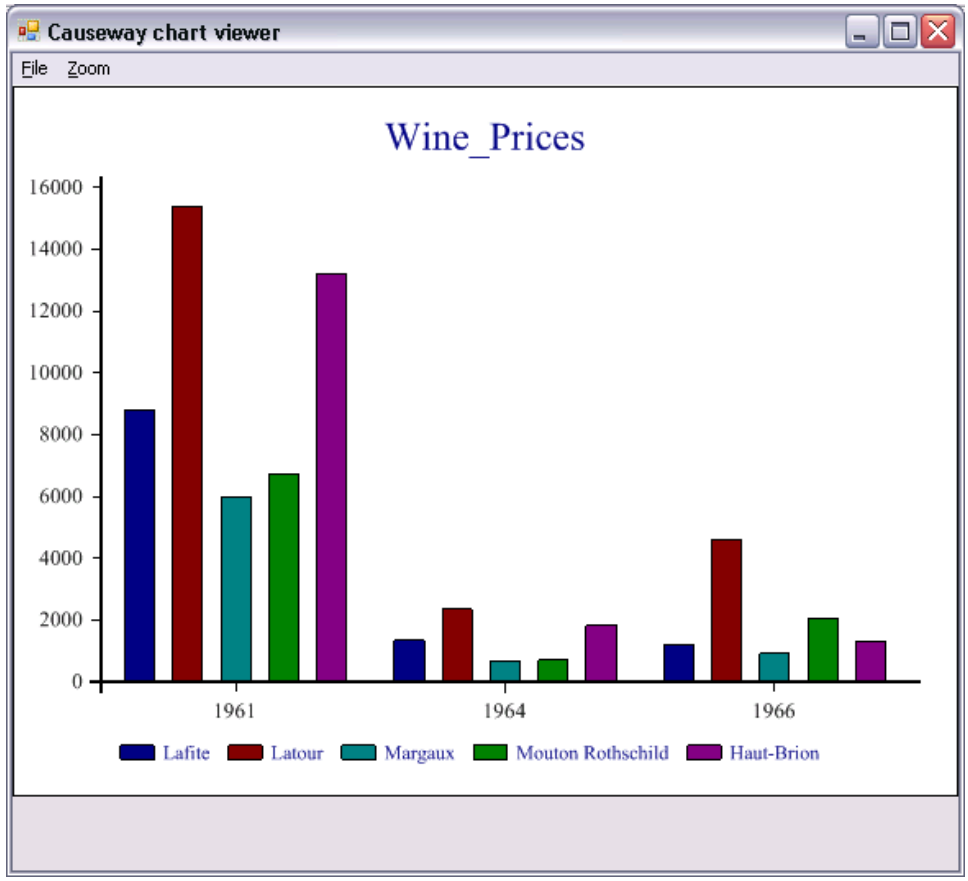
The charting function can plot variables with the following data structures:

- 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.

Examples

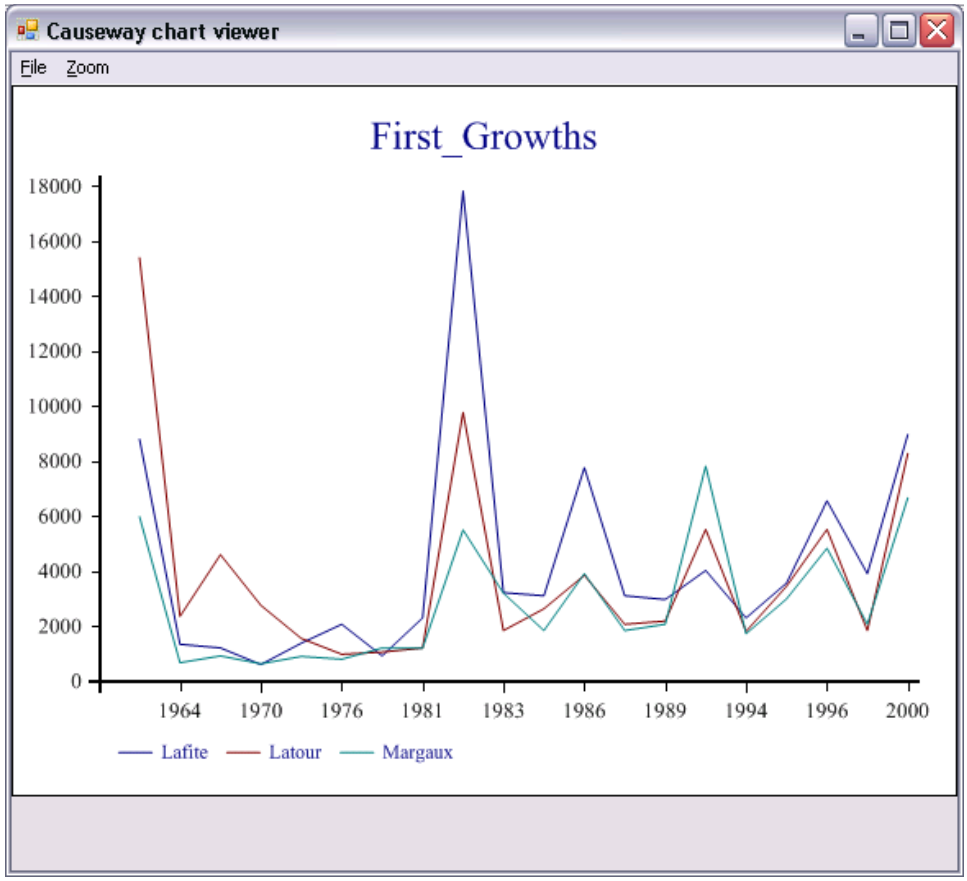
Bar Chart

```
      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
```



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 tools are implemented by four buttons in the Session toolbar. Each 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 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 `\bin\sharpplot.dll` in the Dyalog program directory.

Notes

For further information, please see http://www.sharpplot.com/Docs/default.aspx.

Although `⎕SE.Chart.MyChart` is overwritten by successive uses of the graphical buttons, 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 *System.IO.Path.GetTempFileName*.

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 variables 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.

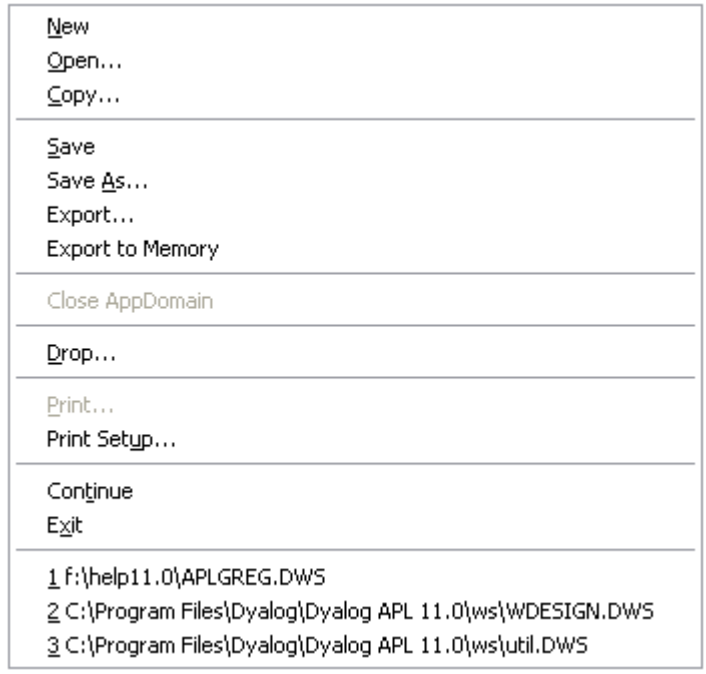
The statusbars in ⎕SE.cbbot, are implemented by two CoolBand objects , bandtb1 and bandtb2, each containing a StatusBar named sb.

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.



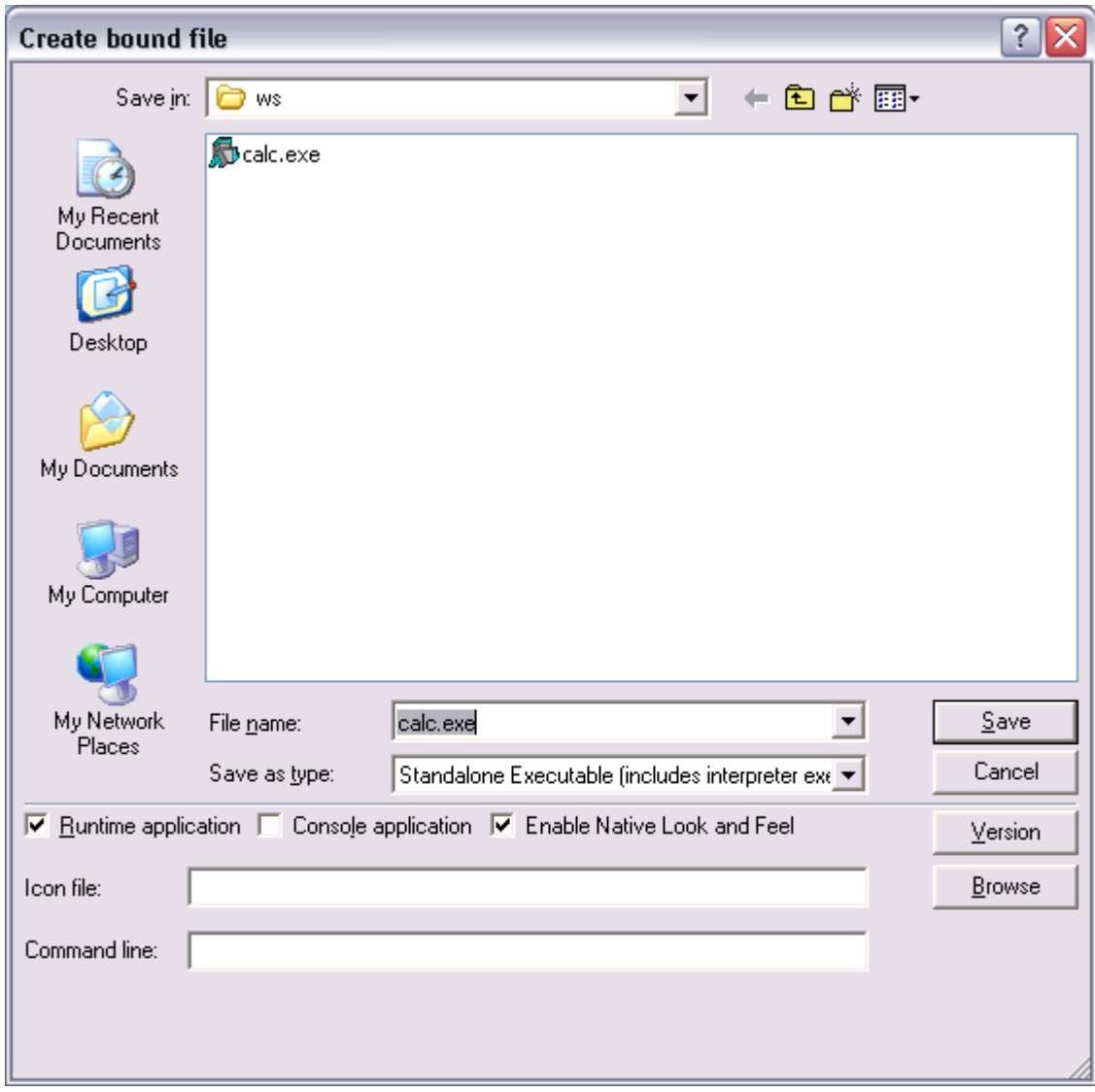
Item	Action	Description
New	`[WSClear]`	Prompts for confirmation, then clears the workspace
Open	`[WSLoad]`	Prompts for a workspace file name, then loads it
Copy	`[WSCopy]`	Prompts for a workspace file name, then copies it
Save	`[WSSave]`	Saves the active workspace
Save As	`[WSSaveas]`	Prompts for a 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 *in-memory* .Net Assembly
Drop	`[WSDrop]`	Prompts for a workspace file name, then erases it
Print Setup	`[PrintSetup]`	Invokes the print set-up dialog box
Continue	`[Continue]`	Saves the active workspace in CONTINUE.DWS and exits APL
Exit	`[Off]`	Prompts for confirmation, then exits APL

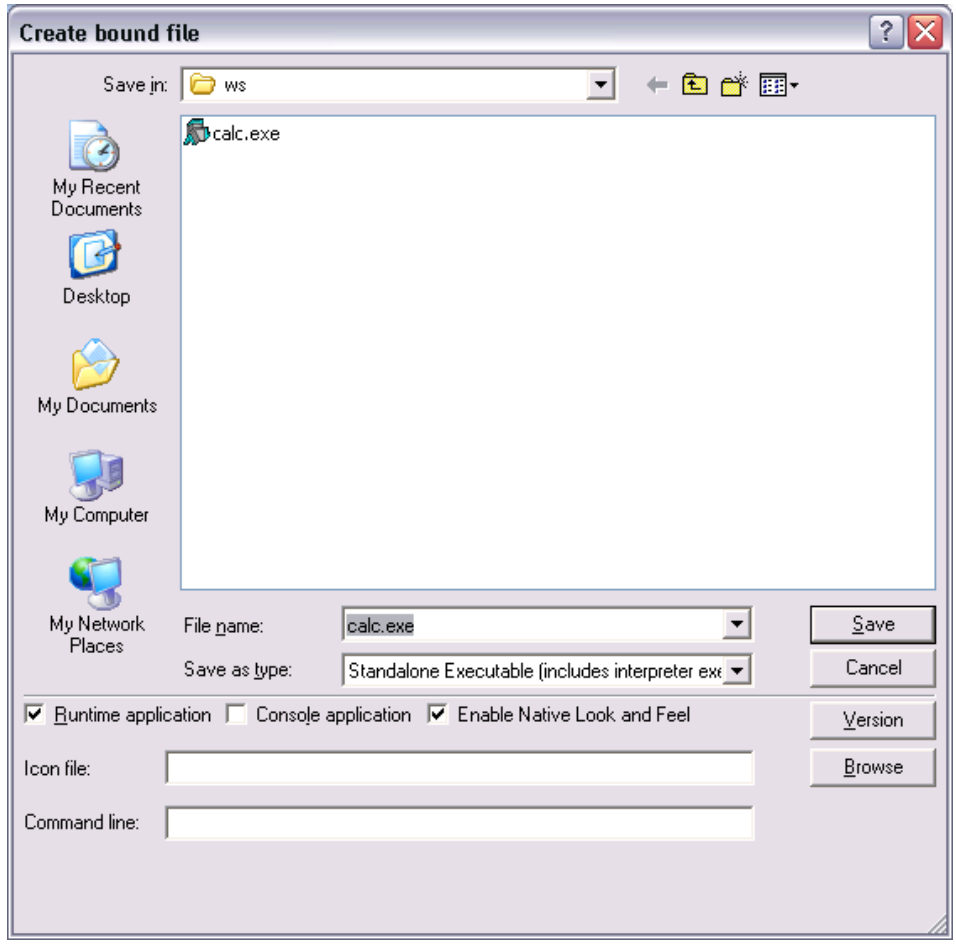
File Menu Operations

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.


Create bound file
Save in:
ws
calc.exe
My Recent Documents
Desktop
My Documents
My Computer
My Network Places
File name:
calc.exe
Save as type:
Standalone Executable (includes interpreter exe
Save
Cancel
Runtime application
Console application
Enable Native Look and Feel
Version
Icon file:
Browse
Command line:

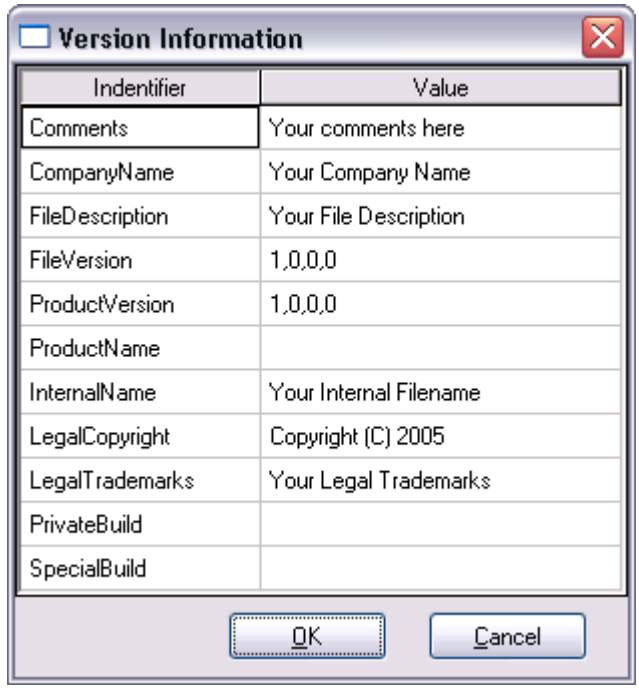

Create bound file
Save in:
ws
calc.exe
My Recent Documents
Desktop
My Documents
My Computer
My Network Places
File name:
calc.exe
Save as type:
Standalone Executable (includes interpreter exe
Save
Cancel
Runtime application
Console application
Enable Native Look and Feel
Version
Icon file:
Browse
Command line:

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 XP Look and Feel	If checked, XP 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.

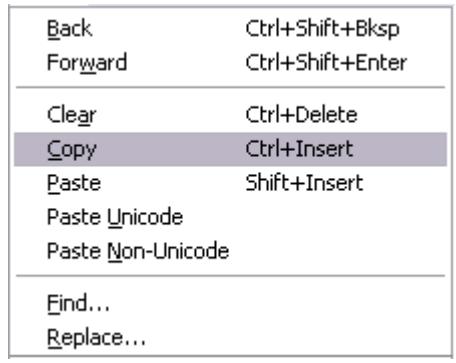
Pressing the Version button brings up the *Version Information* dialog box shown below.

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**

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

Edit menu operations

The View Menu

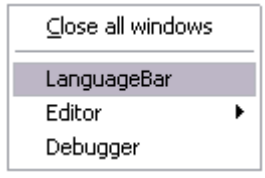
The View menu (`⎕SE.mb.view`) toggles the visibility of the Session Toolbar, StatusBar, and Language Bar.

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

View menu operations

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.



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

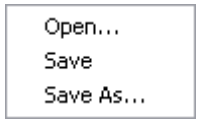
Window menu operations

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

In addition, the Windows 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.



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`.

Session menu operations

The Log Menu

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

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.

Log menu operations

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

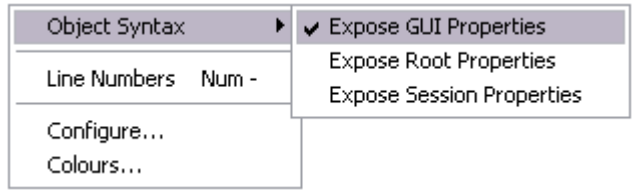
Classic Edition

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`

Action menu operations

The Options Menu

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



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

Options menu operations

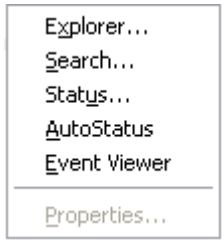
The values associated with the *Expose GUI*, *Expose Root* and *Expose Session* options reflect the values of these settings in your current workspace and are saved in it.

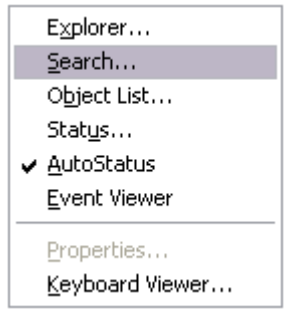
When you change these values through the Options menu, you are changing them in the current workspace only.

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.





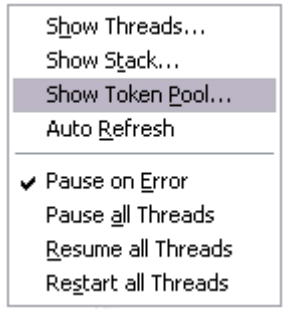
Unicode Edition **Classic Edition**

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
Keyboard Viewer	`N/A`	Displays the APLTeam Keyboard Viewer. **Classic Edition only**.

Tools Menu Operations

The Threads Menu

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



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	`[ThreadsResrartAll]`	Restarts all threads

Threads Menu Operations

The Help Menu

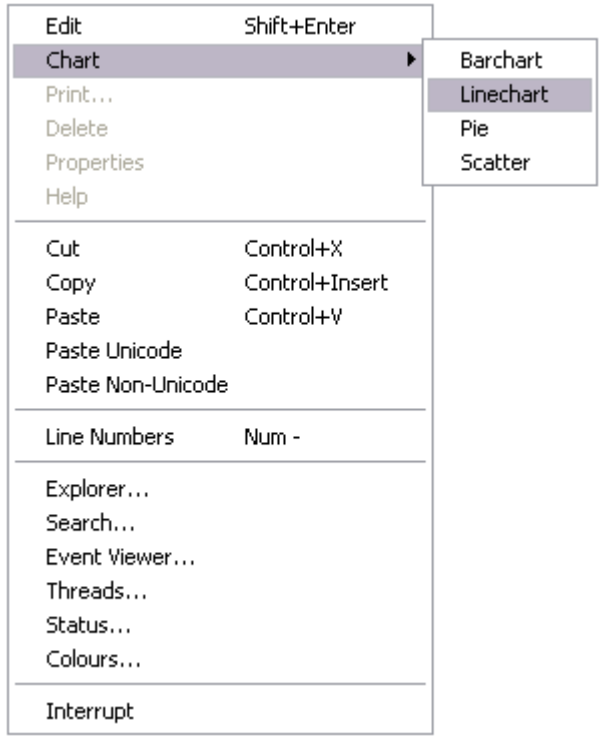
The Help menu (⎕SE.mb.help) provides access to the help system which is packaged as a single *Microsoft HTML Help* compiled help file named help\dyalog.chm.

Label	Action	Description
Documentation Center	[Decanter]	Opens your web browser on help\index.html which displays an index to the on-line PDF documentation and selected internet links.
Latest Enhancements	[Reunites]	Opens help\dyalog.chm, starting at the first topic in the Version 12.1 Release Notes section. Note that the Version 11.0 Release Notes are also included for your convenience.
Language Help	[Lang Help]	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.
Dyalog Web Site	[DyalogWeb]	Opens your web browser on the Dyalog home page.
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

Help menu operations

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.

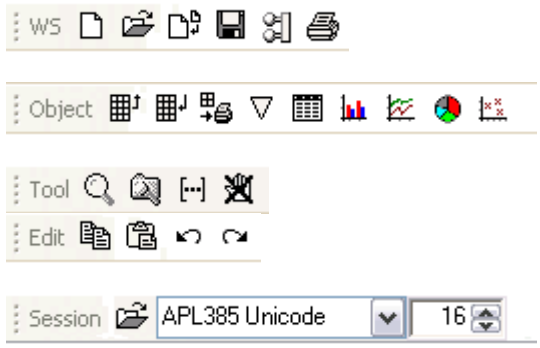


Item	Action	Description
Edit	`[Edit]`	Edits the current object
Print	`[ObjPrint]`	Prints the current object
Delete	`[ObjDelete]`	Erases the current object
Properties	`[GUIHelp]`	Displays the Object Properties dialog box for the current object
Help	`[Help]`	Displays the help topic associated with the current object or the APL symbol under the cursor
Line Numbers	`[LineNumbers]`	Toggles line numbers on/off
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`
Explorer	`[Explorer]`	Displays the Workspace Explorer
Search	`[WSSearch]`	Displays the Find Objects tool
Event Viewer	`[EventViewer]`	Displays the Event Viewer
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

Session popup menu operations

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'(192 192 192))
   'ili'⎕WC'ImageList'('MapCols' 0)('Masked' 1)
   'ili.bm'⎕WC'Bitmap'('' 'tb_inactive')('MaskCol'(192 192 192))
: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 Numbers

Executes a defined function in `⎕SE` that edits the current object (which must be numeric) using a spreadsheet like interface based upon the Grid object.

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.

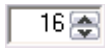
Session Operations

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.

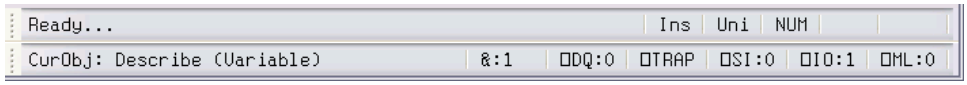


Select Font Size

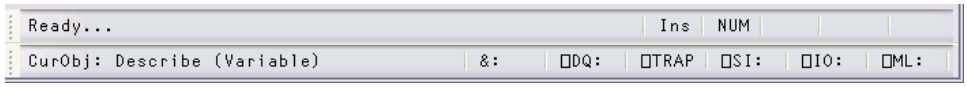
Selects the size of the font to be used in the Session window.

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.

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

Session status fields : first 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

Session status fields : second row

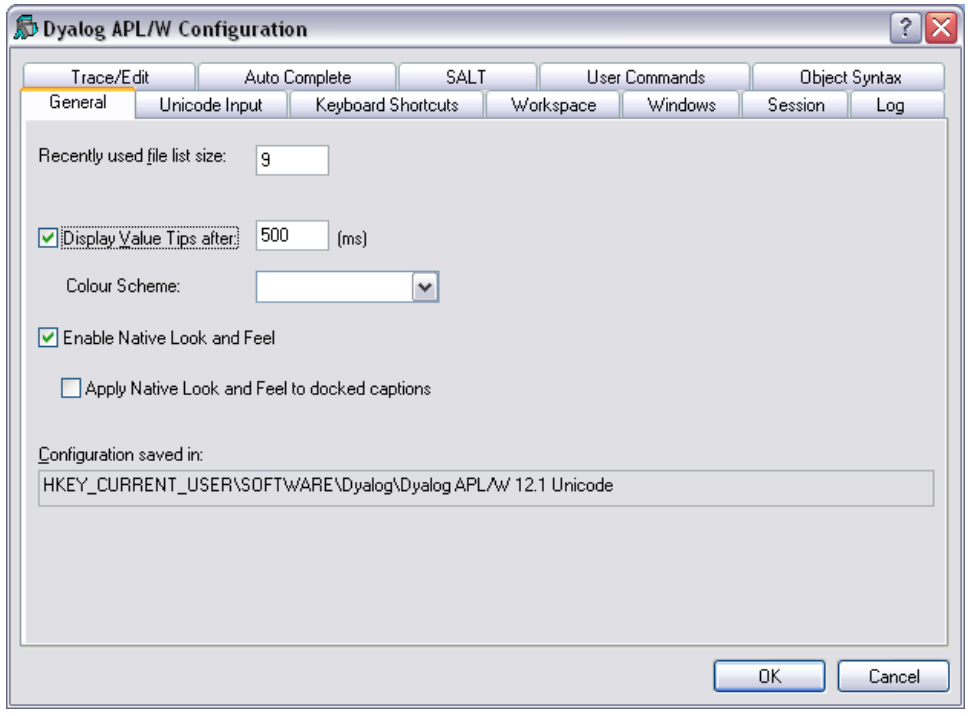
Toggle Status Fields

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.

The Configuration Dialog Box

General Tab



Label	Parameter	Description
Show line numbers	lines_on_functions	Determines whether or not line numbers are shown in edit/trace windows
Recently used file list size	file_stack_size	Specifies the number of the most recently used workspaces displayed in the File menu.
Display Value Tips after	ValueTips/Delay	Specifies the delay before APL will display the value of a variable or the code for a function when the user hovers the mouse over its name.
Colour Scheme	ValueTips/ ColourScheme	Specifies the colour scheme used to display the value of a variable or the code for a function when the user hovers the mouse over its name.
Enable Native Look and Feel	XPLookAndFeel	See below.
Apply Native Look and Feel to docked captions	XPLookAndFeelDocker	Specifies whether or not XP Look and Feel is honoured when drawing the title bars of docked windows, including docked Session windows.
Configuration saved in	inifile	Specifies the full pathname of the registry folder used by APL

Configuration dialog: General

Native Look and Feel

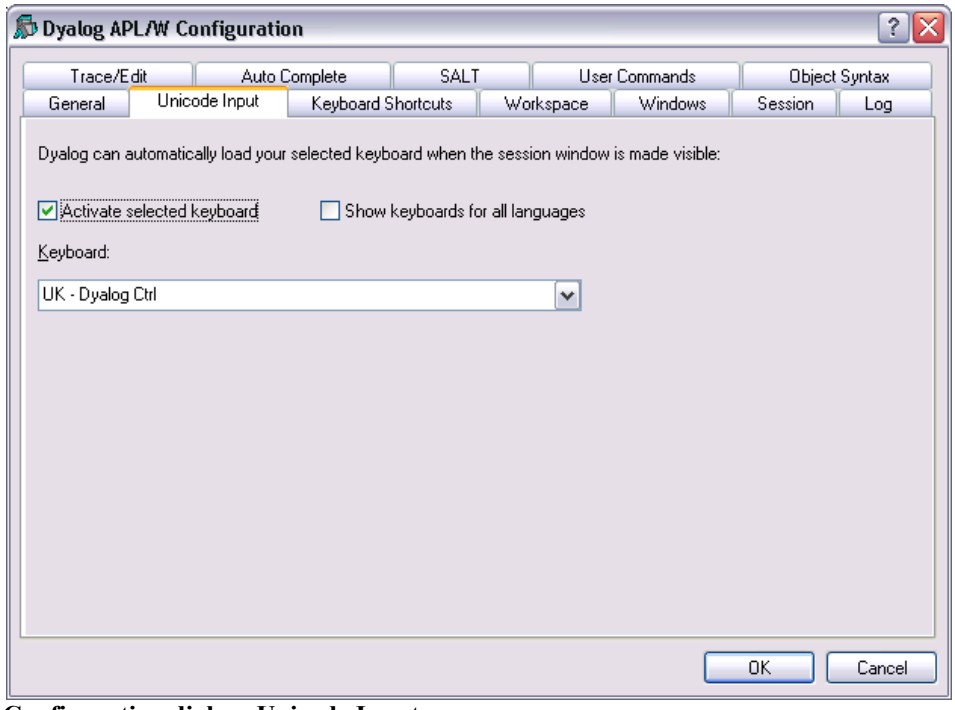
If you check the *Apply Native Look and Feel* option box and close the *Options* dialog by pressing *OK*, APL creates a MANIFEST file. This is a file with the same name as the Dyalog executable program (normally, `dyalog.exe`) with the addition of a `.manifest` suffix (normally, `dyalog.exe.manifest`). If you clear the option box and click *OK*, the manifest file is deleted.

The presence or absence of this file determines whether or not Native (XP) Look and Feel is used for Session windows.

Unicode Input Tab (Unicode Edition Only)

Unicode Edition can optionally select your APL keyboard each time you start APL.

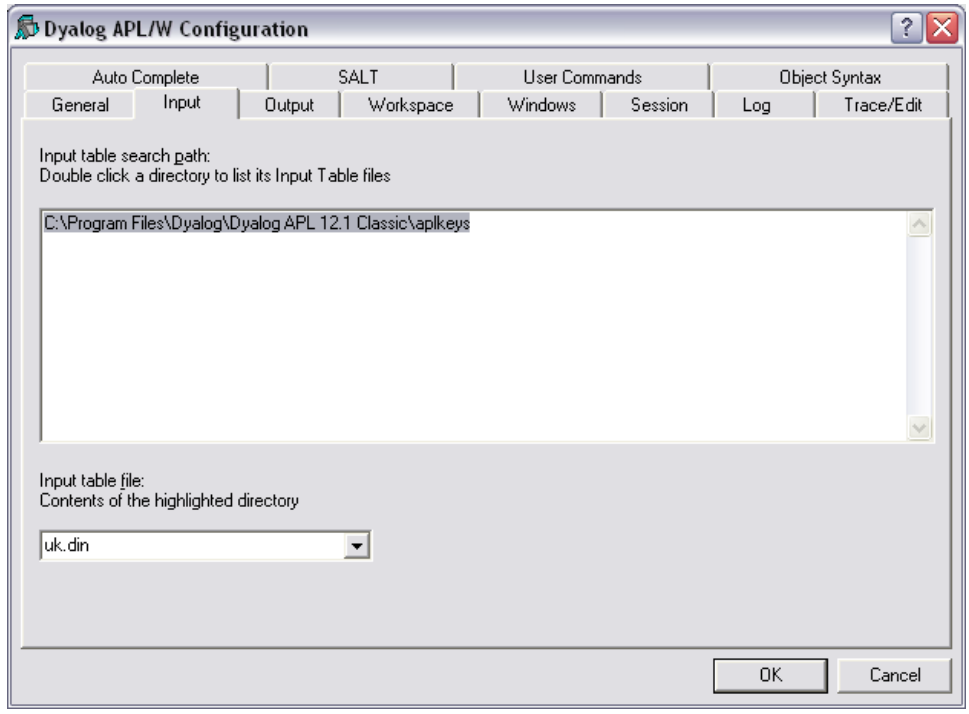
To choose this option, select one of your installed APL keyboards, enable the *Activate selected keyboard* checkbox, then click *OK*



Configuration dialog: Unicode Input

Label	Parameter	Description
Activate selected keyboard	InitialKeyboardLayoutInUse	1 = automatically select the specified APL keyboard on start-up. 0 = no action
Show keyboards for all Languages	InitialKeyboardLayoutShow All	1 = show list of all installed keyboards 0 = show only the Dyalog keyboards
Keyboard	InitialKeyboardLayout	the name of the APL keyboard to be selected.

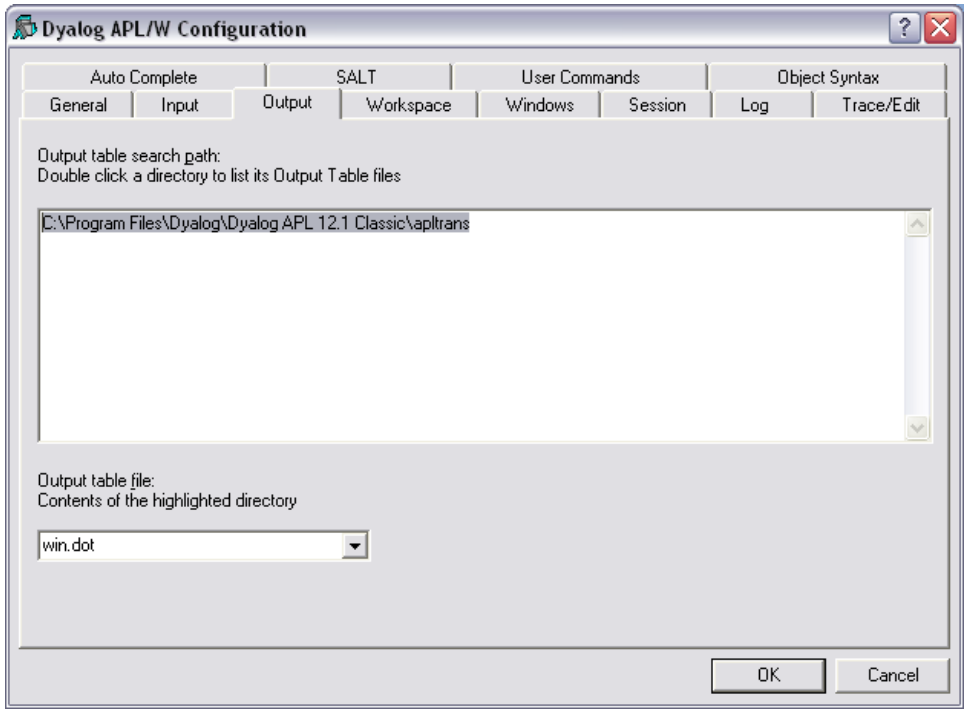
Input Tab (Classic Edition Only)



Label	Parameter	Description
Input table search path	aplkeys	A list of directories to be searched for the specified input table
Input table file	aplk	The name of the input table file (.DIN)

Configuration dialog: Keyboard

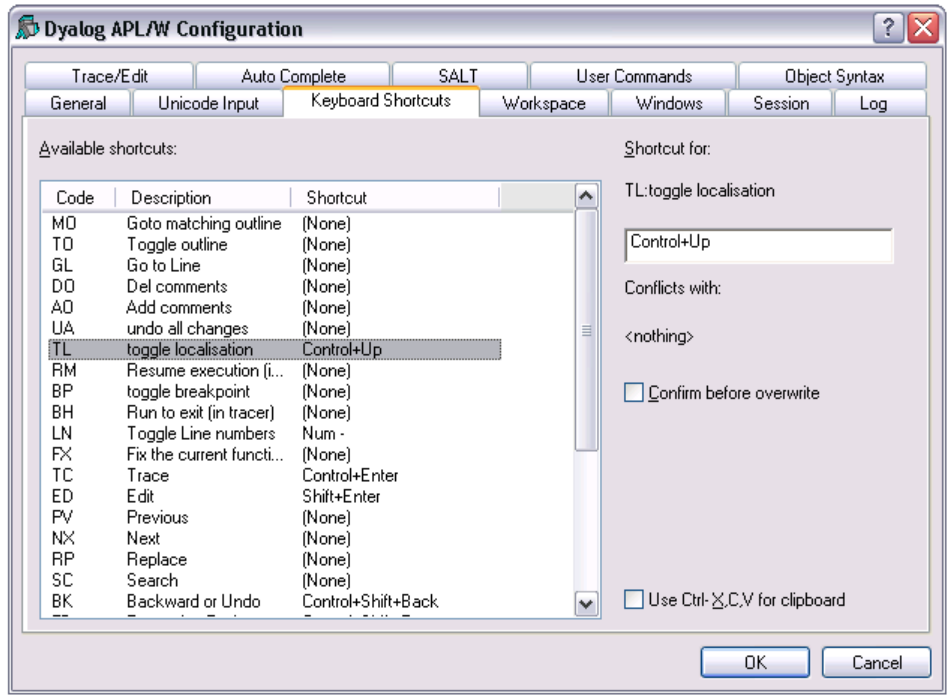
Output Tab (Classic Edition Only)



Label	Parameter	Description
Output table search path	apltrans	A list of directories to be searched for the specified output table
Output table file	aplt	The name of the output table file (.DOT)

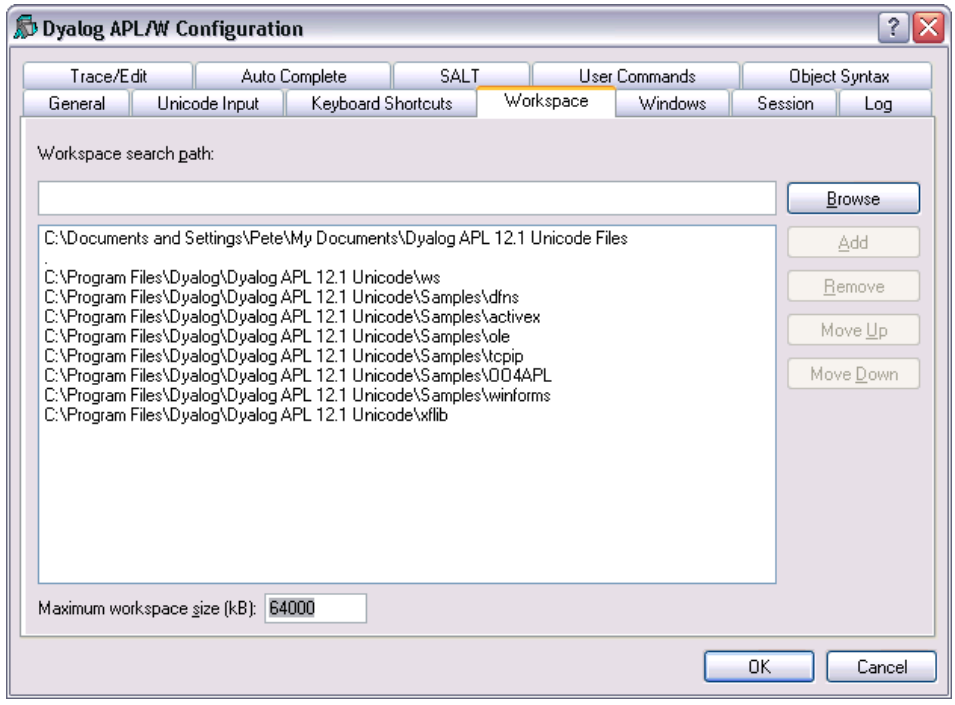
Configuration dialog: Output

Keyboard Shortcuts Tab



To alter the keystroke associated with a particular action, simply select the action required and press the keystroke. For example, to change the keystroke associated with the action <UA> (undo all changes) from (None) to Ctrl+Shift+u, simply select the corresponding row in the list and press Ctrl+Shift+u. If *Confirm before Overwrite* is checked, you will be prompted to confirm or cancel before each and every change is written back to the registry.

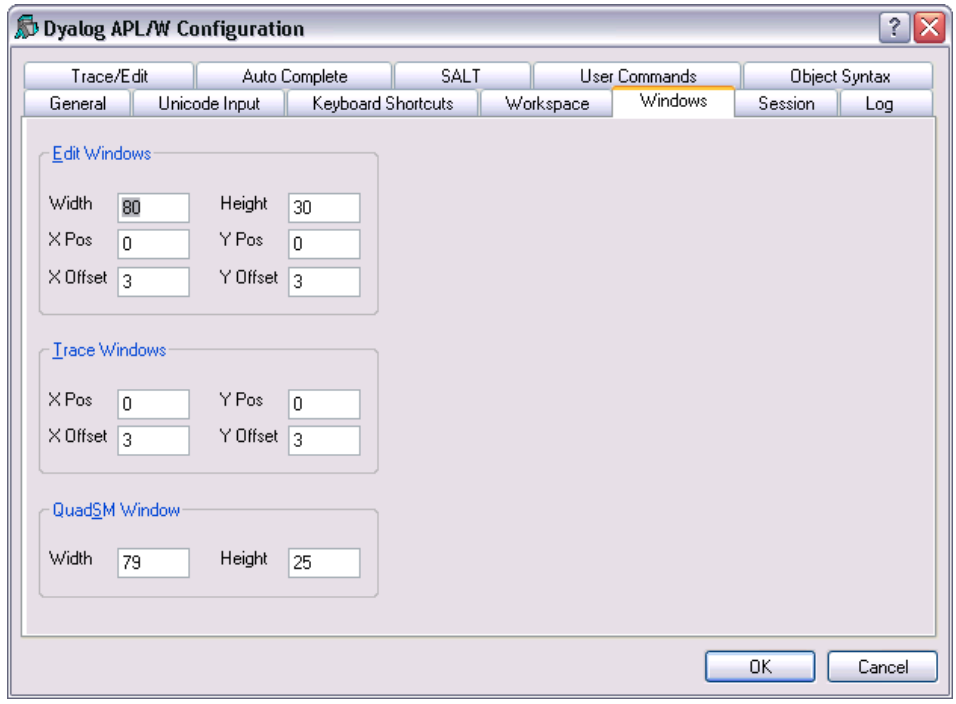
Workspace Tab



Label	Parameter	Description
Workspace search path	wspath	A list of directories to be searched for the specified workspace when the user executes `)LOAD wsname`
Maximum workspace size(kB)	maxws	The maximum size of the workspace in KB. Default is 16384.

Configuration dialog: Workspace

Windows Tab


Dyalog APL/W Configuration
Trace/Edit
Auto Complete
SALT
User Commands
Object Syntax
General
Unicode Input
Keyboard Shortcuts
Workspace
Windows
Session
Log
Edit Windows
Width
80
Height
30
X Pos
0
Y Pos
0
X Offset
3
Y Offset
3
Trace Windows
X Pos
0
Y Pos
0
X Offset
3
Y Offset
3
QuadSM Window
Width
79
Height
25
OK
Cancel

Label	Parameter	Description
Width	edit_cols	The maximum number of rows displayed in a new edit window
Height	edit_rows	The maximum number of columns displayed in a new edit window
X Pos	edit_first_x	The initial horizontal position in characters of the first edit window relative to the Session window
Y Pos	edit_first_y	The initial vertical position in characters of the first edit window relative to the Session window
X Offset	edit_offset_x	The initial horizontal position in characters of the second and subsequent edit windows relative to the previous one
Y Offset	edit_offset_y	The initial vertical position in characters of the second and subsequent edit windows relative to the previous one

Configuration dialog: Windows (Edit Windows)

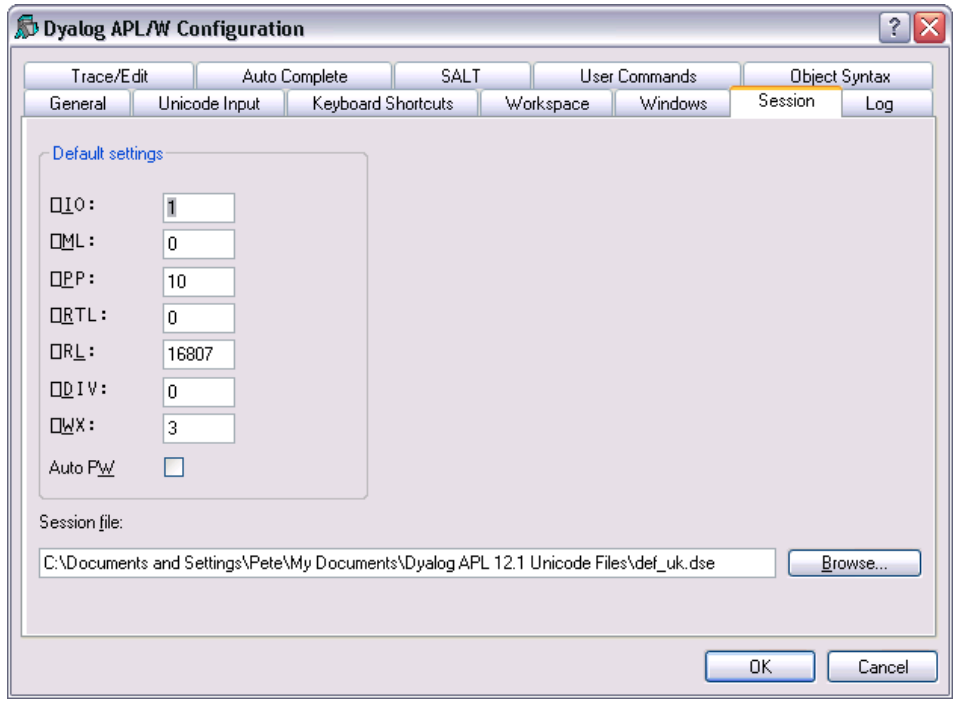
Label	Parameter	Description
X Pos	trace_first_x	The initial horizontal position in characters of the first trace window relative to the Session window
Y Pos	trace_first_y	The initial vertical position in characters of the first trace window relative to the Session window
X Offset	trace_offset_x	The initial horizontal position in characters of the second and subsequent trace windows relative to the previous one
Y Offset	trace_offset_y	The initial vertical position in characters of the second and subsequent trace windows relative to the previous one

Configuration dialog: Windows (Trace Windows)

Label	Parameter	Description
Width	sm_cols	The width of the ⎕SM and prefect windows
Height	sm_rows	The height of the ⎕SM and prefect windows

Configuration dialog: Windows (QuadSM Window)

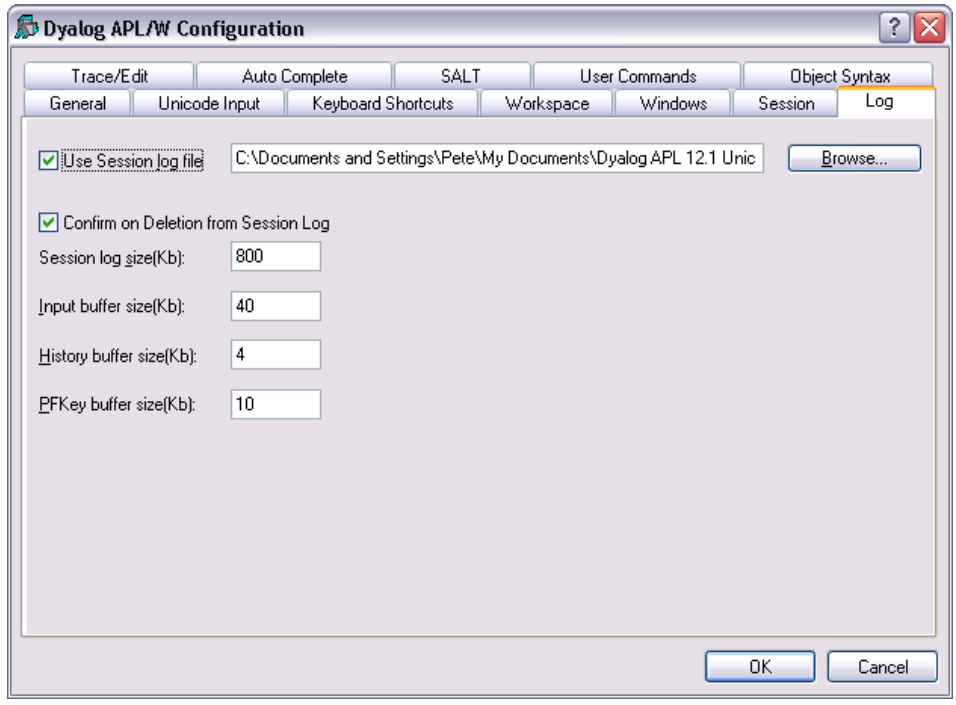
Session Tab


Dyalog APL/W Configuration
Trace/Edit
Auto Complete
SALT
User Commands
Object Syntax
General
Unicode Input
Keyboard Shortcuts
Workspace
Windows
Session
Log
Default settings
⎕IO:
1
⎕ML:
0
⎕PP:
10
⎕RTL:
0
⎕RL:
16807
⎕DIV:
0
⎕WX:
3
Auto PW
Session file:
C:\Documents and Settings\Pete\My Documents\Dyalog APL 12.1 Unicode Files\def_uk.dse
Browse...
OK
Cancel

Label	Parameter	Description
⎕IO	default_io	The default value of ⎕IO in a Clear WS.
⎕ML	default_ml	The default value of ⎕ML in a Clear WS.
⎕PP	default_pp	The default value of ⎕PP in a Clear WS.
⎕RTL	default_rtl	The default value of ⎕RTL in a Clear WS.
⎕RL	default_rl	The default value of ⎕RL in a Clear WS.
⎕DIV	default_div	The default value of ⎕DIV in a Clear WS.
⎕WX	default_wx	The default value of ⎕WX in a Clear WS.
Auto PW	auto_pw	If checked, the value of ⎕PW is dynamic and depends on the width of the Session Window.
Session file	session_file	The name of the Session file in which the definition of your session (⎕SE) is stored.

Configuration dialog: Session

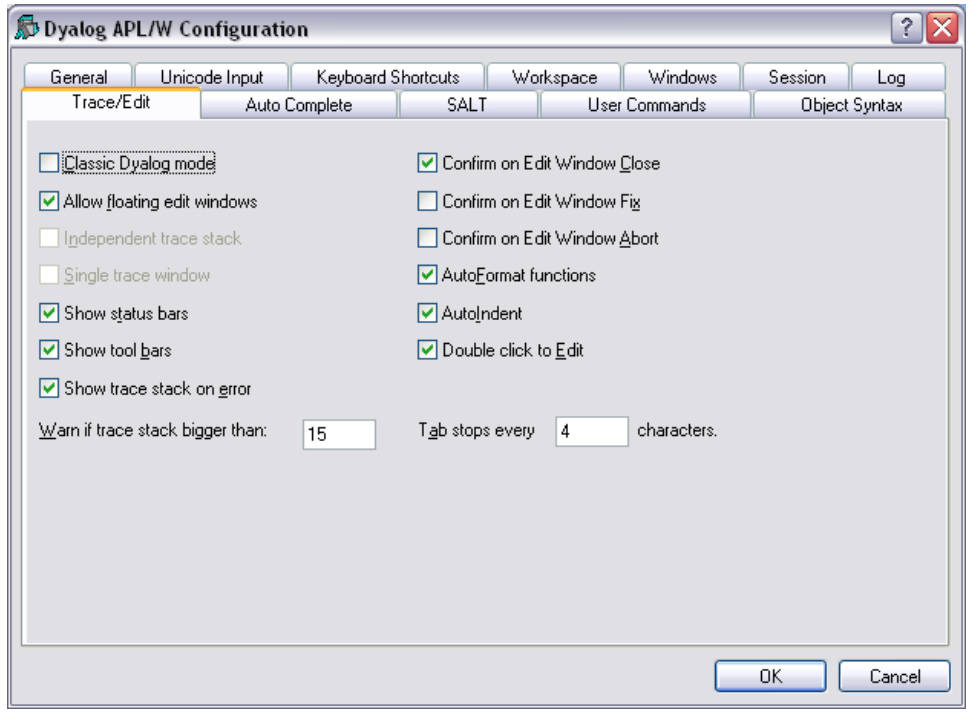
Log Tab


Dyalog APL/W Configuration
Trace/Edit
Auto Complete
SALT
User Commands
Object Syntax
General
Unicode Input
Keyboard Shortcuts
Workspace
Windows
Session
Log
Use Session log file
C:\Documents and Settings\Pete\My Documents\Dyalog APL 12.1 Unic
Browse...
Confirm on Deletion from Session Log
Session log size(Kb):
800
Input buffer size(Kb):
40
History buffer size(Kb):
4
PFKey buffer size(Kb):
10
OK
Cancel

Label	Parameter	Description
Use Session log file	log_file_inuse	Specifies whether or not the Session log is saved in a session log file
Use Session log file	log_file	The full pathname of the Session log file
Confirm on Deletion from Session log	confirm_session_delete	Specifies whether or not you are prompted to confirm the deletion of a line from the Session (and Session log).
Session log size(Kb)	log_size	The size of the Session log buffer in Kb
Input buffer size(Kb)	input_size	The size of the buffer used to store marked lines (lines awaiting execution) in the Session
History size(Kb)	history_size	The size of the buffer used to store previously entered (input) lines in the Session
PFKey buffer size(Kb)	pfkey_size	The size of the buffer used to store PFKey definitions (⎕PFKEY)

Configuration dialog: Log

Trace/Edit Tab

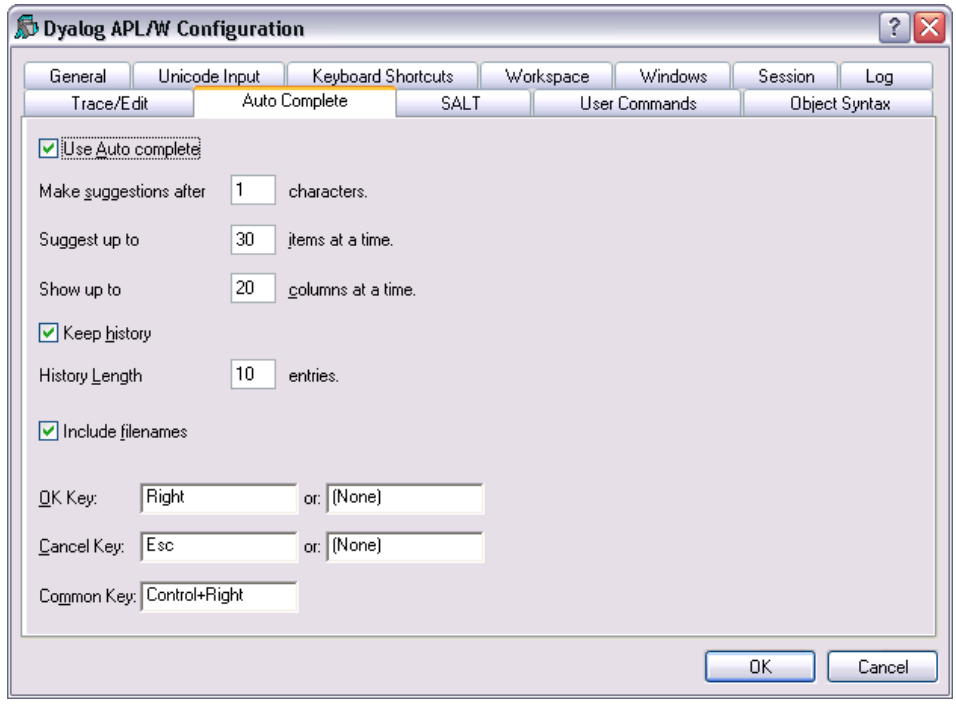


Label	Parameter	Description
Classic Dyalog mode	ClassicMode	Selects pre-Version 9 behaviour for Edit and Trace windows
Allow floating edit windows	DockableEditWindows	Allows individual Edit windows to be undocked from (and re-docked in) the main Edit window
Independent trace stack	IndependentTrace	Specifies whether or not the Trace windows are child windows of the Session.
Single trace window	SingleTrace	Specifies whether or not there is a single Trace window
Show status bars	StatusOnEdit	Specifies whether or not status bars are displayed along the bottom of individual Edit windows
Show tool bars	ToolBarsOnEdit	Specifies whether or not tool bars are displayed along the top of individual Edit windows
Show trace stack on error	Trace_on_error	Specifies whether or not the Tracer is automatically invoked when an error or stop occurs in a defined function
Warn if trace stack bigger than	Trace_level_warn	Specifies the maximum stack size for automatic deployment of the Tracer.
Confirm on edit window close	confirm_close	Specifies whether or not a confirmation dialog is displayed if the user alters the contents of an edit window, then closes it without saving
Confirm on edit window fix	confirm_fix	Specifies whether or not a confirmation dialog is displayed if the user alters the contents of an edit window, then saves it using *Fix* or *Exit*
Confirm on edit window abort	confirm_abort	Specifies whether or not a confirmation dialog is displayed if the user alters the contents of an edit window, then aborts using
Autoformat functions	AutoFormat	Selects automatic indentation for Control Structures when function is opened for editing

Label	Parameter	Description
Autoindent functions	AutoIndent	Selects semi-automatic indentation for Control Structures while editing
Double-click to Edit	DoubleClickEdit	Specifies whether or not double-clicking over a name invokes the editor
Paste text as Unicode	UnicodeToClipboard	Specifies whether or not text transferred to and from the Windows clipboard is to be treated as Unicode
Tab stops every	TabStops	The number of spaces inserted by pressing Tab in an edit window

Configuration dialog: Trace/Edit

Auto Complete Tab



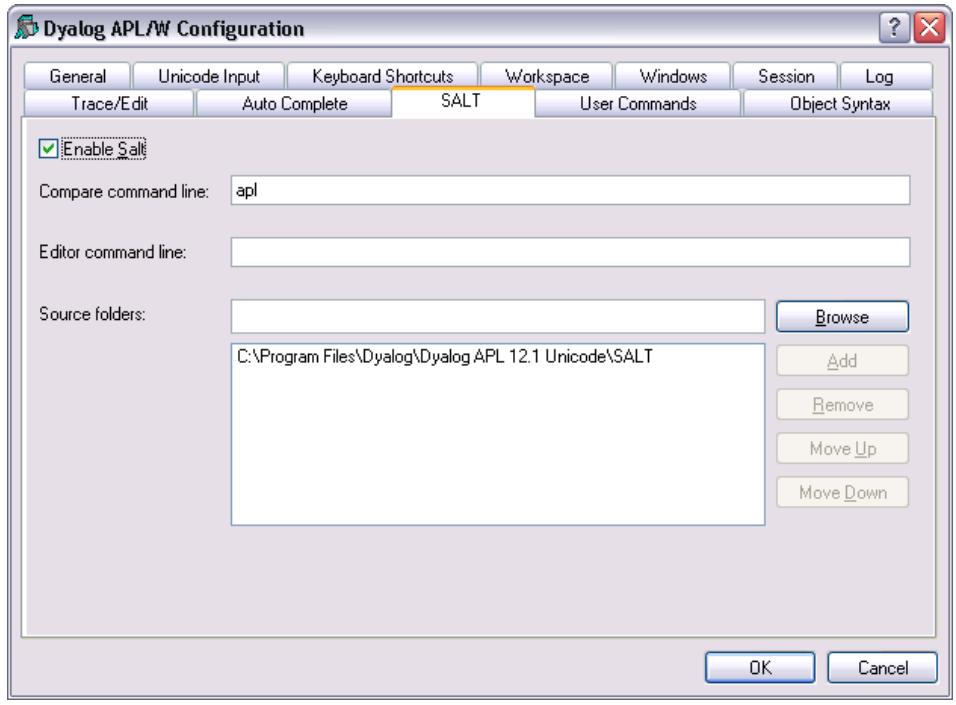
Label	Parameter	Description
Use Auto Complete	Enabled	Specifies whether or not Auto Completion is enabled.
Make suggestions after	PrefixSize	Specifies the number of characters you must enter before Auto Completion begins to make suggestions
Suggest up to	Rows	Specifies the maximum number of rows (height) in the AutoComplete pop-up suggestions box.
Show up to	Cols	Specifies the maximum number of columns (width) in the AutoComplete pop-up suggestion box
Keep History	History	Specifies whether or not AutoComplete maintains a list of previous AutoCompletions.
History Length	HistorySize	Specifies the number of previous AutoCompletions that are maintained
Include filenames	ShowFiles	Specifies whether or not AutoCompletion suggests directory and file names for `)LOAD`, `)COPY` and `)DROP` system commands.
OK Key	CompleteKey1 CompleteKey2	Specifies two possible keys that may be used to select the current option from the Auto Complete suggestion box.
Cancel Key	CancelKey1 CancelKey2	Specifies two possible keys that may be used to cancel (hide) the Auto Complete suggestion box.
Common Key	CommonKey1	Specifies the key that will auto-complete the *common prefix*. This is defined to be the longest string of leading characters in the currently selected name that is shared by at least one other name in the Auto Complete suggestion box...

Configuration dialog: Auto Complete

Note: To enter values in the OK Key and Cancel Key fields, click on the field with the mouse and then press the desired keystroke.

SALT

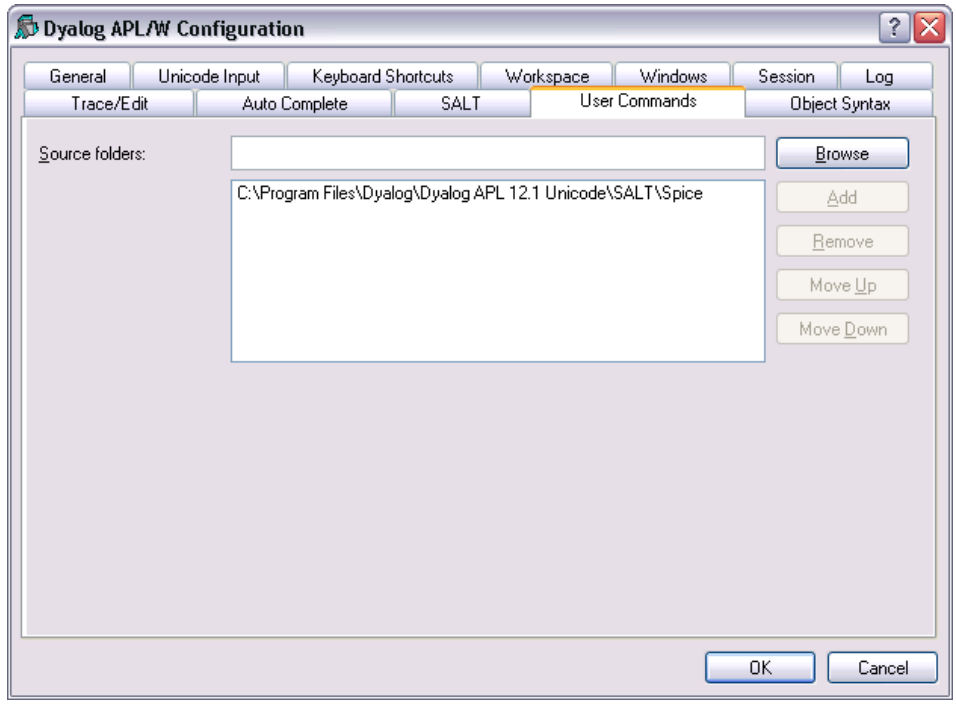
SALT is the Simple APL Library Toolkit, a simple source code management system for Classes and script-based Namespaces. SPICE uses SALT to manage development tools which “plug in” to the Dyalog session



Label	Parameter	Description
Enable Salt	AddSALT	Specifies whether or not SALT is enabled
Compare command line	CompareCMD	The command line for a 3rd party file comparison tool to be used to compare two versions of a file. See note.
Editor	Editor	Name of the program to be used to edit script files (default "Notepad").
Class source folders	SourceFolder	Sets the SALT working directory; a list of folders to be searched for source code.

Configuration dialog: SALT

User Commands Tab

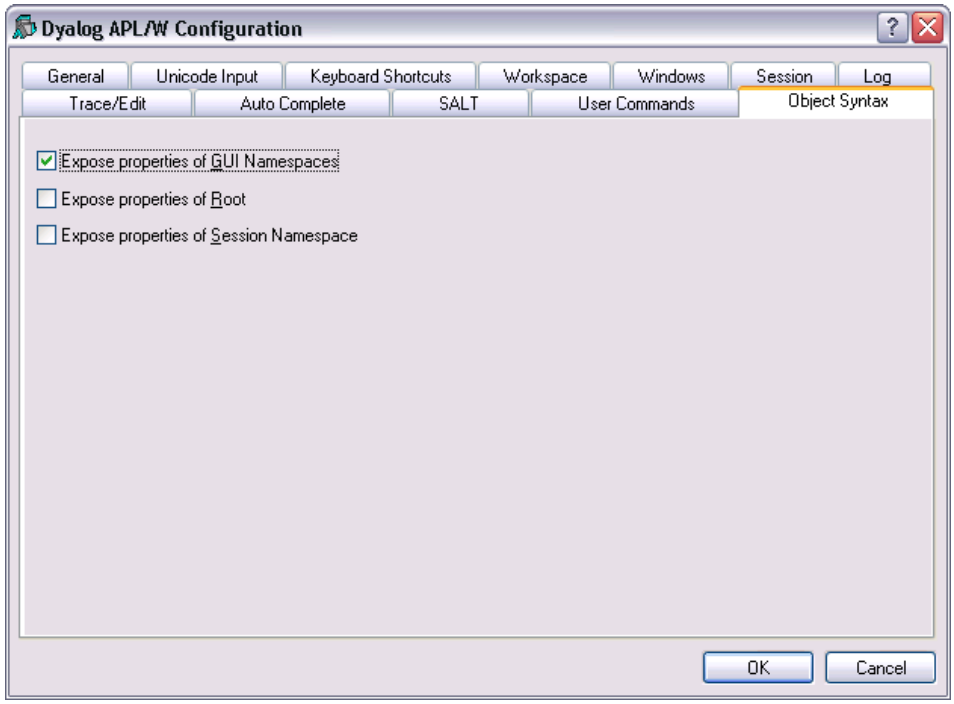


This page is used to specify and organise a list of folders that contain User-Command files. When you issue a User Command, these folders will be searched for the source of the command in the order in which they appear in this list.

Label	Parameter	Description
Source Folders	SALT\CommandFolder	Use this field to add folders to the list of folders that will be searched for User Commands.

Configuration dialog: User Commands

Object Syntax Tab



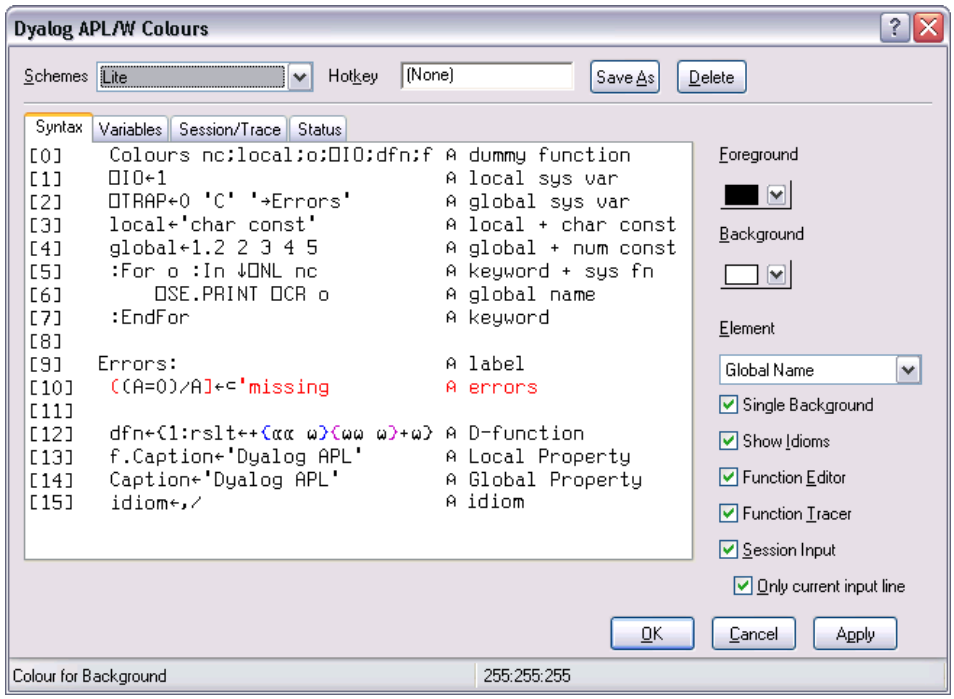
Label	Parameter	Description
Expose properties of GUI Namespaces	default_wx	Specifies the value of ⎕WX in a clear workspace. This in turn determines whether or not the names of properties, methods and events of GUI objects are exposed. If set (⎕WX is 1), you may query/set properties and invoke methods directly as if they were variables and functions respectively. As a consequence, these names may not be used for global variables in GUI objects.
Expose properties of Root	PropertyExposeRoot	Specifies whether or not the names of properties, methods and events of the Root object are exposed. If set, you may query/set the properties of Root and invoke the Root methods directly as if they were variables and functions respectively. As a consequence, these names may not be used for global variables in your workspace.
Expose properties of Session Namespace	PropertyExposeSE	Specifies whether or not the names of properties, methods and events of the Session object are exposed. If set, you may query/set the properties of ⎕SE and invoke ⎕SE methods directly as if they were variables and functions respectively. As a consequence, these names may not be used for global variables in the ⎕SE namespace.

Configuration dialog: Object Syntax

The *Object Syntax* tab of the Configuration dialog is used to set your *default preferences* for Object Syntax.

The Object Syntax settings for the current workspace are reflected by the *Object Syntax* submenu of the *Options* menu. Use *Options/Object Syntax* to change them. These settings are saved in the workspace.

Colour Selection Dialog



The Colour Selection dialog box allows you to select colours for:

- Syntax colouring
- Edit, Trace and Session windows
- Status window

The colour selection dialog box is selected by the `[ChooseColor]` system action which by default is attached to *the Options/Colours* menu item on the Session menubar and to the *Colours* menu item in the Session pop-up menu.

Syntax Colouring

Syntax colouring allows you to visually identify various components in the function edit and session windows by assigning different colours to them, such as:

- Global references (functions and variables)
- Local references (functions and variables)
- Primitive functions
- System functions
- Localised System Variables
- Comments
- Character constants
- Numeric constants
- Labels
- Control Structures
- Unmatched parentheses, quotes, and braces

Schemes

You may define a number of different syntax colouring schemes which are suitable for different purposes and a selection of schemes is provided. Choose the scheme you wish to use from the Combo box provided. If you change a colour allocation, you may overwrite an existing Colour Scheme or define a new one by clicking Save As and then entering the name of the Scheme. You may delete a Colour Scheme using the Delete button.

Changing Colours

To allocate a colour to a syntax element, you must first select the syntax element. You may select a syntax element from the Combo box provided, or by clicking on an example in the sample function provided. Having selected a syntax element, choose a colour using the Foreground or Background selectors as appropriate.

Show Idioms

The Show Idioms checkbox allows you to choose whether or not idioms are to be identified by syntax colouring.

Single Background

The Single Background checkbox allows you to choose whether to impose a single background colour, or to allow the use of different background colours for different syntax elements.

Function Editor

Check this box if you want to enable syntax colouring in Edit windows.

Function Tracer

Check this box if you want to enable syntax colouring in Trace windows.

Session Input

Check this box if you want to enable syntax colouring in the Session window. Note that the colour scheme used for the Session may differ from the colour scheme selected for Edit windows and is specified by the *Session Colour Scheme* box on the Session/Trace tab.

Only current input line

This option only applies if Session syntax colouring is enabled. Check this box if you want syntax colouring to apply only to the current input line. Clear this box, if you want to apply syntax colouring to all the input lines in the current Session window. Note that syntax colouring of input lines is not remembered in the Session log, so input lines from previous sessions do not have syntax colouring.

HotKeys

You may associate different *hot key* with any or all of your colour schemes.

When you depress a hot key over a function in an Edit window, the function is displayed using the scheme associated with the hot key. Releasing the hot key causes it to be displayed in the normal scheme.

This feature is intended to allow you to quickly check for certain syntax elements. For example, you may define a special scheme that only highlights global names and associate a hot key with it. Pressing the hot key will temporarily highlight the globals for you.

To associate a hot key with a colour scheme, click on the *Hotkey* field, and then make the desired keystroke. To disassociate a hot key, use <backspace>.

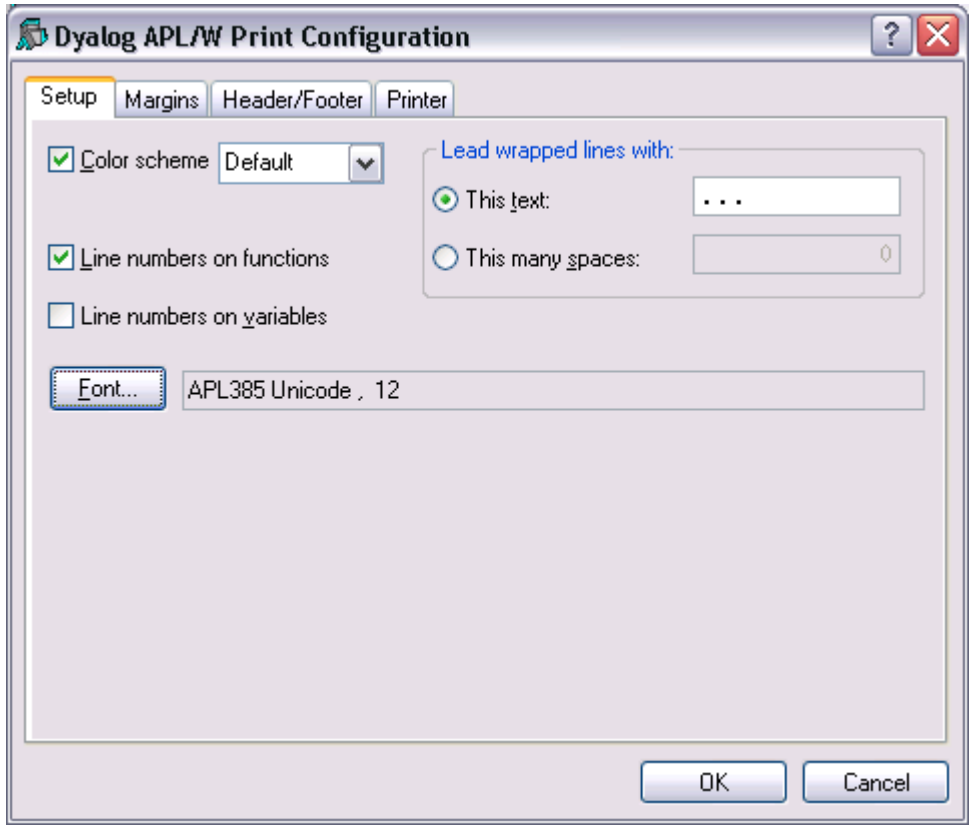
Print Configuration Dialog Box

The Print Configuration dialog box is displayed by the system operation [PrintSetup] that is associated with the File/Print Setup menu item. It is also available from Edit windows and from the Workspace Explorer and Find Objects tools.

There are four separate tabs namely Setup, Margins, Header/Footer and Printer.

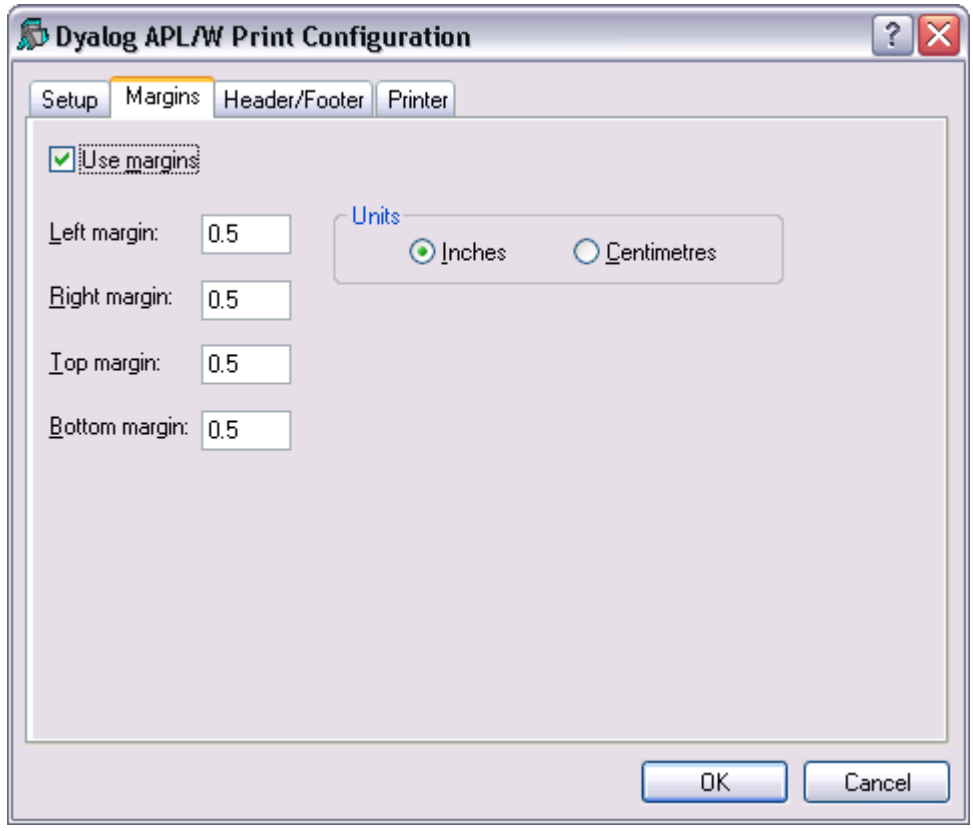
Note that the printing parameters are stored in the Registry in the Printing sub-folder

Setup Tab



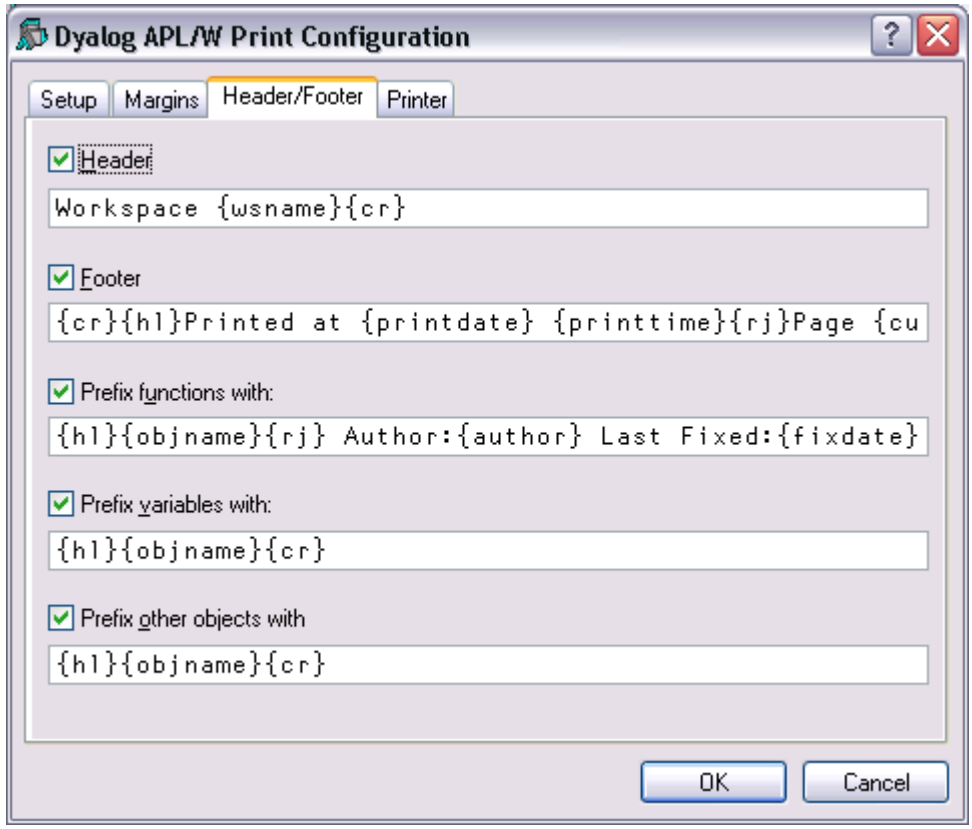
Label	Parameter	Description
Color scheme	InColour	Check this box if you want to print functions with syntax colouring. Note that that printing in colour is slower than printing without colour.
Color scheme	SchemeName	Select the colour scheme to be used for printing.
This text	WrapWithText	Check this option button if you wish to prefix wrapped lines (lines that exceed the width of the paper) with a particular text string
This text	WrapLeadText	Specifies the text for prefixing wrapped lines
This many spaces	WrapWithSpaces	Check this option button if you wish to prefix wrapped lines with spaces.
This many spaces	WrapLeadSpaces	Specifies the number of spaces to be inserted at the beginning of wrapped lines.
Line numbers on functions	LineNumsFns	Check this box if you want line numbers to be printed in defined functions.
Line numbers on variables	LineNumsVars	Check this box if you want line numbers to be printed in variables. If you choose this option, line numbering starts at `⎕IO`.
Font	Font	Click to select the font to be used for printing. Note that only fixed-pitch fonts are supported.

Margins Tab



Label	Parameter	Description
Use margins	UseMargins	Check this box if you want margins to apply
Left margin	MarginLeft	Specifies the width of the left margin
Right margin	MarginRight	Specifies the width of the right margin
Top margin	MarginTop	Specifies the height of the top margin
Bottom margin	MarginBottom	Specifies the height of the bottom margin
Inches	MarginInch	Specifies that the margin units are inches
Centimetres	MarginCM	Specifies that the margin units are centimetres

Header/Footer Tab



Label	Parameter	Description
Header	DoHeader	Specifies whether or not a header is printed at the top of each page
Header	HeaderText	The header text
Footer	DoFooter	Specifies whether or not a footer is printed at the bottom of each page
Footer	FooterText	The footer text
Prefix functions with	DoSepFn	Specifies whether or not text is printed before each defined function
Prefix functions with	SepFnText	The text to be printed before each defined function. This can include its name, timestamp and author
Prefix variables with	DoSepVar	Specifies whether or not text is printed before each variable.
Prefix variables with	SepVarText	The text to be printed before each variable. This can include its name.
Prefix other objects with	DoSepOther	Specifies whether or not text is printed before other objects. These include locked functions, external functions, ⎕NA functions, derived functions and namespaces.
Prefix other objects with	SepOtherText	The text to be printed before other objects. This can include its name.

The specification for headers and footers may include a mixture of your own text, and keywords which are enclosed in braces, e.g. {objname}. Keywords act like variables and are replaced at print time by corresponding values.

Any of the following fields may be included in headers, footers and separators.

{WSName}	{WS}	Workspace name
{NSName}	{NS}	Namespace name
{ObjName}	{OB}	Object name
{Author}	{AU}	Author
{FixDate}	{FD}	Date function was last fixed
{FixTime}	{FT}	Time function was fixed
{PrintDate}	{PD}	Today's date
{PrintTime}	{PT}	Current time
{CurrentPage}	{CP}	Current page number
{TotalPages}	{TP}	Total number of pages
{RightJustify}	{RJ}	Right-justifies subsequent text/fields
{HorizontalLine}	{HL}	Inserts a horizontal line
{CarriageReturn}	{CR}	Inserts a new-line

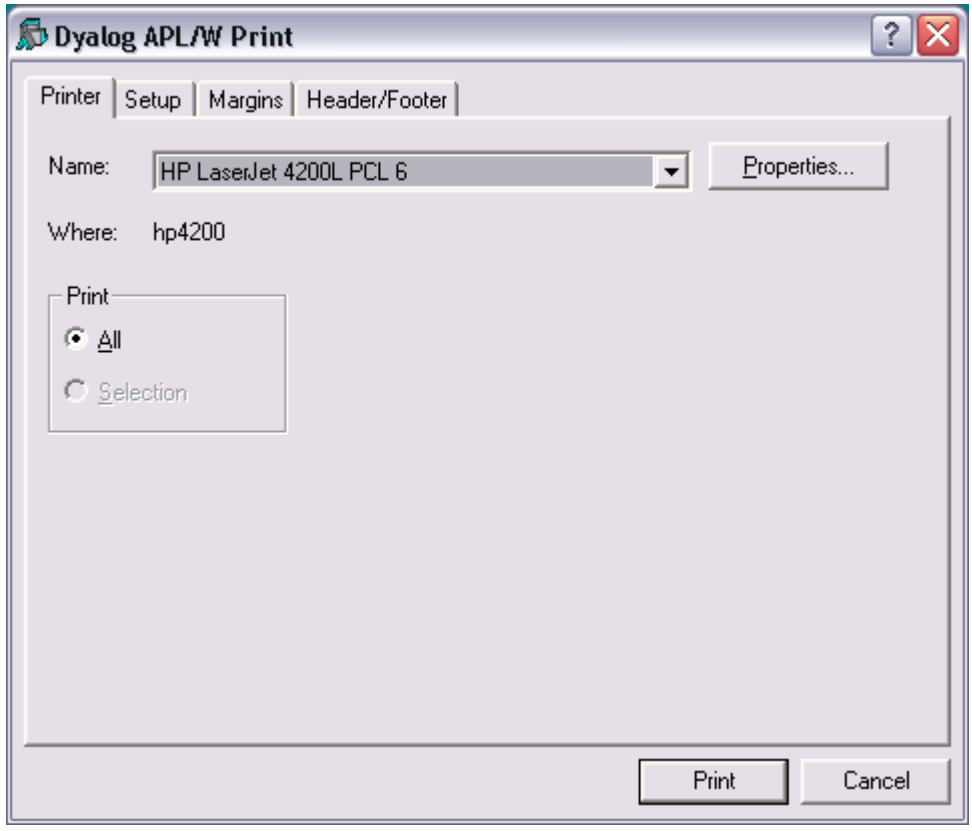
For example, the specification:

Workspace: {wsname} {objname} {rj} Printed {PrintTime} {PrintDate}

would cause the following header, footer or separator to be printed at the appropriate position in each page of output:

Workspace: U:\WS\WDESIGN WIZ_change_toolbar Printed 14:40:11 02 March 1998

Printer Tab



Label	Parameter	Description
Name	PrinterField	The name of the printer to be used for printing from Dyalog APL.
Properties		Click this to set Printer options.
Where		Reports the printer device
Print		Allows you to choose between printing all of the current object or just the selection. Note that this option is present only when the dialog box is displayed in response to selecting *Print*.

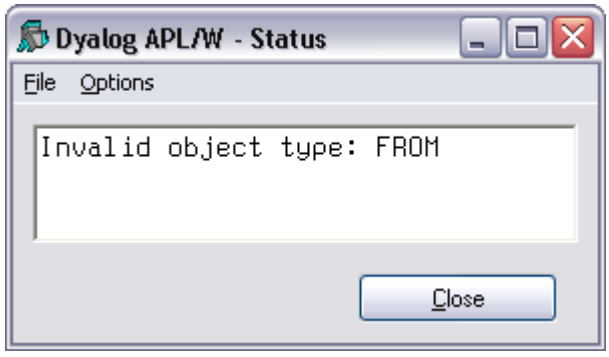
Status Window

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.

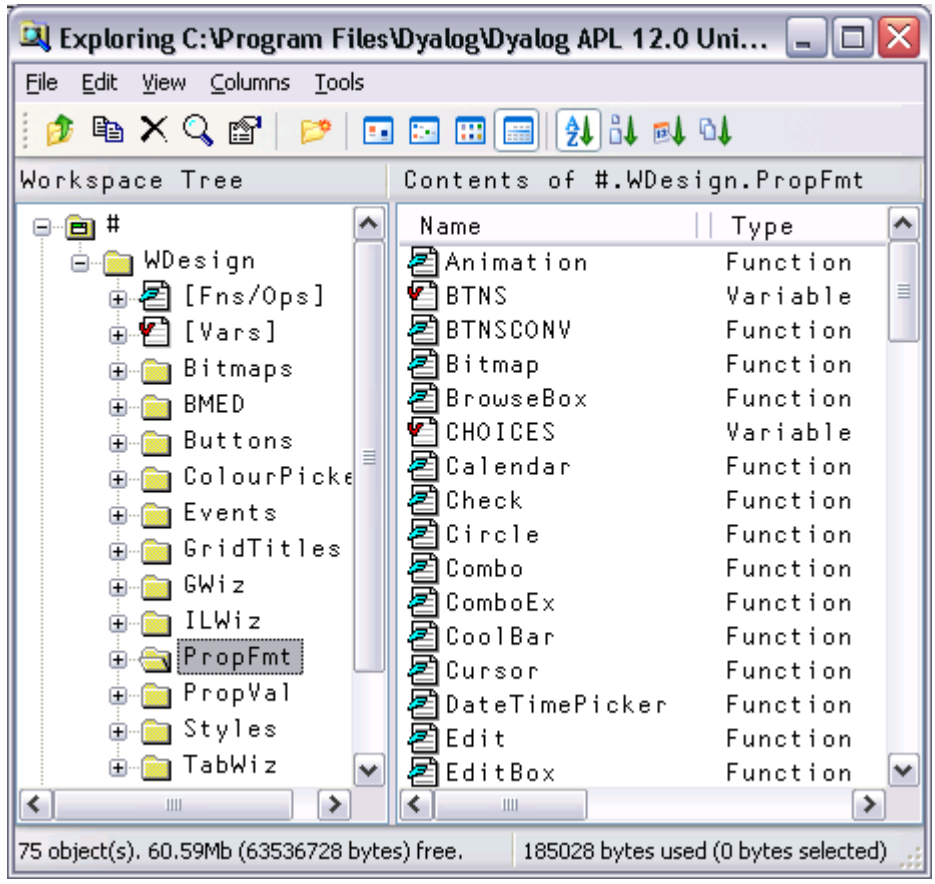
There is also an option to have the Status window appear automatically whenever a new message is written to it. This option is selected using the `[AutoStatus]` system operation which is associated with the *Tools/AutoStatus* menu item.

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.

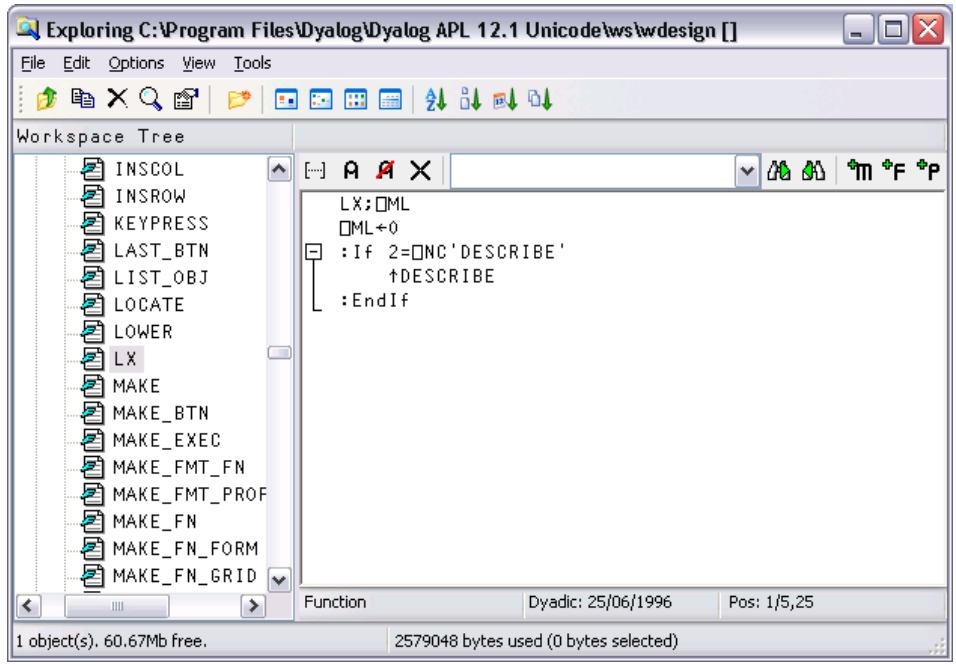
You may also move and copy objects using the Edit menu. To do so, select the object(s) and then choose Move or Copy from the Edit menu. You will be prompted for the name of the namespace into which the objects are to be moved or copied. Enter the namespace and click OK.

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

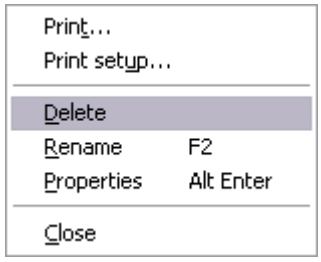
If you open the `Fns/Ops` item, the names of the functions and operators in the namespace are displayed below it alphabetically in the left (tree view) pane. When you select one of these names, the function itself is opened in the right (list view) pane.



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.

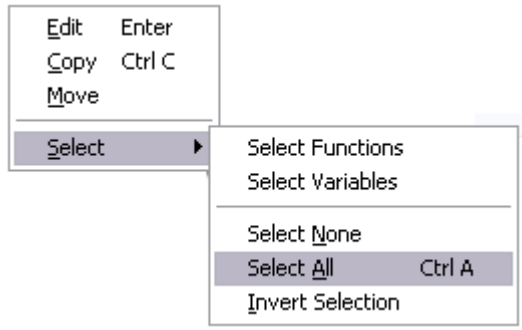
The File Menu



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

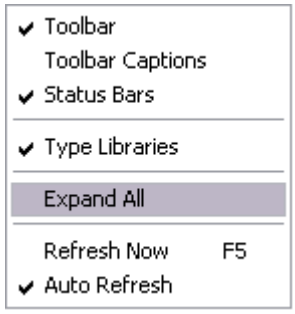
The Edit Menu



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

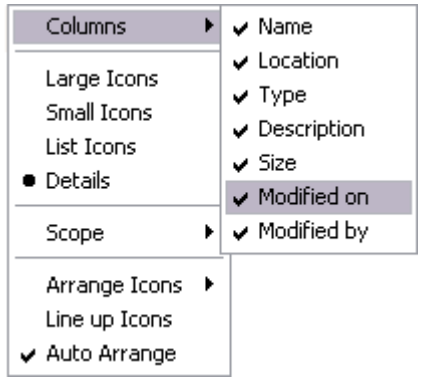


The Options menu, illustrated above, provides the following actions.

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.

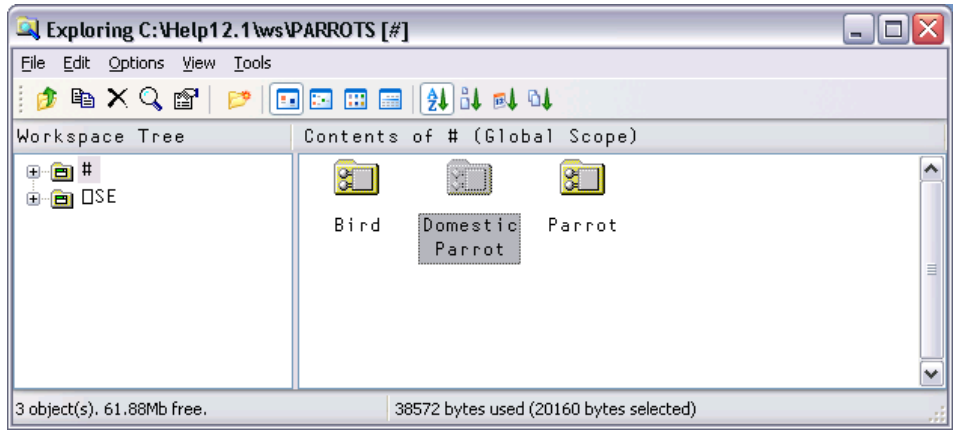
The Tools Menu

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

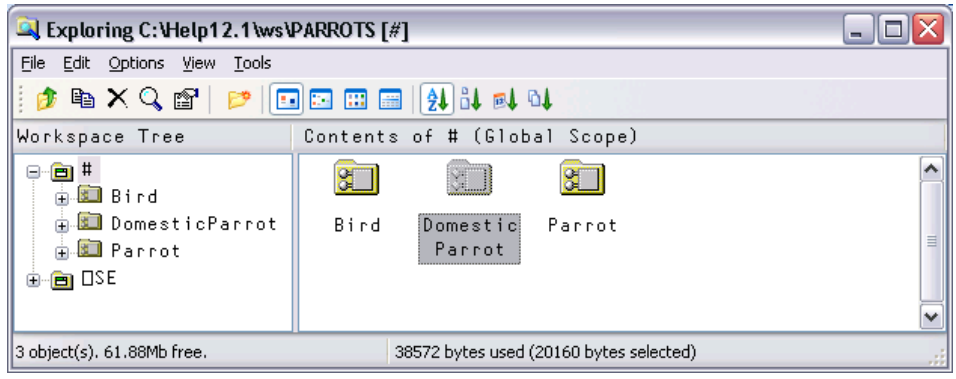
Find	Displays the Find Objects Tool
Go to	Prompts for a namespace and then opens that namespace in the TreeView, displaying its contents in the ListView
Go to Session Space	Opens the namespace in the TreeView control corresponding to the current space in the Session.
Set Session Space	Sets the current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

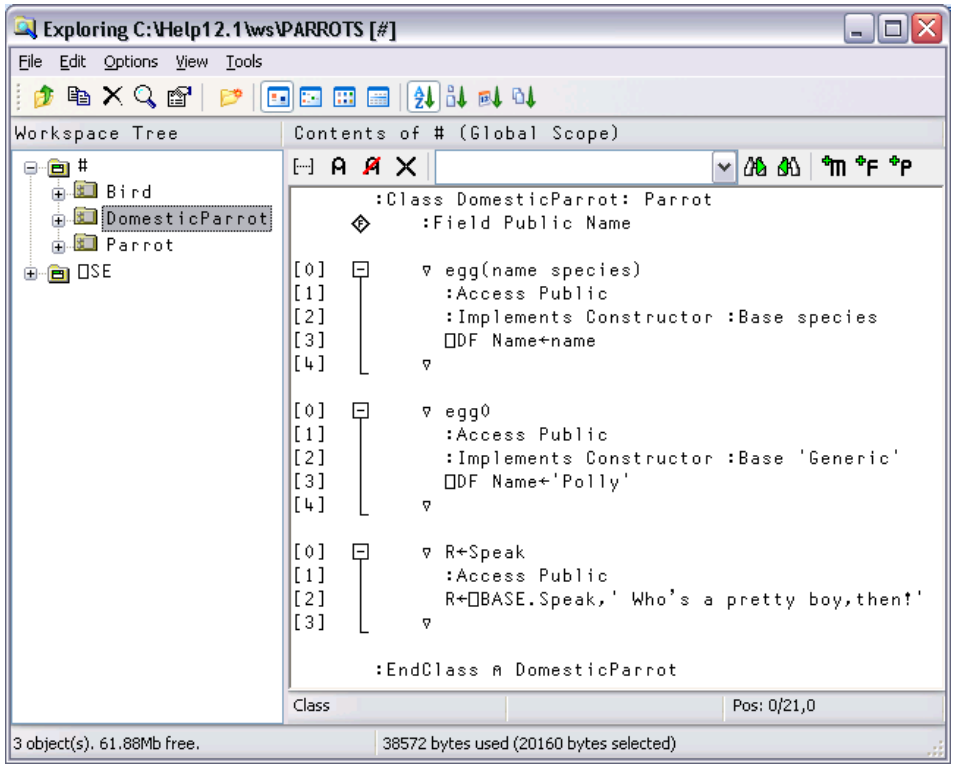


If you open the # node in the left-hand pane, you see the contents of # as a tree.

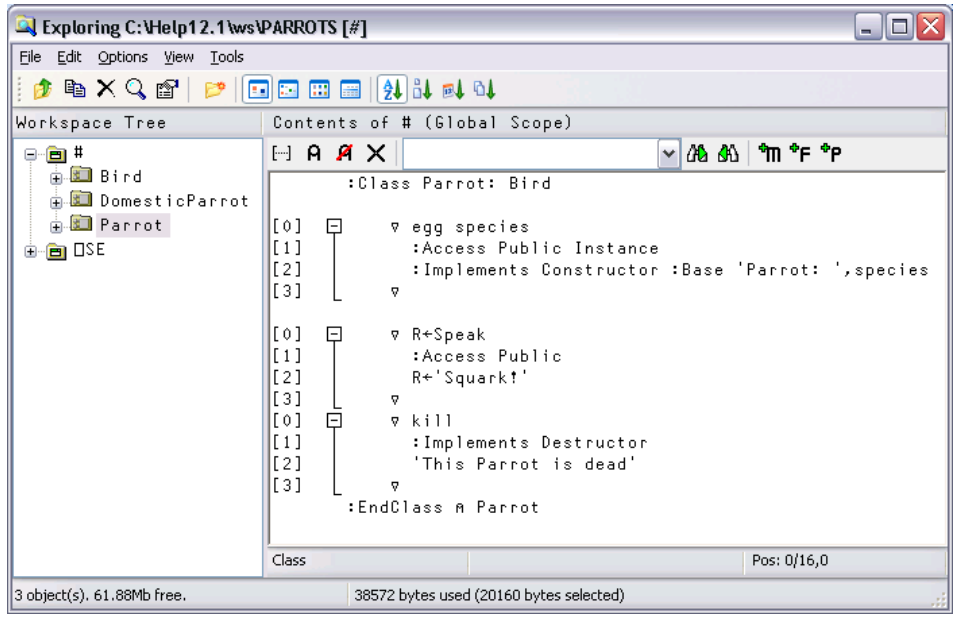


Browsing Class Scripts

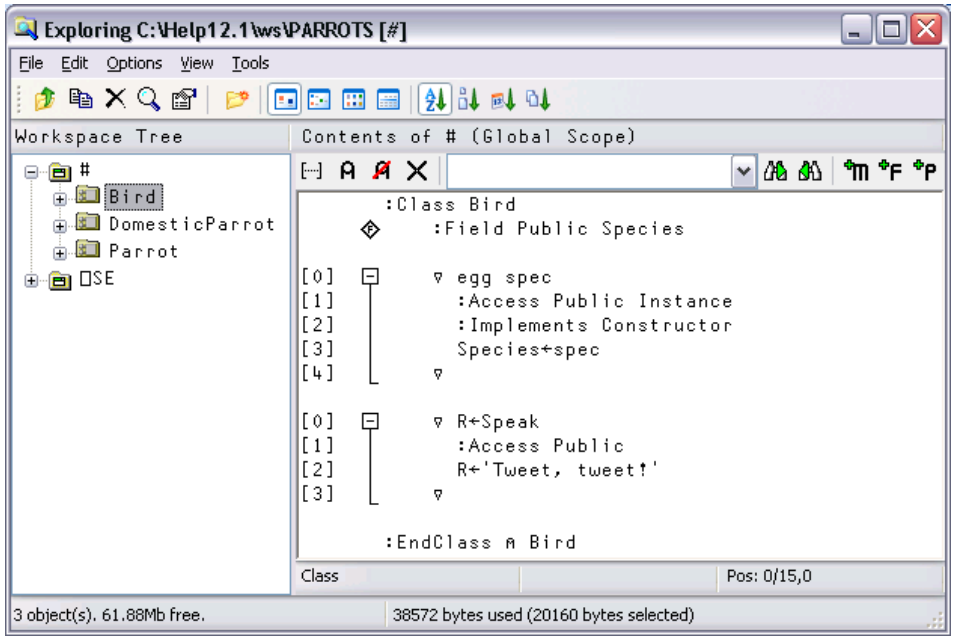
Selecting `DomesticParrot` in the left-hand pane brings up its Class Script in the right-hand pane.



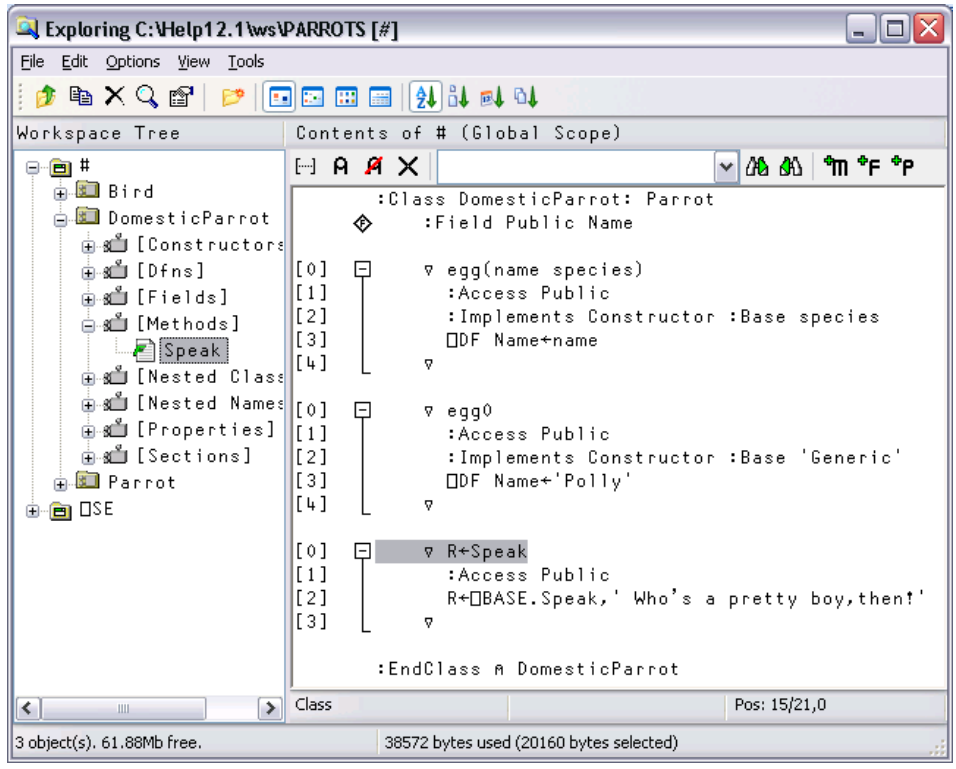
… and selecting `Parrot` in the left-hand pane brings up the Class Script for `Parrot`.



… 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 and .Net Metadata

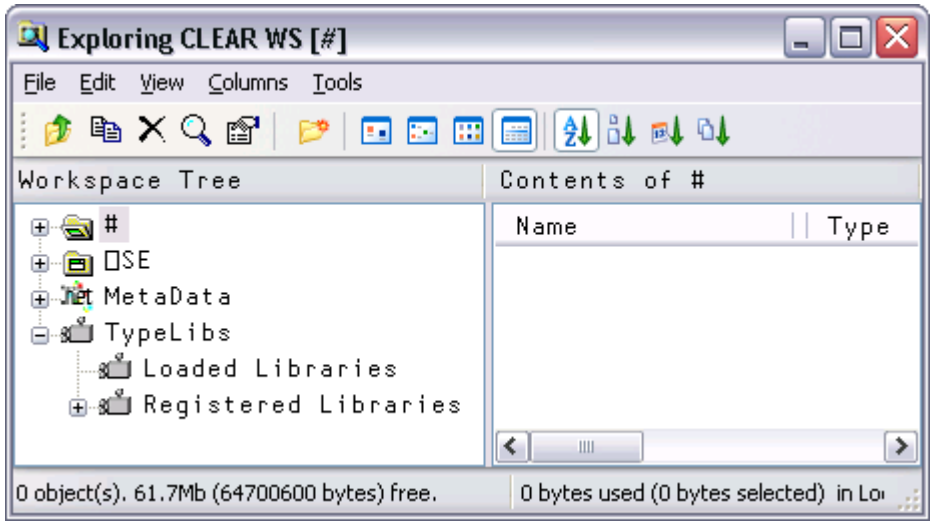
When the *View/Type Libraries* option is enabled, the Workspace Explorer allows you to:

- 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.

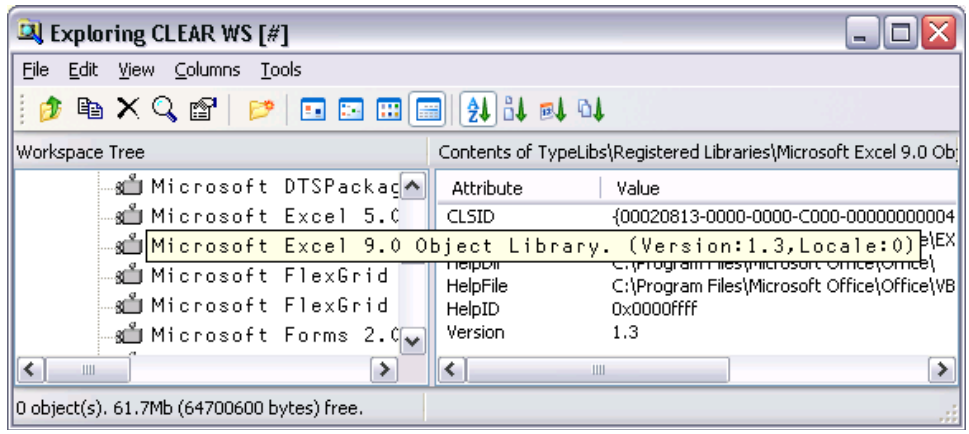


Browsing Registered Libraries

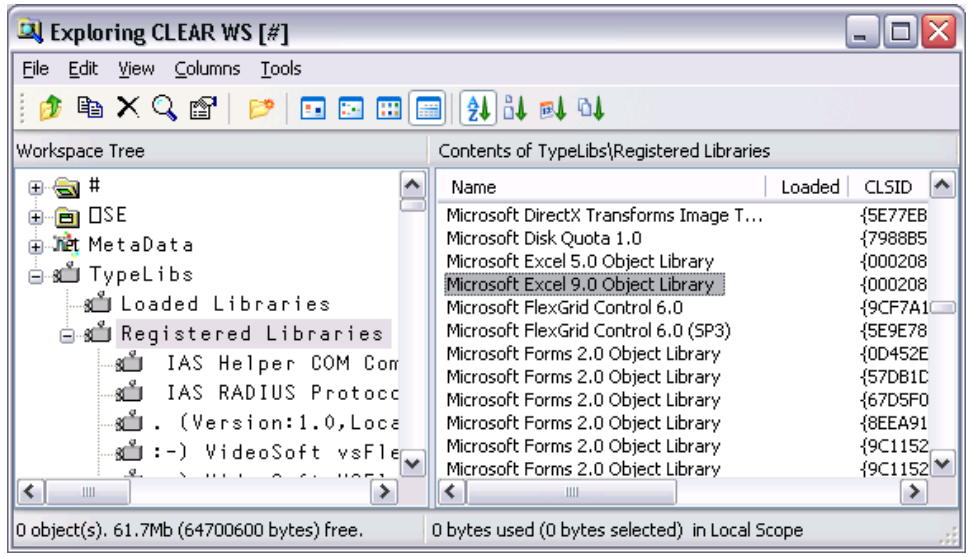
If you open the Registered Libraries folder, the Workspace Explorer will display in the tree view pane the names of all the Type Libraries associated with the COM Server objects that are installed on your computer.

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.

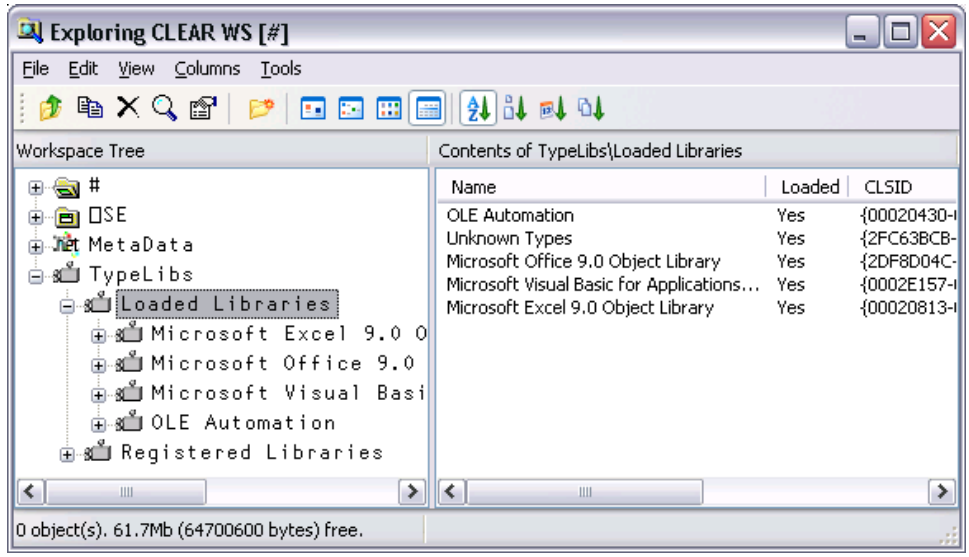
Notice that if the selected Library references any other libraries, they too will be loaded. For example, loading the *Microsoft Excel 9.0 Object Library* brings in the *Microsoft Office 9.0 Object Library* and the *Microsoft Visual Basic for Applications Extensibility 5.3 Library* too. It also contains 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.

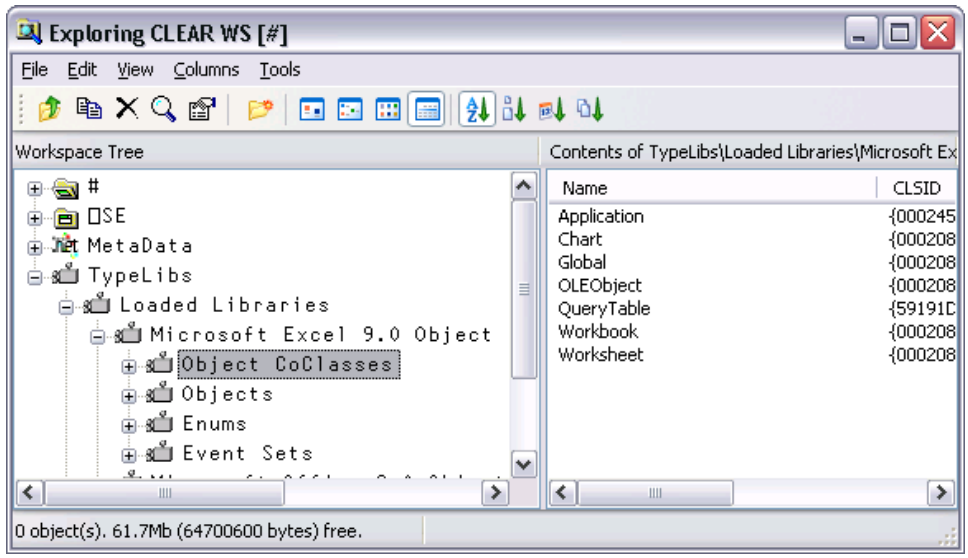
If you select a loaded Type Library, summary information is displayed in the list view pane.

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

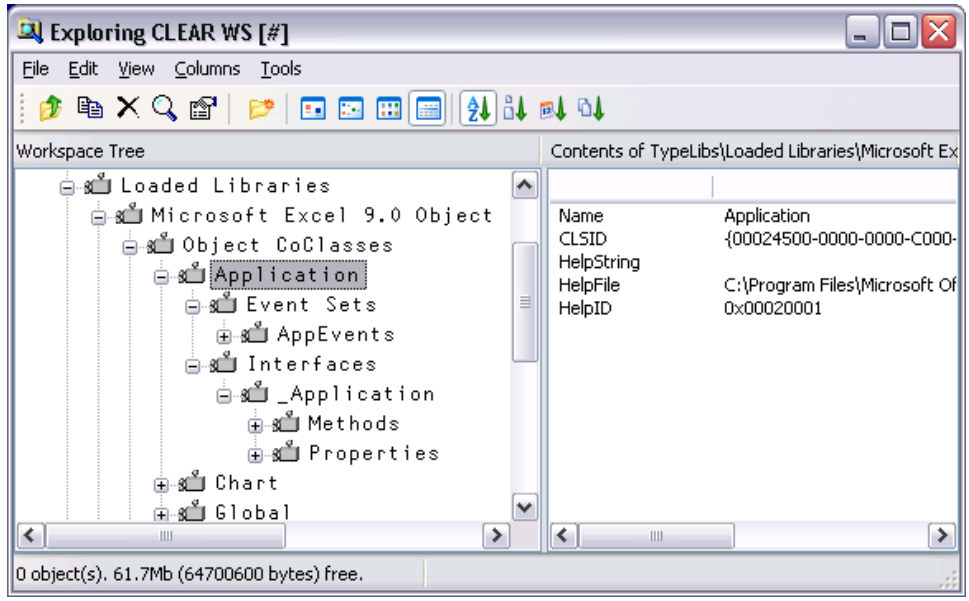
Object CoClasses

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.

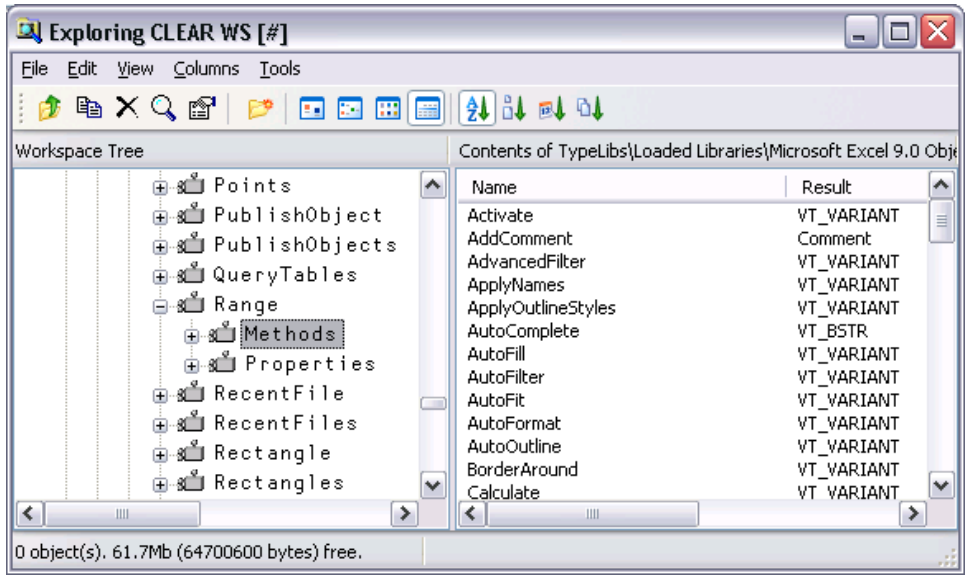


The specific methods, properties and events supported by the CoClass object can be examined by opening the appropriate sub-folder. The same information for these and other objects is also accessible from the Objects and Event Sets folders as discussed below.

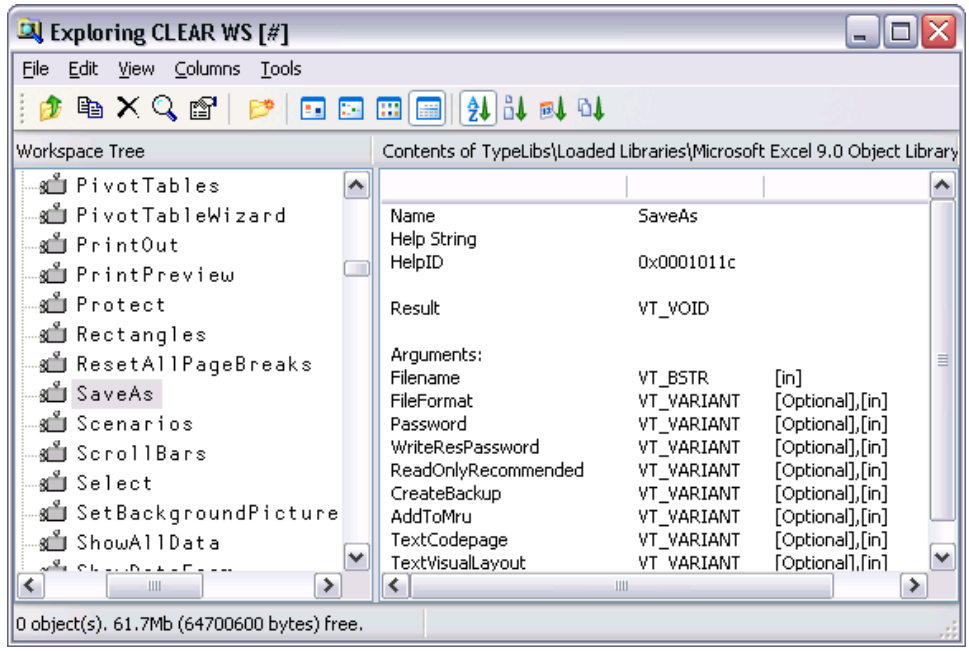
Objects

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.

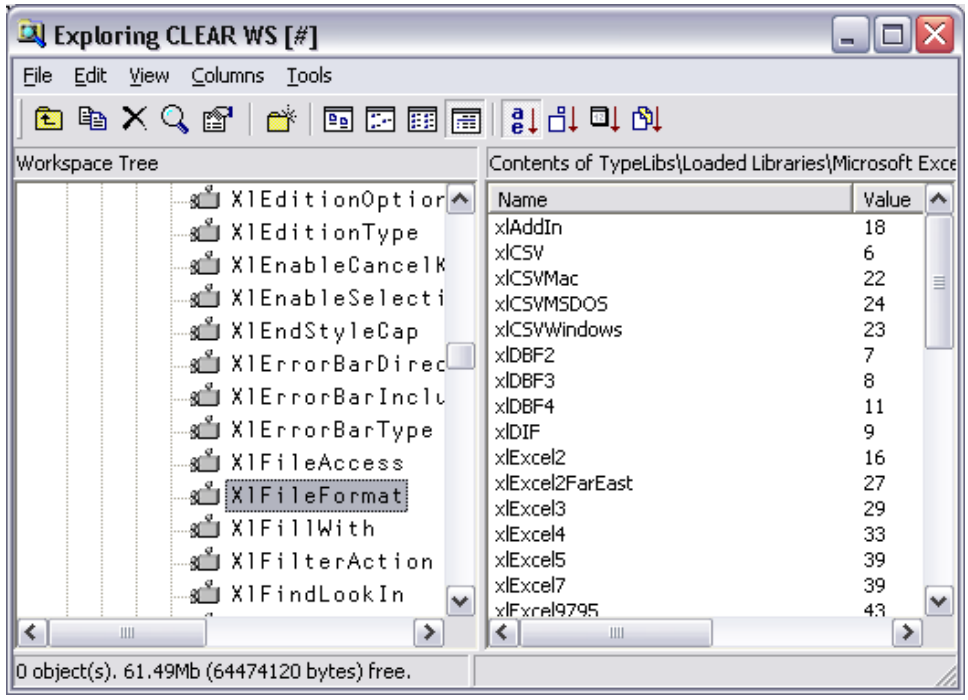


If you open the Methods or Properties subfolder, you can display more detailed information about individual Methods and Properties. For example, the following picture shows information about the SaveAs method exposed by the Microsoft Excel 9.0 Worksheet object.



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
```

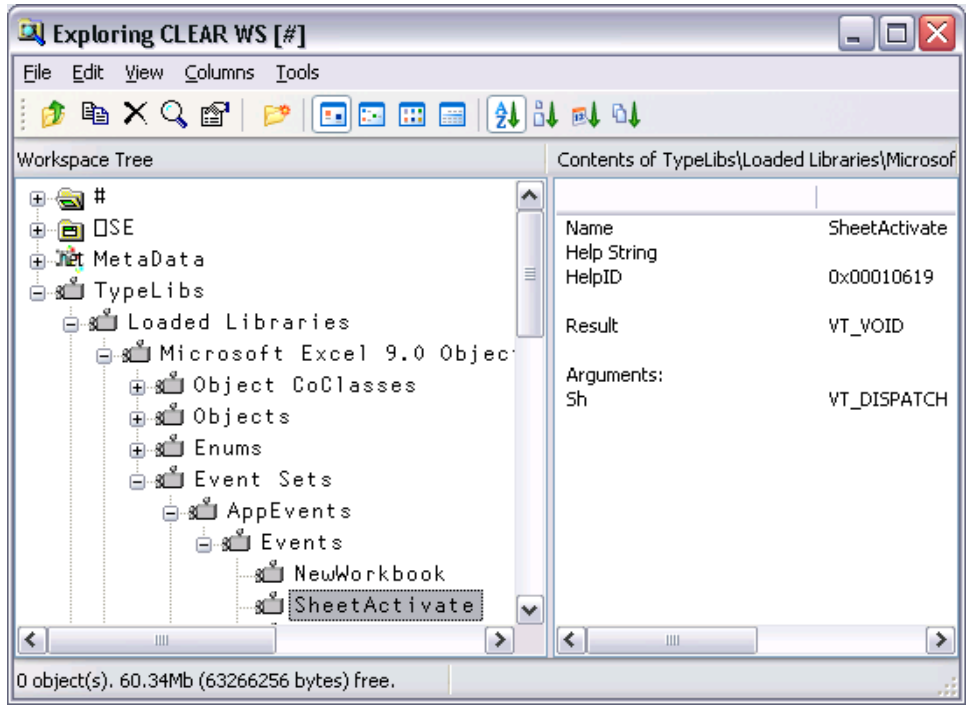
or

```
      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.

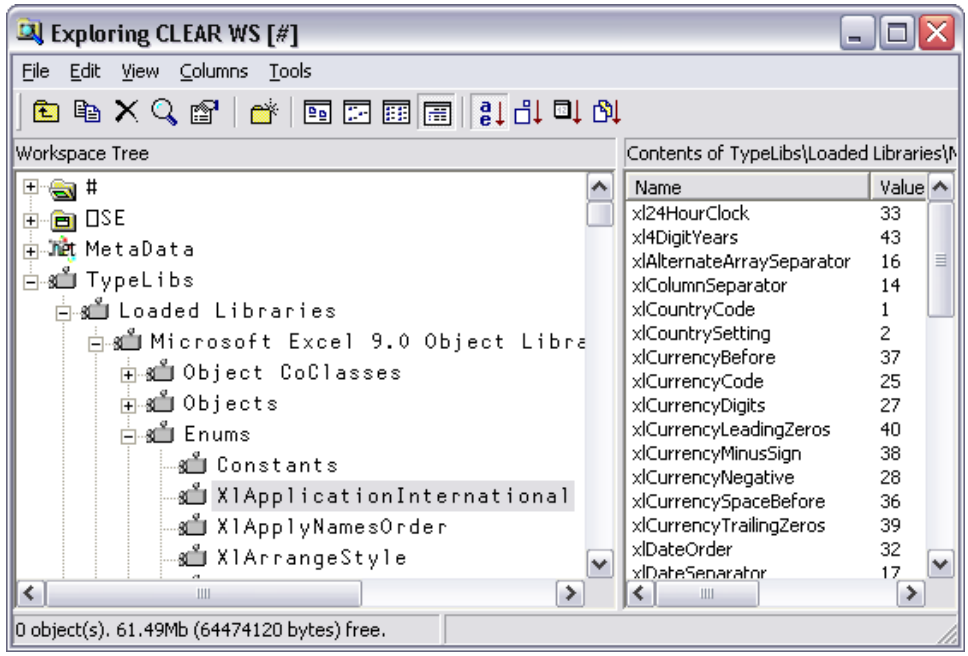


This example shows that when it fires, the SheetActivate event invokes your callback function with a single argument named *Sh* whose datatype is VT_DISPATCH (in practice, a Worksheet object).

Enums

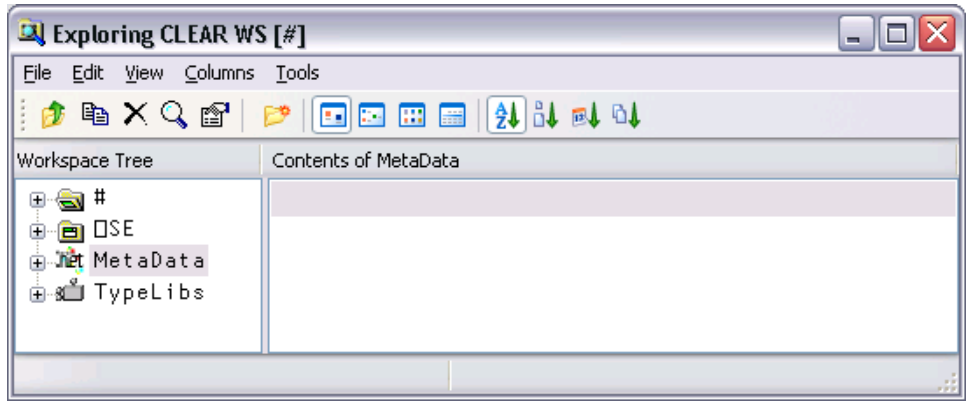
The Enums folder will typically contain several sub-folders each of which represents a named set of enumerated constants.

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.

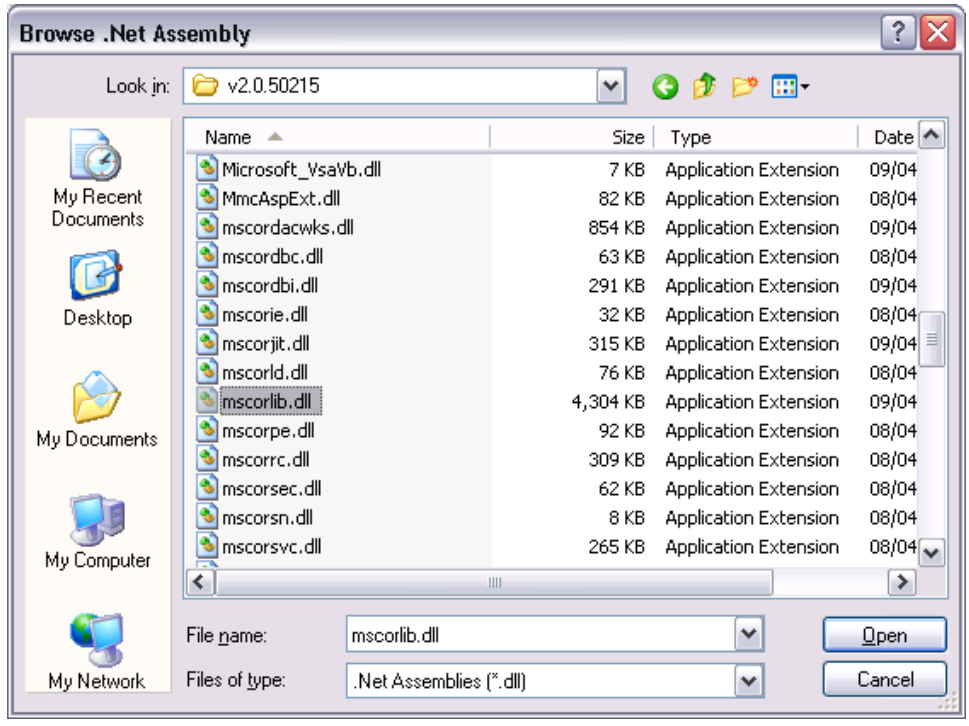


Browsing .Net Classes

If the Microsoft .Net Framework is installed, you may browse the .Net Metadata using the Explorer. To gain information about one or more Net Classes, open the Workspace Explorer, right click the *Metadata* folder, and choose *Load*.



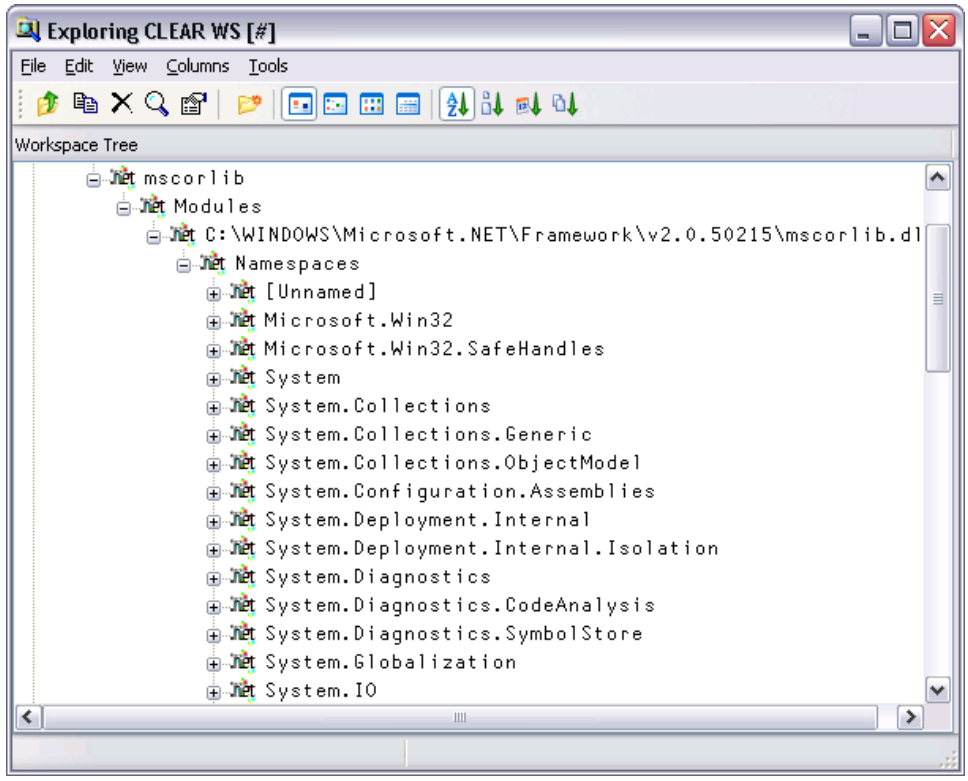
This brings up the *Browse .Net Assembly* dialog box as shown below. Navigate to the .NET assembly of your choice, and click *Open*.



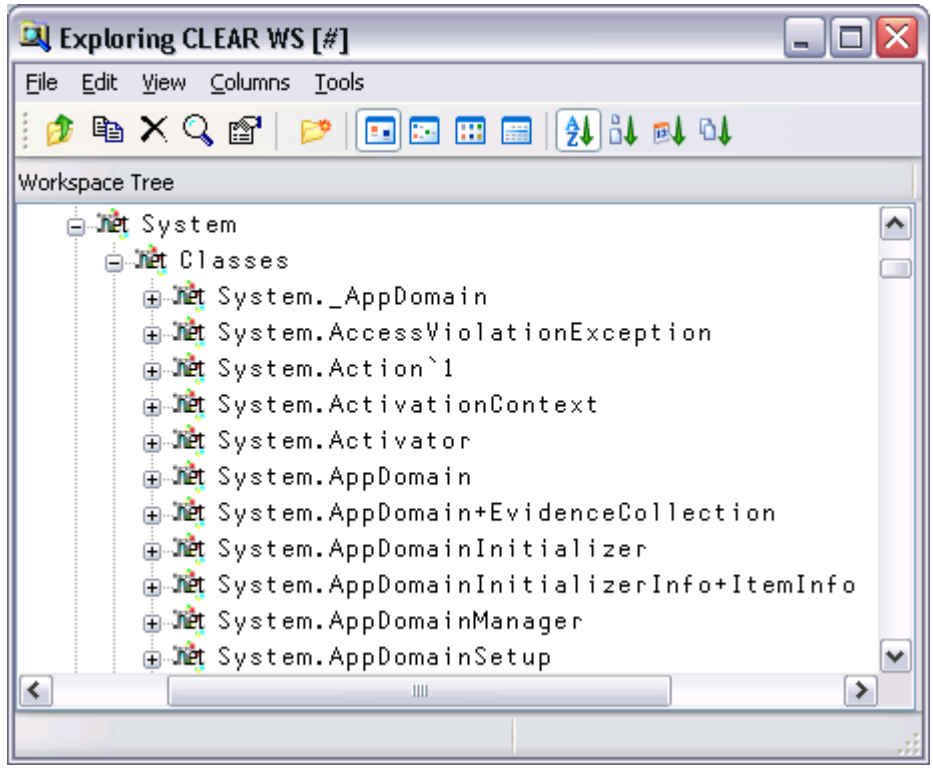
Note that the .NET Classes provided with the .NET Framework are typically located in `C:\WINDOWS\Microsoft.NET\Framework\V2.0.50215`. The last named folder is the Version number.

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.

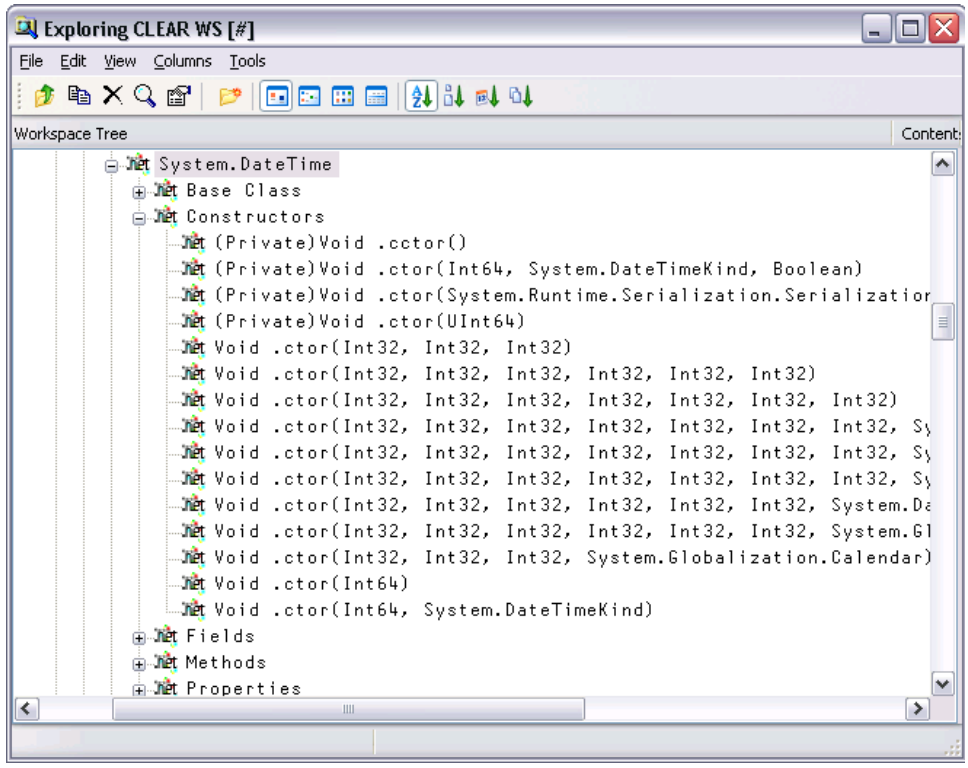


Opening the *System/ Classes* sub-folder causes the Explorer to display the list of classes contained in the .NET Namespace *System* as shown in the picture below.



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`.

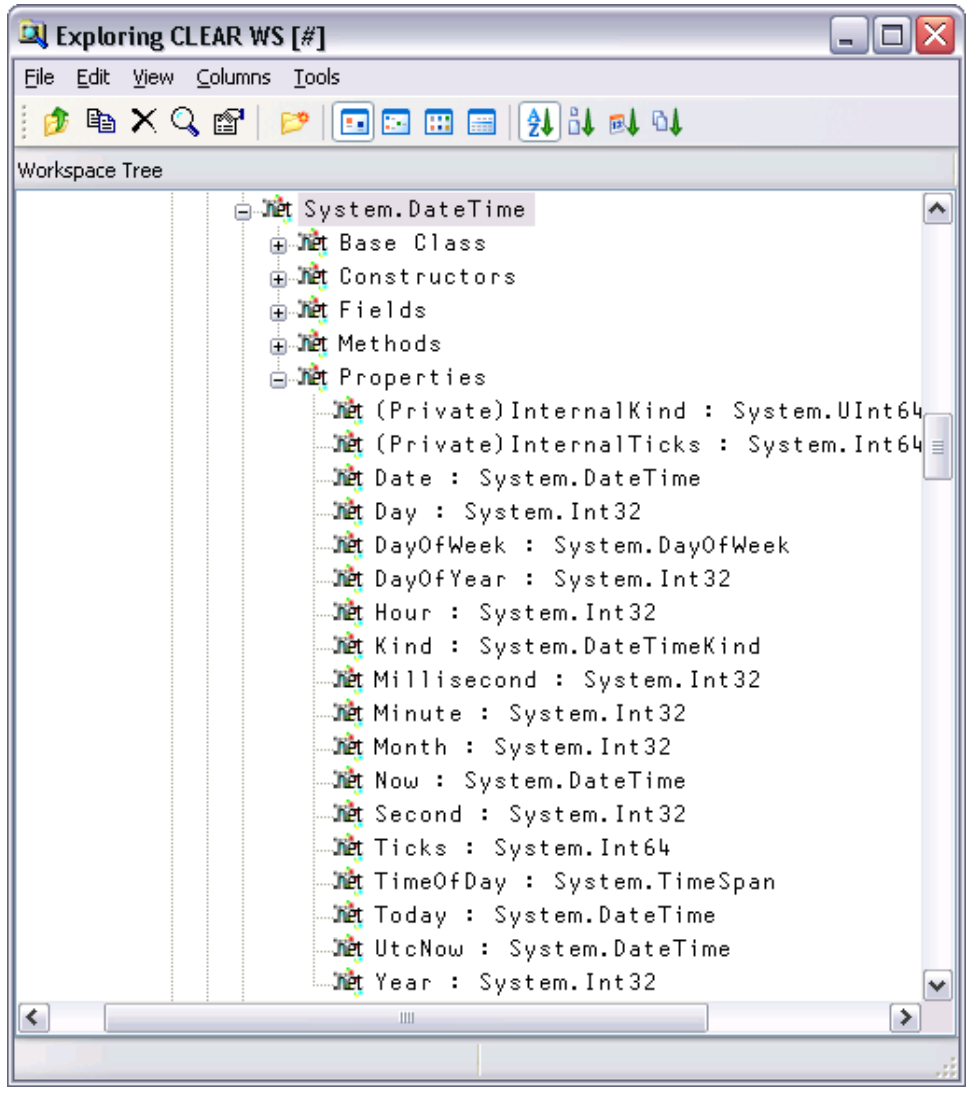


For example, the following statement may be used to create a new instance of `DateTime` (09:30 in the morning on 30th April 2001):

```
      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:

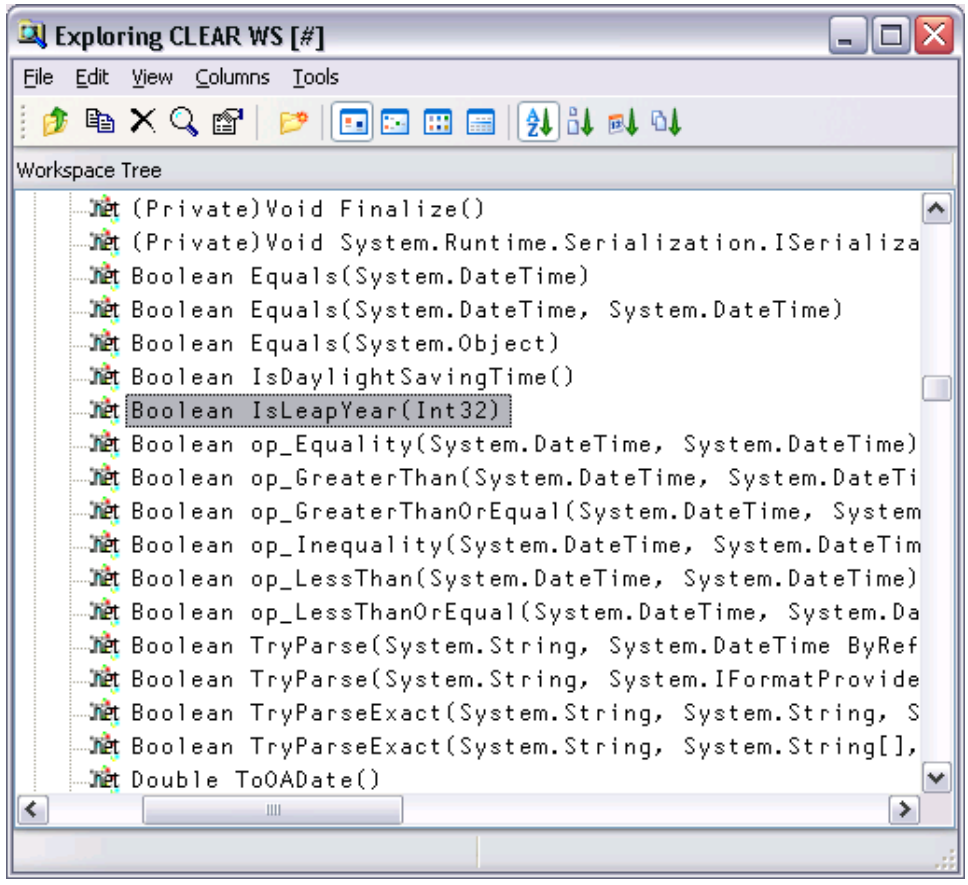
```
      mydt.DayOfWeek
Monday
```

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.

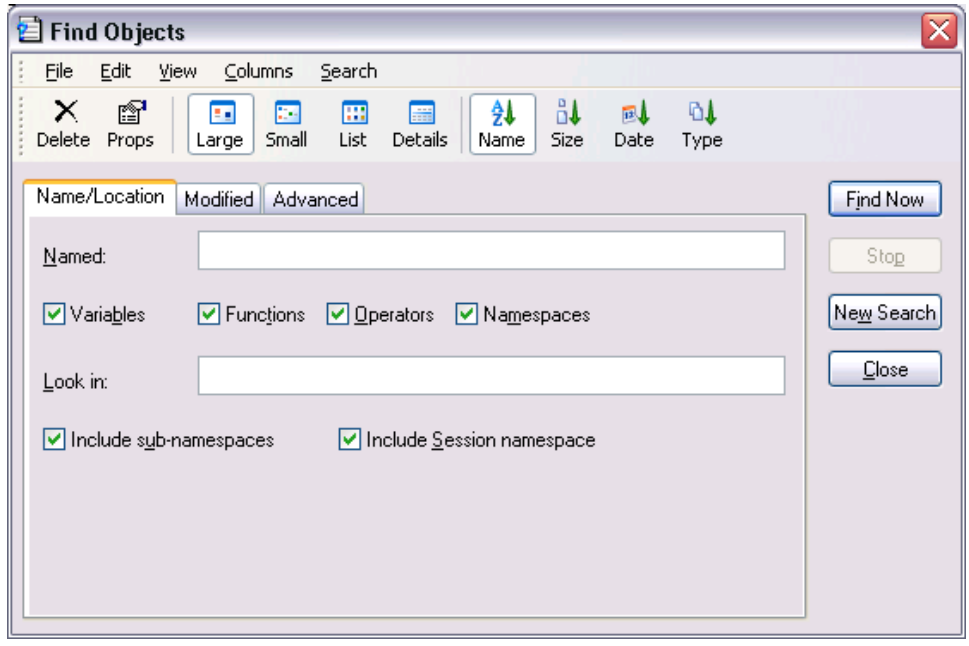
```
      mydt.IsLeapYear 2000
1
```



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.

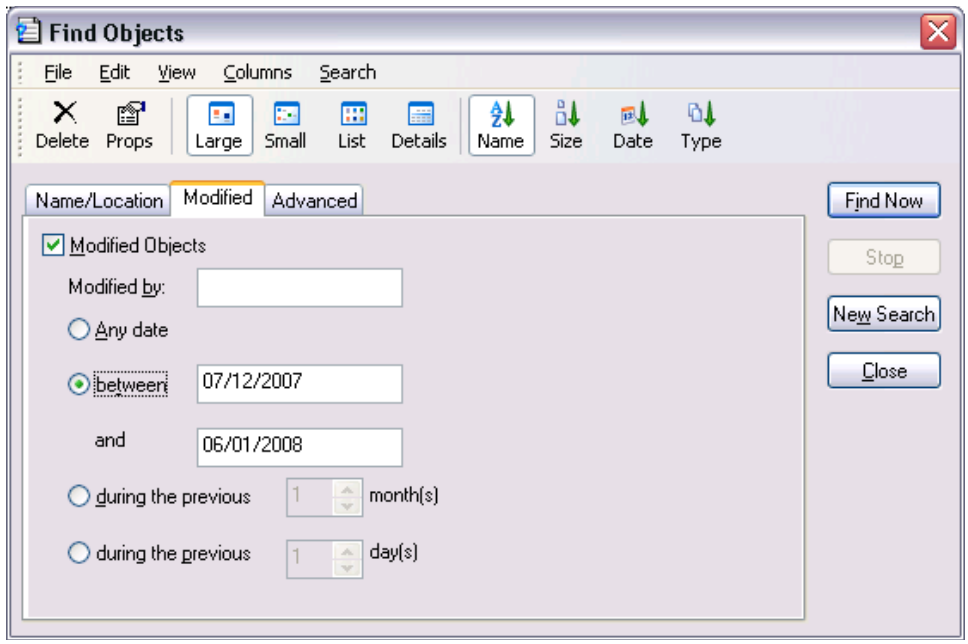


You type the name of the object you wish to find into the field labelled *Named*. To locate all objects beginning with a particular string, enter the string followed by a `'*'` character. For example, if you enter the string `FOO*`, the system will locate all objects whose name begins with `FOO`.

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

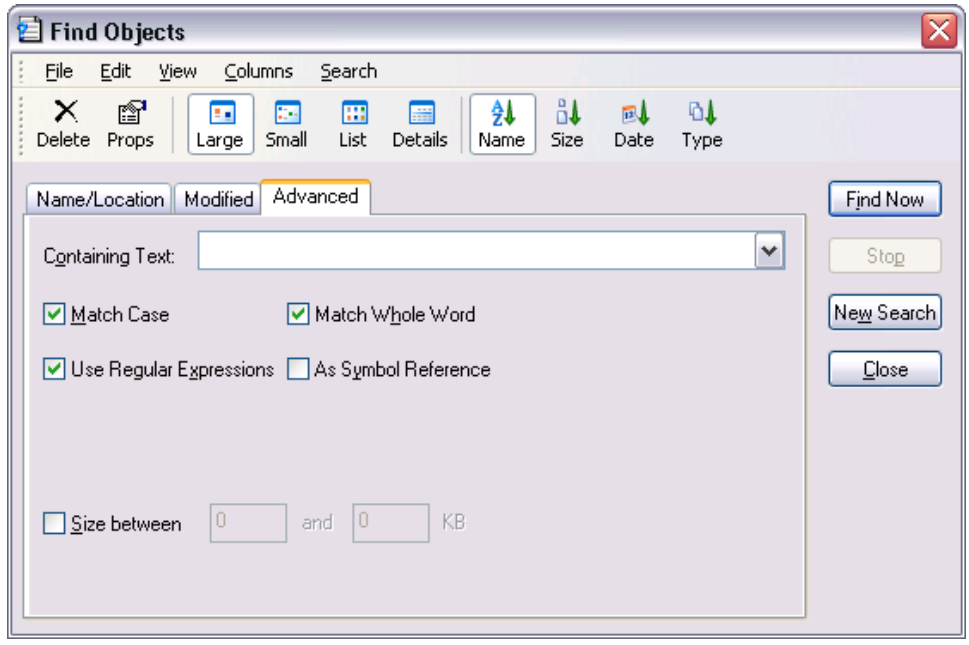


To make the search dependent upon modification, you must check the *Modified Objects* check box.

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.

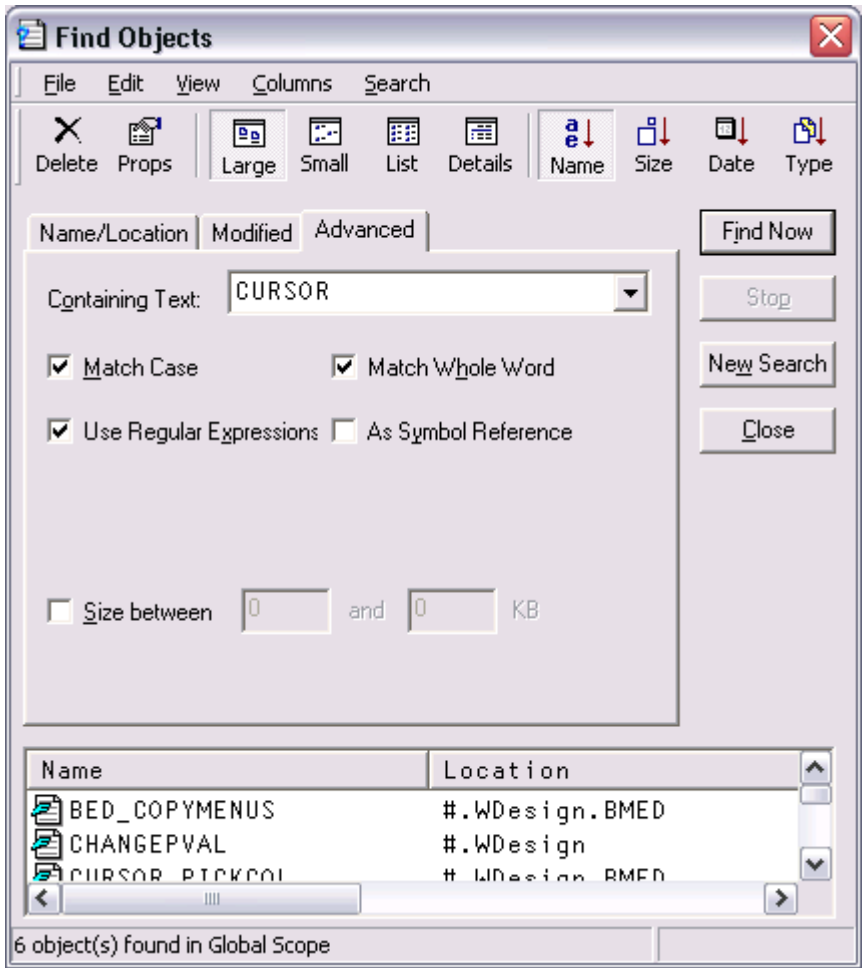
Use Regular Expressions specifies whether or not regular expressions are applicable. For example, if you enter `FOO*` into the field labelled *Containing Text* and check this box, the system will find objects that contain any text string starting with the 3 characters `FOO`. If this box is not checked,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

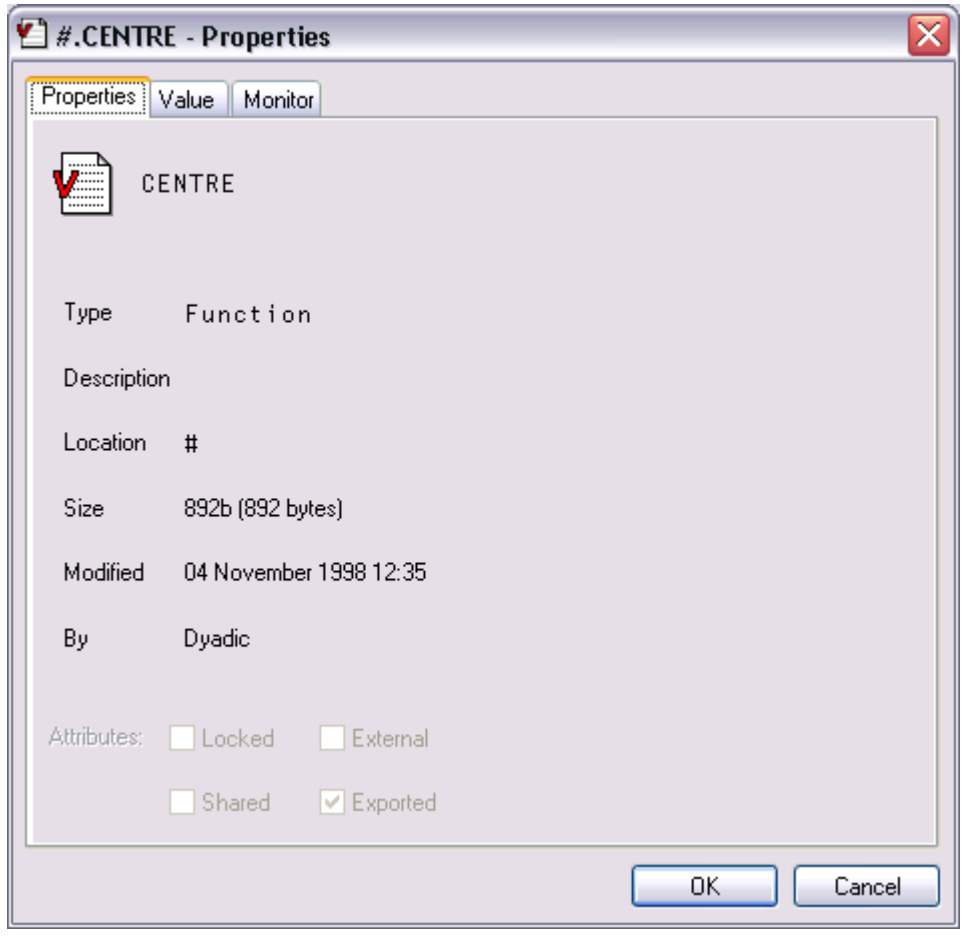
If you wish to specify a completely new set of criteria, press the *New Search* button. This will reset all of the various controls on the 3 pages of the dialog box to their default values.

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.

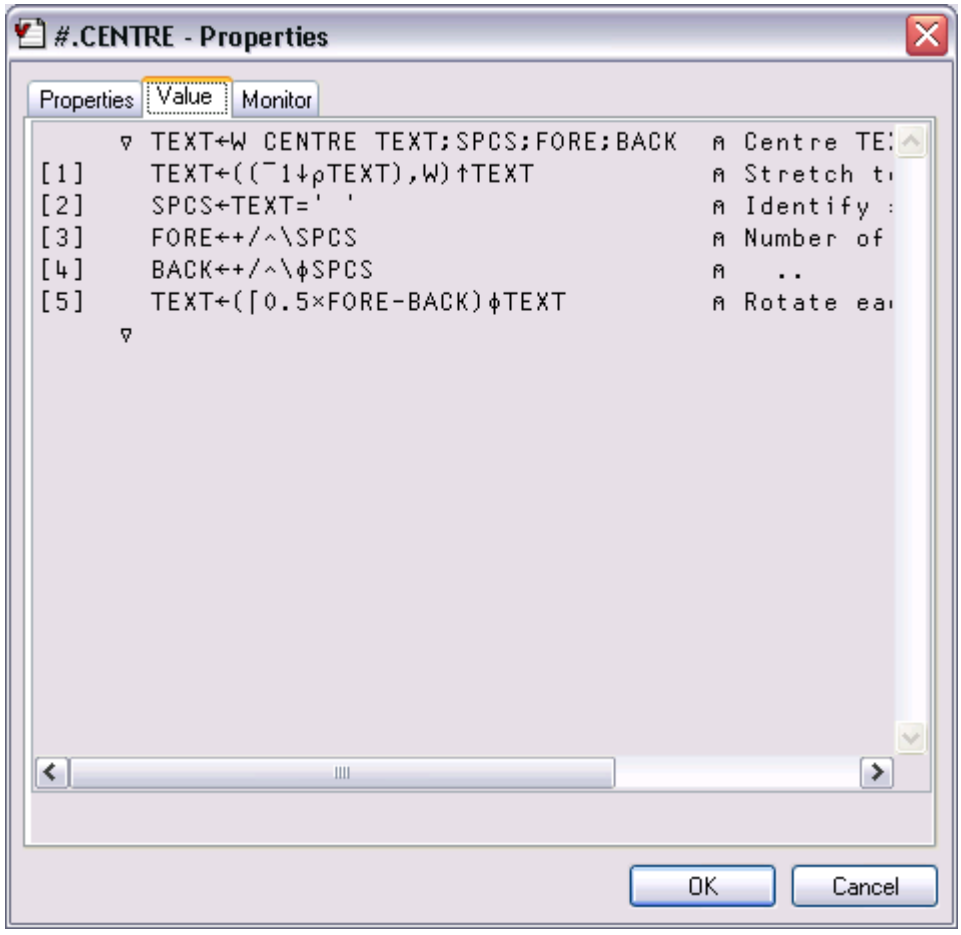
Properties Tab

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.



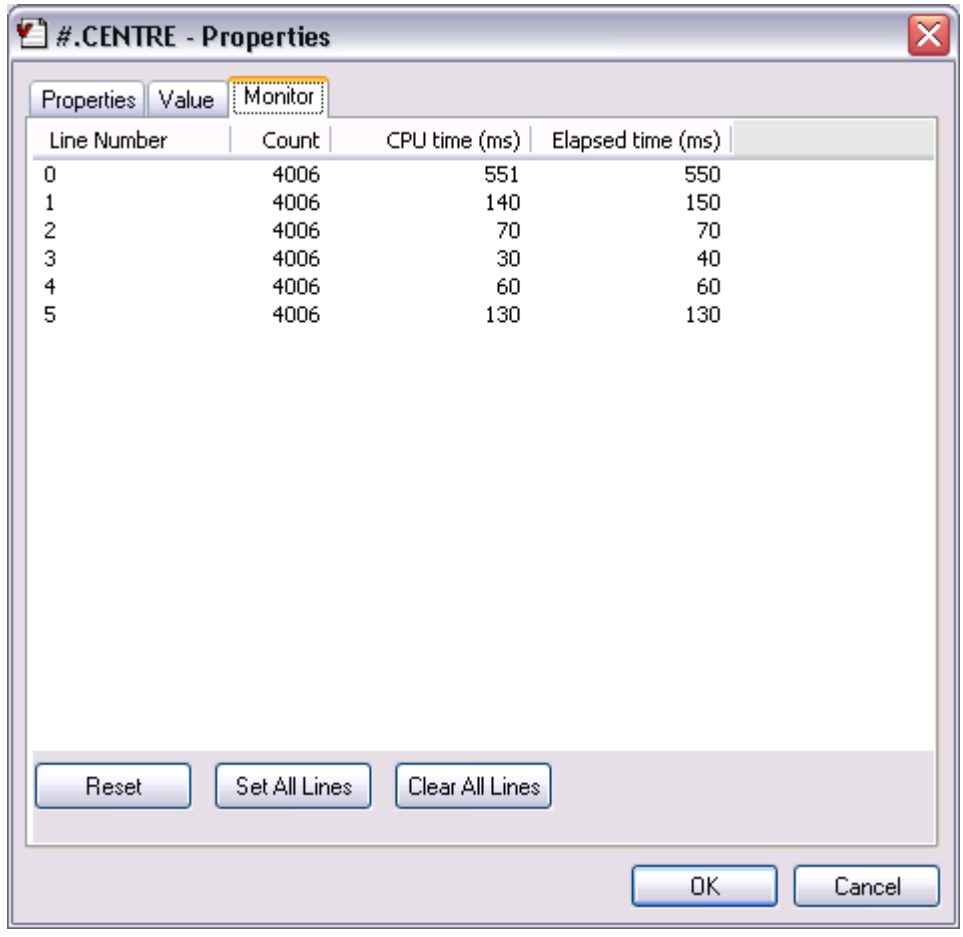
Value Tab

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



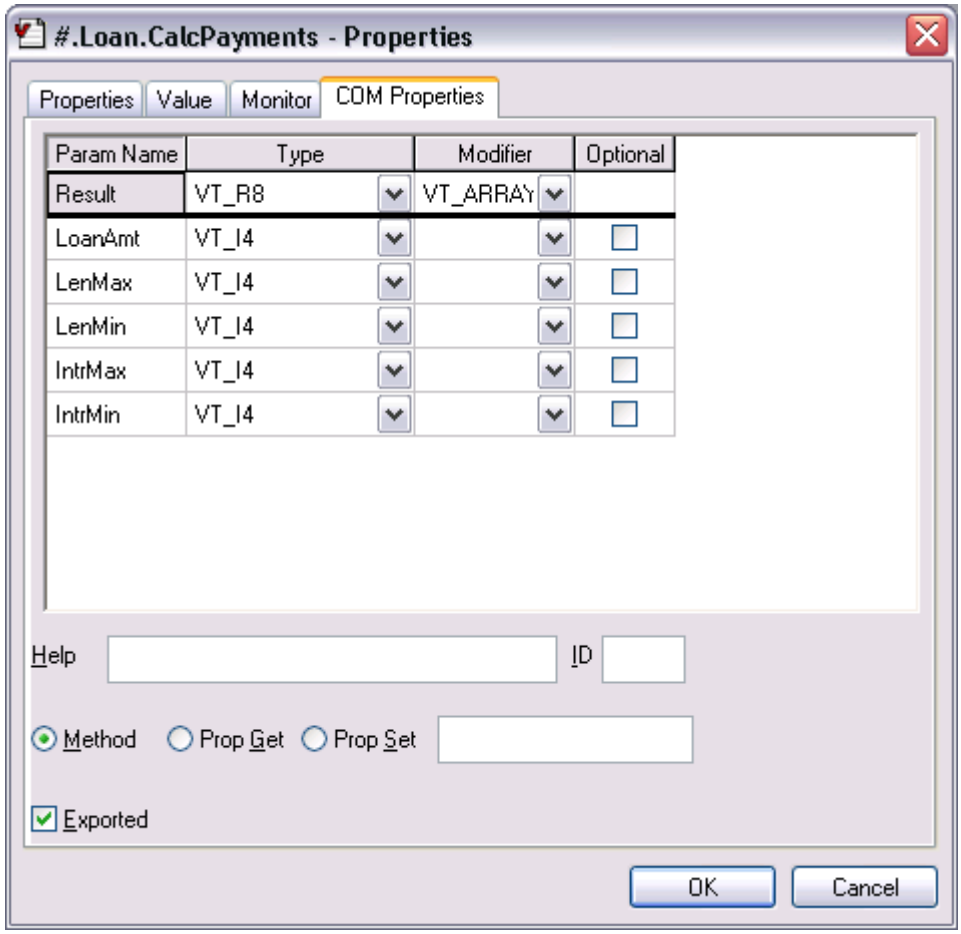
Monitor Tab

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.



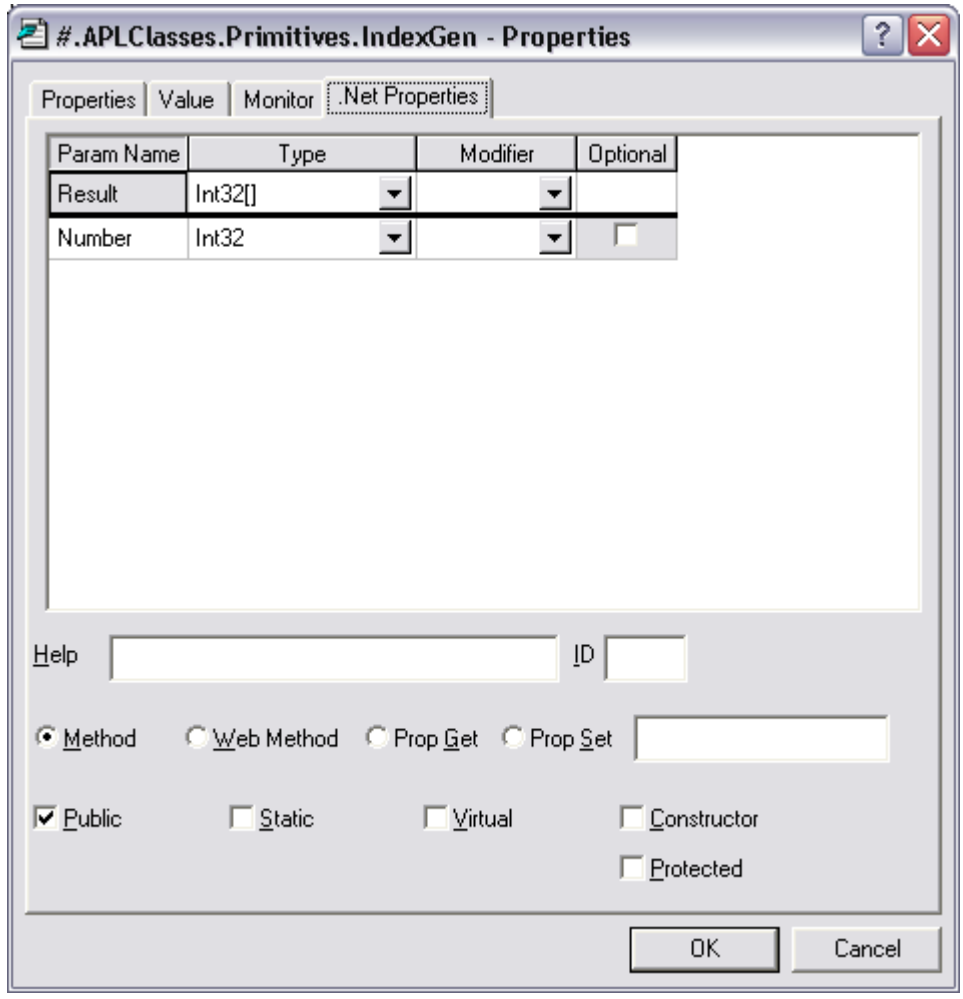
COM Properties Tab

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, Chapters 12 and 13.



Net Properties Tab

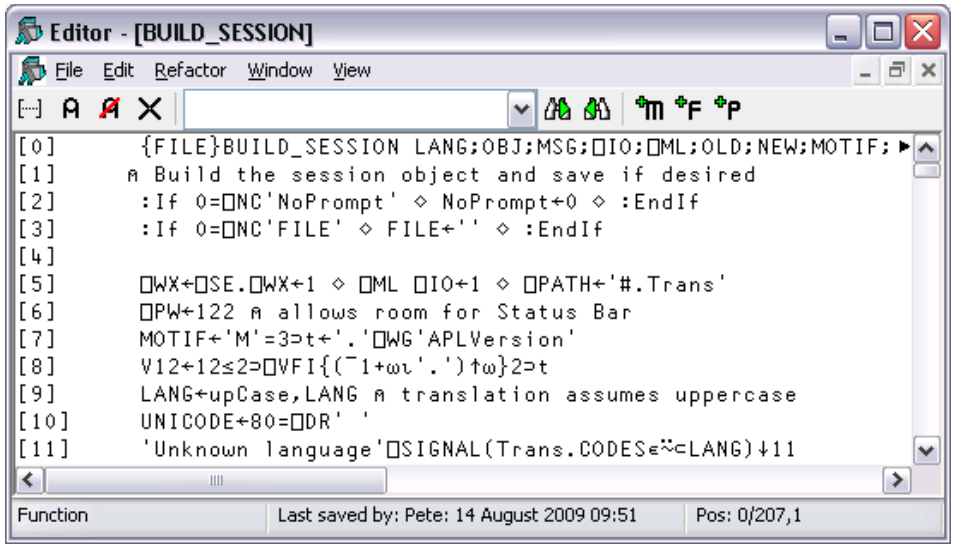
The Net Properties tab applies only to a function in a NetType namespace. The tab is used to define arguments and data types for an exported Method or Property. For further information, see .Net Interface Guide.



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



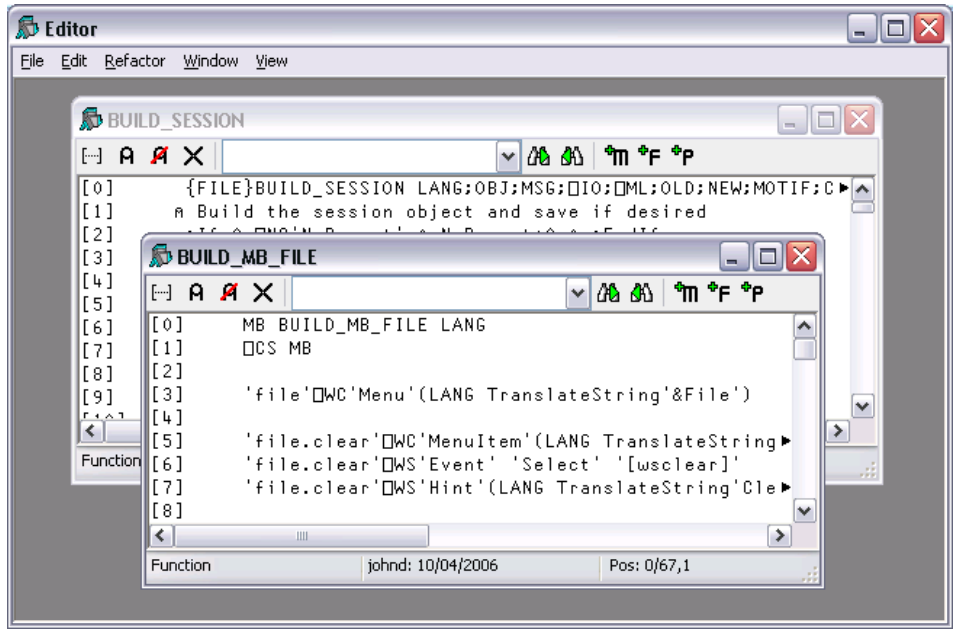
In addition, there is a general *point-and-edit* facility which works in edit and trace windows too. Simply position the input cursor over a name and double-click the left mouse button. Alternatively, you can press Shift+Enter or select Edit from the File menu. The name can appear in the Session, in 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.

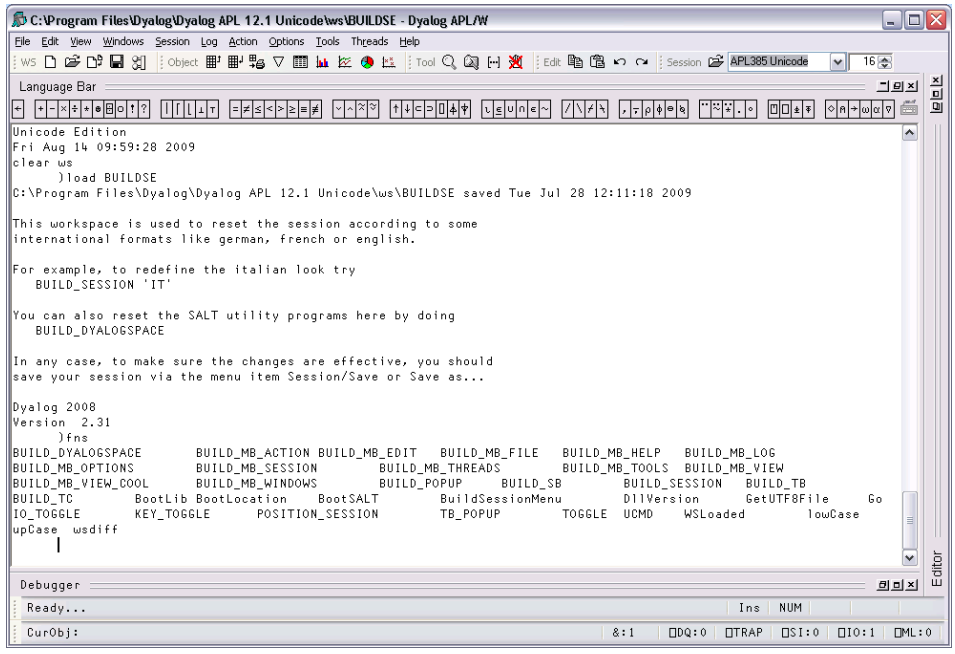
Window Management (Standard)

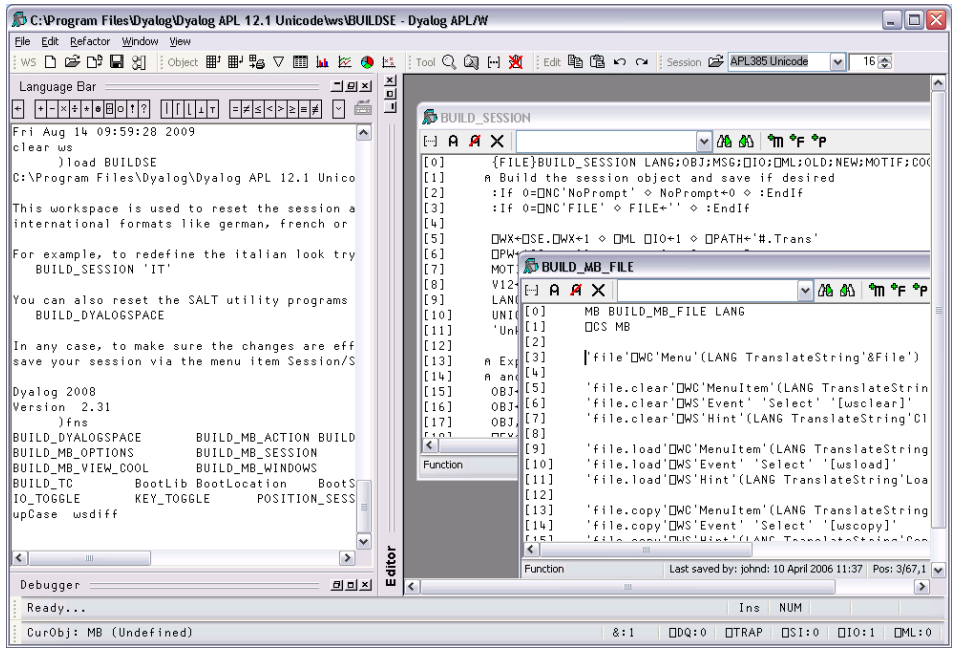
Unless Classic Dyalog mode is selected (*Options/Configure/Trace/Edit*), the Editor is a Multiple Document Interface (MDI) window that may be a stand-alone window, or be docked in the Session window. Each of the objects being edited is displayed in a separate sub-window. Individual edit windows are managed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



The initial size of an edit window is specified by the **edit_rows** and **edit_cols** parameters. The first edit window is positioned at 0 0. Subsequent ones are staggered according to the values of the **edit_offset_y** and **edit_offset_x** 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 overleaf.

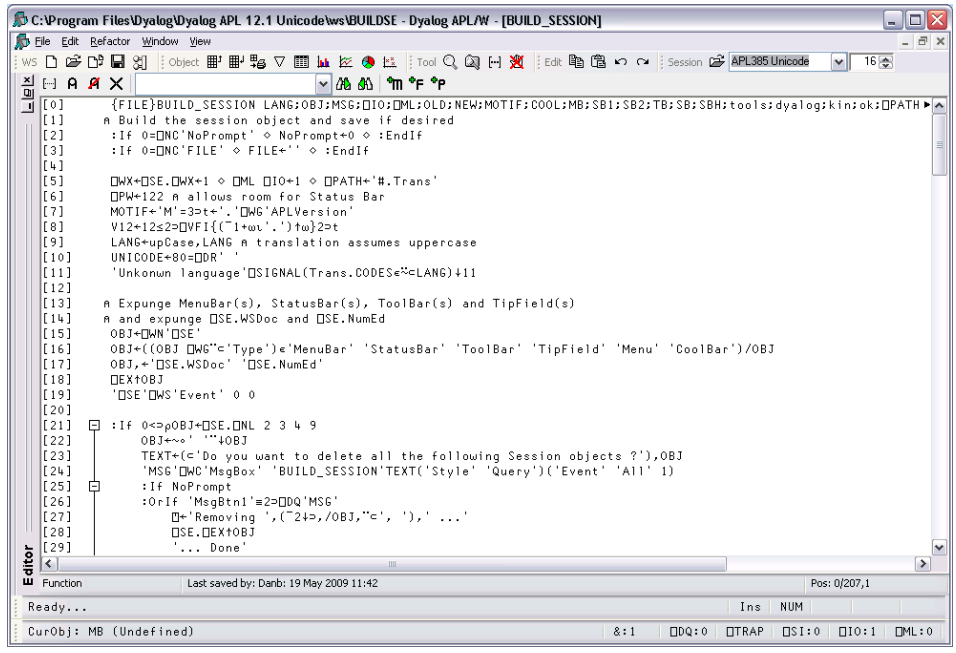

C:\Program Files\Dyalog\Dyalog APL 12.1 Unicode\ws\BUILDSE - Dyalog APL/W
File Edit View Windows Session Log Action Options Tools Threads Help
Language Bar
Unicode Edition
Fri Aug 14 09:59:28 2009
clear ws
)load BUILDSE
C:\Program Files\Dyalog\Dyalog APL 12.1 Unicode\ws\BUILDSE saved Tue Jul 28 12:11:18 2009
This workspace is used to reset the session according to some
international formats like german, french or english.
For example, to redefine the italian look try
BUILD_SESSION 'IT'
You can also reset the SALT utility programs here by doing
BUILD_DYALOGSPACE
In any case, to make sure the changes are effective, you should
save your session via the menu item Session/Save or Save as...
Dyalog 2008
Version 2.31
)fns
BUILD_DYALOGSPACE BUILD_MB_ACTION BUILD_MB_EDIT BUILD_MB_FILE BUILD_MB_HELP BUILD_MB_LOG
BUILD_MB_OPTIONS BUILD_MB_SESSION BUILD_MB_THREADS BUILD_MB_TOOLS BUILD_MB_VIEW
BUILD_MB_VIEW_COOL BUILD_MB_WINDOWS BUILD_POPUP BUILD_SB BUILD_SESSION BUILD_TB
BUILD_TC BootLib BootLocation BootSALT BuildSessionMenu DllVersion GetUTF8File Go
IO_TOGGLE KEY_TOGGLE POSITION_SESSION TB_POPUP TOGGLE UCMD WSLoaded lowCase
upCase wsdiff
Debugger
Editor
Ready...
Ins NUM
CurObj:


C:\Program Files\Dyalog\Dyalog APL 12.1 Unicode\ws\BUILDSE - Dyalog APL/W
File Edit Refactor Window View
Language Bar
BUILD_SESSION
BUILD_MB_FILE
Function
Editor
Debugger
Ready...
Ins NUM
CurObj: MB (Undefined)

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.

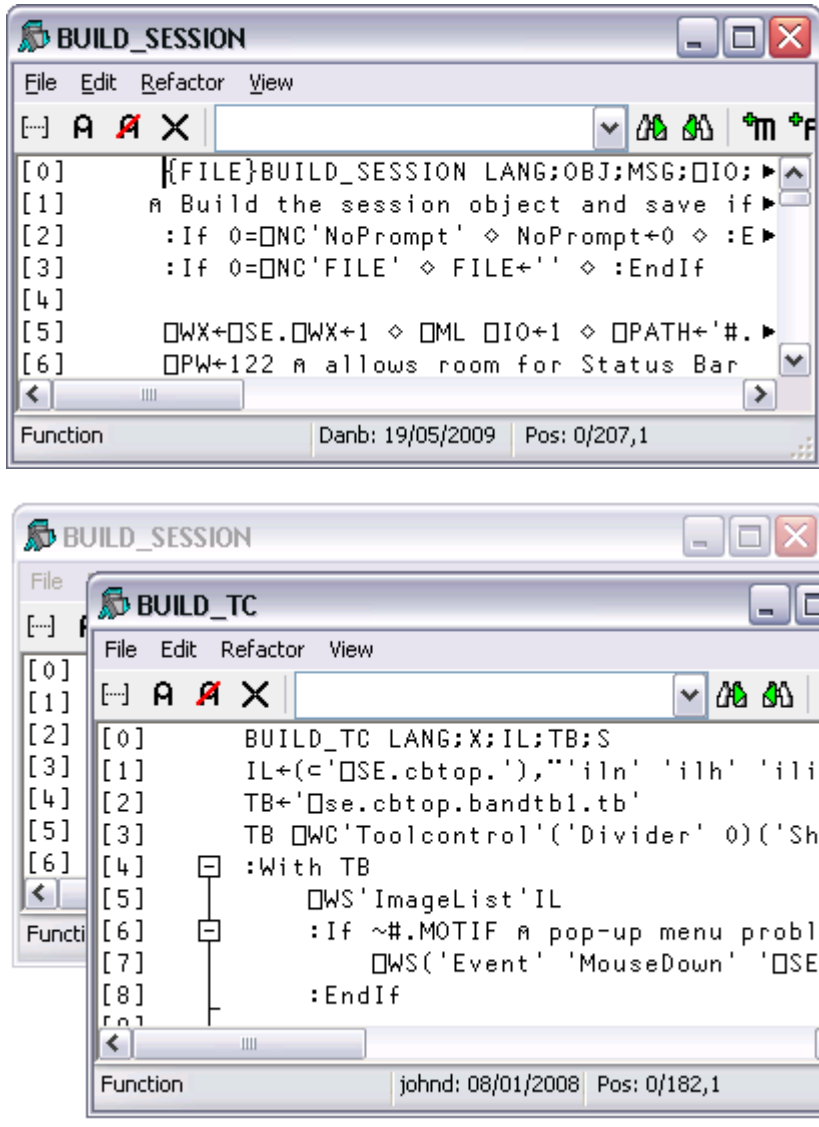


Note that when the Editor has the focus, the Editor menubar is displayed in place of the Session menubar.

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 Edit windows always appear on top of the Session.

The first edit window is created at the position specified by the **edit_first_y** and **edit_first_x** parameters. 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.

Inserts a Method template for the selected name.

Refactor text as method

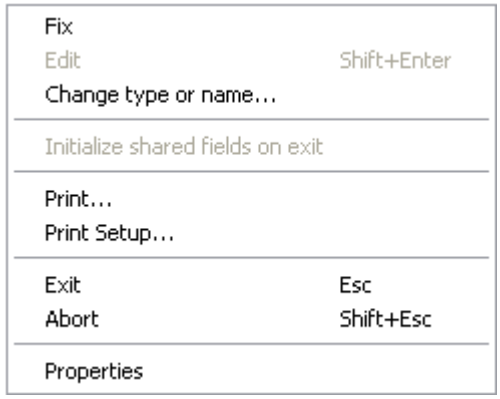
Inserts a Field template for the selected name.

Refactor text as field

Inserts a Property template for the selected name.

Refactor text as property

The File Menu



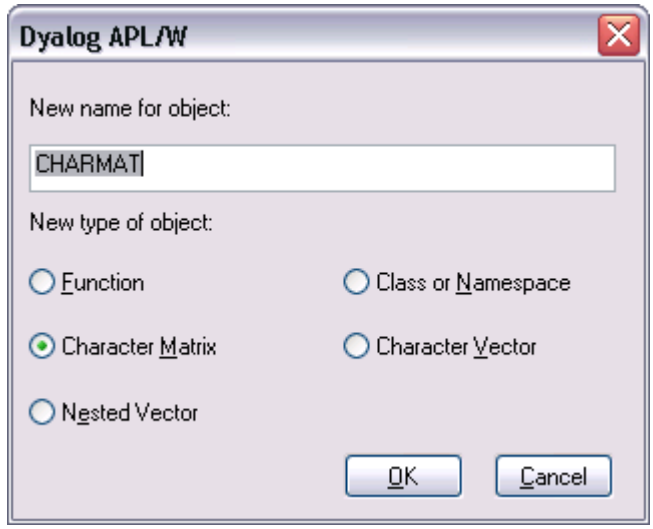
The File Menu

The File menu illustrated above 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.
Edit	Opens an Edit window on the name under the mouse pointer.
Change type or name...	Displays the change type/name dialog box (see below).
Initialize shared fields on exit	
Print	Prints the current contents of the edit window.
Print Setup	Displays the Print Configuration dialog box.
Exit	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.
Abort	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.
Properties	Displays the Object Properties dialog box for the current object.

Change Type or Name

This dialog box allows you to change the name of an object from the editor, or alter its type.



Initialize shared fields on exit

When you fix a Class, the Editor will optionally initialise the shared Fields defined in your Class script. This is typically appropriate when you first define a Class, but may be undesirable when you subsequently edit the Class during debugging. This option allows you to control this behaviour.

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:-

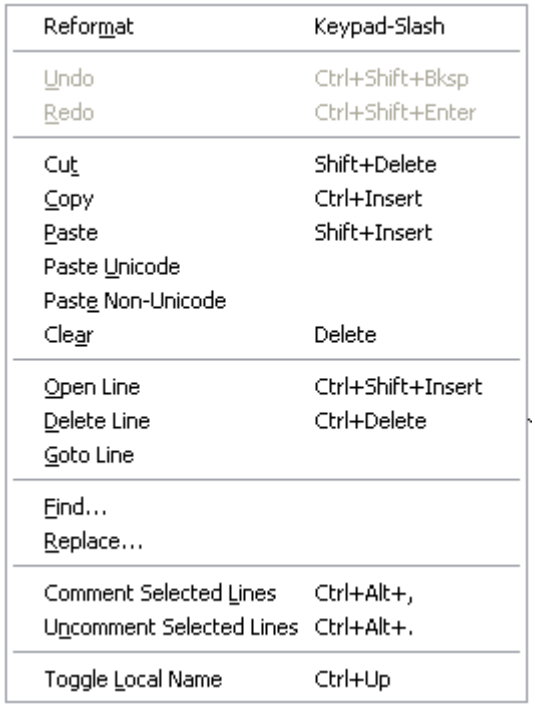
```
:Field shared public List {gets} {zilde}
```

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. When you exit the editor you DO NOT want List reset back to the empty vector.

This option is by default checked for edit windows, and un-checked for trace windows.

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.



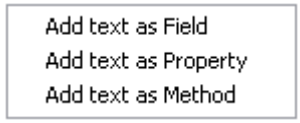
The Edit Menu

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

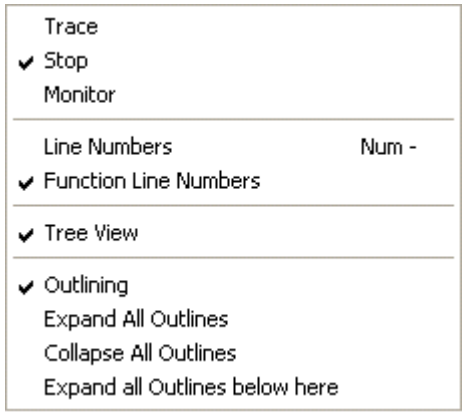


The Refactor Menu

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.

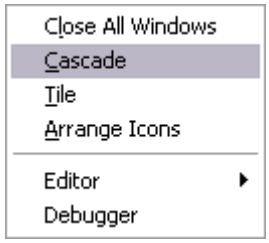
Function Line Numbers

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 *Paste* 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

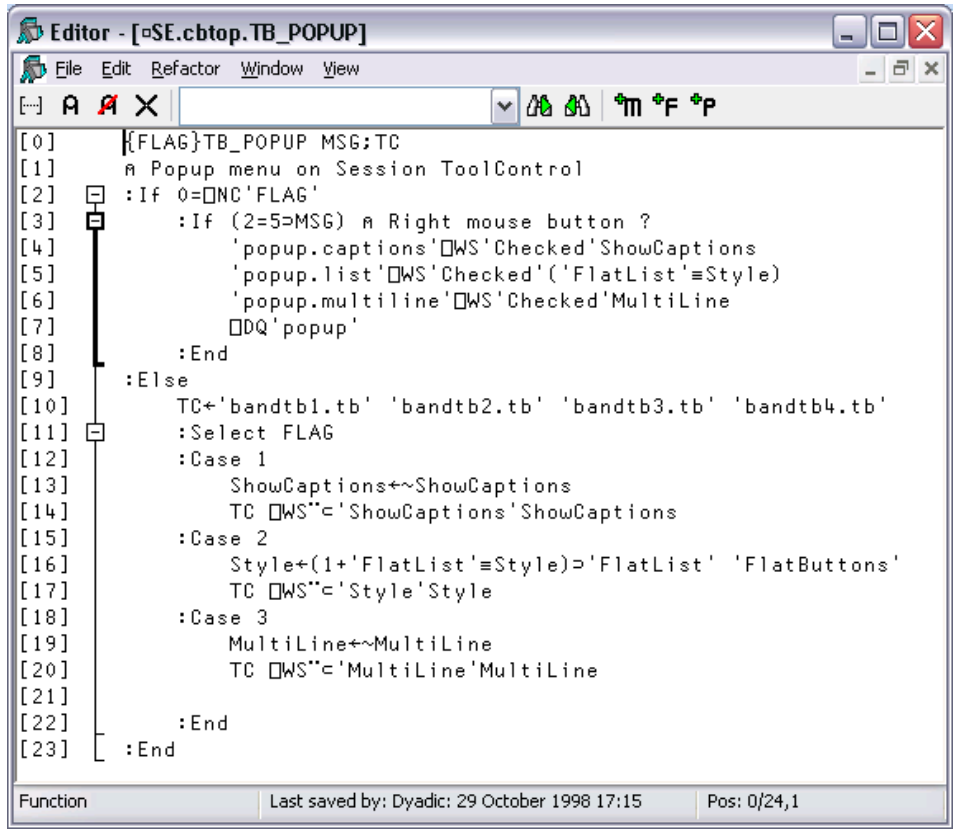
The TL command (which by default is mapped to Ctrl+Up) toggles the localisation of the name under the cursor. If the name is currently global, pressing Ctrl+Up causes the name to be added to the list of locals in the function header. If the name is already localised, pressing Ctrl+Alt+l removes it from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

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

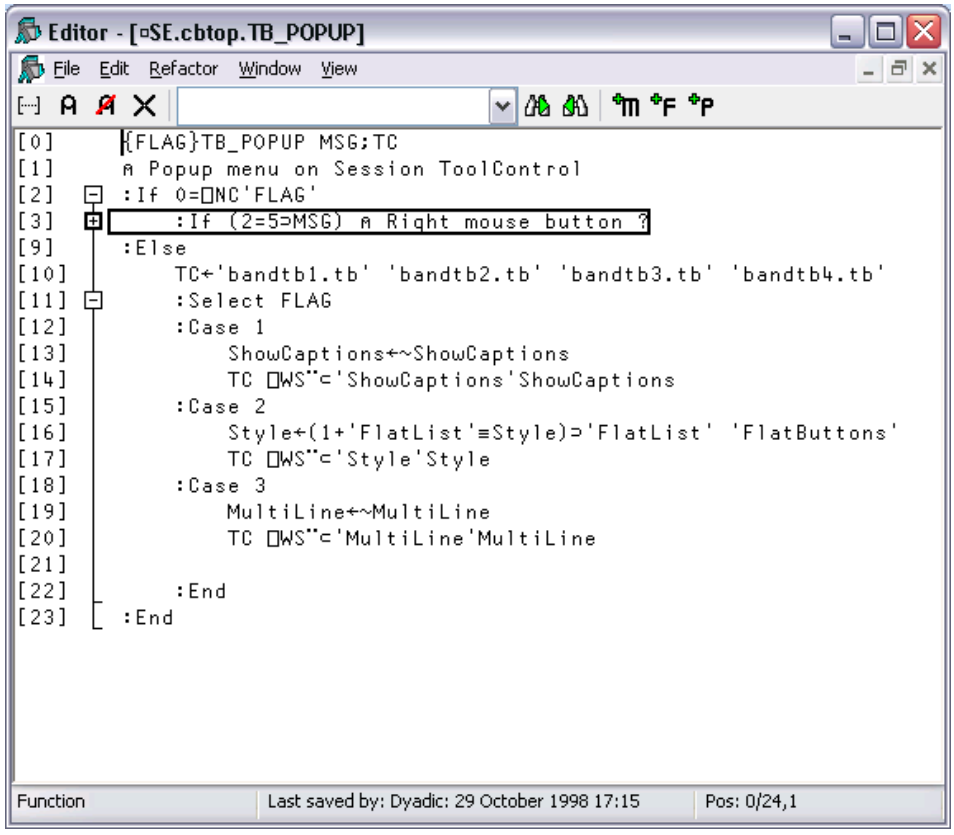
The picture below shows the result of opening the function ⎕SE.cbtop.TB_POPUP.

```
)ed ⎕SE.cbtop.TB_POPUP
```



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

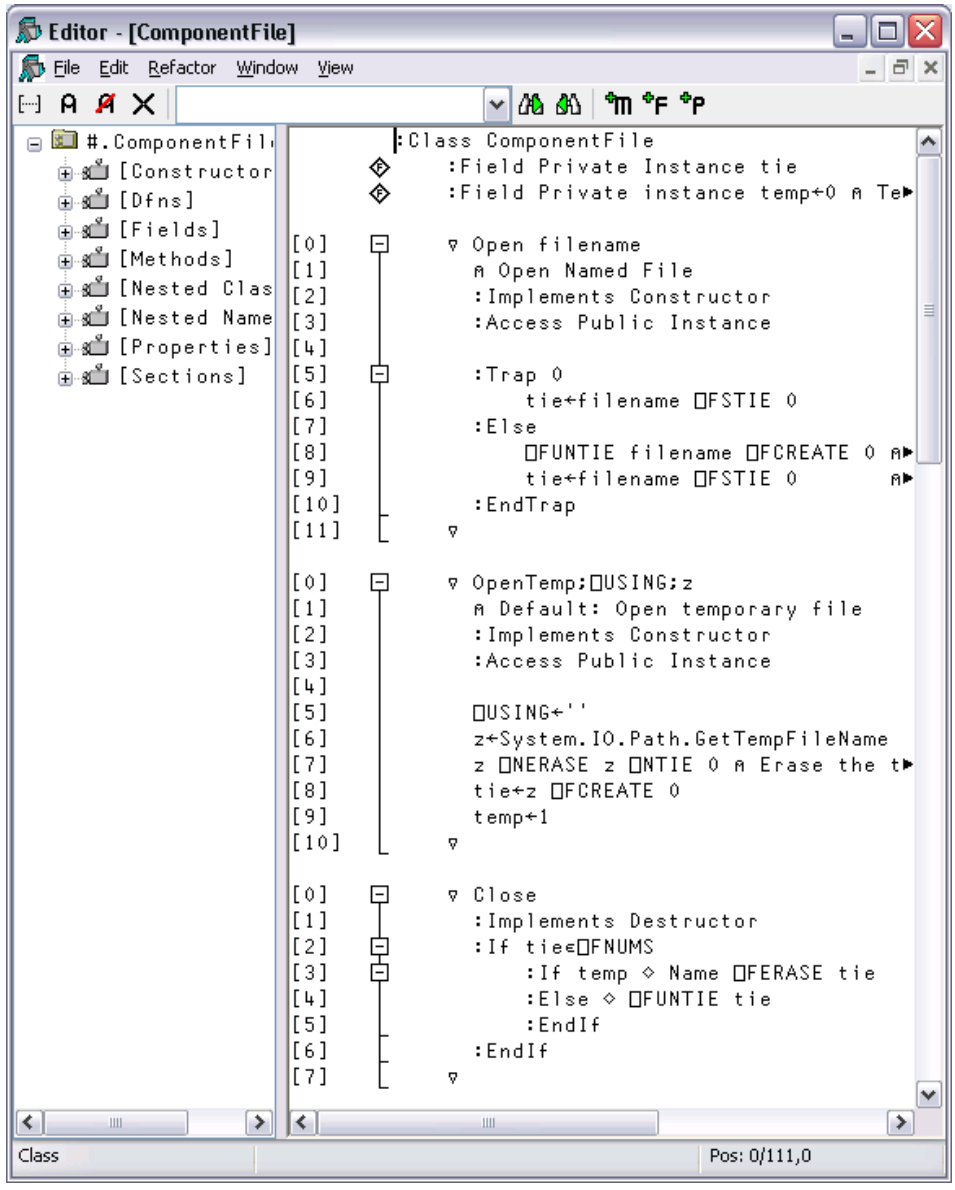
- When you hover the mouse pointer over one of the boxes that mark the start of a block , the line marking the extent of that block becomes highlighted, as shown above.
- If you click on a ⊟ box, the corresponding section collapses, so that only the first line of the block is displayed, as shown below.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



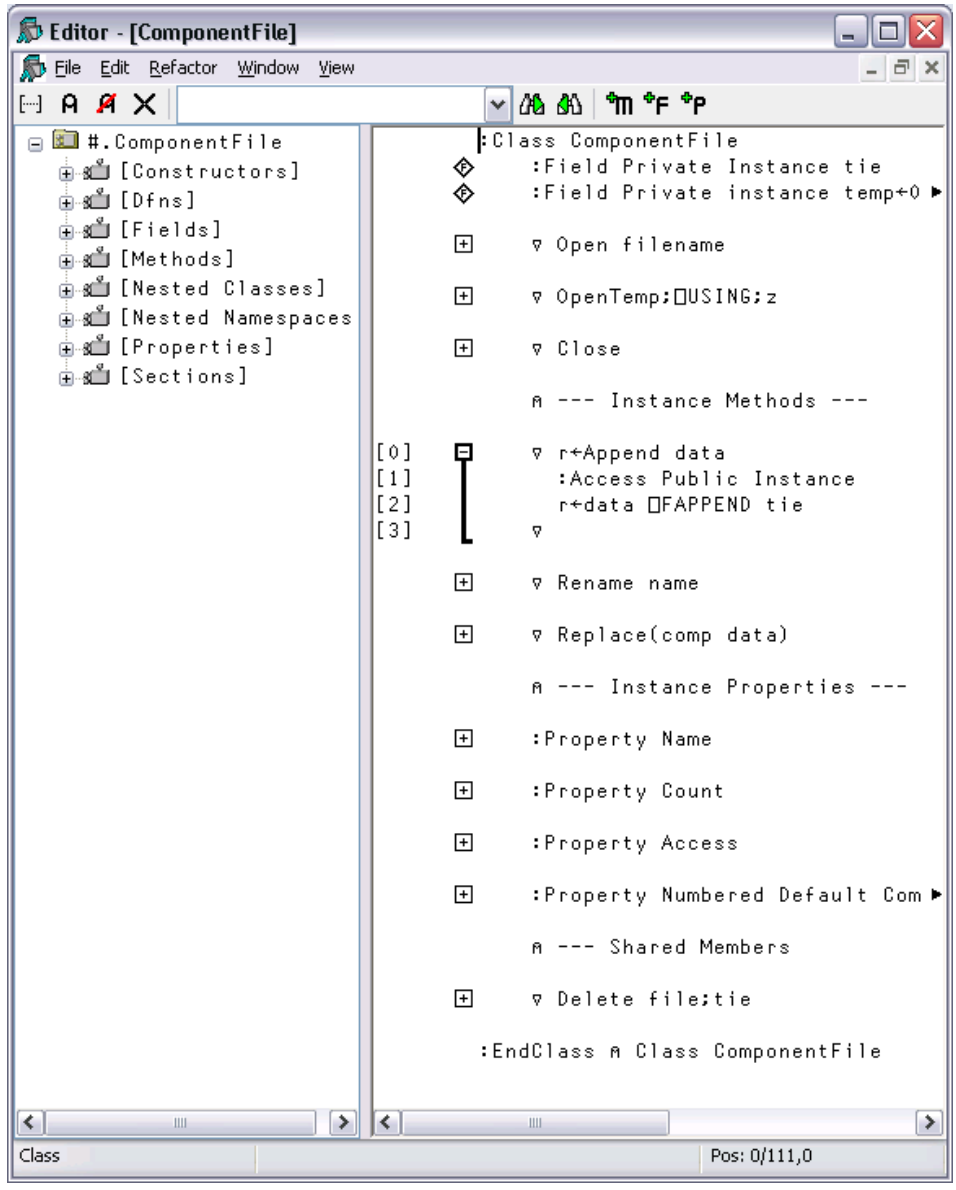
Editing Classes

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.

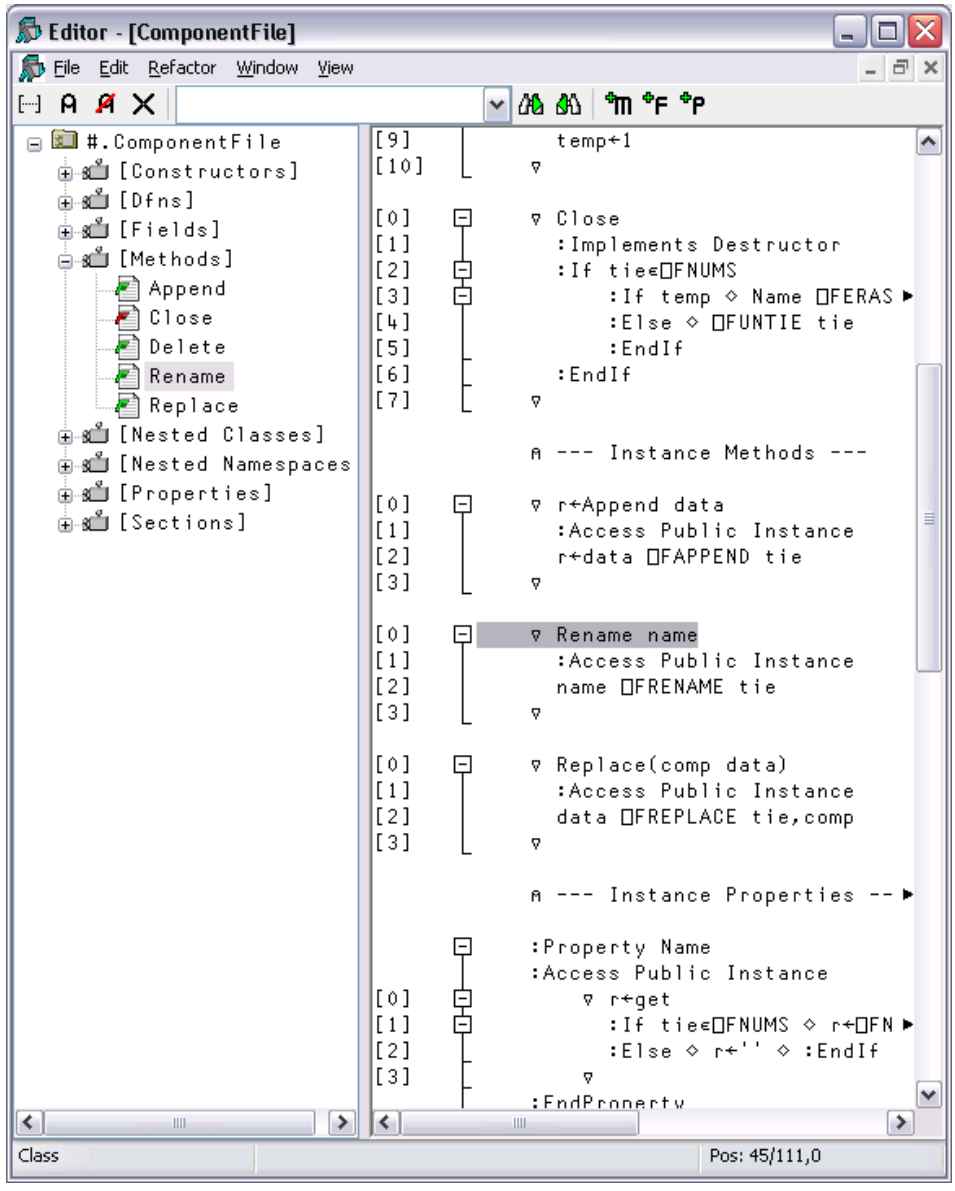
```
)ed ComponentFile
```


Editor - [ComponentFile]
File Edit Refactor Window View
#.ComponentFil
[Constructor
[Dfns]
[Fields]
[Methods]
[Nested Clas
[Nested Name
[Properties]
[Sections]
:Class ComponentFile
:Field Private Instance tie
:Field Private instance temp←0 ⍝ Te
[0] ∇ Open filename
[1] ⍝ Open Named File
[2] :Implements Constructor
[3] :Access Public Instance
[4]
[5] :Trap 0
[6] tie←filename ⎕FSTIE 0
[7] :Else
[8] ⎕FUNTIE filename ⎕FCREATE 0 ⍝
[9] tie←filename ⎕FSTIE 0 ⍝
[10] :EndTrap
[11] ∇
[0] ∇ OpenTemp;⎕USING;z
[1] ⍝ Default: Open temporary file
[2] :Implements Constructor
[3] :Access Public Instance
[4]
[5] ⎕USING←''
[6] z←System.IO.Path.GetTempFileName
[7] z ⎕NERASE z ⎕NTIE 0 ⍝ Erase the t
[8] tie←z ⎕FCREATE 0
[9] temp←1
[10] ∇
[0] ∇ Close
[1] :Implements Destructor
[2] :If tie∈⎕FNUMS
[3] :If temp ◇ Name ⎕FERASE tie
[4] :Else ◇ ⎕FUNTIE tie
[5] :EndIf
[6] :EndIf
[7] ∇
Class
Pos: 0/111,0

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.

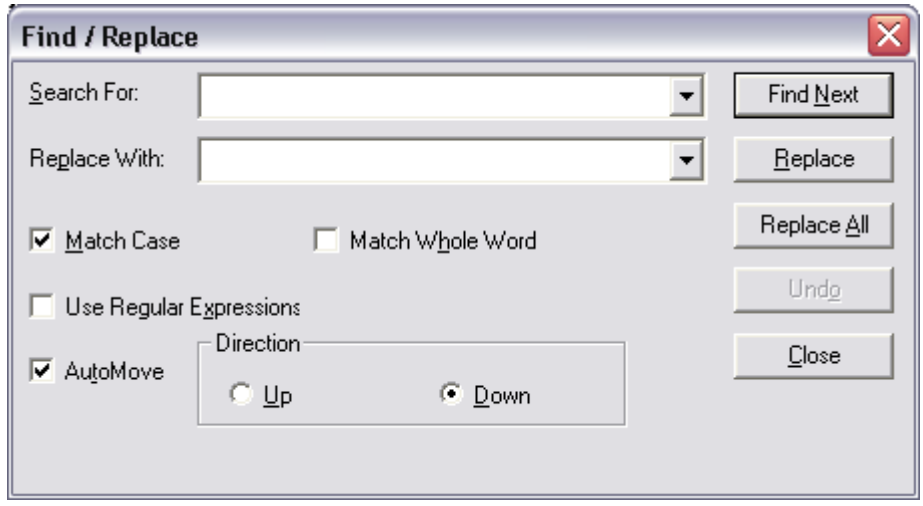


When you edit a class, a separate treeview is optionally displayed in the left pane to make it easy to navigate within the class. When you click on a name in the treeview, the editor automatically scrolls the appropriate section into view (if necessary) and positions the edit cursor at its start. The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



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.

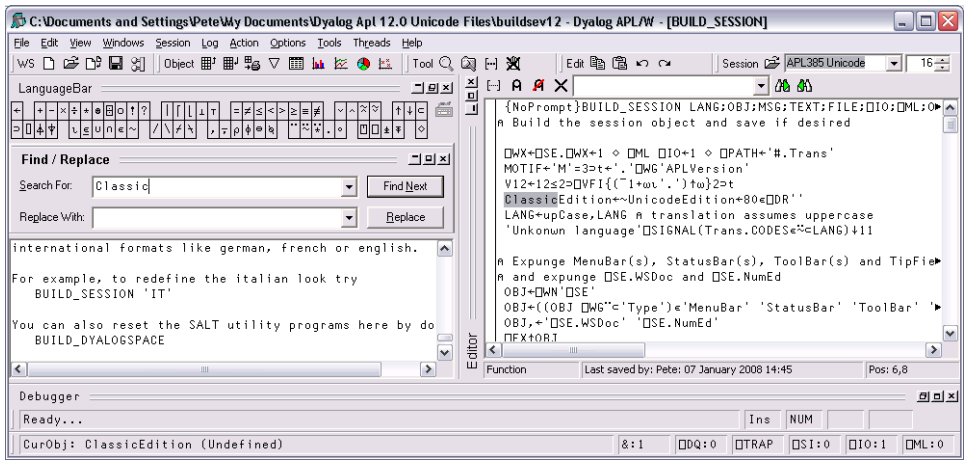
Use Regular Expressions Check this box if you want to use various *wild card* symbols.

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.

Docking the Find/Replace Dialogs

You may dock the Find or Find/Replace dialog boxes in the Session window. If you do so, they are displayed in a slightly abbreviated form, for economy of space. The picture below illustrates the effect of docking the Replace dialog box along the top edge of the Session.



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 then have two further choices:

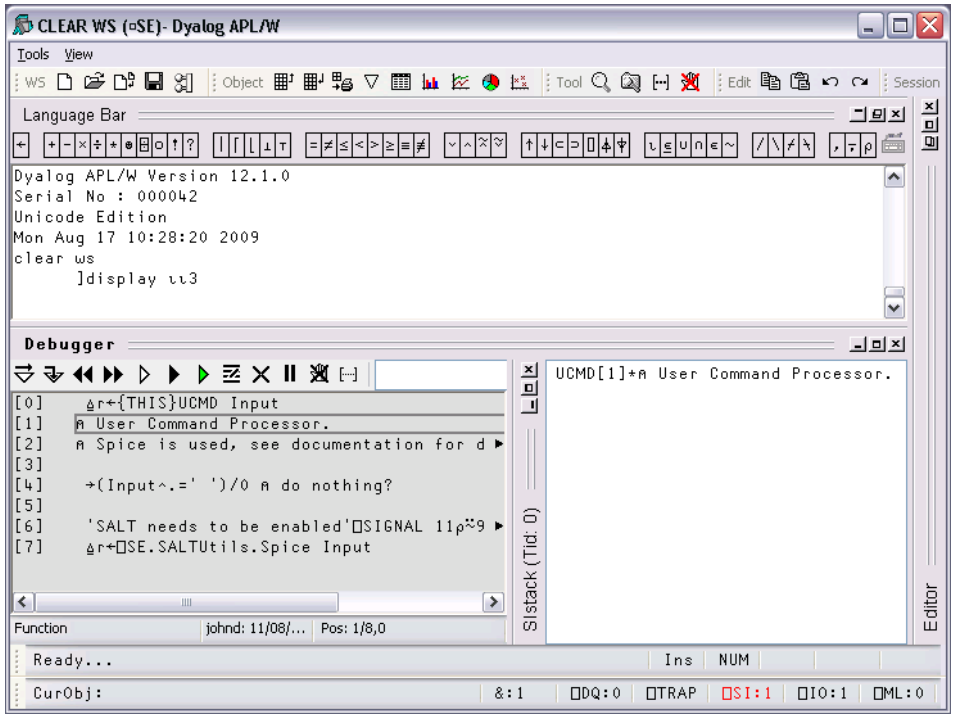
- to have the Tracer operate in multiple windows or in a single window
- to have the Trace window(s) dependant or independent of the Session window.

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.

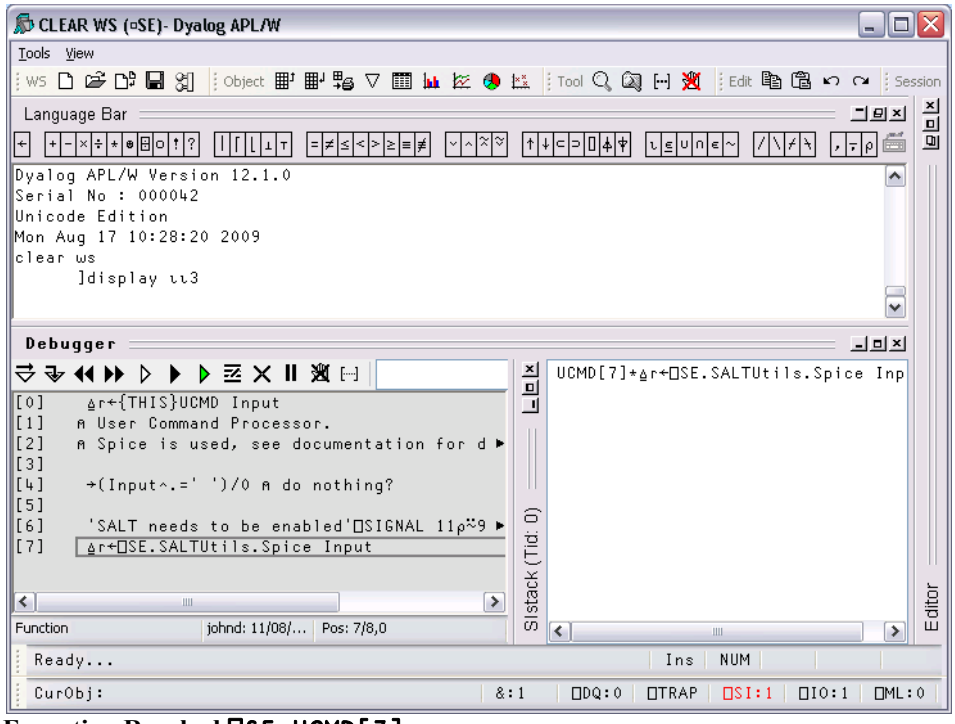
Trace Tools

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 threads			Restarts execution for all threads, closing all trace windows
	Contin ue	BH		Continues execution of the current thread,
	Edit	ED	Shift+Enter	Invokes the Editor
	Exit	EP	Esc	Closes the Trace window, exits the current function
	Intr		Ctrl+Pause	Interrupts execution
	Reset	CS		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.

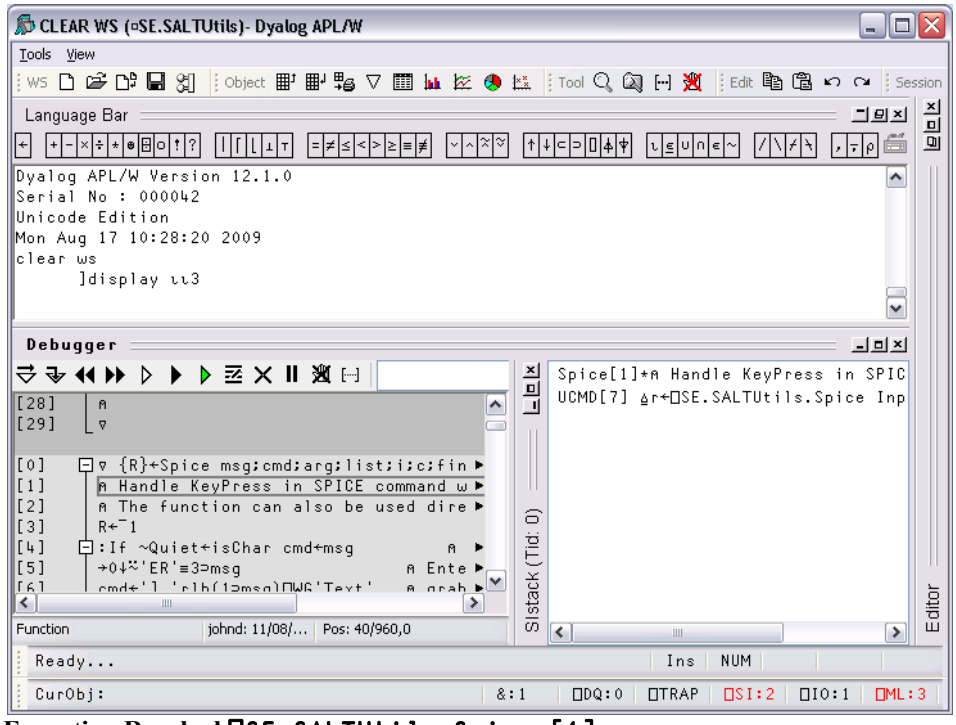
The illustration below shows the state of execution having clicked *Exec* 6 times to reach `⎕SE.UCMD[7]`.



Execution Reached `⎕SE.UCMD[7]`

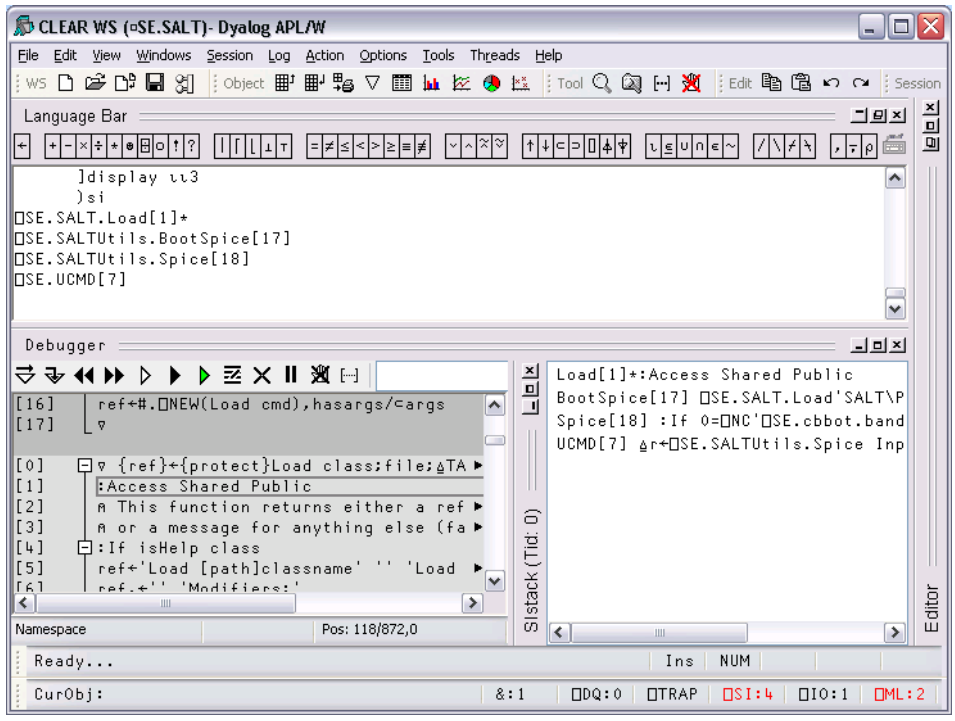
The next illustration shows the result of clicking *Trace* at this point. This caused the system to **trace into** `⎕SE.SaltUtils.Spice`, the function called from `⎕SE.UCMD[7]`.

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 sideways, the cursor is positioned to the left of the suspended line in the top Trace window. If you press Shift+Enter (ED) with the cursor in this position, the trace window becomes an edit window allowing you to edit the function or operator on top of the stack. You can achieve the same thing by selecting *Edit* from the *File* menu, but the input cursor MUST again be in the left-most (empty) column, or the system will attempt to open an edit window for the name under the cursor (point-and-edit).

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

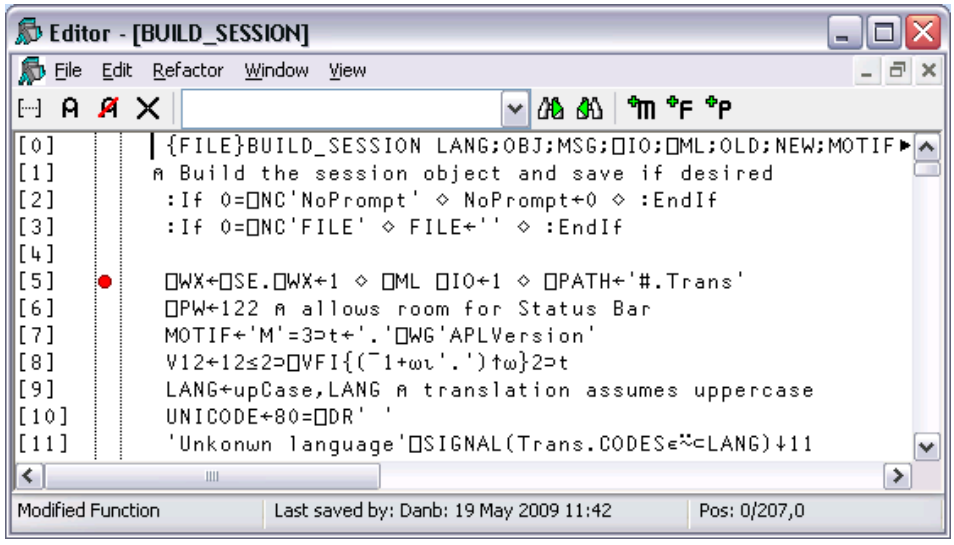
You may also open a new edit window from within the Tracer using point-and-edit.

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.

There are two further options, namely *Single Trace Window* and *Independent Trace Stack*.

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.

Dependent Trace Stack

If you **deselect** the *Independent trace stack* option, trace windows are *owned by* the Session window and, as a consequence, are always shown on top of it. This reflects the behaviour of Dyalog APL prior to Version 8.2.3, and is the default.

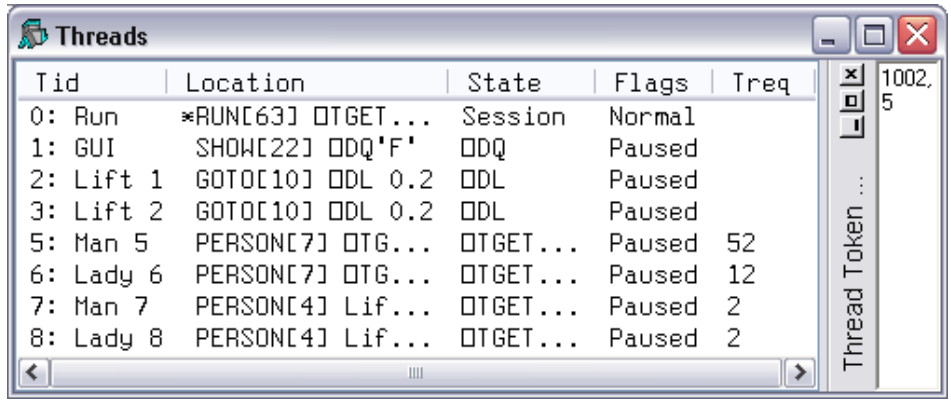
Independent Trace Stack

If you **select** the *Independent trace stack* option, trace windows are *independent of* the Session window and so go behind it when the Session has the focus. Furthermore, the top trace window is a top-level window in its own right and is therefore represented by its own button in the Windows Taskbar. You can switch focus between the session and top trace window in various ways:

- If any part of the target window is visible, click on it with the mouse.
- Click on its associated button in the Windows Taskbar.
- Use Ctrl-Tab to cycle within Dyalog APL application windows.
- Use Alt-Tab to cycle around all applications.

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`)

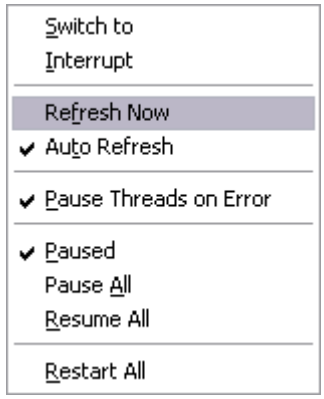
Thread States

State	Description
Pending	Not yet running
Initializing	Not yet running
Defined function	Between lines of a defined function
Dynamic function	Between lines of a dynamic function
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

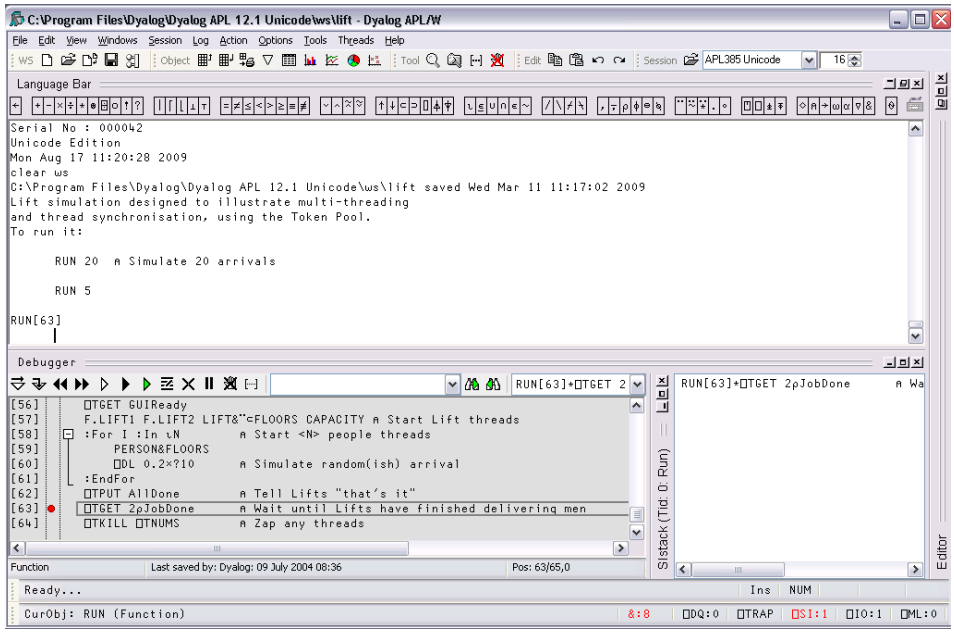


The Pop-up Menu

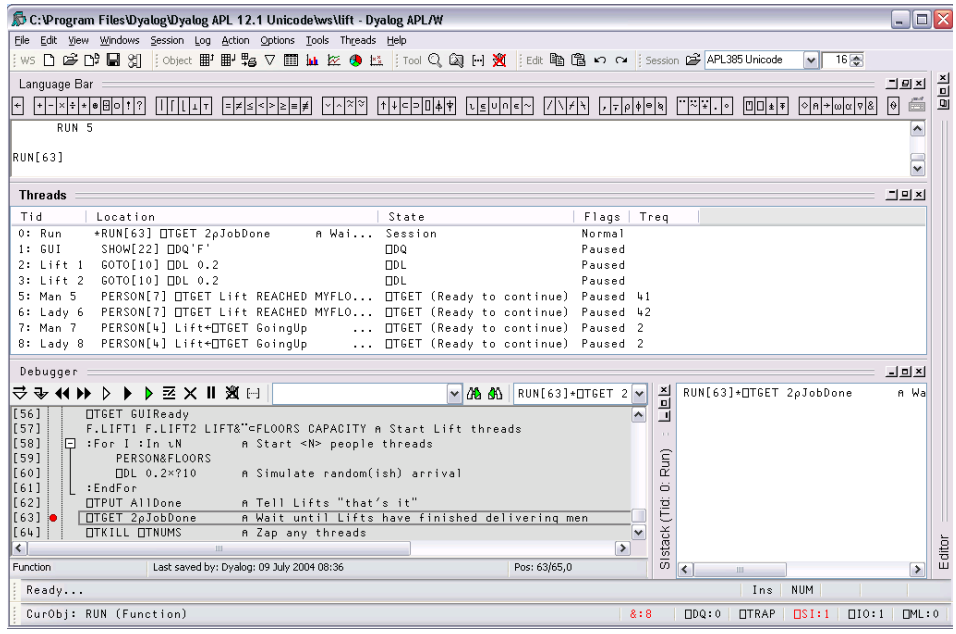
Switch to	Selecting this item causes APL to attempt to suspend (if necessary) and switch to the selected thread, connecting it to the Session and Debugger windows.
Refresh Now	Refreshes the Threads Tool display to show the current position and state of each thread.
Auto Refresh	Selecting this item causes 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

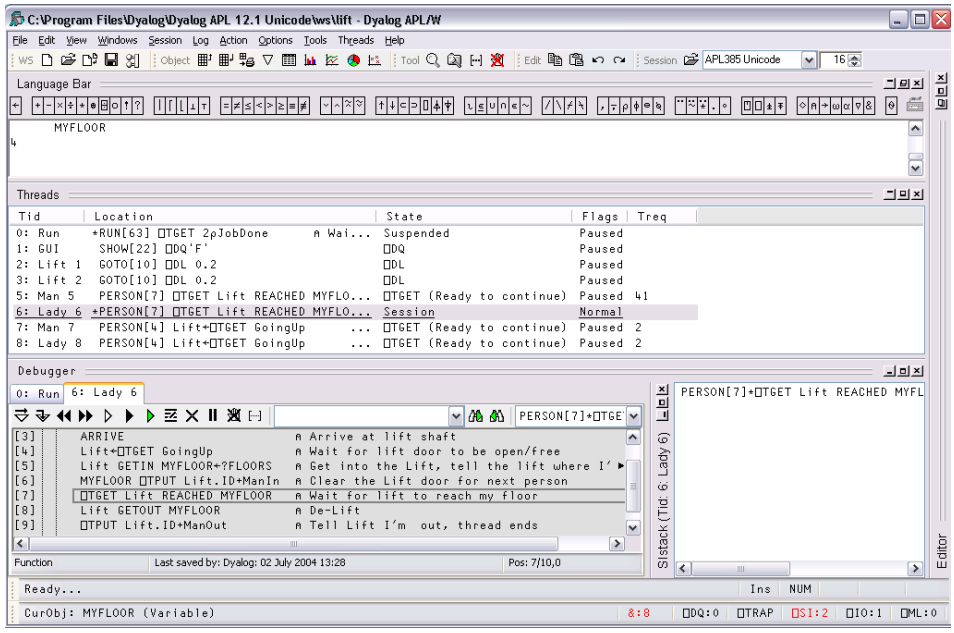
The Debugger provides a tabbed interface that allows you to easily switch between suspended threads for debugging purposes. To keep things simple for non-threaded applications, Tabs are only displayed if there is a thread suspended that is other than Thread 0. The following picture shows the Debugger 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).

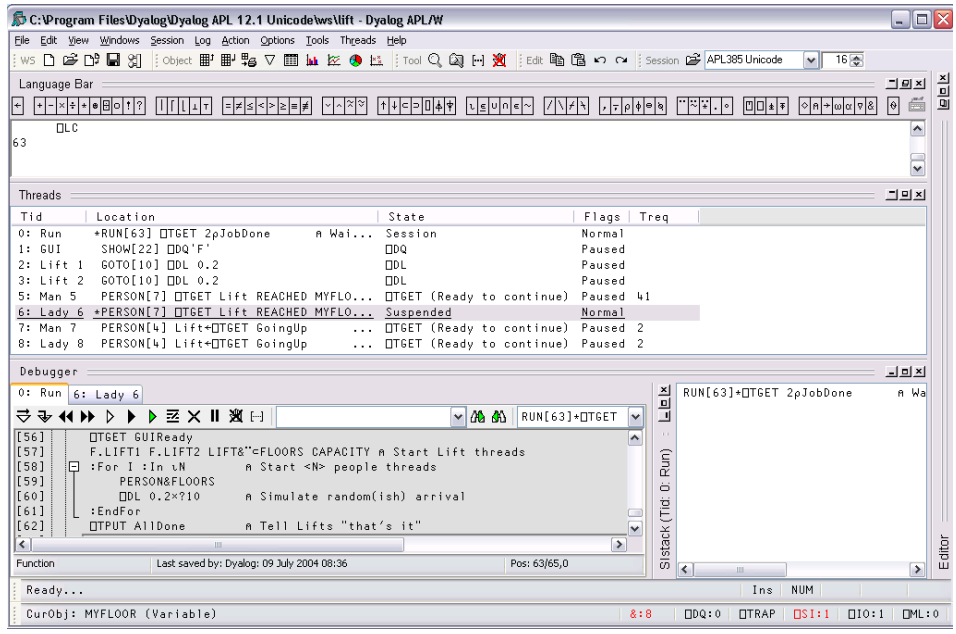


The user then uses the context menu to *Switch To* Thread 6 (whose name is *Lady 6*) which was Paused on `PERSON[7]` in the middle of a `⎕TGET`. The act of switching to this thread caused it to be suspended at the beginning of its current line `PERSON[7]` and the Debugger now displays two Tabs to represent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

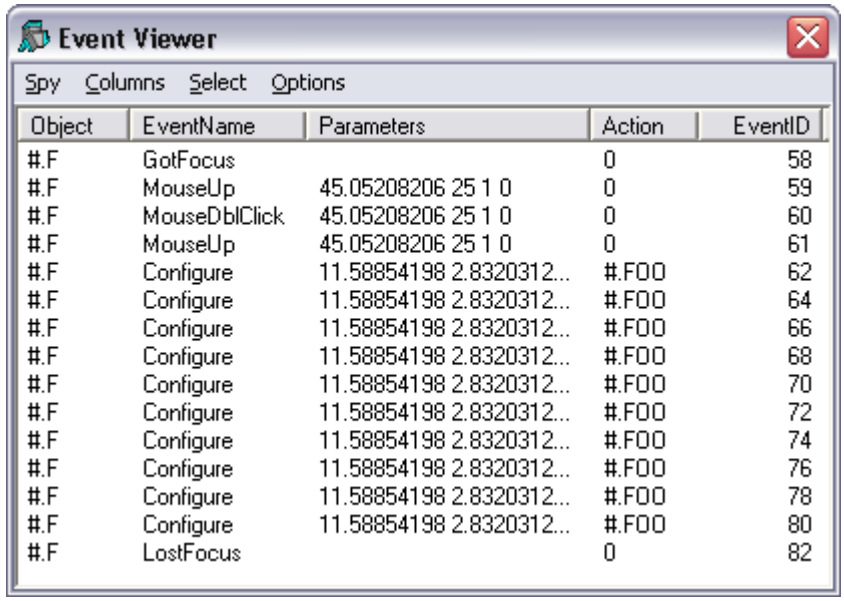


The Event Viewer

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.

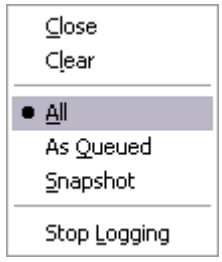
You can choose:

- 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.

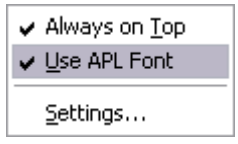
The Select Menu

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

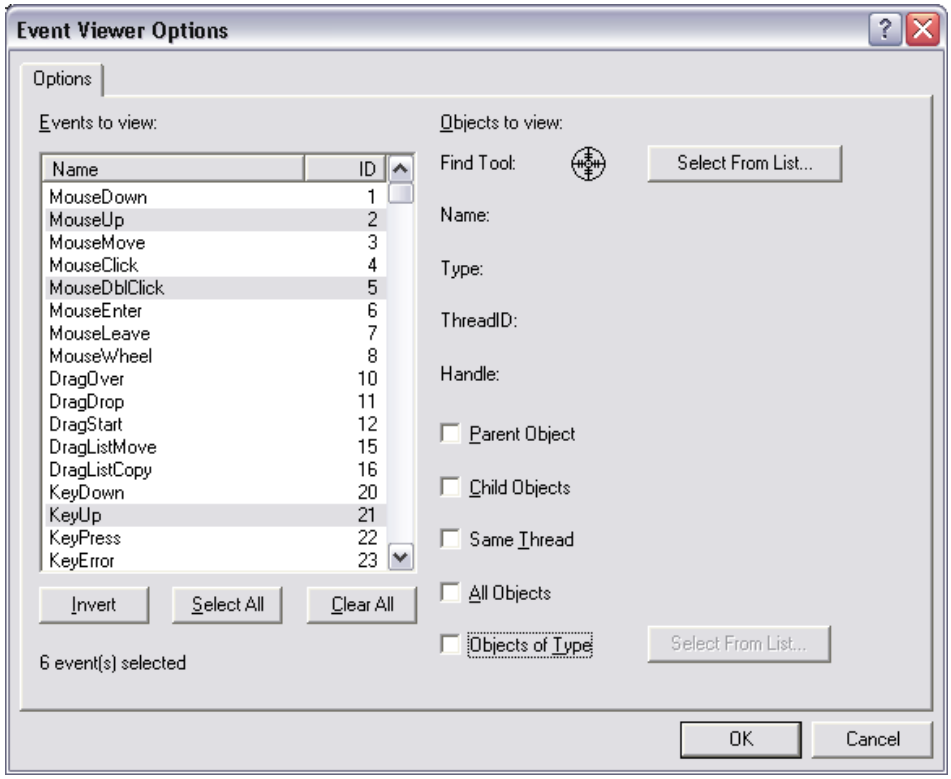


The *Options* menu allows you to choose which information is displayed for the events you are monitoring.

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

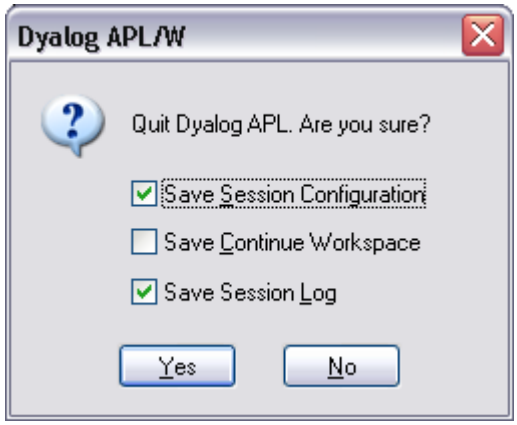
The list box shows all the events that are support by the Dyalog APL GUI and allows you to select which events are to be monitored. Only those events that are selected will be reported. You can sort the events by name or by event number by clicking the appropriate column header.

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.

Closing the Session

When you close the Session window by pressing its X button, or with Alt+F4, the system prompts you with the following dialog box.



Label	Parameter	Description
Save Session Configuration	SaveSessionOnExit	If checked, your current session file will be saved before APL terminates.
Save Continue Workspace	SaveContinueOnExit	If checked, your current workspace will be saved as CONTINUE.DWS before APL terminates.
Save Session Log	SaveLogOnExit	If checked, your session log will be saved before APL terminates.

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

Events Close, Create, FontOK, FontCancel, WorkspaceLoaded

Methods ChooseFont, FileRead, FileWrite

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.

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.

A vector of character vectors containing the names of the methods associated with ⎕SE.

ChildList A vector of character vectors containing the types of object that can be created as a child of ⎕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. For a full description, see the section on *Coord* in *the Object Reference* manual.

Data May be used to associate arbitrary data with the session object ⎕SE. For further details, see the section on *Data* in the *Object Reference* manual

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. For further details, see the section on *HintObj* in the *Object Reference* manual.

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. For further details, see the section on *TipObj* in the *Object Reference* manual.

XRange See the section on *XRange* in the *Object Reference* manual.

YRange See the section on *YRange* in the *Object Reference* manual.

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 *File* and *File/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.

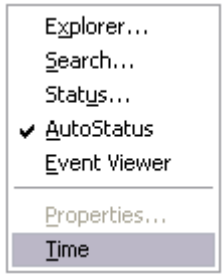
Once again, we will start by changing space into the Tools menu itself

```
      )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

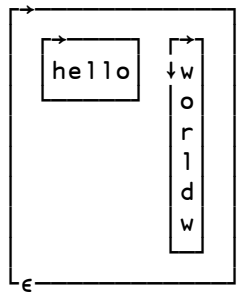
Dyalog APL includes a mechanism to define *User Commands*.

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 3

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

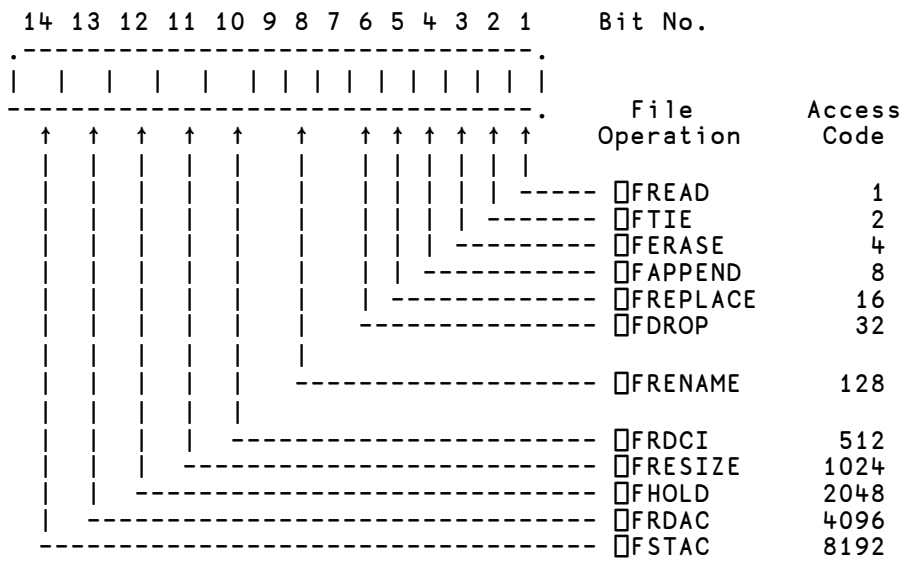
There are two levels of file access control. As a regular data file, the operating system read/write controls for owner and other users apply. In addition, Dyalog APL manages its own access controls using the **access matrix**. This is an integer matrix with 3 columns and any number of rows. Column 1 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 matching row and the user is the owner, no access is granted and the tie fails with `FILE ACCESS ERROR`. If there is no matching row and the user is not the owner, the access matrix is rescanned for the first row with a zero (anybody but the owner) in column 1 and a matching passnumber in column 3. If such a row does not exist, no access is granted and the tie fails with `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

Please 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

⎕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 a component
⎕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
 123
```

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

The techniques discussed in this section apply to both types of file structure.

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. Both types of file systems use the same facility, ⎕FHOLD, to achieve this.

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 external variable
[2]    'personnel' ⎕XT 'DATA'
[3]    NAMES←⊃¨DATA
[4]    →END×⍳('Geoff')∈NAMES
[5]    DATA←DATA,⊂'Geoff' 41 'Hounslow'
[6]  END:
    ∇

    ∇  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 external variable
[2]  'personnel' ⎕XT 'DATA'
[3]  ⎕FHOLD 'personnel'
[4]  NAMES←⊃¨DATA
[5]  →END×⍳(⊂'Geoff')∈NAMES
[6]  DATA←DATA,⊂'Geoff' 41 'Hounslow'
[7] END: ⎕FHOLD ⍳0
    ∇

    ∇ 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 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.

```
      ._.   ._.     ._.
      |2|   |3|     |1|
.-----.-----.-------.------------.
|□□□□□| Six | Three | Bigger One |
---------------------------------.
```

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

Now we'll replace component 3 with something bigger:

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

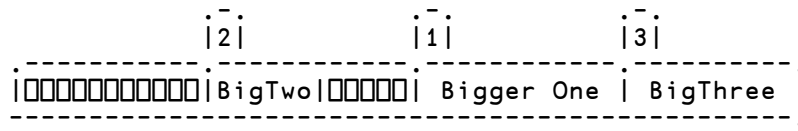
Component 3 is appended to the end of the file, and the area that was used before becomes free:

```
      ._.                 ._.          ._.
      |2|                 |1|          |3|
.-----.-----------------.------------.----------.
|□□□□□| Six |□□□□□□□□□□□□| Bigger One | BigThree |
-----------------------------------------------.
```

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`

In a PC network, or in a single-processor Unix environment, if the FSCB file is missing or inaccessible (restricted access permissions) the report `FILE SYSTEM NOT AVAILABLE` (Error code 28) will be given. The same error will occur under NFS if the **aplfscb** "daemon" is not running.

`FILE SYSTEM TIES USED UP`

The FSCB file has a limited capacity and when that capacity is reached the report `FILE SYSTEM TIES USED UP` (Error code 30) will be given.

`FILE TIED`

A `FILE TIED` error is reported if you attempt to tie a file which another user has exclusively tied. However, it is possible to get **spurious** `FILE TIED` errors in a network for the following reason.

If an APL session has component files tied or has External Variables associated, **and terminates abnormally**, the FSCB will continue to record the file ties, even though the session is no longer running. To prevent another user (or even the same application restarted) from getting spurious `FILE TIED` errors, APL checks whether the process flagged as having a file tied is actually running. If not, the entry is cleared and the new tie honoured.

In a networked environment, it is not possible for a process running on one node to check the status of a process running on another. If a node with component files tied crashes, its file ties will remain (incorrectly) recorded in the FSCB until either that node itself attempts to re-tie the files or until the FSCB is re-initialised.

Limitations

File Tie Quota

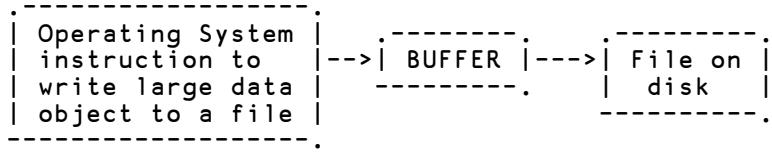
The File Tie Quota is the maximum number of files that a user may tie concurrently. Dyalog APL itself allows a maximum of 256 under Unix and Windows, although in either case your installation may impose a lower limit. When an attempt is made to exceed this limit, the report `FILE TIE QUOTA` (Error code 31) is given. On a UNIX system, there is a system-wide and a per-user limit on the number of open file descriptors. On many systems, the per-user limit is 20, and the system-wide limit about 100. Both limits are usually parameters specified when Unix is installed. Under Windows, the maximum number of open files permitted is specified by the "FILES=" statement in CONFIG.SYS.

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 ⍬` 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.

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 4

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
     ^
```

Now, these sorts of error messages are fine for us clever APL programmers, but meaningless to most of our users. We need to find a way to bypass the default action of APL, so that we can take an action of our own.

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 ⎕fstie 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 ⎕fstie 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(s)	**:**	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:

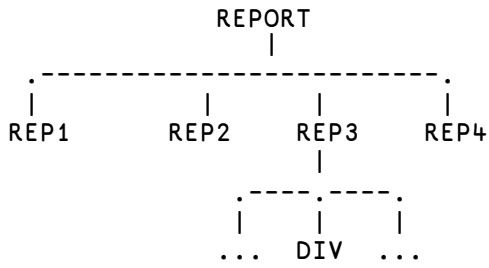
```
    ∇ R←TIE∆FILE FILE;FILES
[1]  ⍝ Tie file FILE with next available tie number
[2]  ⍝
[3]  ⍝ All files in my directory
[4]    FILES←⎕FLIB 'mydir'
[5]  ⍝ Remove trailing blanks
[6]    FILES←dbr¨↓FILES
[7]  ⍝ Required file in list?
[8]    →ERR×⍳~(⊂FILE)∈FILES
[9]  ⍝ Tie file with next number
[10]   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:



where REPORT looks something like this:

```
      ∇  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.

Index

□

.

A

B

C

C

D

E

F

G

H

I

K

S

S

T

U

V

W

X

Y

DYALOG APL

Dyalog Ltd
South Barn
Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom
www.dyalog.com