dyalog APL/W

Dyalog *APL*™ for Windows

User Guide

Dyadic Systems Ltd

Riverside View
Basing Road, Old Basing
Basingstoke
Hampshire, RG24 7AL
United Kingdom

tel: +44 (0)1256 811125
fax: +44 (0)1256 811130
email: support@dyadic.com
http://www.dyadic.com

Dyalog *APL* is a trademark of Dyadic Systems Limited
Copyright © 1982-2003

Copyright © 1982-2003 by Dyadic Systems Limited.

All rights reserved.

Version 10.0

First Edition March 2003

No part of this publication may be reproduced in any form by any means without the prior written permission of Dyadic Systems Limited, Riverside View, Basing Road, Old Basing, Basingstoke, Hampshire, RG24 7AL, United Kingdom.

Dyadic Systems Limited makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Dyadic Systems Limited reserves the right to revise this publication without notification.

TRADEMARKS:

Intel, 386 and 486 are registered trademarks of Intel Corporation.
IBM is a registered trademark of International Business Machines Corporation.
Kermit is copyright of Columbia University Centre for Computing Activities.
Microsoft, MS and MS-DOS are registered trademarks of Microsoft Corporation.
Motif is a trademark of the Open Software Foundation.
POSTSCRIPT is aregistered trademark of Adobe Systems, Inc.
SQAPL is copyright of Insight Systems ApS.
The Dyalog APL True Type font is the copyright of Adrian Smith.
TrueType is aregistered trademark of Apple Computer, Inc.
UNIX is a trademark of X/Open Ltd.
Windows, Windows NT, Visual Basic and Excel are trademarks of Microsoft Corporation.
WATCOM is a trademark of WATCOM Systems, Inc.
X Window System is a trademark of Massachusetts Institute of Technology.

All other trademarks and copyrights are acknowledged.

Printed in the United Kingdom.

Contents

CHAPTER 1

Installation and Configuration

System Requirements

Version 10

Dyalog APL/W Version 10 requires a Pentium PC equipped with a minimum of 16MB. RAM, and one of the following versions of Microsoft Windows:

- Windows XP (all variants)
- Windows 2000 (all variants)
- Windows ME
- Windows NT 3.51 or 4.0
- Windows 98 or 95

.NET Interface

The Dyalog APL .NET interface requires a computer running Windows 2000 or Windows XP Professional with the following elements installed:

- The Microsoft .NET Framework SDK V1.0.3705 or higher.
- Microsoft Internet Information Services (IIS) 5.0 or 5.1
- Microsoft Internet Explorer Version 6.00.

Compatibility

Saved workspaces and `⎕ORs` of functions written in component files produced by previous versions of Dyalog APL, including Version 9.0 and Version 8.2, may be loaded in Version 10.

However, workspaces and `⎕ORs` of functions written by Version 10 may not be loaded using a *previous* release of Dyalog APL. The reason for this restriction is that certain data structures have been changed to accommodate new features. Previous versions of Dyalog APL can have no prior knowledge of these data structures and therefore cannot recognise them.

Installation Overview

Installation Directory

By default, Dyalog APL/W Version 10.0 is installed in the `dyalog10` directory on the drive on which Windows is installed (for example, `c:\dyalog10`). You may select a different drive and directory during installation.

Installation Options

There are 3 installation choices; please see *Installation Instructions* for details:

Local	Select this option to install Dyalog APL/W Version 10.0 on a stand-alone PC. This will copy the software onto your hard disk and update your registry.
Server	Select this option to install Dyalog APL on a network drive. This will copy the software onto the specified network directory but will not update any registry entries. You must subsequently perform a Client installation for each prospective user. See *Client/Server Installation* for details.
Client	Select this option to register yourself as a user of a network installation. This will create your personal registry entries so that you can access Dyalog APL. A number of additional files are copied. See *Client/Server Installation* for details.

Note that if you have a single Windows NT PC that will be used at different times by different users each with a separate log-in, you must perform a Client/Server installation.

In the Local and Client cases, the Dyalog APL Input Method Editor (IME) is installed (See *.Net Interface Guide*). Furthermore, if the setup program detects that the Microsoft .Net Framework is installed, it will ask you whether or not you wish to install the Microsoft .Net Interface. This part of the installation is described later in this chapter.

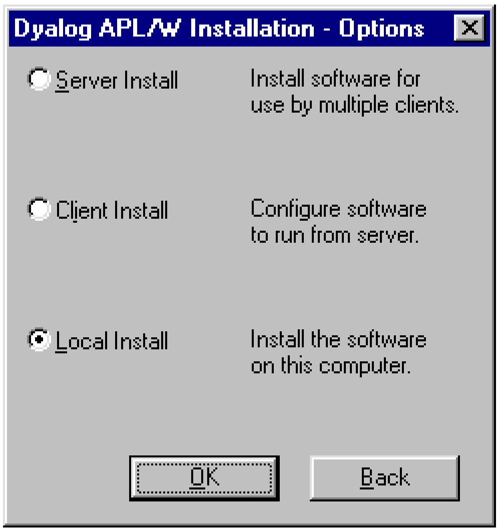
Registry Folders

Unless they already exist, the installation of Dyalog APL/W Version 10.0 creates the following registry folders:

```
HKEY_CURRENT_USER\Software\Dyadic\Dyalog APL/W 10.0
HKEY_CURRENT_USER\Software\Insight\SQApl
```

Stand-alone Installation

1. Insert the CD-ROM labelled Dyalog APL/W Version 10.0 in the drive and open the Dyalog10 directory.
2. To begin the installation, double-click `setup.exe`.
3. The first dialog box displays the Copyright Notice; click Next to continue.
4. The next dialog box asks you to choose the type of installation,



Dyalog APL/W Installation - Options

○ Server Install — Install software for use by multiple clients.

○ Client Install — Configure software to run from server.

● Local Install — Install the software on this computer.

Ensure that Local Install is selected, then press OK.

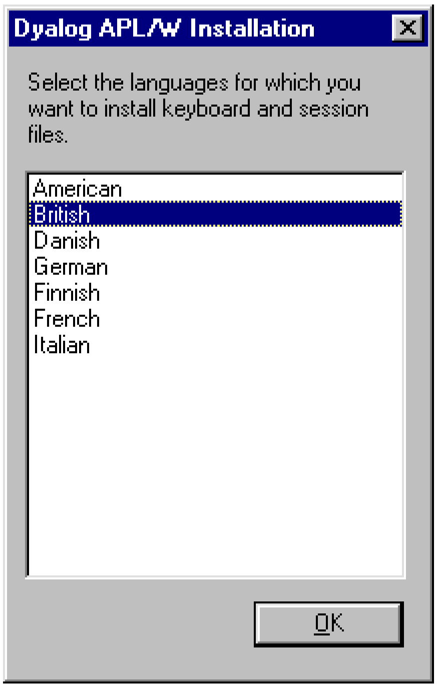
5. The next dialog box asks you to specify the installation directory; the default is Dyalog10 on the drive in which Windows is installed. If you want to install on a different drive or in a different directory, click Browse and change it. Please note that if you install over an existing Dyalog APL/W directory, `setup.exe` will overwrite files of the same name as those on the CD-ROM.
6. You will next be asked to choose between a Typical, Compact or Custom installation.

Typical	This copies all Dyalog APL/W files from the CD-ROM onto your disk and updates your registry. This option requires approximately 16Mb of disk space.
Compact	This copies only the bare minimum number of files required to run `dyalog.exe` onto your system, and updates your registry. This option requires 2.6Mb of disk space.
Custom	This option allows you to omit certain types of file which would otherwise be installed.

7. The next dialog box asks you to choose a Program Folder for the Dyalog APL icons; the default is “Dyalog APL Version 10.0”.
8. If you are installing Dyalog APL/W for the first time, you will then be asked to enter your serial number. This may be found on a label on the back of the CD-ROM.

9. You will then be prompted to select the languages for which you want to install keyboard and session files. See the following table entitled National Language Support files for a list of the files that will be copied onto your system corresponding to each language you select.

   Note that you can select more than one language. For example, if you want to use a French keyboard but have the American session, you should choose both American and French options here.



Dyalog APL/W Installation

Select the languages for which you want to install keyboard and session files.

American
British
Danish
German
Finnish
French
Italian

10. Finally you will be asked to select your Session file and keyboard table. You will be able to choose from all of the .DSE and .DIN files that `setup.exe` finds in your Dyalog10 and Dyalog10\aplkeys directories respectively. This means all the files selected at Step 9 plus any .DSE and/or .DIN files that you already had in these directories.
11. `setup.exe` will now complete the installation process without further prompting and (if you chose a Typical installation) will display the Release Notes on completion.
12. After installing Dyalog APL/W Version 10.0, `setup.exe` will check that your Windows Custom Control Library `comctl32.dll` is Version 4.72 or later. If not, you will be asked to install the update.

National Language Support Files

Language	Files	Description

American	`def_us.dse`
	`apl_us.din`
	`unify_us.din`
	`unify2us.din`
	`combo_us.din`
		US English Session file
		APL/ASCII keyboard
		Unified keyboard
		Unified keyboard
		Combined keyboard

British	`def_uk.dse`
	`apl_uk.din`
	`unify_uk.din`
	`unify2uk.din`
	`combo_uk.din`
	`combo2uk.din`
		UK English Session file
		APL/ASCII keyboard
		Unified keyboard
		Unified keyboard
		Combined keyboard
		Combined keyboard

Danish	`def_uk.dse` `unify_dk.din`	UK English Session file Unified keyboard

German	`def_gr.dse`
	`apl_gr.din`
	`unify_gr.din`
	`unify2gr.din`
		German Session file
		APL/ASCII keyboard
		Unified keyboard
		Unified keyboard

Finnish	`def_fi.dse`
	`apl_fi.din`
	`unify_fi.din`
		Finnish Session file
		APL/ASCII keyboard
		Unified keyboard

French	`def_fr.dse`
	`apl_fr.din`
	`unify_fr.din`
		French Session file
		APL/ASCII keyboard
		Unified keyboard

Italian	`def_it.dse` `apl_it.din`	Italian Session file APL/ASCII keyboard

Notes:

Using `unify_uk.din`, `unify_us.din`, `combo_uk.din` (in unified mode) and `combo_us.din` (in unified mode) you input an APL quote using Ctrl+k and an ASCII quote using the standard quote keystroke. Using `unify2uk.din`, `unify2us.din` and `combo2uk.din` (in unified mode) these are reversed; Ctrl+k produces an ASCII quote and the standard quote keystroke produces an APL quote.

The Finnish, French, Italian and Danish files were supplied by Dinosoft OY, Legrand Consultants, APL Italiana, and Insight Systems ApS respectively. `apl_gr.din` and `unify_gr.din` were donated by Peter Brahm. `unify2gr.din` was supplied by APL Software team.

Dyadic is grateful to all those who have supplied these files.

Client/Server Installation

The purpose of a Client/Server installation is to copy the software onto a single network directory and to perform individual installation tasks for each user that will run Dyalog APL/W. It also applies if several different user log-ons share the same computer.

Logged in as the network or system administrator:

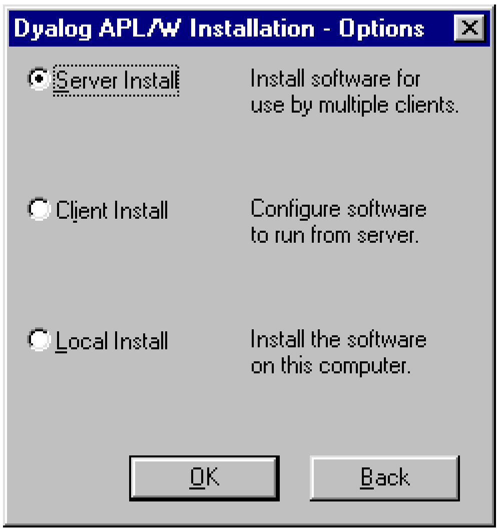
Perform a *Server Install* as described below. This will copy the Dyalog APL/W software onto the specified drive; that is all.

Logged in as a user:

Perform a *Client Install* as described below. This will update your registry, install essential DLL files in your Windows directory, and install the Dyalog APL fonts on your system. This step must be repeated for each and every Dyalog APL user.

Server Installation

1. Insert the CD-ROM labelled Dyalog APL/W Version 10.0 in the drive and open the Dyalog10 directory.
2. To begin the installation process, double-click `setup.exe`.
3. The first dialog box displays the Copyright Notice; click *Next* to continue.
4. The next dialog box asks you to choose the type of installation,



Dyalog APL/W Installation - Options

● Server Install — Install software for use by multiple clients.

○ Client Install — Configure software to run from server.

○ Local Install — Install the software on this computer.

Select *Server Install*, then press *OK*.

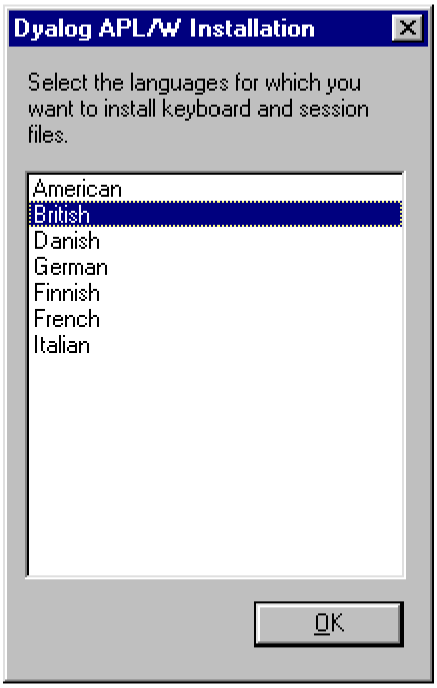
5. The next dialog box asks you to specify the installation directory; the default is dyalog10 on the drive in which Windows is installed. If you want to install on a different drive or in a different directory, click *Browse* and change it. Please note that if you install over an existing Dyalog APL/W directory, `setup.exe` will overwrite files of the same name as those on the CD-ROM.

6. You will next be asked to choose between a *Typical*, *Compact* or *Custom* installation.

Typical	This copies all Dyalog APL/W files from the CD-ROM onto your disk and updates your registry. This option requires approximately 16Mb of disk space.
Compact	This copies only the bare minimum number of files required to run `dyalog.exe` onto your system, and updates your registry. This option requires 2.6Mb of disk space.
Custom	This option allows you to omit certain types of file which would otherwise be installed.

7. You will then be prompted to select the languages for which you want to install keyboard and session files. See *National Language Support Files* earlier in this chapter.

   Note that you can select more than one language. For example, if a user wanted to have a French keyboard but the American session, you should choose both American and French options here. Please ensure that you select all of the languages that your users are likely to require.



Dyalog APL/W Installation

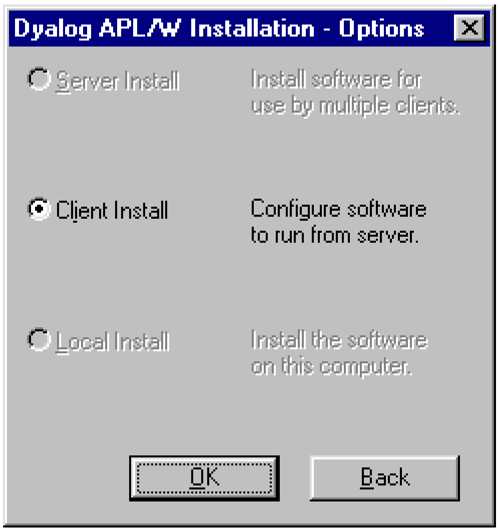
Select the languages for which you want to install keyboard and session files.

- American
- British
- Danish
- German
- Finnish
- French
- Italian

8. `setup.exe` will now complete the *Server Install* process without further prompting.

Client Installation

1. Use File Manager to start `setup.exe` in the Dyalog APL/W directory on your network drive. Note that you must run `setup.exe` from *this* directory and not from the CD-ROM.
2. The first dialog box displays the Copyright Notice; click *Next* to continue.
3. The next dialog box asks you to confirm a *Client Install*; other options are disabled. Click *OK* to continue.


Dyalog APL/W Installation - Options

○ Server Install — Install software for use by multiple clients.

● Client Install — Configure software to run from server.

○ Local Install — Install the software on this computer.

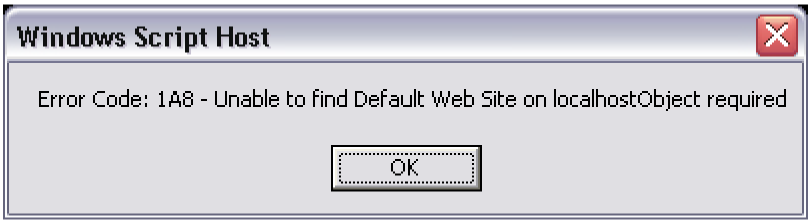
4. The next dialog box asks you to specify your Dyalog APL/W directory. This is going to be your *working* directory for Dyalog APL/W Version 10.0. The default is dyalog10 on the drive in which Windows is installed. If you want to choose a different drive or directory, click *Browse* and change it. Please note that no files will be copied into this directory.
5. If you are installing Dyalog APL Version 10 for the first time, you will then be asked to enter your serial number. This may be found on a label on the back of the CD-ROM.
6. Then you will be asked to select your Session file and keyboard table. You will be able to choose from all of the .DSE and .DIN files that were copied onto your network drive according to the languages that your System Administrator selected during the *Server Install*.
7. The next dialog box asks you to choose a Program Folder for your Dyalog APL icons; the default is “Dyalog APL Version 10.0”.
8. `setup.exe` will now complete the installation process without further prompting and will display the Version 10.0 Release Notes on completion.

Microsoft .NET Interface Installation

If the Microsoft .NET Framework is installed on the computer, `setup.exe` asks you whether or not you wish to install the Microsoft .Net Interface. If you say yes, `setup.exe` performs the following additional tasks:

1. Removes incompatible elements that were installed or created by previous Dyalog.Net releases.
2. Modifies the .NET configuration file `machine.config`, adding a section to register and associate the .NET language "apl" with the Dyalog `APLScript` *compiler*.
3. Optionally associates Notepad with .aspx, asax, asmx and .ascx files. This makes it easy to use Notepad to edit the sample APLScript files.
4. Optionally, modifies your IIS configuration by setting the service attribute *Allow service to interact with desktop*. In addition, `setup.exe` optionally changes the IIS user from ASPNET to SYSTEM. These changes are both necessary to enable the use the Dyalog APL Session and debugger under IIS.
5. Installs some sample Dyalog APL Web applications as IIS Virtual Directories.

   Note that this only works if you have an IIS Web Site named *Default Web Site* (the default for Windows 2000). If not, you will see the message box shown below and you will have to install the APL Web applications manually as described in Chapter 6.



Error Code: 1A8 - Unable to find Default Web Site on localhostObject required

To install the Microsoft .Net Interface separately, by insert your Dyalog APL Version 10 CD-ROM and run the `setup.exe` in the sub-directory `dyalog10\dotnet_setup`.

COMCTL32.DLL Update

For full functionality, Dyalog APL/W Version 10.0 requires Version 4.72 or later of the Windows Custom Control Library `comctl32.dll` which is installed in your WINDOWS\SYSTEM directory.

Your Dyalog APL CD-ROM contains the file 40COMUPD.EXE. This is an official Microsoft update for `comctl32.dll` which Dyadic is authorised to distribute with Dyalog APL.

After installing Dyalog APL, `setup.exe` checks the Version number of `comctl32.dll` and, if necessary, notifies you that you should install the update.

If you need the update to `comctl32.dll`, you should install it prior to running Dyalog APL/W Version 10.0. To do so, simply run 40COMUPD.EXE from the CD-ROM.

If you choose *not* to install the update, you will be unable to use many of the new objects such as the CoolBar and Calendar objects. Furthermore, the standard Version 10.0 Session files (which use CoolBars) will not load correctly.

If you prefer not to install the update to `comctl32.dll`, you may use a Dyalog APL/W Version 8.1 Session file, or you may build your own Version 10.0 Session file using BUILDSE.DWS. To do this :

```
      )LOAD BUILDSE
C:\Dyalog10\ws\BUILDSE saved Thu Feb 27 13:03:36 2003
      BUILD_SESSION 'UK' ⍝ See Trans.CODES
...
```

Note that the right argument to BUILD_SESSION is one of the country codes specified in the variable `Trans.CODES`.

If your `comctl32.dll` is prior to Version 4.72, `BUILD_SESSION` will create you a simpler (Version 8.1 style) Session file that avoids the use of CoolBars and other objects that require this Version of `comctl32.dll`.

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

APL Fonts

Two types of APL fonts in two different layouts are provided.

The standard layout contains the APL underscored alphabet A̲-Z̲. The alternative layout 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 standard font layout to the alternative one, you will see the symbol Á (A-acute) instead of the symbol A̲.

The two types of APL font are bitmap (screen) and TrueType. The bitmap fonts were designed by Dyadic and, if desired, may be edited using the FONTS workspace that is provided. The bitmap fonts are named *Dyalog Std* and *Dyalog Alt*.

The TrueType fonts were designed by Adrian Smith and are distributed under agreement by Dyadic. These fonts have a traditional 2741-style italic appearance and are named *Dyalog Std TT* and *Dyalog Alt TT*.

You may use either a bitmap font or a TrueType font in your APL session (see *Chapter 2* for details). Depending upon the capabilities of your graphics card, bitmaps fonts typically provide better screen display performance than TrueType fonts, but are not printable (this is a Windows limitation). You MUST use a TrueType font for printing APL functions.

In addition, the *APL385 Unicode* font © Adrian Smith is provided. This font may be used in conjunction with APLScript files.

Help Files

Dyalog APL/W includes two different sets of Windows help files. These are accessed from the *Help* menu on the Session or by running `winhelp.exe` or `winhlp32.exe` directly.

The files in the top-level directory `dyalog10` are those which are accessed from the Session and are compiled using the *Dyalog Std* bitmap font. This means that they are not printable because Windows cannot print bitmap fonts in help files. Another set of help files is provided in the `tt_help` sub-directory. These help files contain the identical information, but are compiled using the *Dyalog Std TT* TrueType font and are therefore printable. The help files built using the bitmap font are generally considered to be easier to read on-screen; especially at lower resolutions.

If you prefer to be able to print your help topics on demand from the APL session (at the expense of somewhat less legible screen appearance), you may copy the files from the `tt_help` sub-directory into the top-level `dyalog10` directory. This will ensure that APL loads the TrueType help files. If you do so, it is recommended that you backup the bitmap font help files first.

The APL Command Line

The command line for Dyalog APL/W is as follows :

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

where:

[options]

-x	No ⎕LX execution on workspace loads.
-a	Start in USER mode.
-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:\dyalog10\dyalog.exe myapp maxws=8192
c:\dyalog10\dyalog.exe session_file=special.dse
c:\dyalog10\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`.

Workspace Integrity and the APLCORE File

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 saves the workspace, together with diagnostic information, in a file called **aplcore** and exits with a message `SYSERROR nnn`.

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 Dyadic 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.

A syserror report and an **aplcore** file may also be produced if the Dyalog APL program fails (crashes) for another reason.

Objects may often (but not always) be recovered from **aplcore** using `)COPY`. Note that because 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.
```

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

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

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.

Note that if an error occurs in a Windows API function, or in a dynamic link library (DLL) that has been called from APL, the system reports a *syserror 999*. In this case, the crash may not be due to a bug in Dyalog APL, but could be caused by a bug in a DLL, or an error in a `⎕NA` call. Please check these potential causes before reporting the problem to Dyadic.

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\Dyadic\Dyalog APL/W 10.0` is assumed. The value of any other parameter (including **dyalog**) 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.

AutoComplete Parameters

The paremeters used to specify the behaviour of the Auto Complete feature are stored in a sub-folder named AutoComplete. These parameters are described in detail at the end of this section.

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 4 for further details.

aplk

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.

aplkeys

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. For further details, see the description of the Output Table in Chapter 3.

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.

AutoTrace

This parameter specifies whether or not the Trace Tools window pops up automatically when you initiate a Trace, and disappears again when you have finished tracing. It is either 1 (Trace Tools appear automatically) or 0. The default value is 1. **Autotrace** determines your overall preference for the *Trace Tools* window that persists between APL sessions. You may still control the appearance of the *Trace Tools* window on an ad hoc basis from the *Trace Tools* sub-menu of the *Options* menu on the Session Window.

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 delet lines from the Session Log. Its value is either 1 (confirmation is required) or 0. The default is 0.

default_div

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

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.

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.

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\Dyadic\mysettings
```

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

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.

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

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. The default value is 2048 (2 Mb). If you want a larger (or smaller) workspace you must change this value. For example, to get a 4 MB workspace :

```
MAXWS=4096
```

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.

pfkey_size

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

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.

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.

ShowStatusOnOutput

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.

tmp

This parameter specifies the name of the directory which is to be used by `⎕CMD` for its temporary file. This file is used to collect the output produced by running the command given in the argument to `⎕CMD`.

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 default is `.\;WS;.\XFLIB`. The following example causes `)COPY`, `)LOAD` and `)LIB` to look first in the current directory, then in D:\MYWS, and then in the standard places.

```
wspath=.;D:\MYWS;C:\dyalog10\WS;C:\dyalog10\XFLIB
```

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
```

Auto Complete Configuration Parameters

Cols

This parameter specifies the maximum width in characters of the AutoComplete suggestion box. The default value is 20.

Enabled

This parameter specifies whether or not the Auto Complete feature is enabled. Its default value is 1.

History

This parameter specifies whether or not the Auto Complete feature maintains a history of previous Auto Complete operations that the user has chosen. The default value is 1.

HistorySize

This parameter specifies the number of previous Auto Complete operations that are maintained The default value is 10.

PrefixSize

This parameter specifies the number of characters that the user must enter before the Auto Complete feature starts to make suggestions. Its default value is 1.

Rows

This parameter specifies the maximum height in characters of the AutoComplete suggestion box. The default value is 3.

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.

Workspace Management

Workspace Size and Compaction

The amount of memory allocated to a Dyalog APL workspace is defined by the **maxws** parameter. The default value is 4096 KB.

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 you want to create an object larger than any of the available free areas, APL reorganises the workspace and amalgamates all the free areas into one contiguous block. This is termed a COMPACTION. If you try to create an object which is larger than free space, APL reports `WS FULL`.

The following commands force a compaction :

```
⎕WA, )RESET, )SAVE
```

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

)COPY

`)COPY` is implemented in a manner designed to provide optimum performance under normal circumstances. `)COPY` actually `)LOAD`s the target workspace into memory. Some or all of the contents of the target workspace are then copied into the primary workspace. During the copy, all processing of the two symbol tables, stacks, and internal pointers occurs in memory without further (random) disk accesses. In general this design makes `)COPY` (which is traditionally a slow operation) very fast. If however the target workspace is very large it can cause problems with excessive paging. If the source workspace is too large to be loaded into the APL task, the system reports `ws too large`.

Interface with DOS

DOS 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*. The NTUTILS and DOSUTILS workspaces provide direct access to certain commonly used DOS commands. See *Chapter 6*.

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.

```
      'EXAMPLE' ⎕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 were 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.

```
┌─────────────────────────┐
│      APL PROCESS        │
├─────────────────────────┤
│      )CLEAR             │
│clear ws                 │
│                         │  start AP    ┌──────────────┐
│    'EXAMPLE' ⎕CMD ''    │------------->│  AP EXAMPLE  │
│                         │              ├──────────────┤
│                         │ info about   │Send info on  │
│                         │<-------------│external fns  │
│                         │  functions   │              │
│    )FNS                 │              │  wait ...    │
│DATE_TO_IDN IDN_TO_DATE  │              │              │
│                         │function code │              │
│    IDN_TO_DATE 19407    │------------->│call relevant │
│                         │    19407     │ subroutine   │
│    wait ...             │              │              │
│                         │<-18 Feb 53-- │ send result  │
│                         │              │              │
│                         │  terminate   │              │
│      )CLEAR             │------------->│     EXIT     │
│clear ws                 │  and stop    └──────────────┘
│                         │
└─────────────────────────┘
```

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 controlled using a Dyalog APL utility program XVAR.EXE. The three ways you can use XVAR.EXE are as follows:

a) Give external variable MYVAR shared status :

```
C:> xvar /s myvar.dxv
```

b) Make the external variable MYVAR exclusive status, i.e. prevent other users from accessing it :

```
C:> xvar /x myvar.dxv
```

c) Query current access status of external variable MYVAR:

```
C:> xvar myvar.dxv
```

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.

Creating Executables

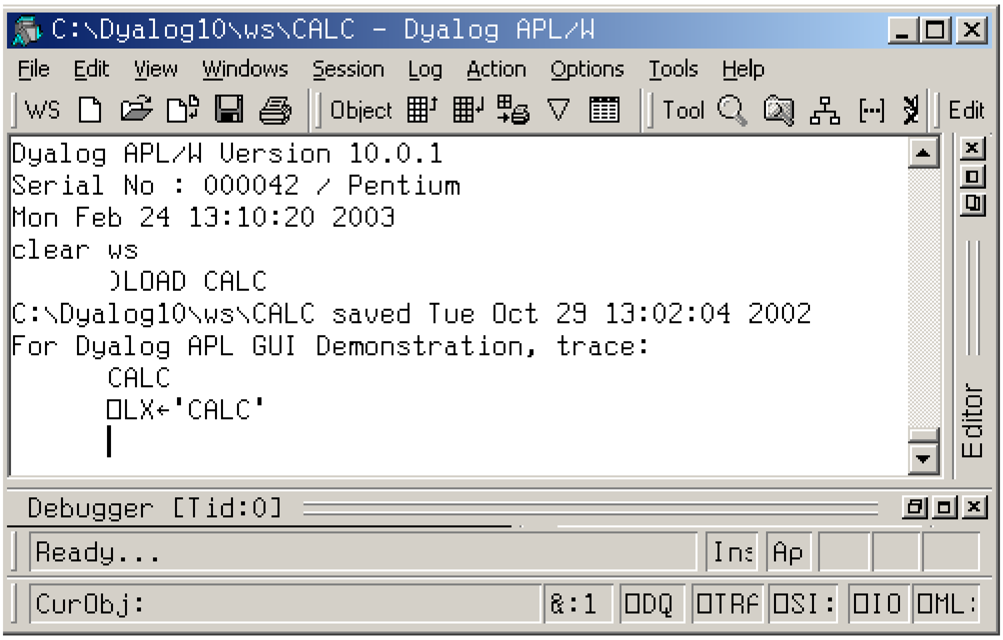
Dyalog APL provides the facility to package an APL workspace as a Windows executable (EXE). This may be done by selecting *Export …* from the *File* menu of the APL Session window.

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 uses the Microsoft .Net Framework. If you select this option, your EXE will automatically attempt to initialise the .Net Framework when it starts. If the .Net Framework is not installed, your application will not fail (at this stage) but the operation takes a few seconds and is best avoided if .Net is not required.

A Dyalog APL application packaged as a EXE file must be accompanied by the Dyalog APL Dynamic Link Library (`dyalog10.dll` or `dyalog10rt.dll`) which should be installed in the same directory (as the EXE) or in the Windows System directory.

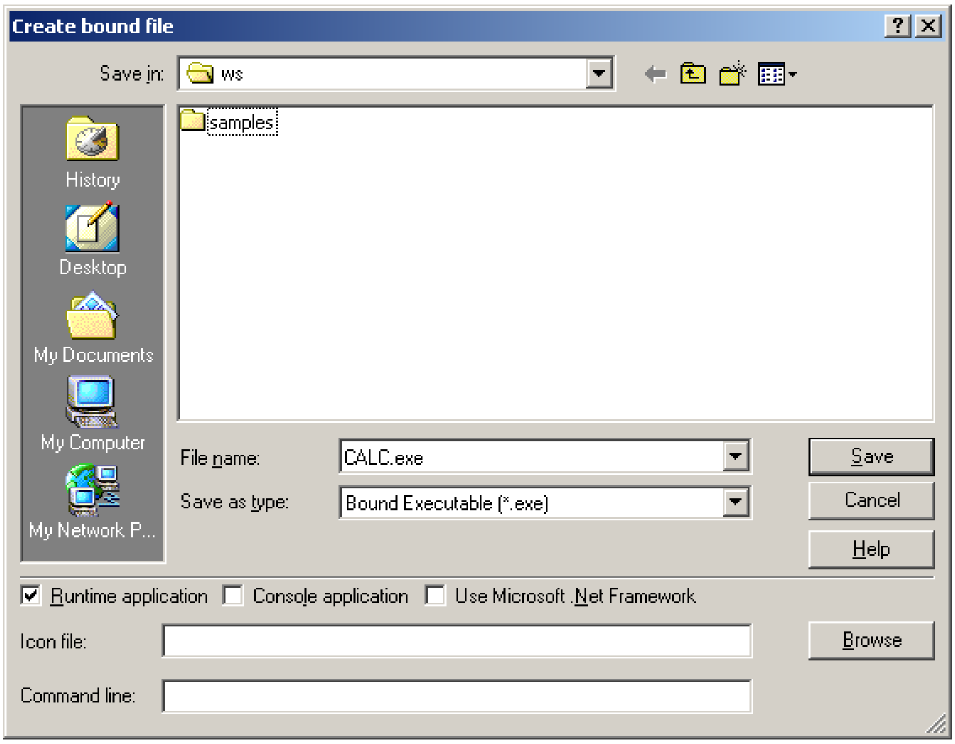
The following example illustrates how you can package the supplied workspace `calc.dws` as an executable. Before making the executable, it is essential to set up the latent expression to run the program using ⎕*LX* as shown. Notice that in this case it is not necessary to execute ⎕*OFF*; the `calc.exe`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



```
Dyalog APL/W Version 10.0.1
Serial No : 000042 / Pentium
Mon Feb 24 13:10:20 2003
clear ws
      )LOAD CALC
C:\Dyalog10\ws\CALC saved Tue Oct 29 13:02:04 2002
For Dyalog APL GUI Demonstration, trace:
      CALC
      ⎕LX←'CALC'
```

C:\Dyalog10\ws\CALC - Dyalog APL/W
File Edit View Windows Session Log Action Options Tools Help
WS
Object
Tool
Edit
Editor
Debugger [Tid:0]
Ready...
Ins
Ap
CurObj:
&:1
⎕DQ
⎕TRAP
⎕SI:
⎕IO
⎕ML:

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



Create bound file

Save in: ws

samples

File name: CALC.exe

Save as type: Bound Executable (*.exe)

☑ Runtime application ☐ Console application ☐ Use Microsoft .Net Framework

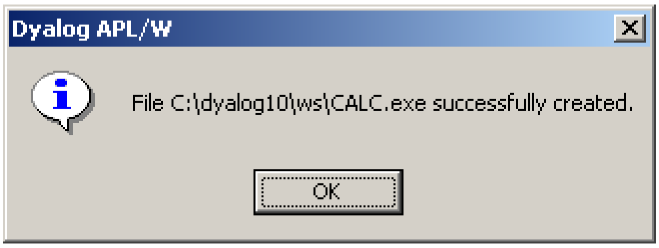
Icon file:

Command line:

In the example shown, the program is to be saved in `dyalog10\ws`, the directory from which the workspace was loaded (the default). The *Runtime application* checkbox is checked, indicating that calc.exe is to be bound to the run-time dynamic link library, `dyalog10rt.dll`.

As this is a GUI application and does not require the Microsoft .Net Framework, the *Console application* and *Use Microsoft .Net Framework* checkboxes have been cleared.

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



Dyalog APL/W

File C:\dyalog10\ws\CALC.exe successfully created.

Run-Time Applications and Components

Using Dyalog APL you may create the following types of run-time applications and components. Subject to the terms of the Dyalog APL No Charge Run-Time Agreement, which you must register in advance with Dyadic, you may distribute run-time applications and components free of charge.

Bound 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 bound to the Dyalog APL Run-Time Dynamic Link Library (`dyalog10rt.dll`). To distribute your application, you need to supply and install

1. Your bound executable (EXE)
2. `dyalog10rt.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 `dyalog10rt.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 `bridge.dll` and `dyadic.dll.` These files must be installed in the *global assembly cache* (GAC) using the `gacutil.exe` utility program. In addition, `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 (`dyalogrt.exe`) and a separate workspace. To distribute your application, you need to supply and install:

1. Your workspace
2. `dyalogrt.exe`
3. whatever additional files that may be required by your application

The command-line for your application invokes `dyalogrt.exe`, passing it start-up parameters required for the Dyalog APL Run-Time Program (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 `bridge.dll` and `dyadic.dll.` These files must be installed in the *global assembly cache* (GAC) using the `gacutil.exe` utility program. In addition, `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 `dyalogrt.exe`, passing it start-up parameters required for the Dyalog APL Run-Time Program (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. `dyalogrt.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 Dyalog APL Run-Time Dynamic Link Library (`dyalog10rt.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. `dyalog10rt.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 (`dyalog10rt.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. `dyalog10rt.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 Dyalog APL Run-Time Dynamic Link Library (`dyalog10rt.dll`).

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

1. Your Assembly file (DLL)
2. `dyalog10rt.dll`, `bridge.dll` and `dyadic.dll`.
3. whatever additional files that may be required by your .Net Assembly.

`bridge.dll` and `dyadic.dll` must be installed in the *global assembly cache* (GAC) using the `gacutil.exe` utility program. In addition, `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 the additional files `cndya30e.dll`, `sqapl.ini`, `sqapl.err and aplunicd.ini`.

`cndya30e.dll` must be installed in the user's Windows directory or be on the user's path. `cndya30e.dll` may be found in your Windows directory.

`sqapl.ini`, `sqapl.err and aplunicd.ini` may be found in your `dyalog10` directory.

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` file, 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. Note too that the `NFILES.EXE` Auxiliary Processor may require translate tables such as `ASCII.DIN`.

DYALOG32.DLL

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. `dyalog32.dll` may be found in your Windows directory where it is put during installation of Version 10.0.

DOS_U32.DLL

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. `dos_u32.dll` may be found in your Windows directory where it is put during installation of Version 10.0.

Registry Entries for Run-Time Applications

`dyalog10rt.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.

`dyalogrt.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 `dyalogrt.exe`.

For example, your application may requires a greater `MAXWS` parameter (workspace size) than the default value of 4Mb. 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 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 10.0 will use the registry folder named

```
HKEY_CURRENT_USER\Software\Dyadic\Dyalog APL/W 10.0
```

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.EXE` or `REGEDIT32.EXE` 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:

   ```
   REGEDIT4

   [HKEY_CURRENT_USER\Software\Dyadic\Dyalog APL/W
   10.0]
   "aplnid"="42"
   "maxws"="8096"
   ```

   Note that the file is identified as a `REGEDIT` script by the first line being `REGEDIT4`. The format of a registry script file is reasonably obvious. In case of doubt, it is suggested that you experiment by saving portions of your own registry to file using `REGEDIT.EXE` and then examining the results.

COM Objects and the Dyalog APL DLL

Introduction

There are two versions of the Dyalog APL Dynamic Link Library, named `dyalog10.dll` and `dyalog10rt.dll` respectively.

`dyalog10.dll` is the complete Dyalog APL development system packaged as a Dynamic Link Library.

`dyalog10rt.dll` is the run-time version of `dyalog10.dll`.

In the remainder of this section, the term *the Dyalog APL DLL* is used to refer to either 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 APLK, APLNID, FILE_CONTROL and so forth from the registry, command-line or environment variables. This means that all such parameters will have their default values.

COMCTL32.DLL Version

For full functionality, Dyalog APL/W Version 10.0 requires Version 4.72 of the Windows Custom Control Library `comctl32.dll`. This file is installed in the WINDOWS\SYSTEM directory.

If your application uses any of the objects that require Version 4.72, and there is any doubt as to which Version will be installed on your end-user computers, you should check the Version as part of your installation procedure and include an update with your software. To check the Version from APL, you may use function *`DllVersion`* in workspace BUILDSE.DWS.

Provided that you accept the terms of the Microsoft End-User License Agreement (“EULA”) for `comctl32.dll` (obtainable from the Microsoft web site), you may redistribute the update file 40COMUPD.EXE which is included on your Dyalog APL CD-ROM. However, it is recommended that you first check the Microsoft web site for more recent information.

CHAPTER 2

The APL Environment

Introduction

The Dyalog APL Development Environment consists of 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.

Session Configuration

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.

Keyboard Configuration

Dyalog APL 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 (using a unified layout) 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. See *Chapter 3, Input and Output Tables* for further details.

Using the Mouse

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.

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 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.

Standard Keyboard Layouts

Although you may configure your keyboard as you wish, three different types of standard layout are provided.

The first is a traditional 2-mode APL/ASCII layout in which you switch between APL mode (for entering APL symbols and upper-case alphabetic characters) and ASCII mode (for all other characters). APL/ASCII keyboard tables are named APL_XX.DIN, where XX represents the country code.

The second type of layout is known as the *Unified* keyboard. Instead of working with modes, the user can enter any symbol directly, using combinations of Shift and Ctrl to extend the meaning of individual keys. Unified keyboard tables are named UNIFY_XX.DIN, where XX represents the country code.

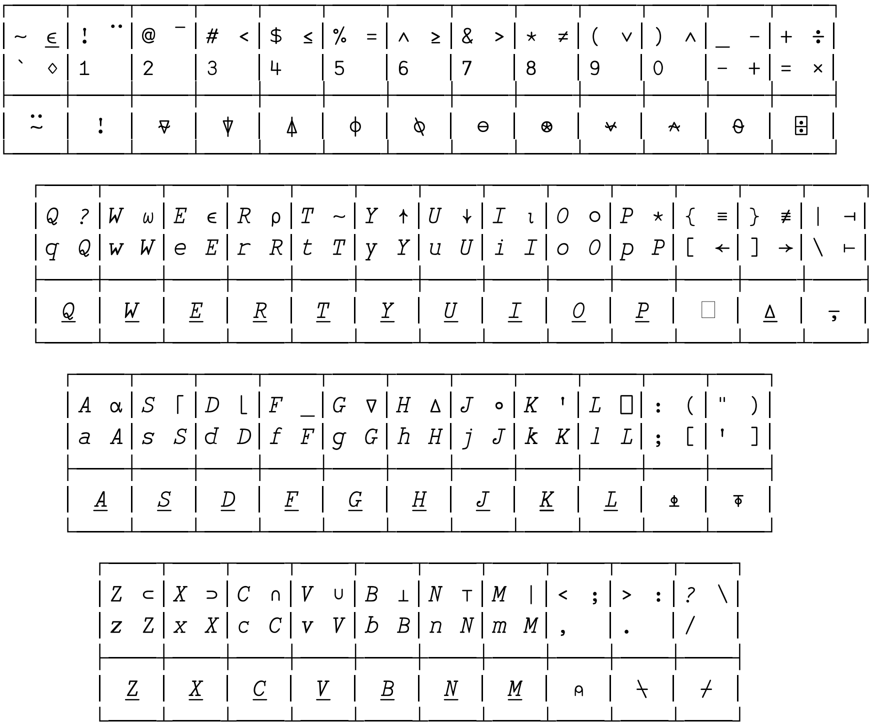
The third type of layout is a 3-mode table in which modes 1 and 2 behave as the APL/ASCII layout and mode 3 the Unified layout. A single keystroke is provided to toggle between APL/ASCII and Unified. These tables are named COMBO_XX.DIN where XX represents the country code.

APL/ASCII Keyboard

Select APL mode by typing Ctrl+n, and ASCII mode by typing Ctrl+o. Alternatively, you can toggle between modes using the MODE command which is (by default) mapped to the "+" key on the numeric keypad.

Mode	Keystroke
APL	Ctrl+n (or MODE)
ASCII	Ctrl+o (or MODE)

The APL symbols are positioned using the standard *typewriter-pairing* conventions which are adopted by most ASCII/APL terminals. The Alt key, which is traditionally used in APL mode to generate composite characters on IBM 3270 terminals, is reserved by Windows for menu selection. Dyalog APL/W therefore uses Ctrl+Shift in place of Alt.


~ ⍷ | ! ¨ | @ ¯ | # < | $ ≤ | % = | ^ ≥ | & > | * ≠ | (∨ |) ∧ | _ − | + ÷
` ⋄ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | - + | = ×
⍨ | ! | ⍫ | ⍒ | ⍋ | ⌽ | ⍉ | ⊖ | ⍟ | ⍱ | ⍲ | ⍬ | ⌹
Q ? | W ⍵ | E ∊ | R ⍴ | T ∼ | Y ↑ | U ↓ | I ⍳ | O ○ | P ⋆ | { ≡ | } ≢ | | ⊣
q Q | w W | e E | r R | t T | y Y | u U | i I | o O | p P | [← |] → | \ ⊢
Q̲ | W̲ | E̲ | R̲ | T̲ | Y̲ | U̲ | I̲ | O̲ | P̲ | □ | ⍙ | ⍪
A ⍺ | S ⌈ | D ⌊ | F _ | G ∇ | H ∆ | J ∘ | K ' | L ⎕ | : (| ")
a A | s S | d D | f F | g G | h H | j J | k K | l L | ; [| ']
A̲ | S̲ | D̲ | F̲ | G̲ | H̲ | J̲ | K̲ | L̲ | ⍎ | ⍕
Z ⊂ | X ⊃ | C ∩ | V ∪ | B ⊥ | N ⊤ | M | | < ; | > : | ? \
z Z | x X | c C | v V | b B | n N | m M | , | . | /
Z̲ | X̲ | C̲ | V̲ | B̲ | N̲ | M̲ | ⍝ | ⍀ | ⌿

Each key is capable of generating up to five different symbols. Those on the left are produced in ASCII mode. Those to the right are produced in APL mode. The lower symbols are produced by pressing the key normally (unshifted). The upper symbols are produced by pressing Shift + the key. If Caps Lock is on, the sense of the shift is reversed, but **only** for the alphabetic keys. The fifth symbol, shown alone at the bottom is produced by pressing Ctrl + Shift + the key.

Consider the 5 symbols shown on the "A" key. In ASCII mode, pressing the key alone produces "`a`" and pressing Shift + the key generates `A`. In APL mode, pressing the key alone produces `A` and pressing Shift + the key generates ⍺. In either mode, pressing Ctrl + Shift + the key generates A̲.

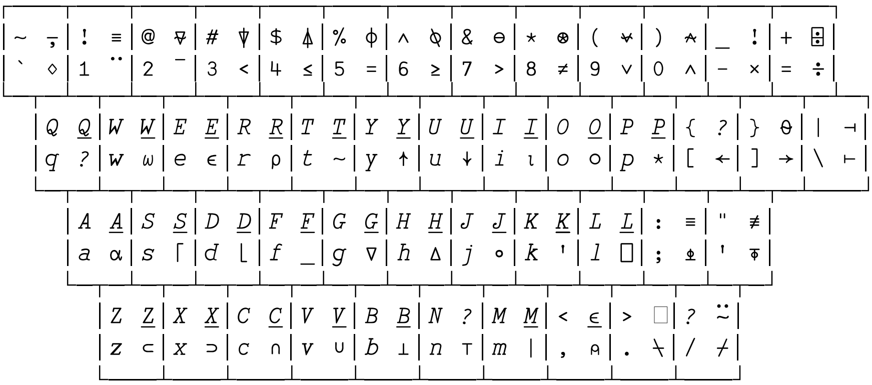
Unified Keyboard

Using the Unified keyboard layout, symbols are generated as follows.

Pressing a key on its own or with Shift generates the corresponding ASCII character in the same way as in any other Windows application. These symbols are shown to the left in each of the keys in the diagram below. The lower of the two is produced by the key on its own, the upper by Shift + the key.

Pressing a key with Ctrl generates the APL symbol shown in the bottom right corner of each of the keys in the diagram. Pressing a key with Ctrl + Shift produces the APL symbol shown in the top right corner.

Consider the 4 symbols shown on the A key. Pressing the key on its own generates `a`. pressing the key with Shift produces `A`. Pressing the key with Ctrl generates ⍺. Finally, pressing the key with Ctrl + Shift produces A̲.



```
|~ ⍪|! ≡|@ ⍫|# ⍒|$ ⍋|% ⌽|^ ⍉|& ⊖|* ⍟|( ⍱|) ⍲|_ !|+ ⌹|
|` ⋄|1 ¨|2 ¯|3 <|4 ≤|5 =|6 ≥|7 >|8 ≠|9 ∨|0 ∧|- ×|= ÷|

  |Q Q̲|W W̲|E E̲|R R̲|T T̲|Y Y̲|U U̲|I I̲|O O̲|P P̲|{ ?|} ⍬|| ⊣|
  |q ?|w ⍵|e ∊|r ⍴|t ∼|y ↑|u ↓|i ⍳|o ○|p *|[ ←|] →|\ ⊢|

    |A A̲|S S̲|D D̲|F F̲|G G̲|H H̲|J J̲|K K̲|L L̲|: ≡|" ≢|
    |a ⍺|s ⌈|d ⌊|f _|g ∇|h ∆|j ∘|k '|l ⎕|; ⍎|' ⍕|

      |Z Z̲|X X̲|C C̲|V V̲|B B̲|N ?|M M̲|< ⍷|> □|? ⍨|
      |z ⊂|x ⊃|c ∩|v ∪|b ⊥|n ⊤|m ||, ⍝|. ⍀|/ ⌿|
```

Combo Keyboard

Using the Combo keyboard layout, you can toggle between the APL/ASCII and the Unified layout by pressing the '*' key (keycode 106) on the numeric keypad.

Line-Drawing Symbols

Dyalog APL/W supports 11 single-line graphics characters for drawing lines and boxes, although it is possible to add more if required. Using either the APL/ASCII keyboard layout, or the Unified keyboard layout, the line-drawing characters are entered using the keys on the numeric keypad in conjunction with the Ctrl key as shown below. Num Lock must be **on**.

```
     Normal                Ctrl
┌─────┬─────┬─────┐    ┌─────┬─────┬─────┐
│  7  │  8  │  9  │    │  ┌  │  ┬  │  ┐  │
├─────┼─────┼─────┤    ├─────┼─────┼─────┤
│  4  │  5  │  6  │    │  ├  │  ┼  │  ┤  │
├─────┼─────┼─────┤    ├─────┼─────┼─────┤
│  1  │  2  │  3  │    │  └  │  ┴  │  ┘  │
├─────┴─────┼─────┤    ├─────┴─────┼─────┤
│     0     │  .  │    │     │     │  ─  │
└───────────┴─────┘    └───────────┴─────┘
```

Note that to accommodate other characters, line-drawing symbols are located in the non-printable area of the font layout. Although these characters can be used in output to the session (function: `DISP` in the UTIL workspace uses them), many printer drivers are unable to display characters from these positions in the font.

Interrupts

To generate an interrupt, click on the Dyalog APL icon in the Windows System Tray; then choose: Weak Interrupt, 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.

Commands

The term *command* is used herein to describe a keystroke that generates an *action*, rather than one that produces a symbol. 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. See *Input & Output Tables* for further details.

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

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. Alternatively, as in older implementations of Dyalog APL, colours may 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
255	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

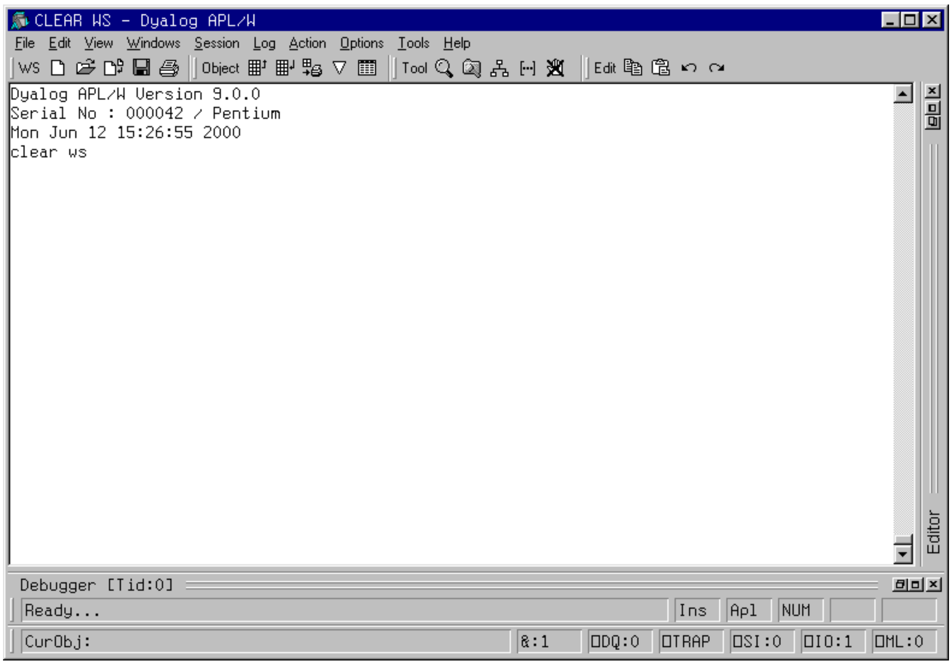
As an adjunct to the overall Session Colour Scheme, you may choose to apply a *syntax colouring scheme* to the current Session Input line(s). You may also extend syntax colouring to previously entered input lines, although this only applies to input lines in the current session; syntax colouring information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



```
Dyalog APL/W Version 9.0.0
Serial No : 000042 / Pentium
Mon Jun 12 15:26:55 2000
clear ws
```

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.

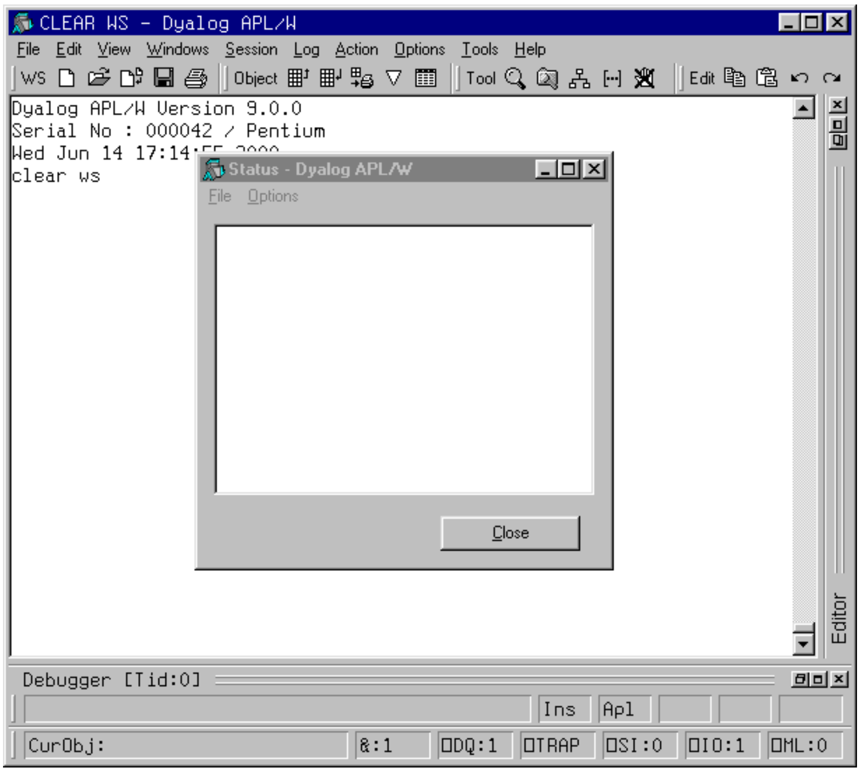
In addition to the Session Window itself, there are various subsidiary windows which are described later in the Chapter. In general, these subsidiary windows may be docked inside the Session window, or may be stand-alone floating windows. You may dock and undock these windows as required. The standard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

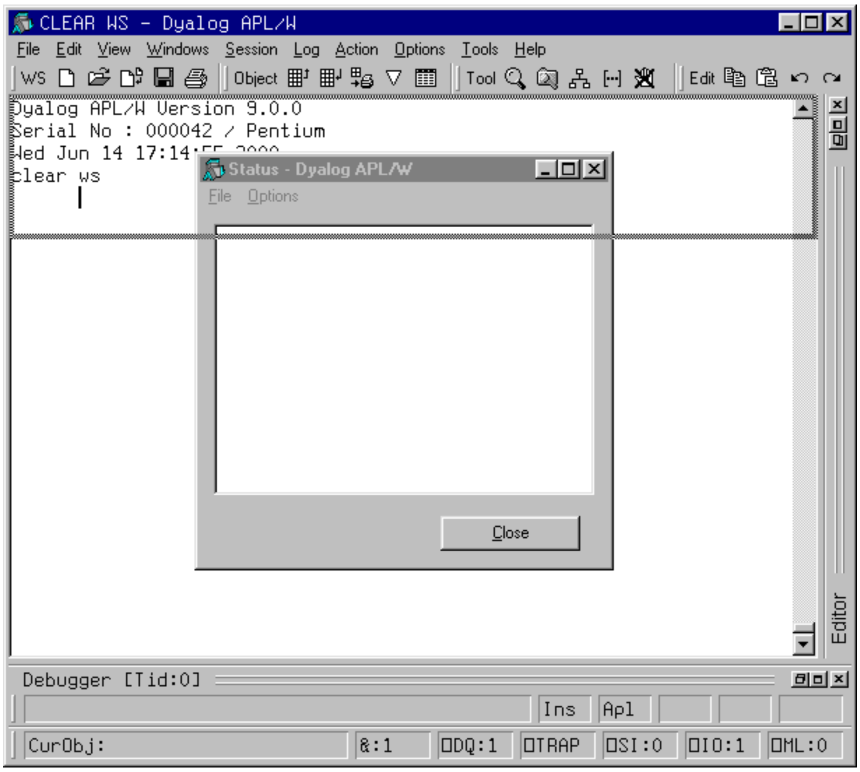
Nearly all of the windows used in the Dyalog APL IDE may be docked in the Session window or be stand-alone floating windows. When windows are docked in the Session, the Session window is split into resizable panes, separated by splitters. The following example, using the Status window, illustrates the 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. Its initially appears as a floating (undocked) window as shown below.



```
Dyalog APL/W Version 9.0.0
Serial No : 000042 / Pentium
Wed Jun 14 17:14:55 2000
clear ws
```

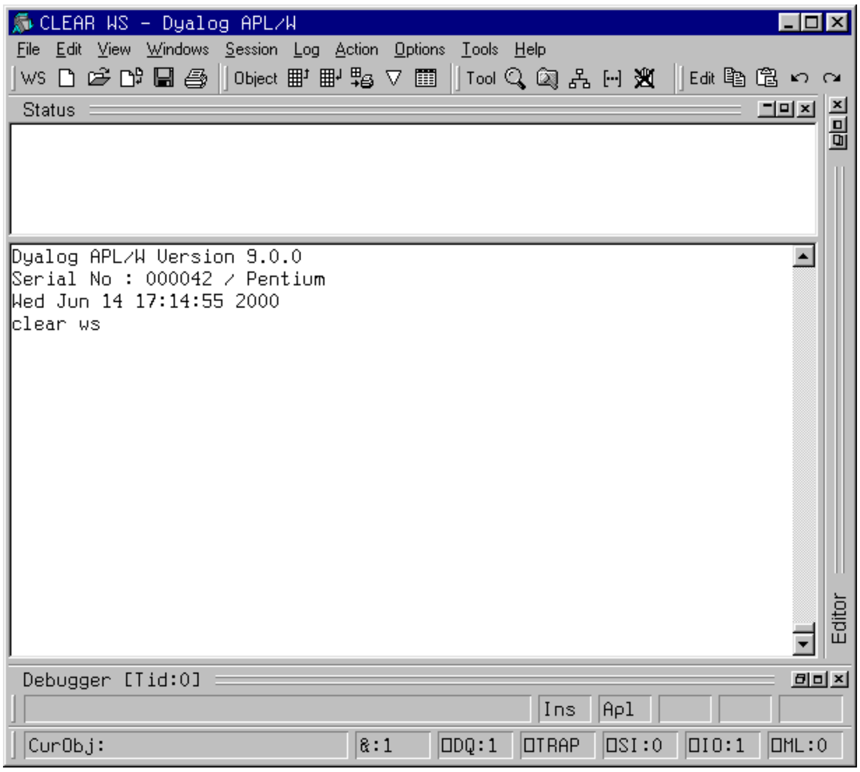
If you press the left mouse button down over the Status window title bar, and drag it, you will find that when the mouse pointer is close to an edge of the Session window, the drag rectangle indicates a docking zone as shown below. This indicates the space that the window will occupy if you now release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



```
Dyalog APL/W Version 9.0.0
Serial No : 000042 / Pentium
Wed Jun 14 17:14:55 2000
clear ws
```

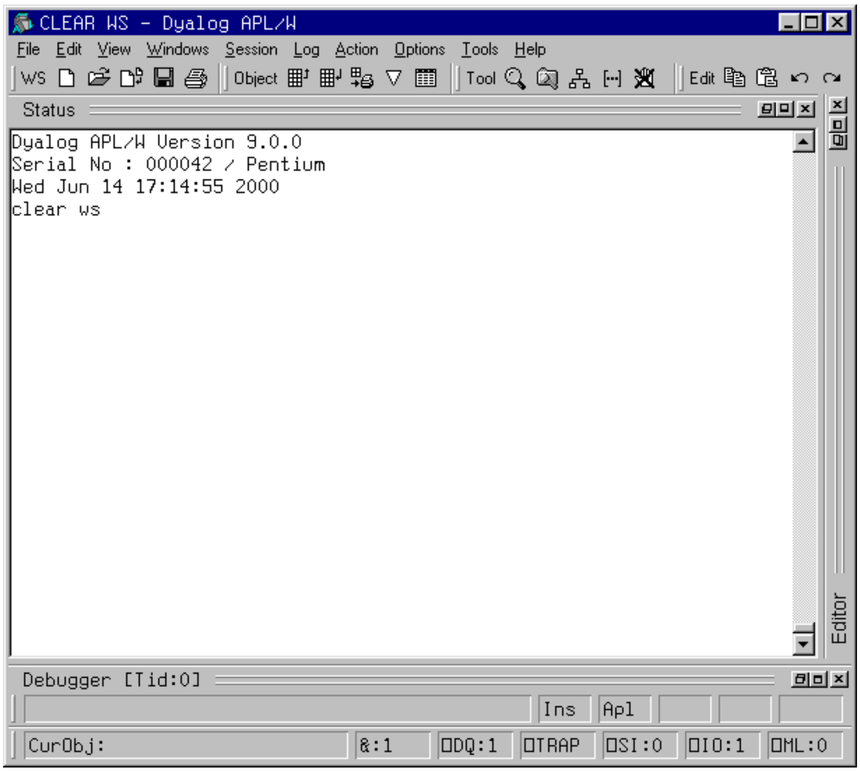
The next picture shows the result of the docking operation. The Session window is now split into 2 panes, with the Status window in the upper pane and the Session log window in the lower pane. You can resize the panes by dragging with the mouse.

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.



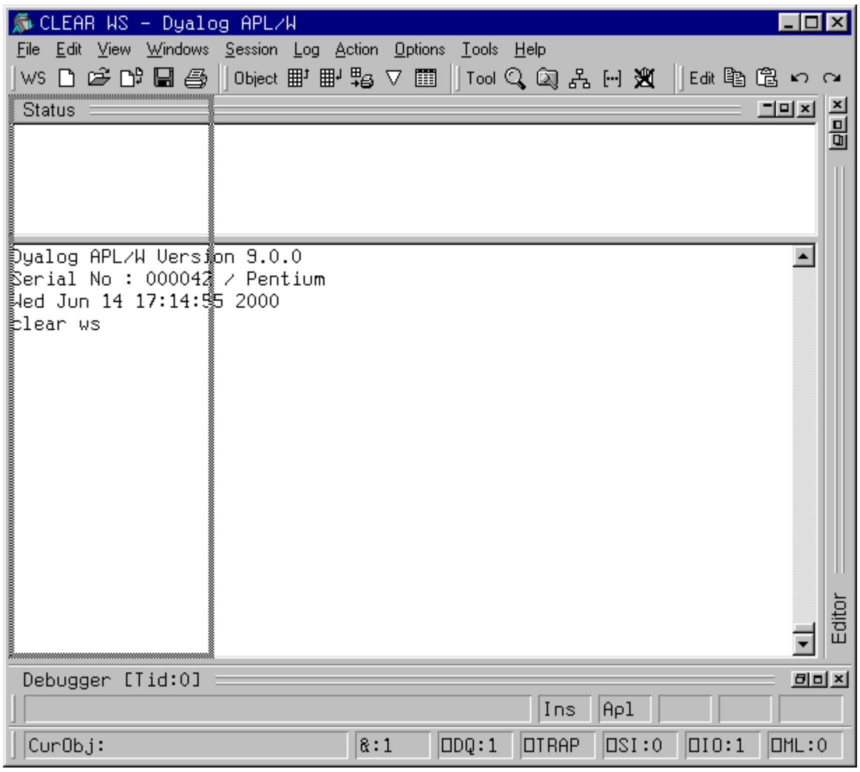
```
Dyalog APL/W Version 9.0.0
Serial No : 000042 / Pentium
Wed Jun 14 17:14:55 2000
clear ws
```

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.



```
Dyalog APL/W Version 9.0.0
Serial No : 000042 / Pentium
Wed Jun 14 17:14:55 2000
clear ws
```

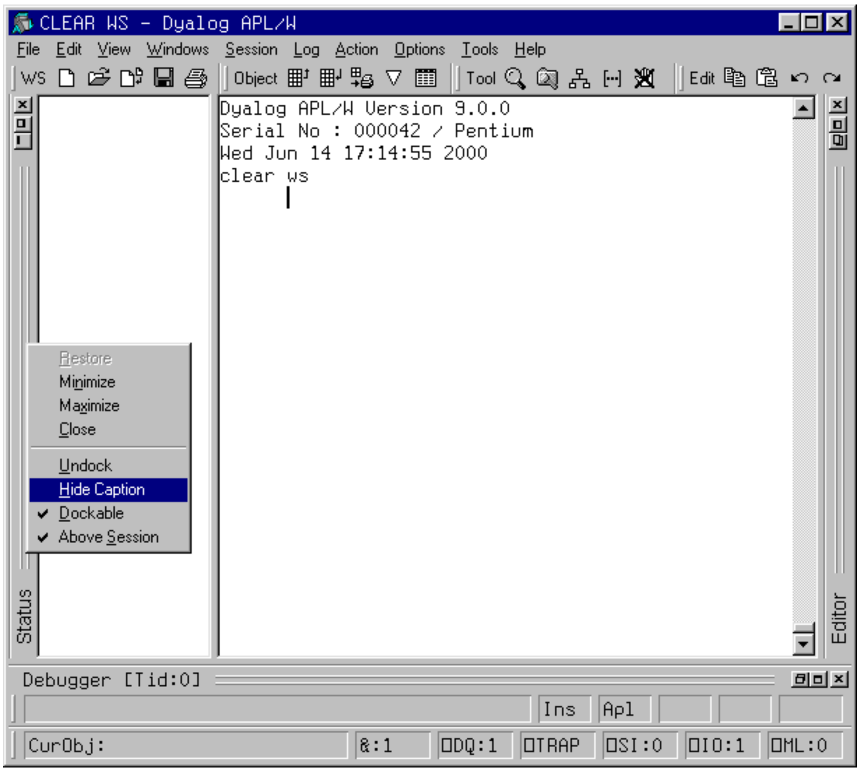
You can pick up a docked window and then re-dock it along a different edge of the Session as illustrated below.



```
Dyalog APL/W Version 9.0.0
Serial No : 000042 / Pentium
Wed Jun 14 17:14:55 2000
clear ws
```

CurObj: &:1 ⎕DQ:1 ⎕TRAP ⎕SI:0 ⎕IO:1 ⎕ML:0

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.

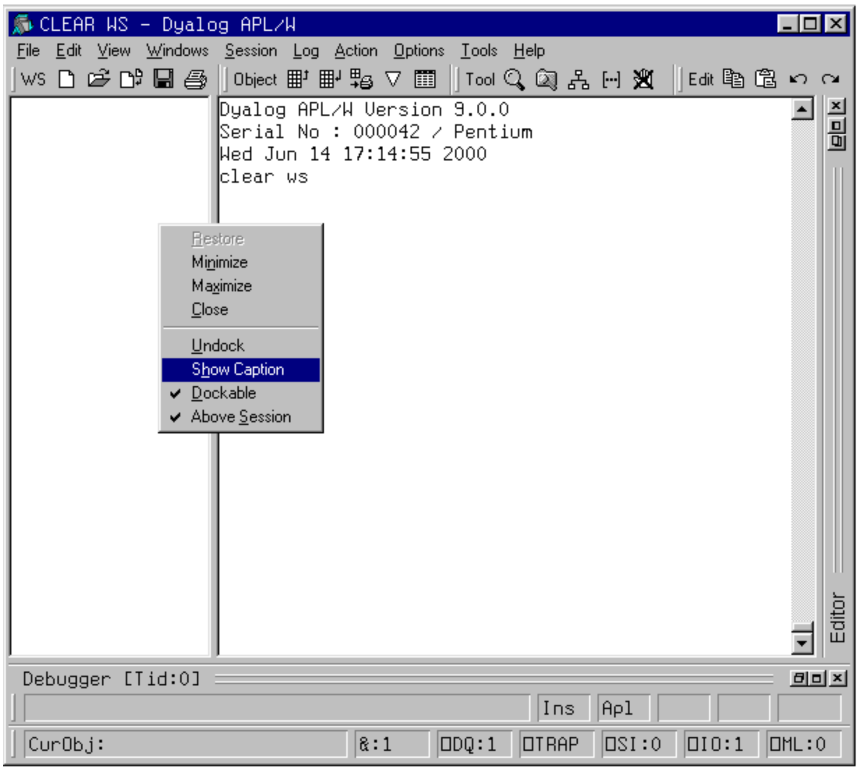


```
Dyalog APL/W Version 9.0.0
Serial No : 000042 / Pentium
Wed Jun 14 17:14:55 2000
clear ws
```

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.



```
Dyalog APL/W Version 9.0.0
Serial No : 000042 / Pentium
Wed Jun 14 17:14:55 2000
clear ws
```

The Session Log

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.

The session log is displayed using colour 250, which by default is Black on White. The input line is displayed using colour 249, which by default is Red on White. 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.

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.

Moving around the Session

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.

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 users terminal or scrren. 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 beome 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.

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`.

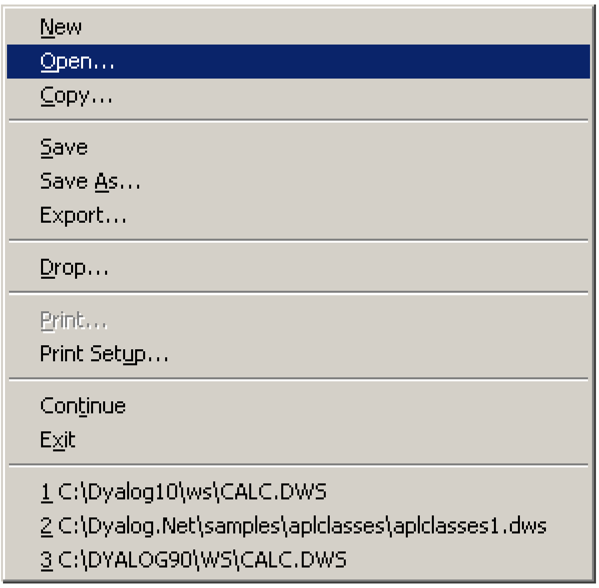
The statusbars in `⎕SE.cbbot`, are implemented by two CoolBand objects , `bandtb1` and `bandtb2`, each containing a StatusBar named `sb`.

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.


New
Open...
Copy...
Save
Save As...
Export...
Drop...
Print...
Print Setup...
Continue
Exit
1 C:\Dyalog10\ws\CALC.DWS
2 C:\Dyalog.Net\samples\aplclasses\aplclasses1.dws
3 C:\DYALOG90\WS\CALC.DWS

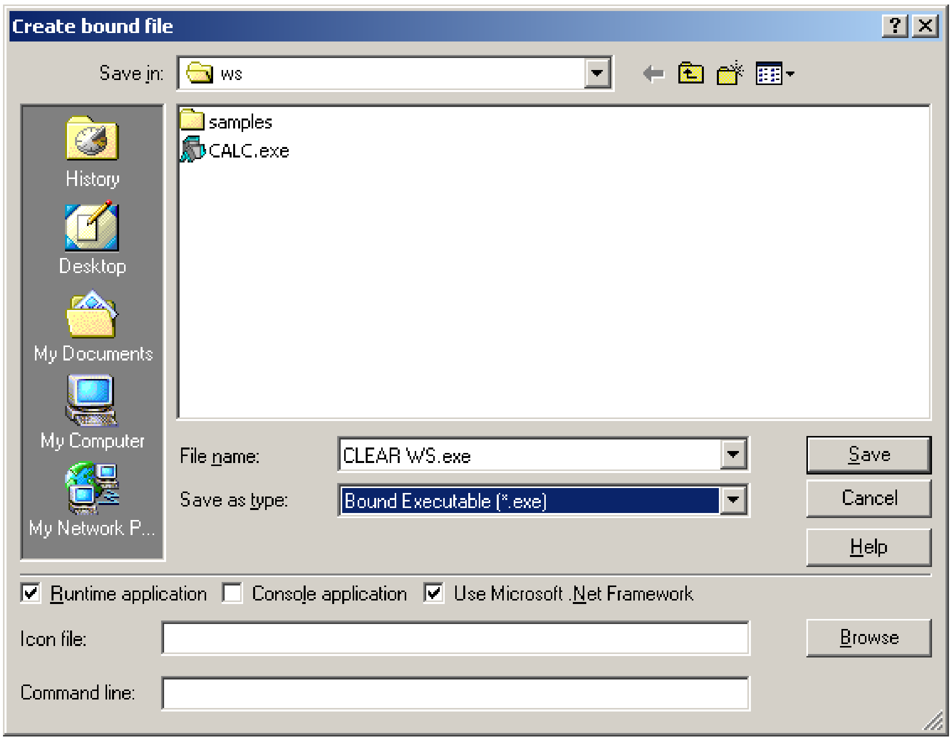
Item	Action	Description
New	`[WSClear]`	Prompts for confirmation, then clears the workspace
Open	`[WSLoad]`	Prompts for a workspace file name, then loads it
Copy	`[WSCopy]`	Prompts for a workspace file name, then copies it
Save	`[WSSave]`	Saves the active workspace
Save As	`[WSSaveas]`	Prompts for a workspace file name, then saves it
Export	`[Makeexe]`	Creates a bound executable, an OLE Server, an ActiveX Control, or a .Net Assembly
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.



Save in: ws

samples
CALC.exe

File name: CLEAR WS.exe

Save as type: Bound Executable (*.exe)

☑ Runtime application ☐ Console application ☑ Use Microsoft .Net Framework

Icon file:

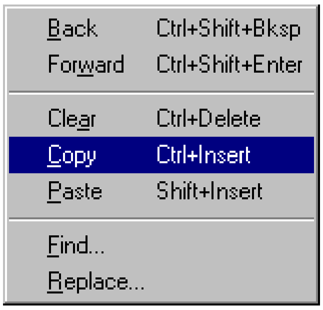
Command line:

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 `dyalog10rt.dll`. If not, it will be bound with `dyalog10.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.
Use Microsoft .Net Framework	If checked, your program will attempt to initialise the .Net Framework at start-up. Check this box only if your executable requires the .Net Framework.
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 specifiy the command line for the process. For a bound executable, this allows you to specify command-line parameters for the corresponding Dyalog APL DLL.

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. It may also contain the menu items to display the *Find* and *Find/Replace* dialog boxes as shown below. An alternative would be to place these in a separate *Search* menu. A typical Edit menu is shown below and is described in the table below.

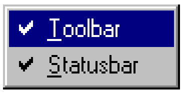


Item	**Action**	**Description**
Back	`[Undo]`	Displays the previous input line. Repeated use of this command cycles back through the input history.
Forward	`[Redo]`	Displays the next input line. Repeated use of this command cycles forward through the input history.
Clear	`[Delete]`	Clears the selected 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.
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 and Statusbars.

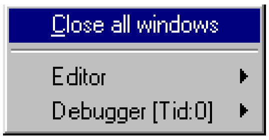


Item	Action	Description
Toolbar		Hides Session toolbars
Statusbar		Hides Session statusbars

View menu operations

The Windows 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

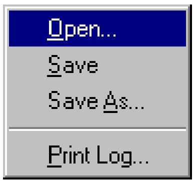
Windows menu operations

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

In addition, the Windows menu will contain options to switch 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.

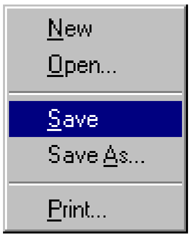


Item	Action	Description
Open	`[SELoad]`	Prompts for a session file name, then loads the session from it, replacing the current one. Sets the File property of `⎕SE` to the name of the file from which the session was loaded.
Save	`[SESave]`	Saves the current session (as defined by `⎕SE`) to the session 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`.
Print Log	`[PrintLog]`	Prints the contents of the session log.

Session menu operations

The Log Menu

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



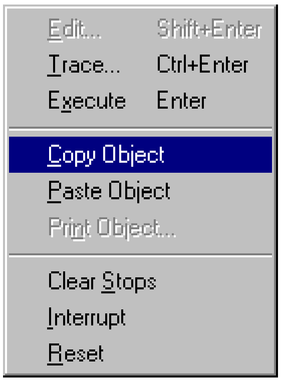
Item	Action	Description
New	`[NewLog]`	Prompts for confirmation, then empties the current Session log.
Open	`[OpenLog]`	Prompts for a Session log file, then loads it into memory, replacing the current Session log
Save	`[SaveLog]`	Saves the current Session log in the current log file, replacing its previous 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.

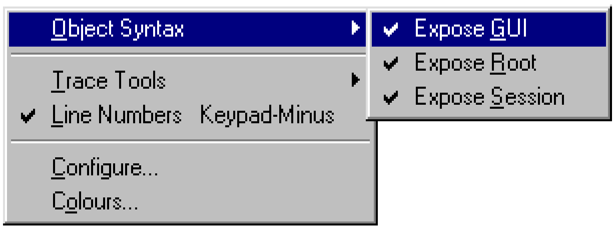


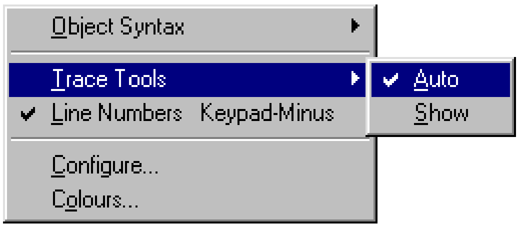
Item	**Action**	**Description**
Edit	`[Edit]`	Edit the current object
Trace	`[Trace]`	Executes the current line under the control of the Tracer
Execute	`[Execute]`	Executes the current line
Copy Object	`[ObjCopy]`	Copies the contents of the current object to the clipboard.
Paste Object	`[ObjPaste]`	Pastes 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	`[ExposeGUI]`	Exposes the names of properties, methods and events in GUI objects
Expose Root	`[ExposeRoot]`	Exposes the names of the properties, methods and events of the Root object
Expose Session	`[ExposeSession]`	Exposes the names of the properties, methods and events of `⎕SE`
Trace Tools/Auto	`[ToolsAuto]`	Causes the Trace Tools to appear at the start of a Trace and to disappear at the end
Trace Tools/Show	`[ToolsShow]`	Display Trace Tools
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 or hides the Object List dialog box

Options menu operations

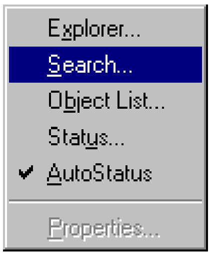
The values associated with the *Expose GUI*, *Expose Root* and *Expose Session* options reflect the values of these settings in your current workspace and are saved in it.

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.

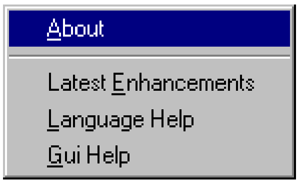


Item	Action	Description
Explorer	`[Explorer]`	Displays the workspace Explorer tool
Search	`[WSSearch]`	Displays the workspace Search tool
Object List	`[NameList]`	Displays or hides the Object List dialog box
Status	`[Status]`	Displays or hides the Status window
AutoStatus	`[AutoStatus]`	Toggle; if checked, causes the Status window to be displayed when a new message is generated for it
Properties	`[ObjProps]`	Displays a property sheet for the current object

Tools Menu Operations

The Help Menu

The Help menu (`⎕SE.mb.help`) provides access to the three help files distributed with Dyalog APL/W.

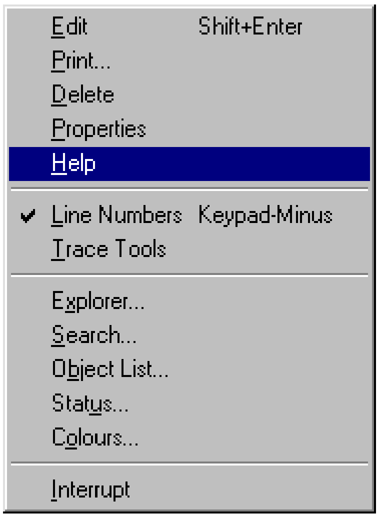


Item	Action	Description
About	`[About]`	Displays an *About* dialog box
Latest Enhancements	`[RelNotes]`	Provides help on the enhancements and improvements made since the previous Version
Language Help	`[LangHelp]`	Provides help on Dyalog APL language topics
Gui Help	`[GUIHelp]`	Provides help on topics concerning the interface between Dyalog APL/W and Windows

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.

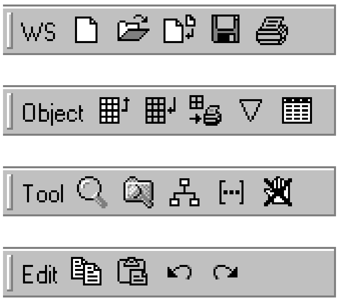


Item	Action	Description
Edit	`[Edit]`	Edits the current object
Print	`[ObjPrint]`	Prints the current object
Delete	`[ObjDelete]`	Erases the current object
Properties	`[GUIHelp]`	Displays the Object Properties dialog box for the current object
Help	`[Help]`	Displays the help topic associated with the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

Tools

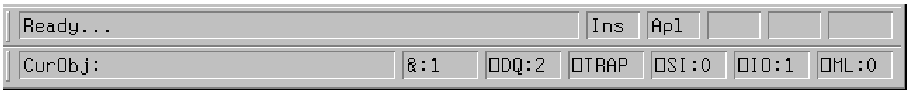
Explorer	Executes the system operation `[Explorer]` which displays the workspace Explorer tool.
Search	Executes the system operation `[WSSearch]` which displays the workspace Search tool.
Objects	Executes the system operation `[NameList]` which shows or hides the Object List dialog box.
Line Numbers	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.

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.



Ready... Ins Apl

CurObj: &:1 ⎕DQ:2 ⎕TRAP ⎕SI:0 ⎕IO:1 ⎕ML:0

The Session status bar

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.
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

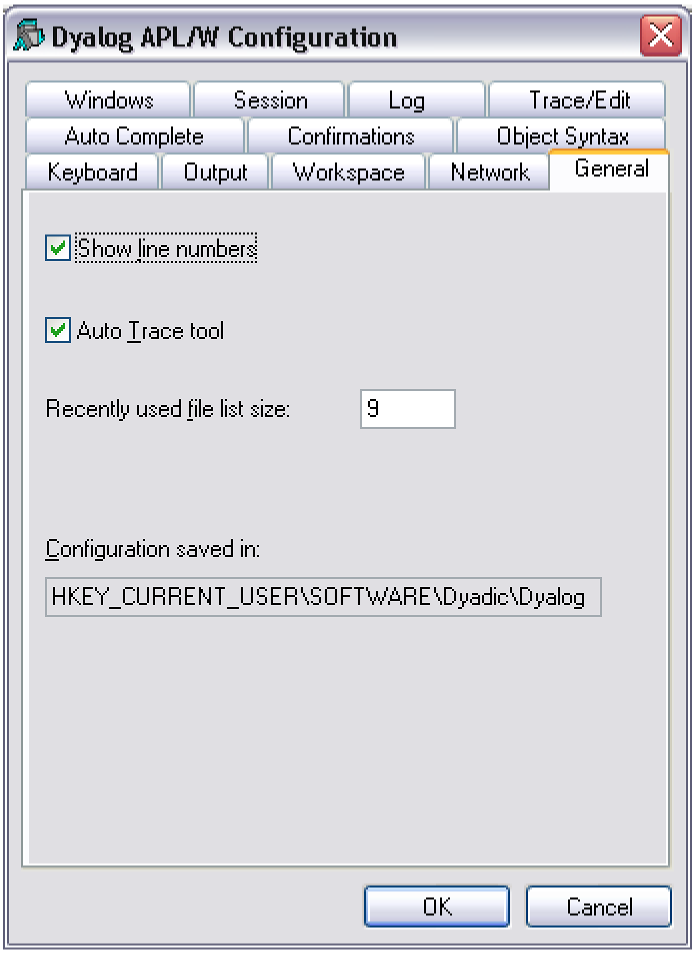
Toggle Status Fields

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 (Apl/Asc/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. Note that the keyboard mode only applies if you are using one of the APL_xx or the COMBO_xx keyboard tables.

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



☑ Show line numbers

☑ Auto Trace tool

Recently used file list size: 9

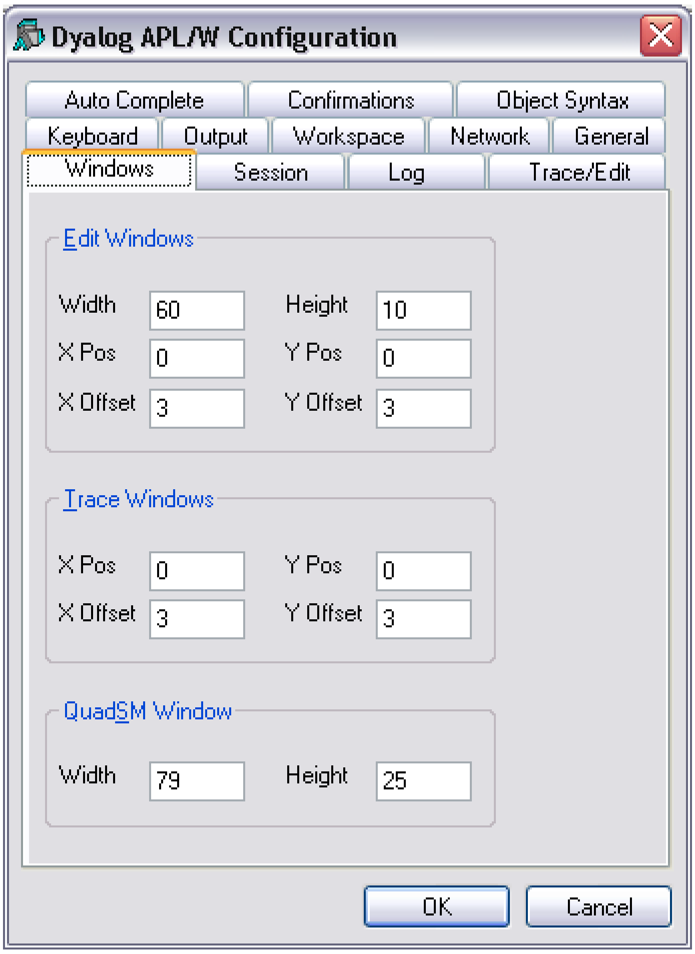
Configuration saved in:

HKEY_CURRENT_USER\SOFTWARE\Dyadic\Dyalog

Label	Parameter	Description
Show line numbers	lines_on_functions	Determines whether or not line numbers are shown in edit/trace windows
Auto_Trace tool	AutoTrace	Determines whether or not the Trace Tools are displayed when an error or stop occurs in a defined function
Recently used file list size	file_stack_size	Specifies the number of the most recently used workspaces displayed in the File menu.
Configuration saved in	inifile	Specifies the full pathname of the registry folder used by APL

Configuration dialog: General

Windows Tab



Edit Windows

Width 60
Height 10
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

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)

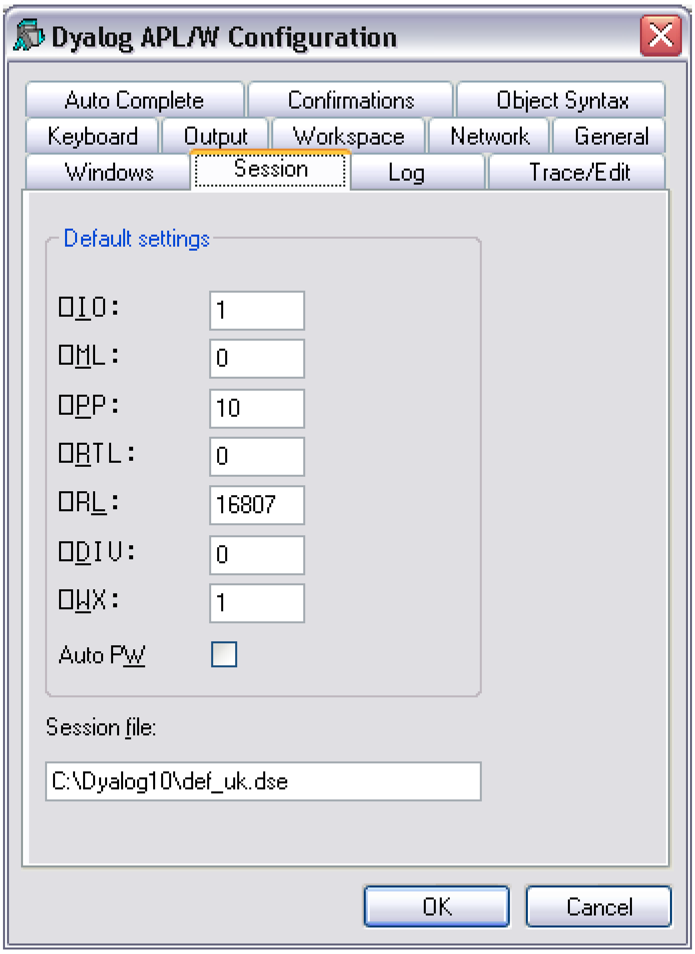
Label	Parameter	Description
X Pos	trace_first_x	The initial horizontal position in characters of the first trace window relative to the Session window
Y Pos	trace_first_y	The initial vertical position in characters of the first trace window relative to the Session window
X Offset	trace_offset_x	The initial 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



⎕IO: 1

⎕ML: 0

⎕PP: 10

⎕RTL: 0

⎕RL: 16807

⎕DIV: 0

⎕WX: 1

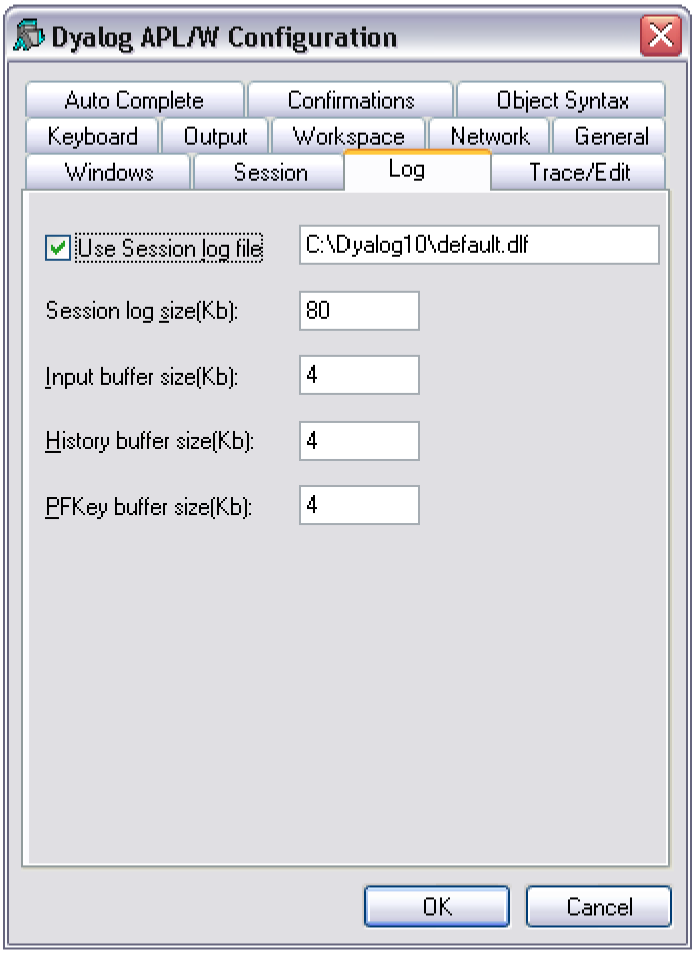
Auto PW ☐

Session file: C:\Dyalog10\def_uk.dse

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



☑ Use Session log file: C:\Dyalog10\default.dlf

Session log size(Kb): 80

Input buffer size(Kb): 4

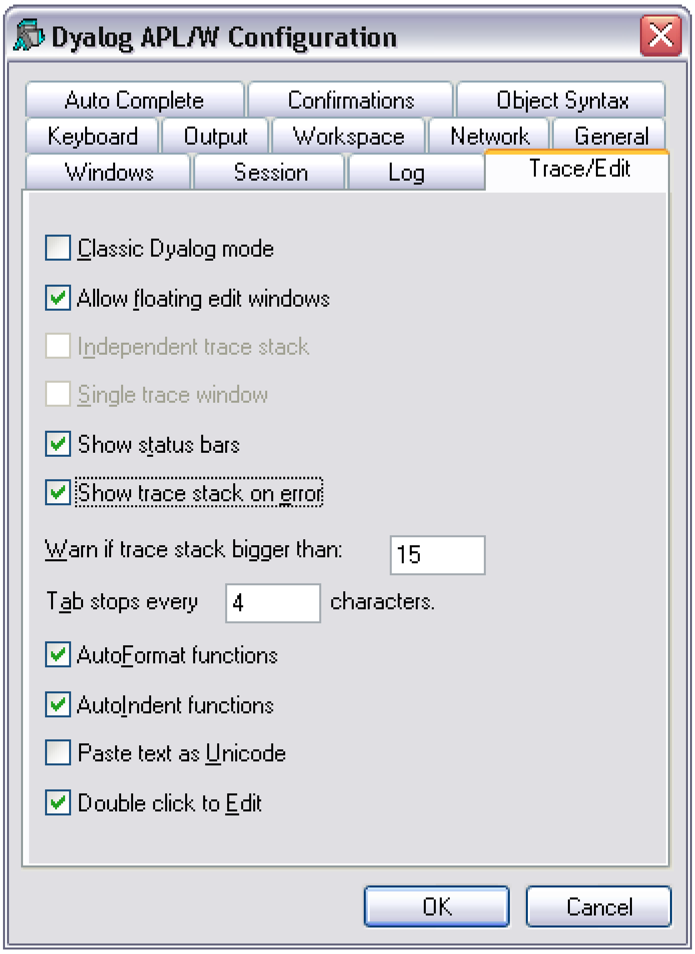
History buffer size(Kb): 4

PFKey buffer size(Kb): 4

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
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



☐ Classic Dyalog mode

☑ Allow floating edit windows

☐ Independent trace stack

☐ Single trace window

☑ Show status bars

☑ Show trace stack on error

Warn if trace stack bigger than: 15

Tab stops every 4 characters.

☑ AutoFormat functions

☑ AutoIndent functions

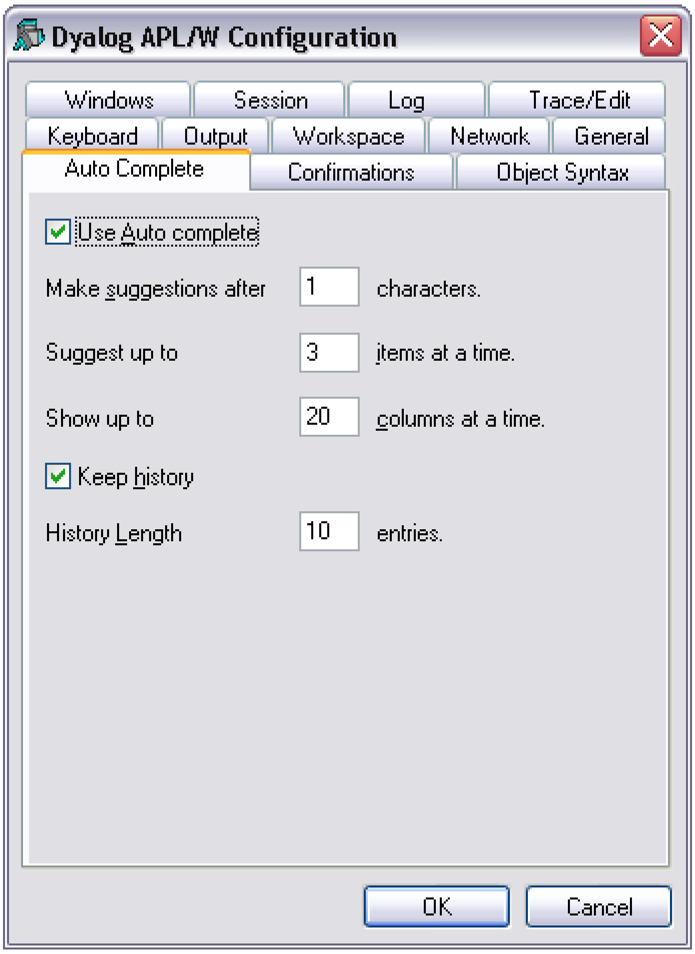
☐ Paste text as Unicode

☑ Double click to Edit

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

Configuration dialog: Trace/Edit

Auto Complete Tab



☑ Use Auto complete

Make suggestions after 1 characters.

Suggest up to 3 items at a time.

Show up to 20 columns at a time.

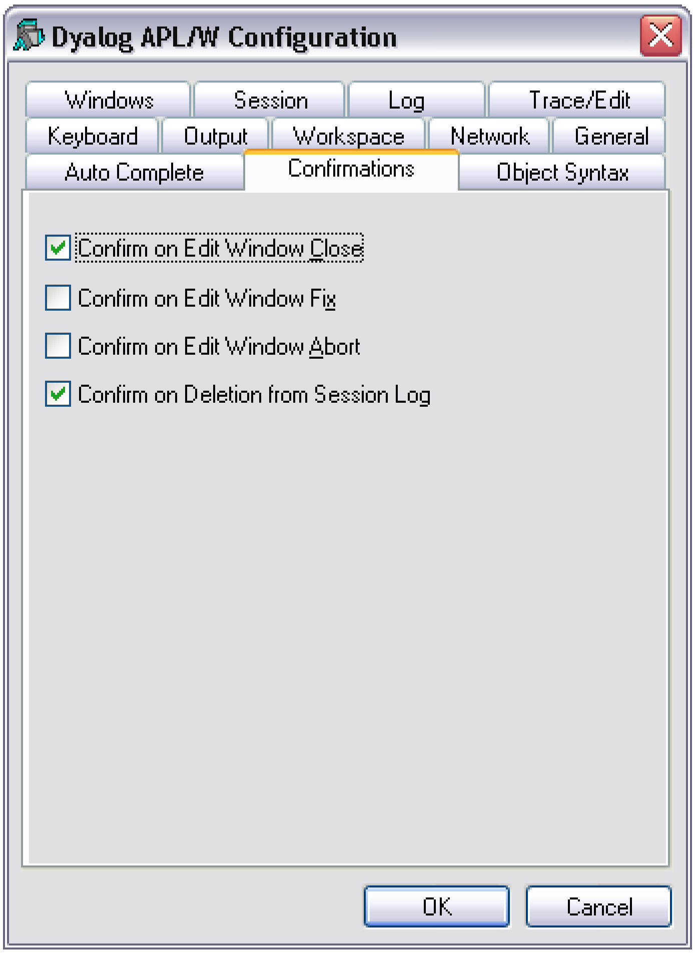
☑ Keep history

History Length 10 entries.

Label	Parameter	Description
Use Auto Complete	Enabled	Specifies whther 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

Configuration dialog: Auto Complete

Confirmations Tab



☑ Confirm on Edit Window Close

☐ Confirm on Edit Window Fix

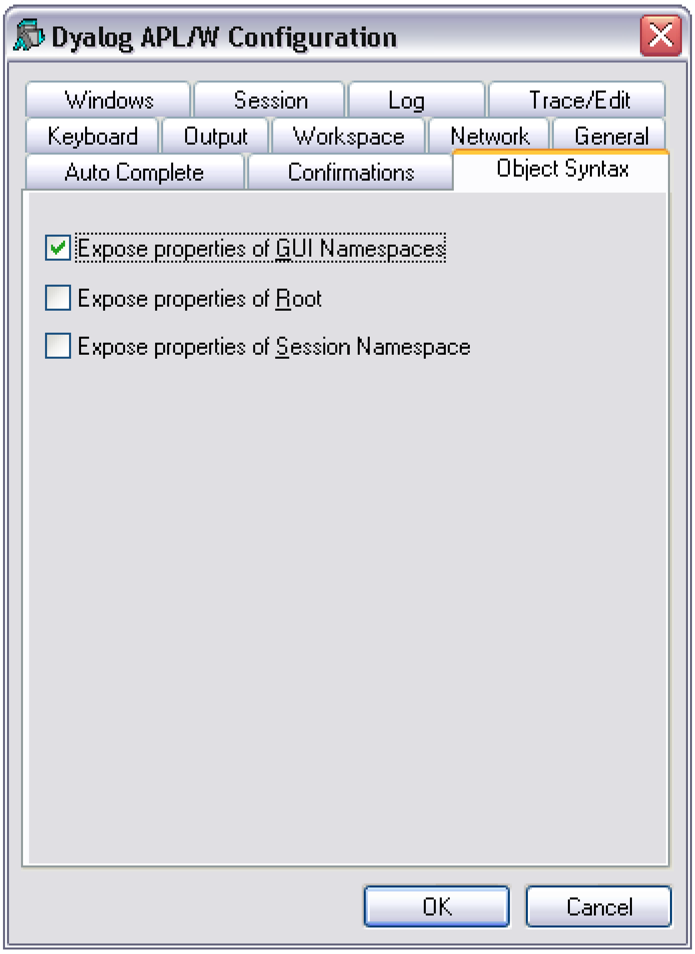
☐ Confirm on Edit Window Abort

☑ Confirm on Deletion from Session Log

Label	**Parameter**	**Description**
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

Configuration dialog: Confirmations

Object Syntax Tab



☑ Expose properties of GUI Namespaces

☐ Expose properties of Root

☐ Expose properties of Session Namespace

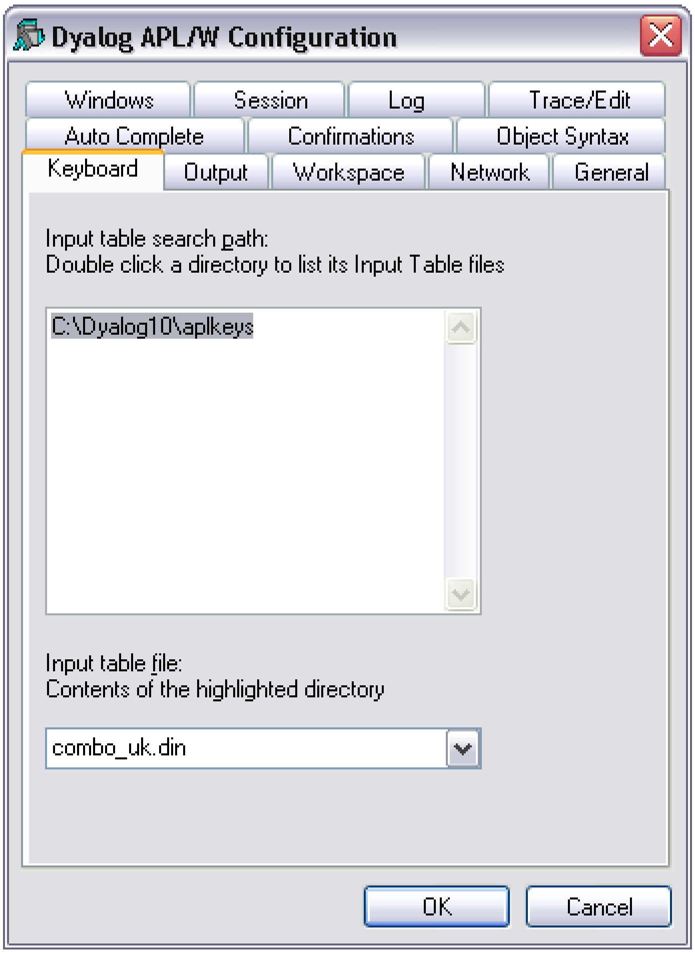
Label	**Parameter**	**Description**
Expose properties of GUI Namespaces	default_wx	Specifies the value of ⎕WX in a clear workspace. This in turn determines whether or not the names of properties, methods and events of GUI objects are exposed. If set (⎕WX is 1), you may query/set properties and invoke 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 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 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.

Keyboard Tab



Input table search path:
Double click a directory to list its Input Table files

C:\Dyalog10\aplkeys

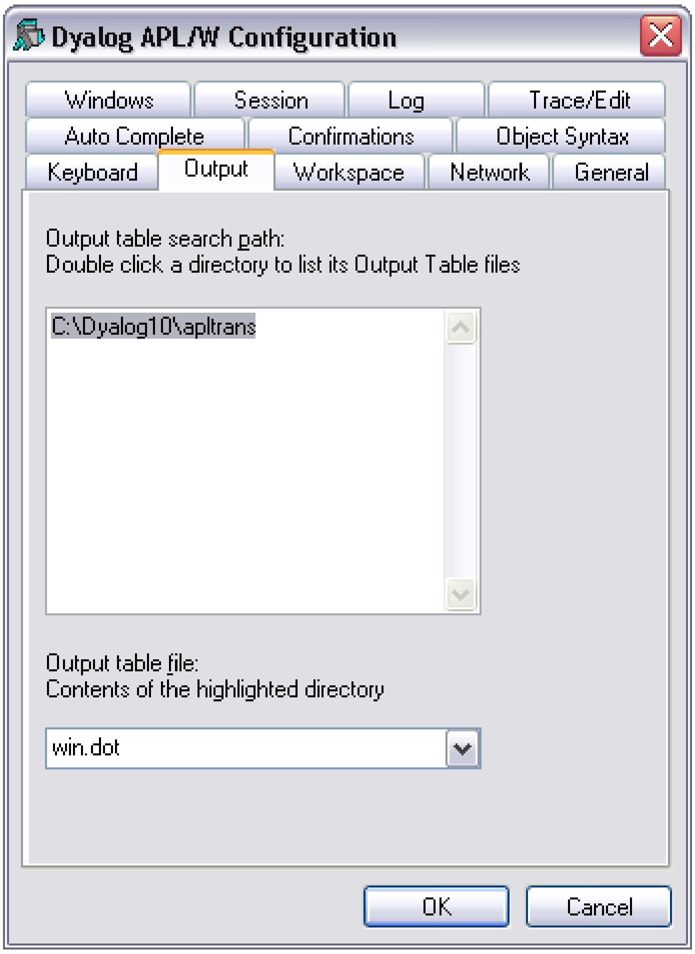
Input table file:
Contents of the highlighted directory

combo_uk.din

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



Output table search path:
Double click a directory to list its Output Table files

C:\Dyalog10\apltrans

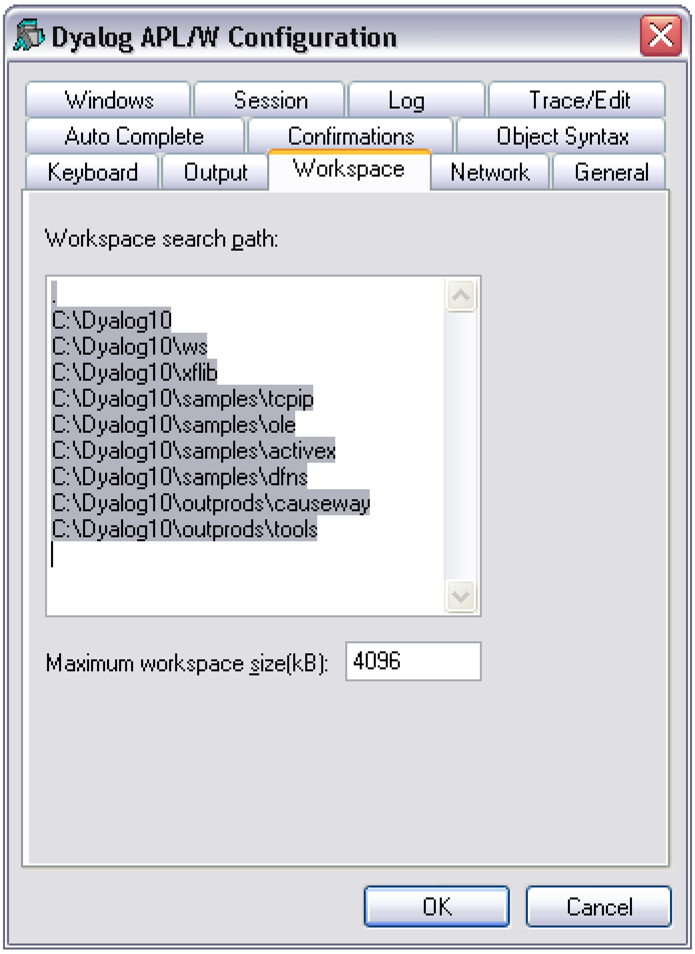
Output table file:
Contents of the highlighted directory

win.dot

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

Workspace Tab



Workspace search path:

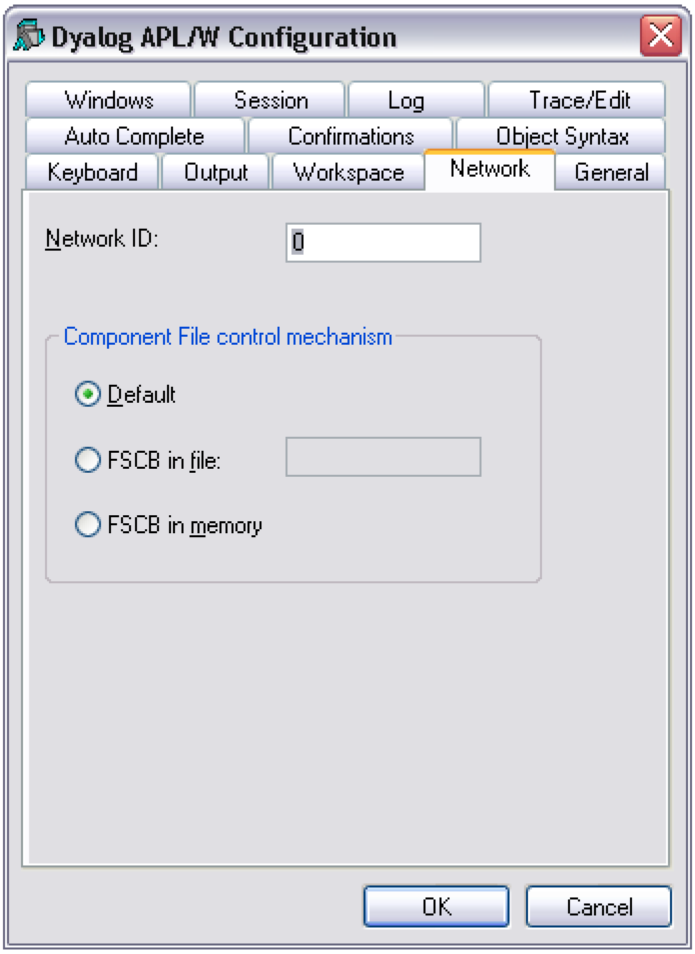
```
.
C:\Dyalog10
C:\Dyalog10\ws
C:\Dyalog10\xflib
C:\Dyalog10\samples\tcpip
C:\Dyalog10\samples\ole
C:\Dyalog10\samples\activex
C:\Dyalog10\samples\dfns
C:\Dyalog10\outprods\causeway
C:\Dyalog10\outprods\tools
```

Maximum workspace size(kB): 4096

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.

Configuration dialog: Workspace

Network Tab



Network ID: 0

Component File control mechanism

● Default

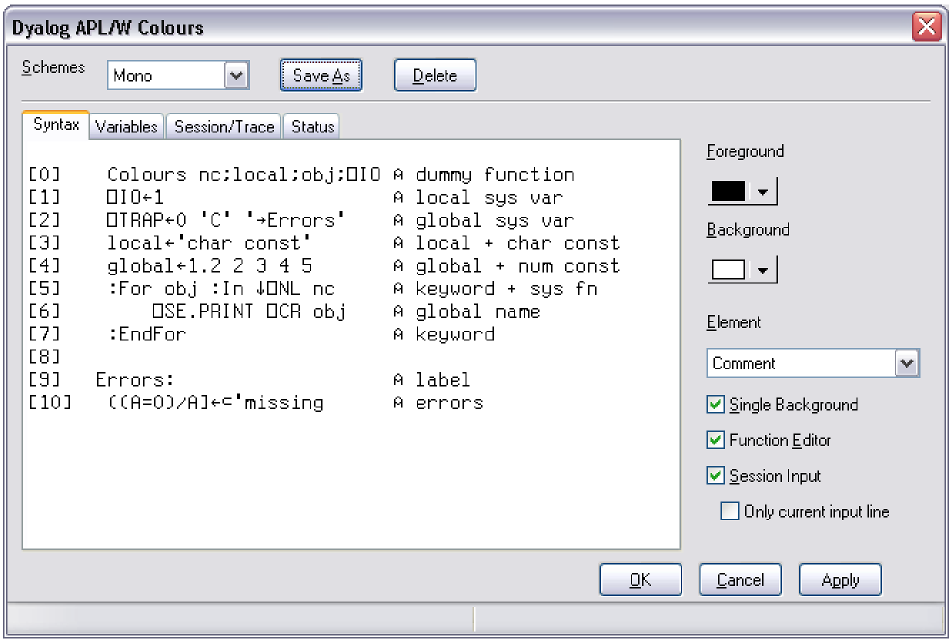
○ FSCB in file:

○ FSCB in memory

Label	**Parameter**	**Description**
Network ID	aplnid	A number that uniquely identifies the user for component file system access control
Default	File_Control (2)	Specifies that the component file system uses file locking to control multi-user access
FSCB in file	File_Control (1)	Specifies that the component file system uses a file-based FSCB to control multi-user access
FSCB in memory	File_Control (0)	Specifies that the component file system uses a memory-based FSCB to control multi-user access

Configuration dialog: Workspace

Colour Selection Dialog



Schemes: Mono

```
[0]    Colours nc;local;obj;⎕IO ⍝ dummy function
[1]    ⎕IO←1                    ⍝ local sys var
[2]    ⎕TRAP←0 'C' '→Errors'    ⍝ global sys var
[3]    local←'char const'       ⍝ local + char const
[4]    global←1.2 2 3 4 5       ⍝ global + num const
[5]    :For obj :In ↓⎕NL nc     ⍝ keyword + sys fn
[6]        ⎕SE.PRINT ⎕CR obj    ⍝ global name
[7]    :EndFor                  ⍝ keyword
[8]
[9]   Errors:                   ⍝ label
[10]   ((A=0)/A]←⊂'missing      ⍝ errors
```

Element: Comment

☑ Single Background
☑ Function Editor
☑ Session Input
☐ Only current input line

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.

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.

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.

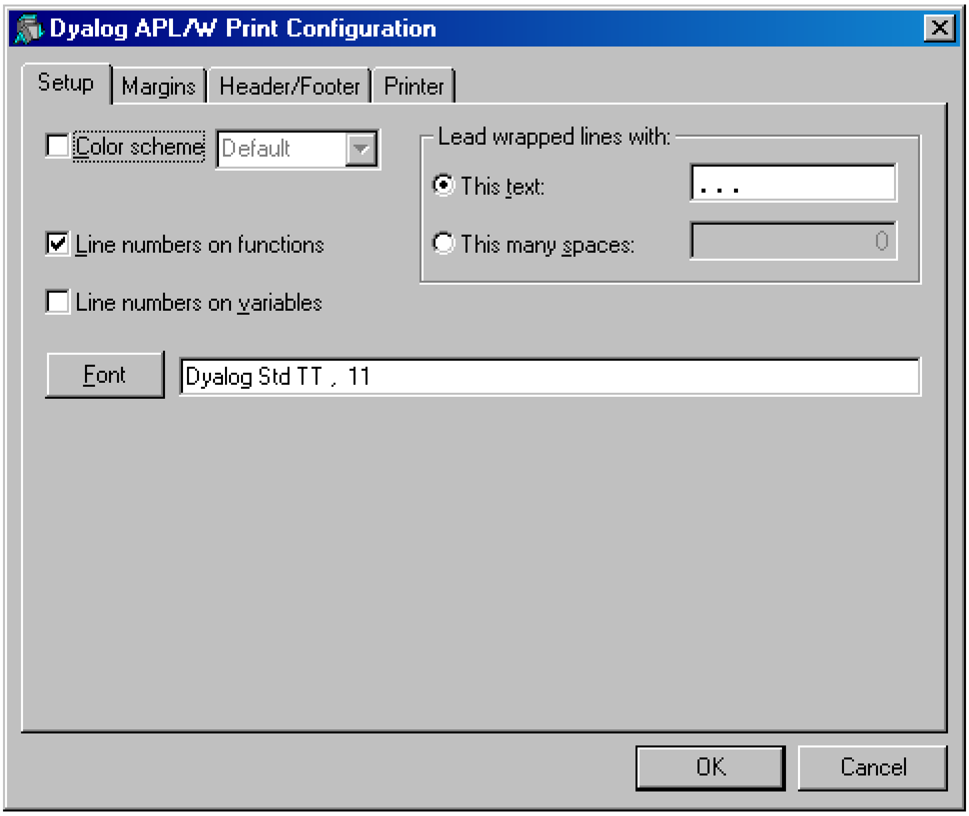
Print Configuration Dialog Box

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



Dyalog APL/W Print Configuration

Setup | Margins | Header/Footer | Printer

☐ Color scheme: Default

Lead wrapped lines with:

● This text: . . .

○ This many spaces: 0

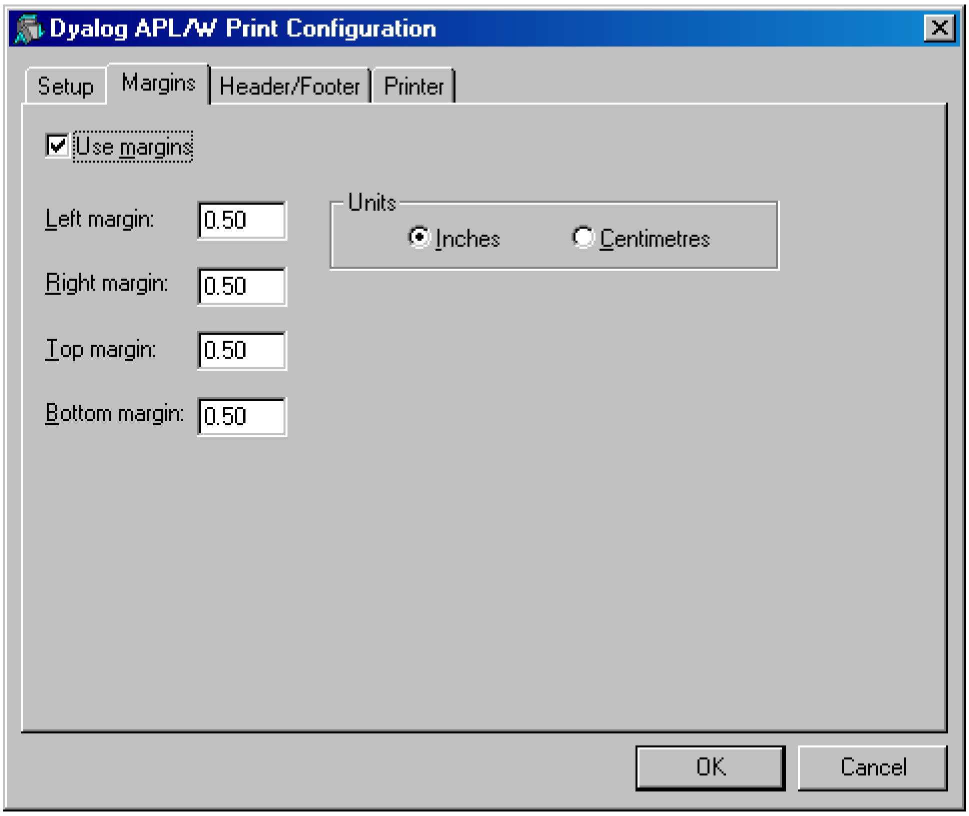
☑ Line numbers on functions

☐ Line numbers on variables

Font: Dyalog Std TT , 11

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



☑ Use margins

Left margin: 0.50

Right margin: 0.50

Top margin: 0.50

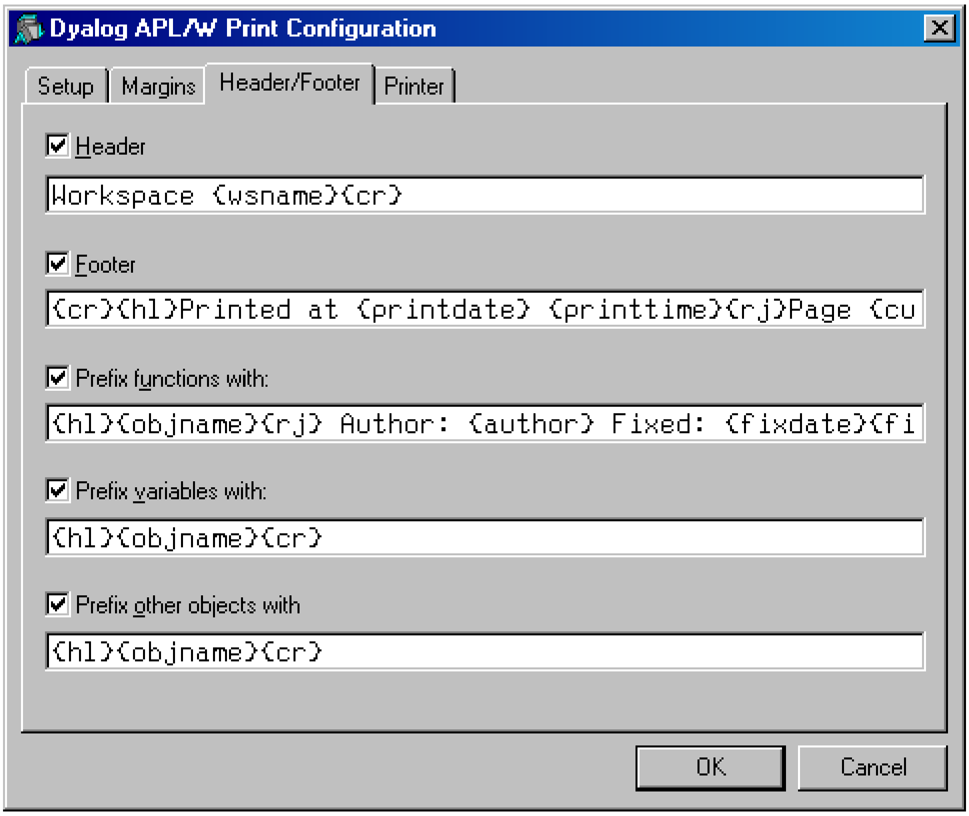
Bottom margin: 0.50

Units

● Inches ○ Centimetres

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



☑ Header

Workspace {wsname}{cr}

☑ Footer

{cr}{hl}Printed at {printdate} {printtime}{rj}Page {cu

☑ Prefix functions with:

{hl}{objname}{rj} Author: {author} Fixed: {fixdate}{fi

☑ Prefix variables with:

{hl}{objname}{cr}

☑ Prefix other objects with

{hl}{objname}{cr}

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

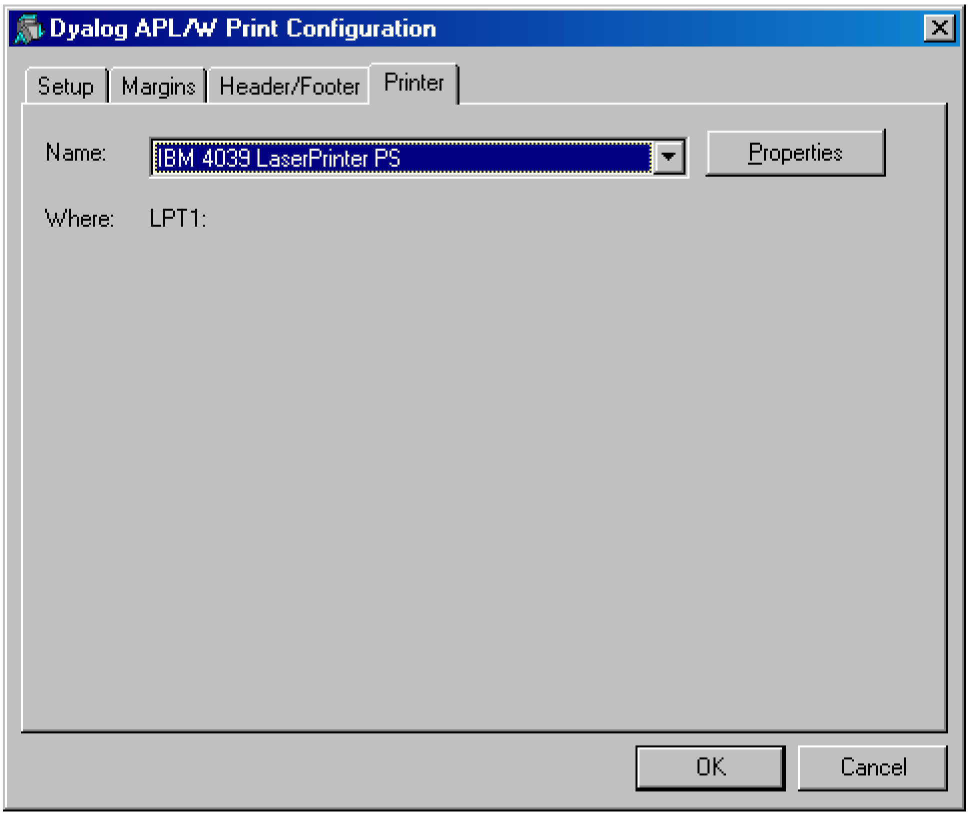
For example, the specification :

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



Name: IBM 4039 LaserPrinter PS

Where: LPT1:

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

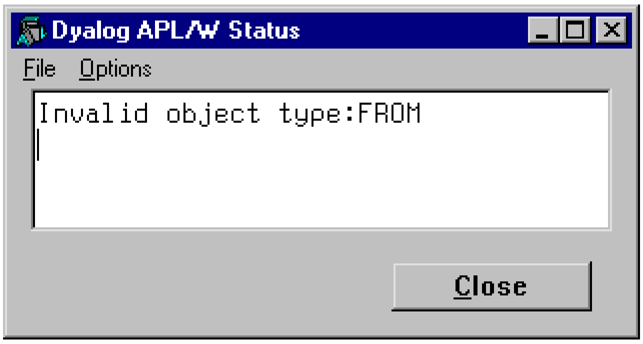
Status Window

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'
     ^
```



```
Invalid object type:FROM
```

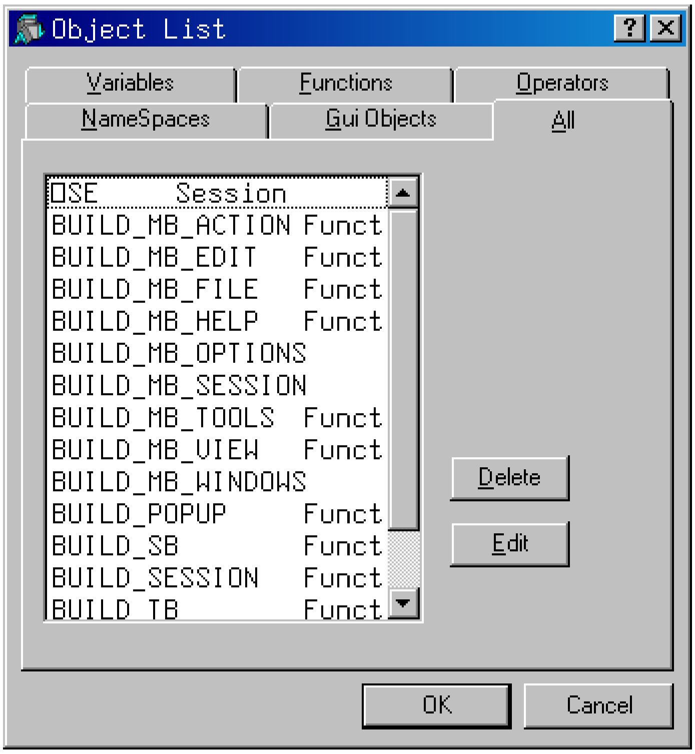
The Status window can be explicitly displayed or hidden using the `[Status]` system operation which is associated with the *Tools/Status* menu item.

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 Object List

The Object List dialog box displays a list of object in the active workspace or namespace. You may selectively display objects of different types and you may use the list to pick objects to be edited or deleted. A typical Object List is illustrated below.



⎕SE	Session
BUILD_MB_ACTION	Funct
BUILD_MB_EDIT	Funct
BUILD_MB_FILE	Funct
BUILD_MB_HELP	Funct
BUILD_MB_OPTIONS	
BUILD_MB_SESSION	
BUILD_MB_TOOLS	Funct
BUILD_MB_VIEW	Funct
BUILD_MB_WINDOWS	
BUILD_POPUP	Funct
BUILD_SB	Funct
BUILD_SESSION	Funct
BUILD_TB	Funct

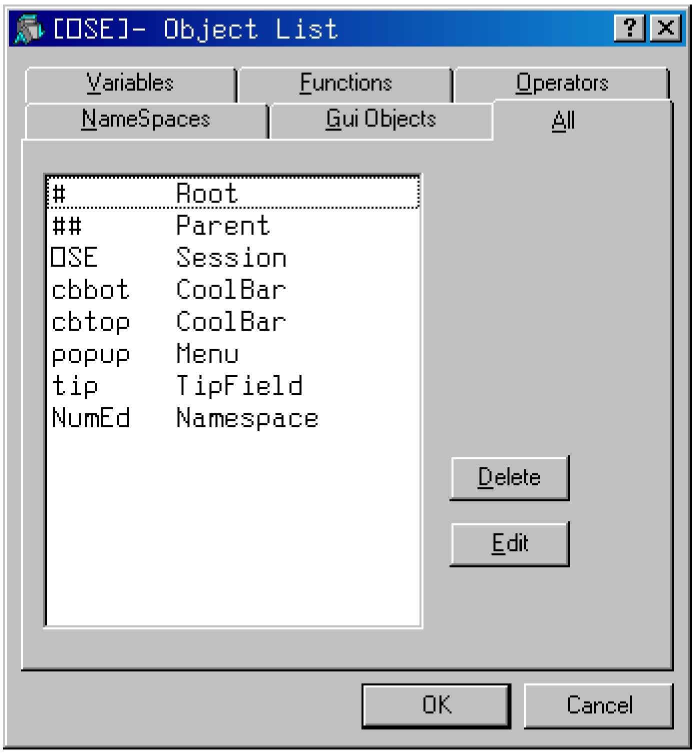
The Object List

To edit a function or variable, double-click its name or select its name and click Edit. You may also select several names (drag or Shift+Click extends the selection; Ctrl+Click selects/de-selects an item) and then select *Edit* or *Delete* as appropriate.

You will notice that the name of the Session object `⎕SE` always appears in the Object List.

Using the Object List with Namespaces

The edit operation on a namespace is interpreted as a request to change to that namespace. You can therefore use the Object List to navigate down a namespace structure and back up again. For example, if you double-click on ⎕SE, the system changes into ⎕SE; the same as if you had typed `)CS ⎕SE`. The Object List now shows only the objects visible in the ⎕SE, as illustrated below.

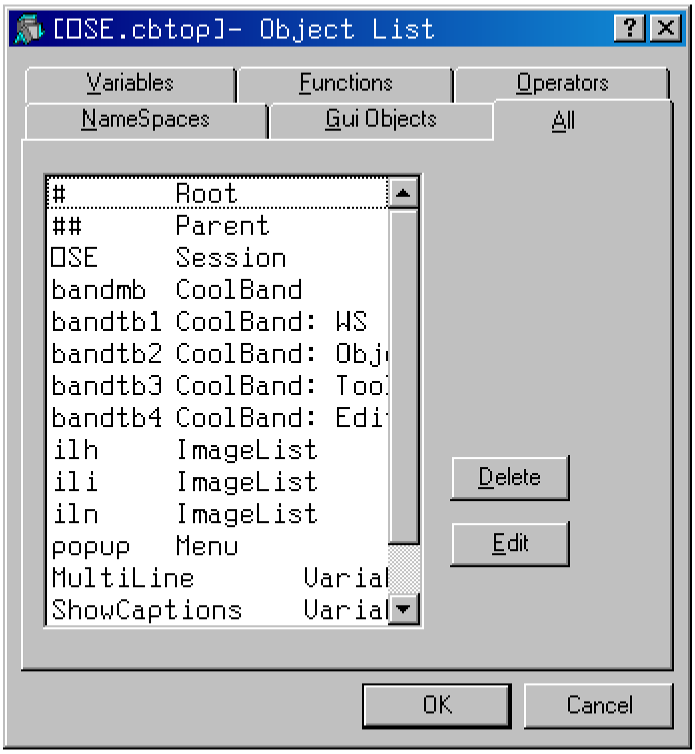


```
#       Root
##      Parent
⎕SE     Session
cbbot   CoolBar
cbtop   CoolBar
popup   Menu
tip     TipField
NumEd   Namespace
```

Within the ⎕SE namespace

Once in a namespace, you are offered two further choices that enable you to switch back to the parent namespace (`##`) or right back to the root workspace (`#`).

If the namespace contains other namespaces, you can go further into the structure. For example, once in *⎕SE* you may switch into the *⎕SE.cbtop* namespace by clicking *mb*. Once within this object, the Object List will then appear as shown below.



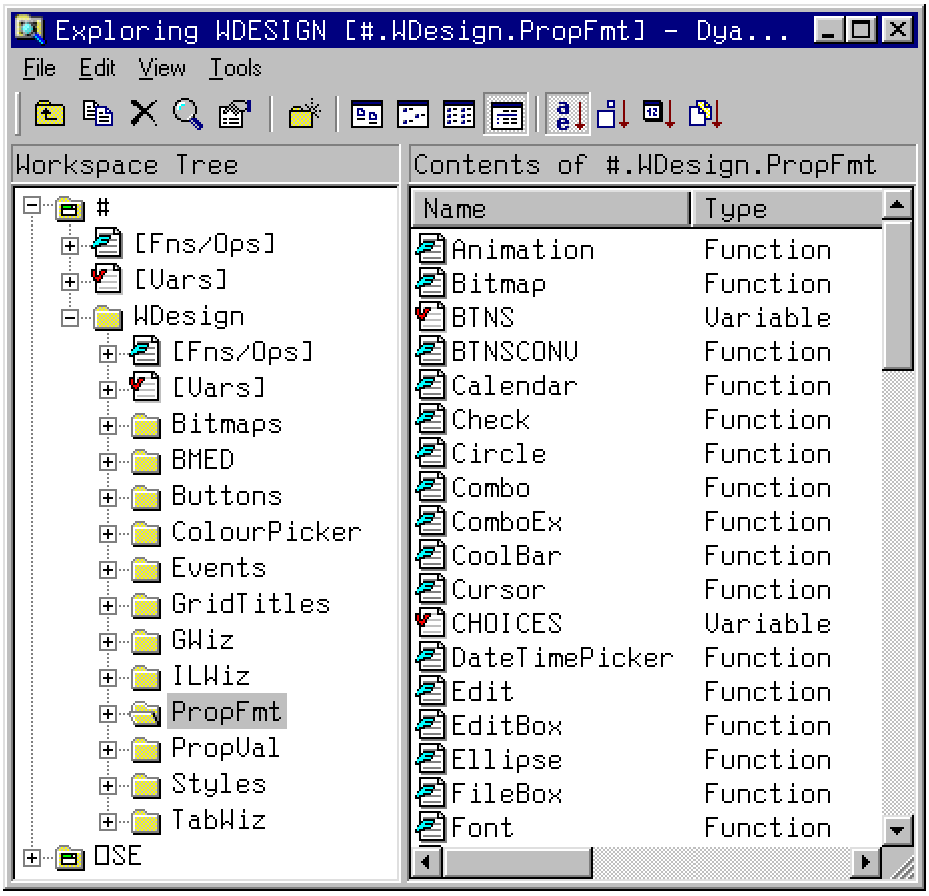
```
#       Root
##      Parent
⎕SE     Session
bandmb  CoolBand
bandtb1 CoolBand: WS
bandtb2 CoolBand: Obj
bandtb3 CoolBand: Too
bandtb4 CoolBand: Edi
ilh     ImageList
ili     ImageList
iln     ImageList
popup   Menu
MultiLine       Varial
ShowCaptions    Varial
```

Within the *⎕SE.cbtop* namespace

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.



Name	Type
Animation	Function
Bitmap	Function
BTNS	Variable
BTNSCONV	Function
Calendar	Function
Check	Function
Circle	Function
Combo	Function
ComboEx	Function
CoolBar	Function
Cursor	Function
CHOICES	Variable
DateTimePicker	Function
Edit	Function
EditBox	Function
Ellipse	Function
FileBox	Function
Font	Function

The Explorer is closely modelled on the *Windows Explorer* in Windows 98 and the facilities it provides are very similar. For Windows 98 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.

📁	Namespace (closed)
📁	GUI Namespace (closed)
📂	Namespace (open)
📂	GUI Namespace (open)
F	Function
V	Variable
OP	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 report 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.

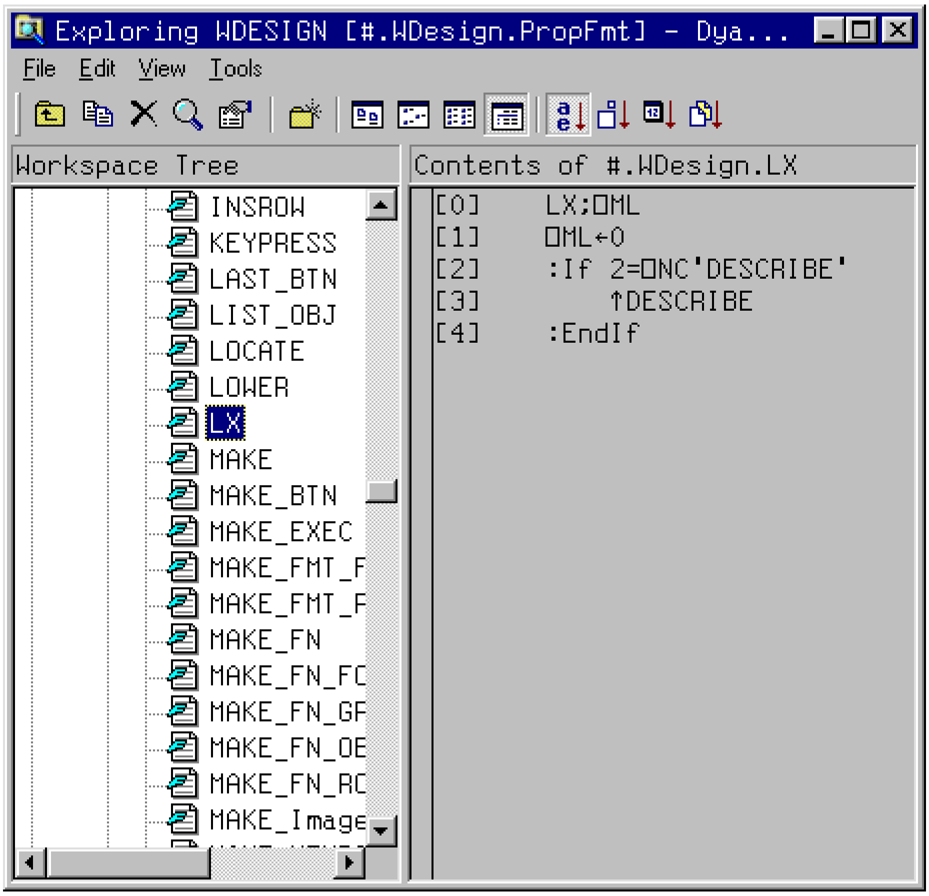
You may also move and copy objects using the Edit menu. To do so, select the object(s) and then choose Move or Copy from the Edit menu. You will be prompted for the name of the namespace into which the objects are to be moved or copied. Enter the namespace and click OK.

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.

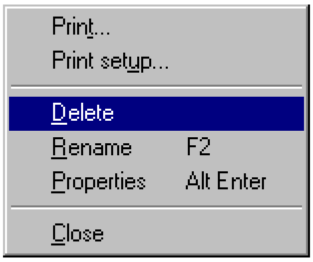


```
[0]     LX;⎕ML
[1]     ⎕ML←0
[2]     :If 2=⎕NC'DESCRIBE'
[3]         ↑DESCRIBE
[4]     :EndIf
```

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.

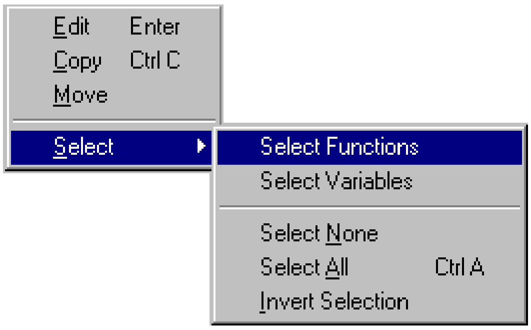
The File Menu



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

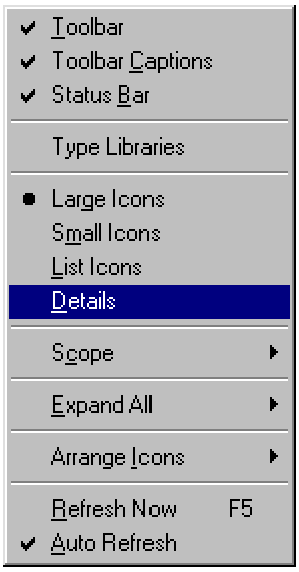
The Edit Menu



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 View Menu



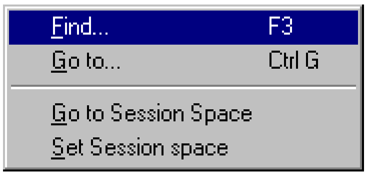
The View menu, illustrated above, provides the following actions.

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
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.
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`.
Arrange Icons	Sorts the items in the ListView by name, type, size or date.
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 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 Type Libraries and .Net Metadata

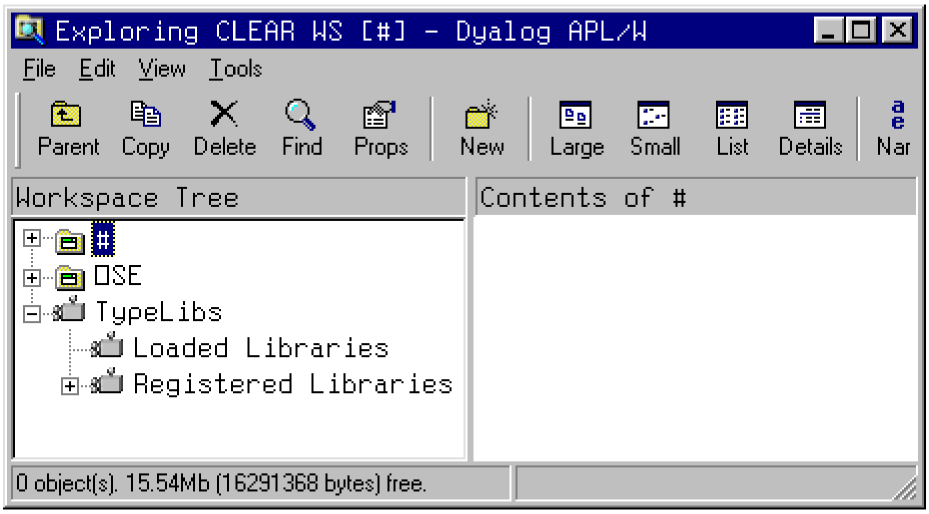
When the *View/Type Libraries* option is enabled, the Workspace Explorer allows you to:

- 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.

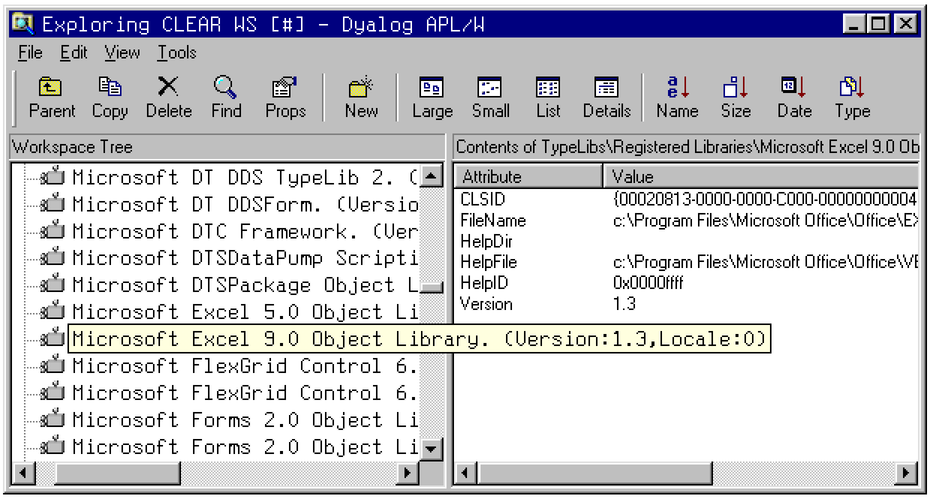


Browsing Registered Libraries

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.

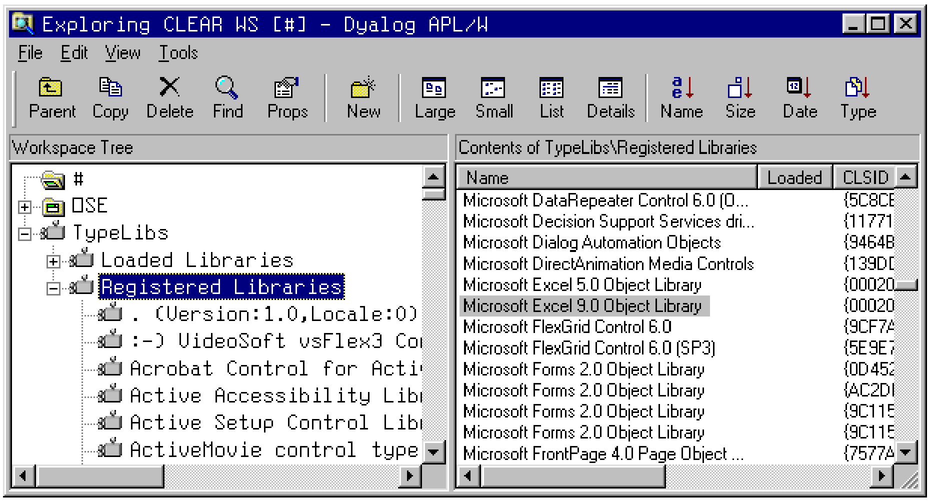
For example, the result of selecting the Microsoft Excel 9.0 Object Library is illustrated below.


Exploring CLEAR WS [#] - Dyalog APL/W
File Edit View Tools
Parent Copy Delete Find Props New Large Small List Details Name Size Date Type
Workspace Tree
Contents of TypeLibs\Registered Libraries\Microsoft Excel 9.0 Ob

Microsoft DT DDS TypeLib 2. (
Microsoft DT DDSForm. (Versio
Microsoft DTC Framework. (Ver
Microsoft DTSDataPump Scripti
Microsoft DTSPackage Object L
Microsoft Excel 5.0 Object Li
Microsoft Excel 9.0 Object Library. (Version:1.3,Locale:0)
Microsoft FlexGrid Control 6.
Microsoft FlexGrid Control 6.
Microsoft Forms 2.0 Object Li
Microsoft Forms 2.0 Object Li

Attribute	Value
CLSID	{00020813-0000-0000-C000-000000000004
FileName	c:\Program Files\Microsoft Office\Office\E:
HelpDir	
HelpFile	c:\Program Files\Microsoft Office\Office\VE
HelpID	0x0000ffff
Version	1.3

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


Exploring CLEAR WS [#] - Dyalog APL/W
File Edit View Tools
Parent Copy Delete Find Props New Large Small List Details Name Size Date Type
Workspace Tree
Contents of TypeLibs\Registered Libraries

- #
- ⎕SE
- TypeLibs
  - Loaded Libraries
  - Registered Libraries
    - . (Version:1.0,Locale:0)
    - :-) VideoSoft vsFlex3 Co
    - Acrobat Control for Acti
    - Active Accessibility Lib
    - Active Setup Control Lib
    - ActiveMovie control type

Name	Loaded	CLSID
Microsoft DataRepeater Control 6.0 (O...		{5C8CE
Microsoft Decision Support Services dri...		{11771
Microsoft Dialog Automation Objects		{9464B
Microsoft DirectAnimation Media Controls		{139DC
Microsoft Excel 5.0 Object Library		{00020
Microsoft Excel 9.0 Object Library		{00020
Microsoft FlexGrid Control 6.0		{9CF7A
Microsoft FlexGrid Control 6.0 (SP3)		{5E9E7
Microsoft Forms 2.0 Object Library		{0D452
Microsoft Forms 2.0 Object Library		{AC2DI
Microsoft Forms 2.0 Object Library		{9C115
Microsoft Forms 2.0 Object Library		{9C115
Microsoft FrontPage 4.0 Page Object ...		{7577A

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 loading a Type Library may take a few moments to complete.

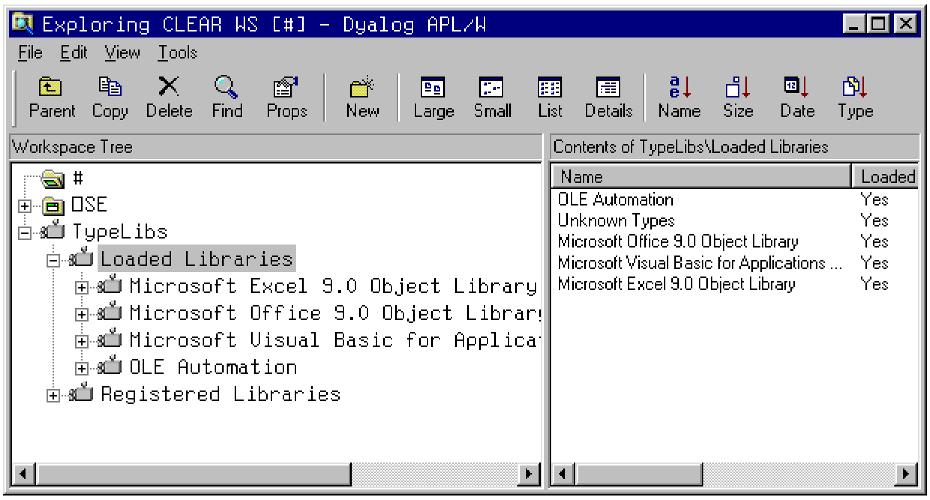
Notice that if the selected Library references any other libraries, they too will be loaded. For example, loading the *Microsoft Excel 9.0 Object Library* brings in the *Microsoft Office 9.0 Object Library* and the *Microsoft Visual Basic for Applications Extensibility 5.3 Library* too. It also contains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



#
⎕SE
TypeLibs
Loaded Libraries
Microsoft Excel 9.0 Object Library
Microsoft Office 9.0 Object Librar
Microsoft Visual Basic for Applica
OLE Automation
Registered Libraries

Name	Loaded
OLE Automation	Yes
Unknown Types	Yes
Microsoft Office 9.0 Object Library	Yes
Microsoft Visual Basic for Applications ...	Yes
Microsoft Excel 9.0 Object Library	Yes

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

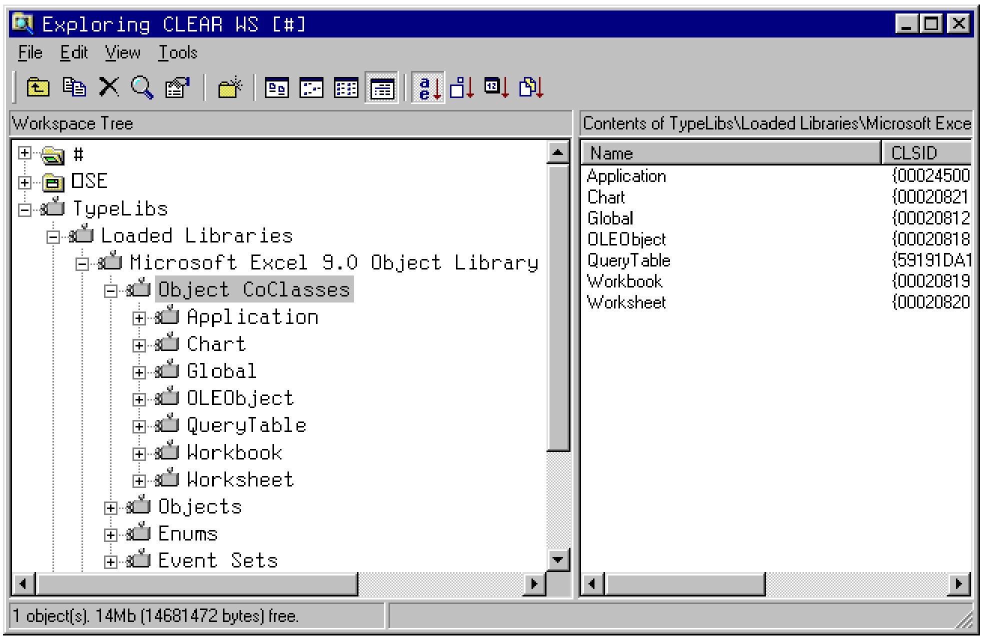
If you select a loaded Type Library, summary information is displayed in the list view pane.

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*.

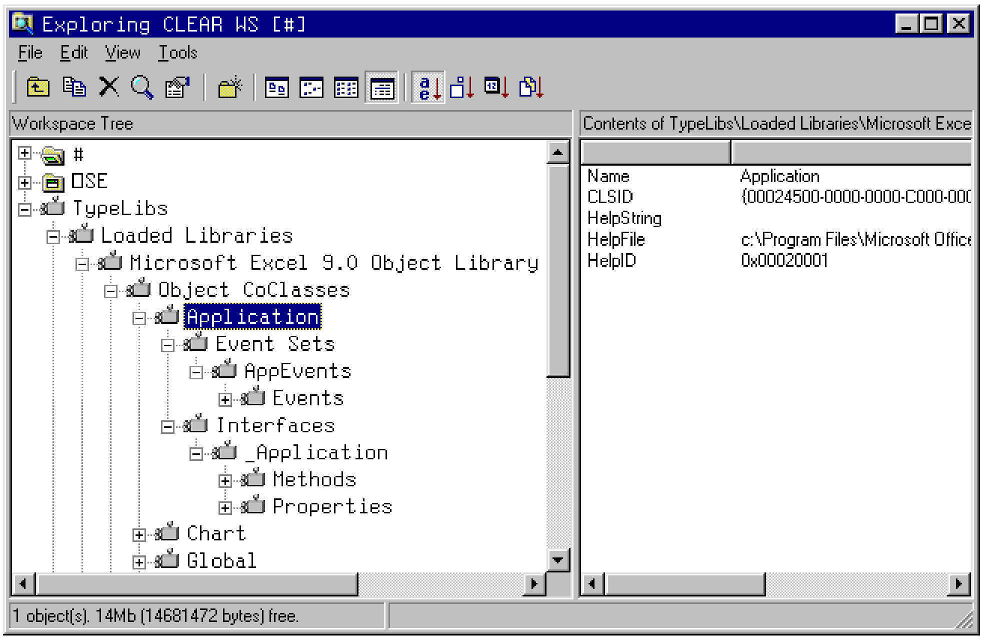
For example, the Microsoft Excel 9.0 Object Library contains seven CoClasses named *Application*, *Chart*, *Global* etc as shown below.



Name	CLSID
Application	{00024500
Chart	{00020821
Global	{00020812
OLEObject	{00020818
QueryTable	{59191DA1
Workbook	{00020819
Worksheet	{00020820

1 object(s). 14Mb (14681472 bytes) free.

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



Name	Application
CLSID	{00024500-0000-0000-C000-00
HelpString	
HelpFile	c:\Program Files\Microsoft Offic
HelpID	0x00020001

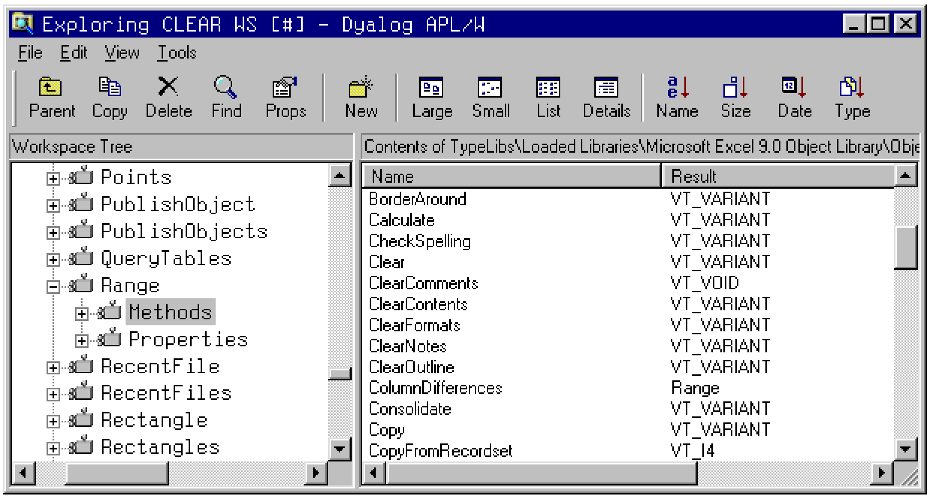
1 object(s). 14Mb (14681472 bytes) free.

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.

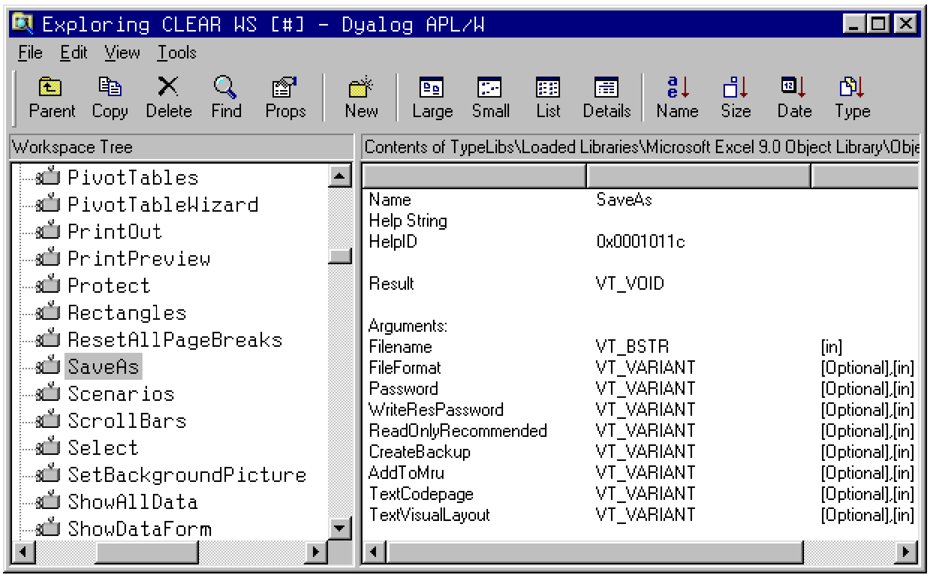
Each object folder contains two sub-folders named Methods and Properties. Selecting one of these causes the list of Methods or Properties to be displayed in the list view pane. The picture below shows the Methods exposed by the Microsoft Excel 9.0 Range object.



Contents of TypeLibs\Loaded Libraries\Microsoft Excel 9.0 Object Library\Obje

Name	Result
BorderAround	VT_VARIANT
Calculate	VT_VARIANT
CheckSpelling	VT_VARIANT
Clear	VT_VARIANT
ClearComments	VT_VOID
ClearContents	VT_VARIANT
ClearFormats	VT_VARIANT
ClearNotes	VT_VARIANT
ClearOutline	VT_VARIANT
ColumnDifferences	Range
Consolidate	VT_VARIANT
Copy	VT_VARIANT
CopyFromRecordset	VT_I4

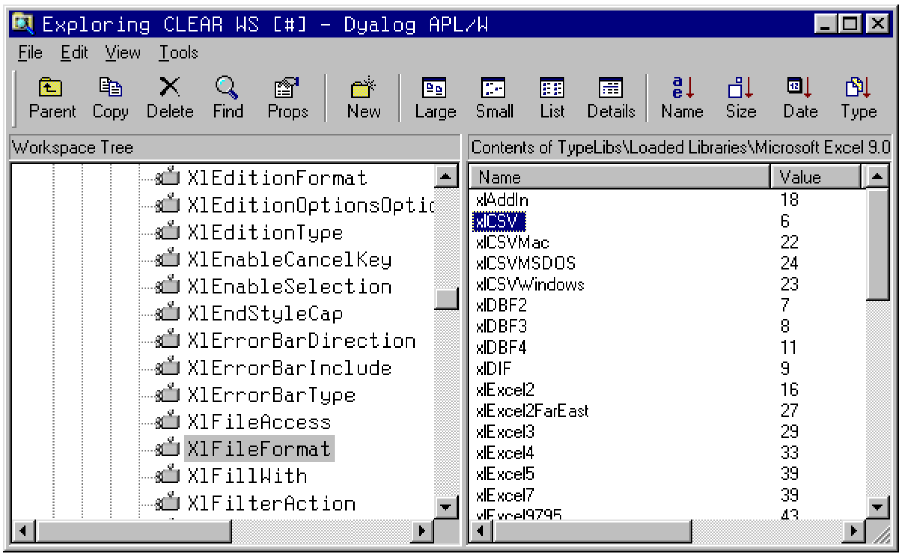
If you open the Methods or Properties subfolder, you can display more detailed information about individual Methods and Properties. For example, the following picture shows information about the SaveAs method exposed by the Microsoft Excel 9.0 Worksheet object.



Name	SaveAs	
Help String		
HelpID	0x0001011c	
Result	VT_VOID	
Arguments:		
Filename	VT_BSTR	[in]
FileFormat	VT_VARIANT	[Optional],[in]
Password	VT_VARIANT	[Optional],[in]
WriteResPassword	VT_VARIANT	[Optional],[in]
ReadOnlyRecommended	VT_VARIANT	[Optional],[in]
CreateBackup	VT_VARIANT	[Optional],[in]
AddToMru	VT_VARIANT	[Optional],[in]
TextCodepage	VT_VARIANT	[Optional],[in]
TextVisualLayout	VT_VARIANT	[Optional],[in]

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.



XlEditionFormat
XlEditionOptionsOptio
XlEditionType
XlEnableCancelKey
XlEnableSelection
XlEndStyleCap
XlErrorBarDirection
XlErrorBarInclude
XlErrorBarType
XlFileAccess
XlFileFormat
XlFillWith
XlFilterAction

Name	Value
xlAddIn	18
xlCSV	6
xlCSVMac	22
xlCSVMSDOS	24
xlCSVWindows	23
xlDBF2	7
xlDBF3	8
xlDBF4	11
xlDIF	9
xlExcel2	16
xlExcel2FarEast	27
xlExcel3	29
xlExcel4	33
xlExcel5	39
xlExcel7	39
xlExcel9795	43

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
```

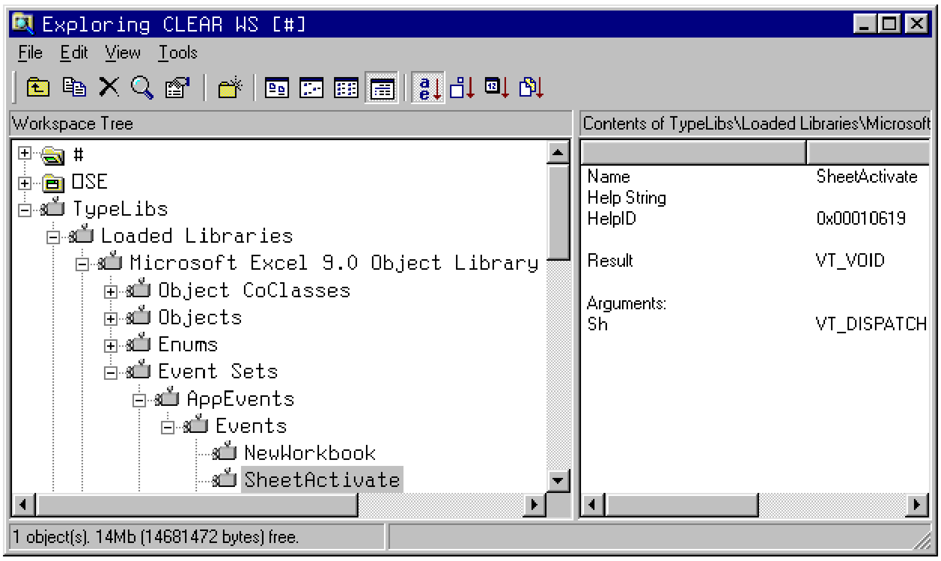
or

```
      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.



```
⊞ #
⊞ ⎕SE
⊟ TypeLibs
  ⊟ Loaded Libraries
    ⊟ Microsoft Excel 9.0 Object Library
      ⊞ Object CoClasses
      ⊞ Objects
      ⊞ Enums
      ⊟ Event Sets
        ⊟ AppEvents
          ⊟ Events
              NewWorkbook
              SheetActivate
```

Name	SheetActivate
Help String	
HelpID	0x00010619
Result	VT_VOID
Arguments:	
Sh	VT_DISPATCH

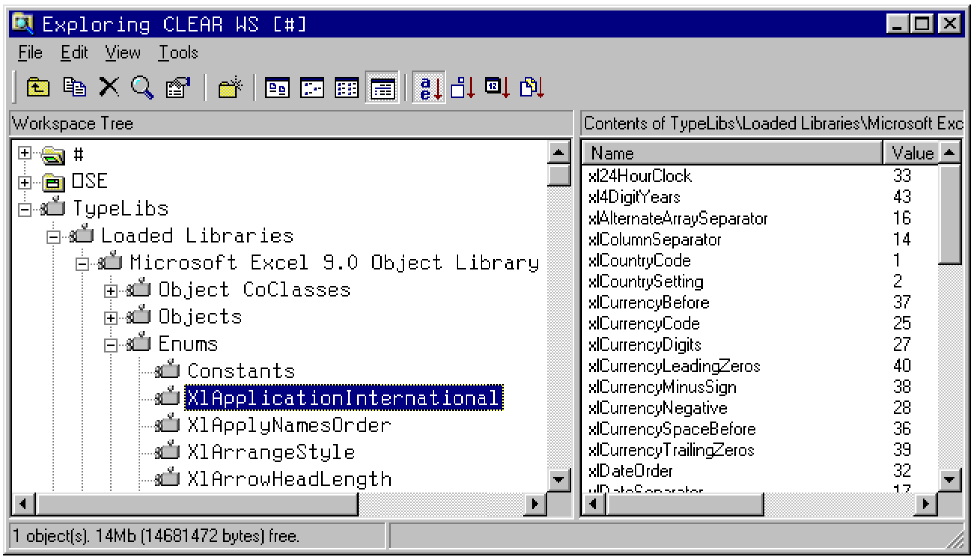
1 object(s). 14Mb (14681472 bytes) free.

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.

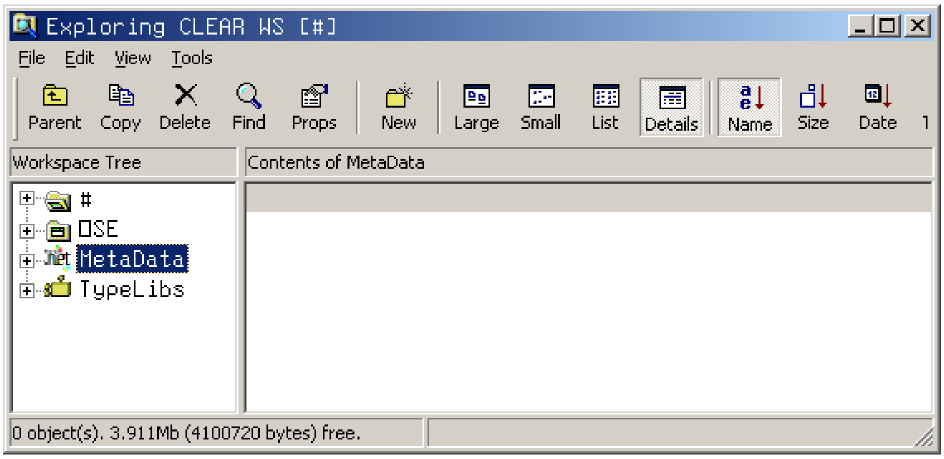


Name	Value
xl24HourClock	33
xl4DigitYears	43
xlAlternateArraySeparator	16
xlColumnSeparator	14
xlCountryCode	1
xlCountrySetting	2
xlCurrencyBefore	37
xlCurrencyCode	25
xlCurrencyDigits	27
xlCurrencyLeadingZeros	40
xlCurrencyMinusSign	38
xlCurrencyNegative	28
xlCurrencySpaceBefore	36
xlCurrencyTrailingZeros	39
xlDateOrder	32

1 object(s). 14Mb (14681472 bytes) free.

Browsing .Net Classes

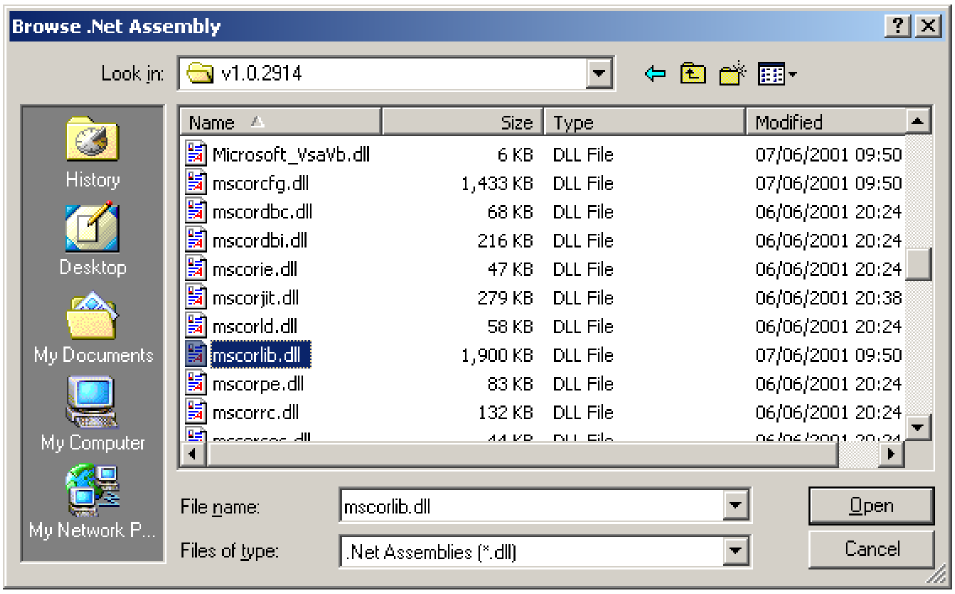
If the Microsoft .Net Framework is installed, you may browse the .Net Metadata using the Explorer. To gain information about one or more Net Classes, open the Workspace Explorer, right click the *Metadata* folder, and choose *Load*.



0 object(s). 3.911Mb (4100720 bytes) free.

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:\WINNT\Microsoft.NET\Framework\V1.0.3705`.



Look in: v1.0.2914

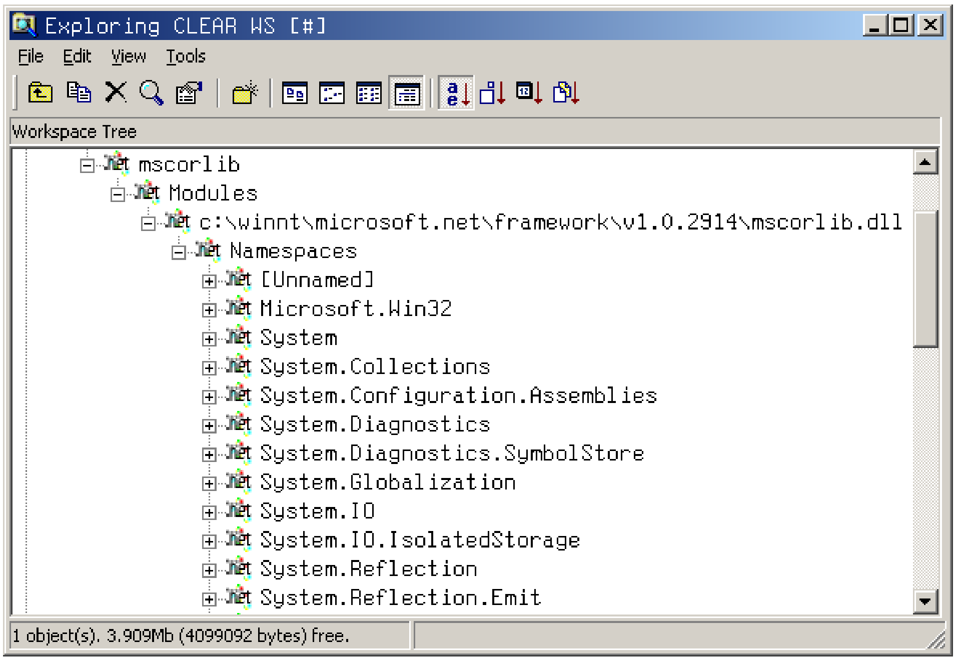
Name	Size	Type	Modified
Microsoft_VsaVb.dll	6 KB	DLL File	07/06/2001 09:50
mscorcfg.dll	1,433 KB	DLL File	07/06/2001 09:50
mscordbc.dll	68 KB	DLL File	06/06/2001 20:24
mscordbi.dll	216 KB	DLL File	06/06/2001 20:24
mscorie.dll	47 KB	DLL File	06/06/2001 20:24
mscorjit.dll	279 KB	DLL File	06/06/2001 20:38
mscorld.dll	58 KB	DLL File	06/06/2001 20:24
mscorlib.dll	1,900 KB	DLL File	07/06/2001 09:50
mscorpe.dll	83 KB	DLL File	06/06/2001 20:24
mscorrc.dll	132 KB	DLL File	06/06/2001 20:24

File name: mscorlib.dll

Files of type: .Net Assemblies (*.dll)

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.

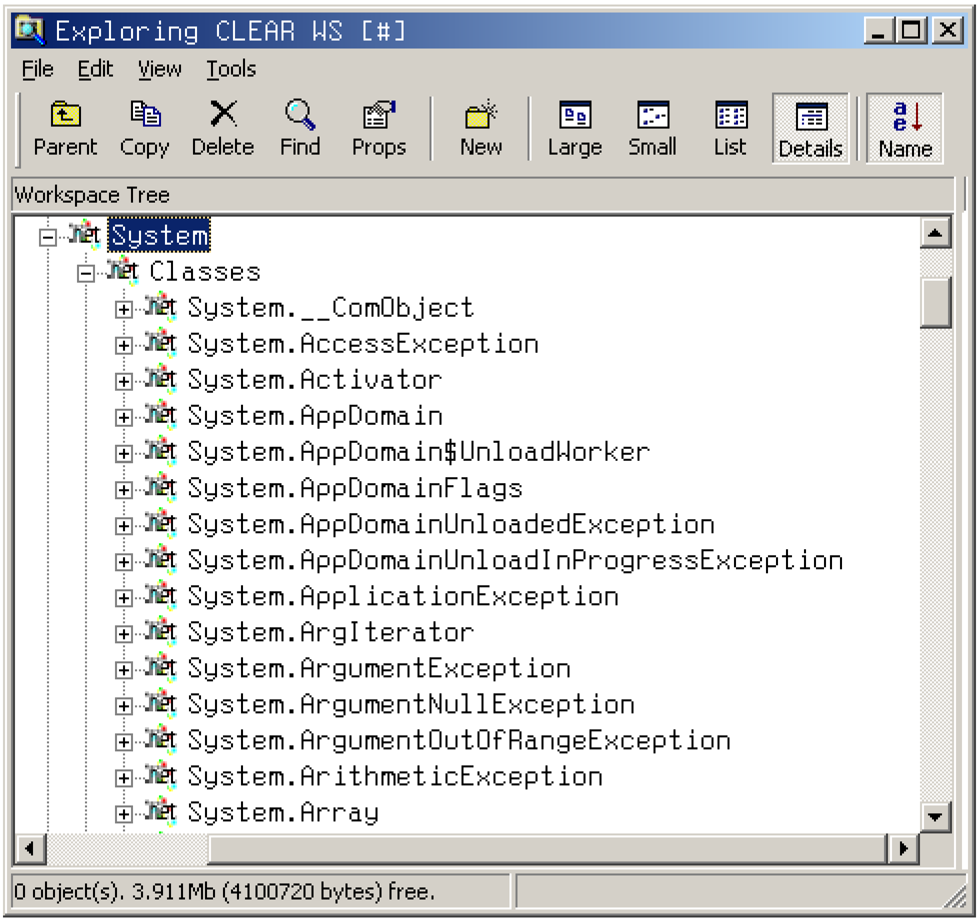
The result of opening this Assembly is illustrated in the following screen shot. The somewhat complex tree structure that is shown in the Workspace Explorer merely reflects the structure of the Metadata itself.



```
mscorlib
  Modules
    c:\winnt\microsoft.net\framework\v1.0.2914\mscorlib.dll
      Namespaces
        [Unnamed]
        Microsoft.Win32
        System
        System.Collections
        System.Configuration.Assemblies
        System.Diagnostics
        System.Diagnostics.SymbolStore
        System.Globalization
        System.IO
        System.IO.IsolatedStorage
        System.Reflection
        System.Reflection.Emit
```

1 object(s). 3.909Mb (4099092 bytes) free.

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

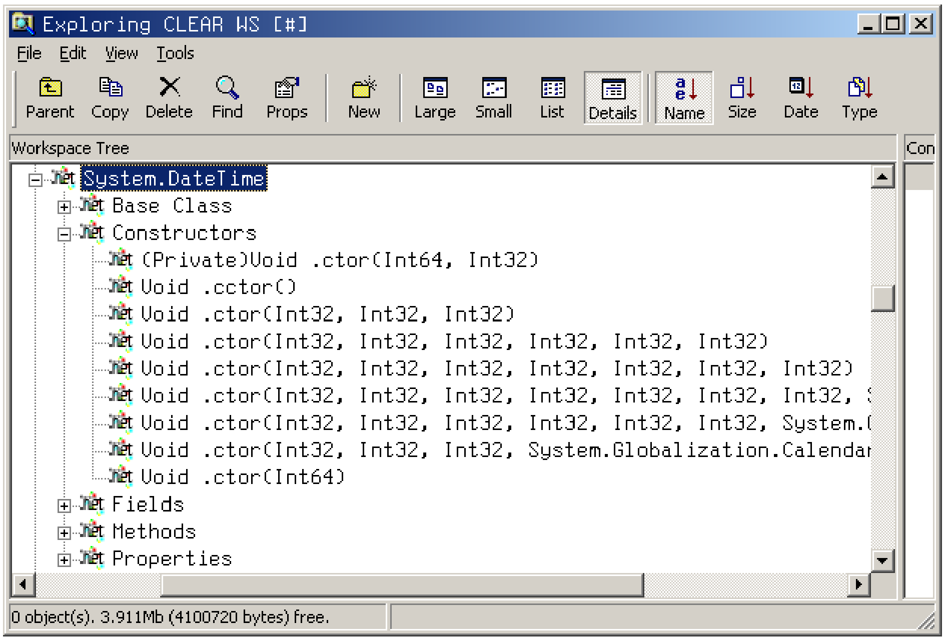


```
System
  Classes
    System.__ComObject
    System.AccessException
    System.Activator
    System.AppDomain
    System.AppDomain$UnloadWorker
    System.AppDomainFlags
    System.AppDomainUnloadedException
    System.AppDomainUnloadInProgressException
    System.ApplicationException
    System.ArgIterator
    System.ArgumentException
    System.ArgumentNullException
    System.ArgumentOutOfRangeException
    System.ArithmeticException
    System.Array
```

0 object(s). 3.911Mb (4100720 bytes) free.

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`.



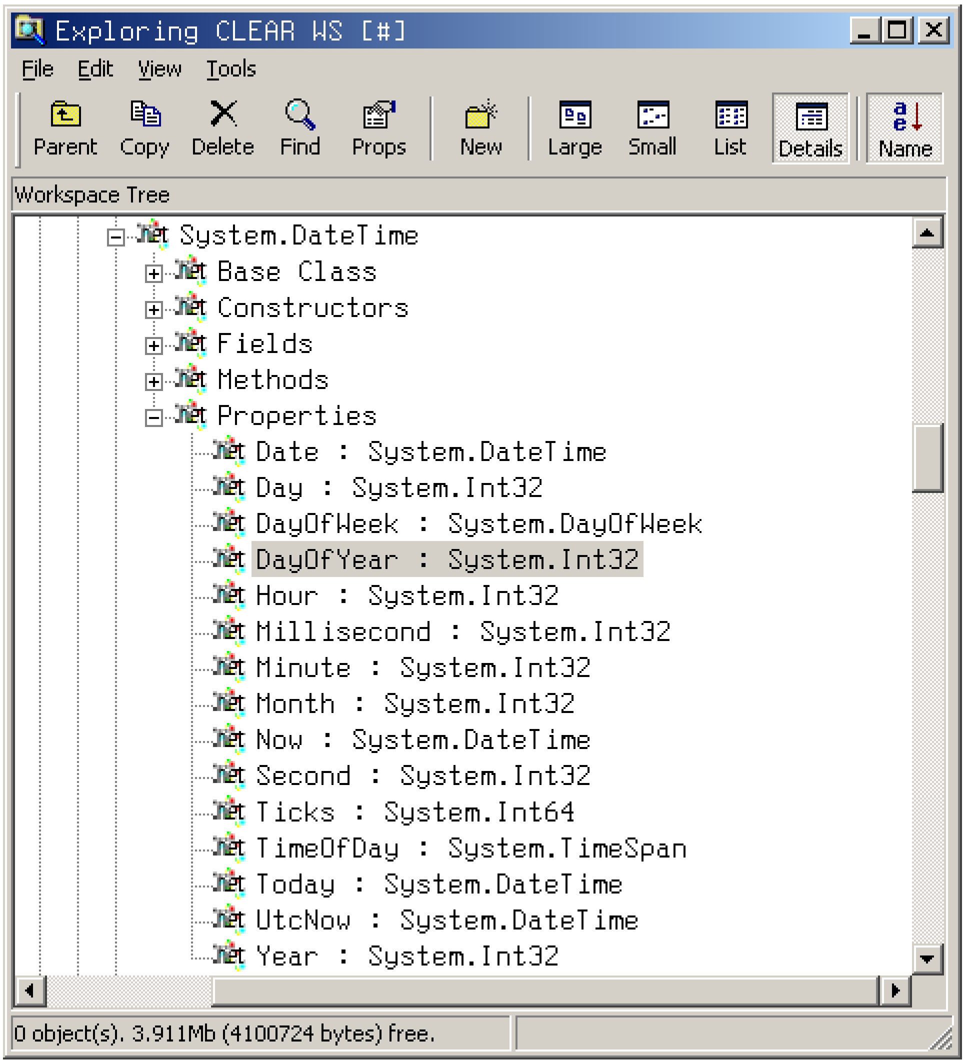
```
System.DateTime
  Base Class
  Constructors
    (Private)Void .ctor(Int64, Int32)
    Void .cctor()
    Void .ctor(Int32, Int32, Int32)
    Void .ctor(Int32, Int32, Int32, Int32, Int32, Int32)
    Void .ctor(Int32, Int32, Int32, Int32, Int32, Int32, Int32)
    Void .ctor(Int32, Int32, Int32, Int32, Int32, Int32, Int32, S
    Void .ctor(Int32, Int32, Int32, Int32, Int32, Int32, System.(
    Void .ctor(Int32, Int32, Int32, System.Globalization.Calendar
    Void .ctor(Int64)
  Fields
  Methods
  Properties
```

0 object(s). 3.911Mb (4100720 bytes) free.

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←DateTime.New 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`.



```
System.DateTime
  Base Class
  Constructors
  Fields
  Methods
  Properties
    Date : System.DateTime
    Day : System.Int32
    DayOfWeek : System.DayOfWeek
    DayOfYear : System.Int32
    Hour : System.Int32
    Millisecond : System.Int32
    Minute : System.Int32
    Month : System.Int32
    Now : System.DateTime
    Second : System.Int32
    Ticks : System.Int64
    TimeOfDay : System.TimeSpan
    Today : System.DateTime
    UtcNow : System.DateTime
    Year : System.Int32
```

0 object(s). 3.911Mb (4100724 bytes) free.

You can query a property by direct reference:

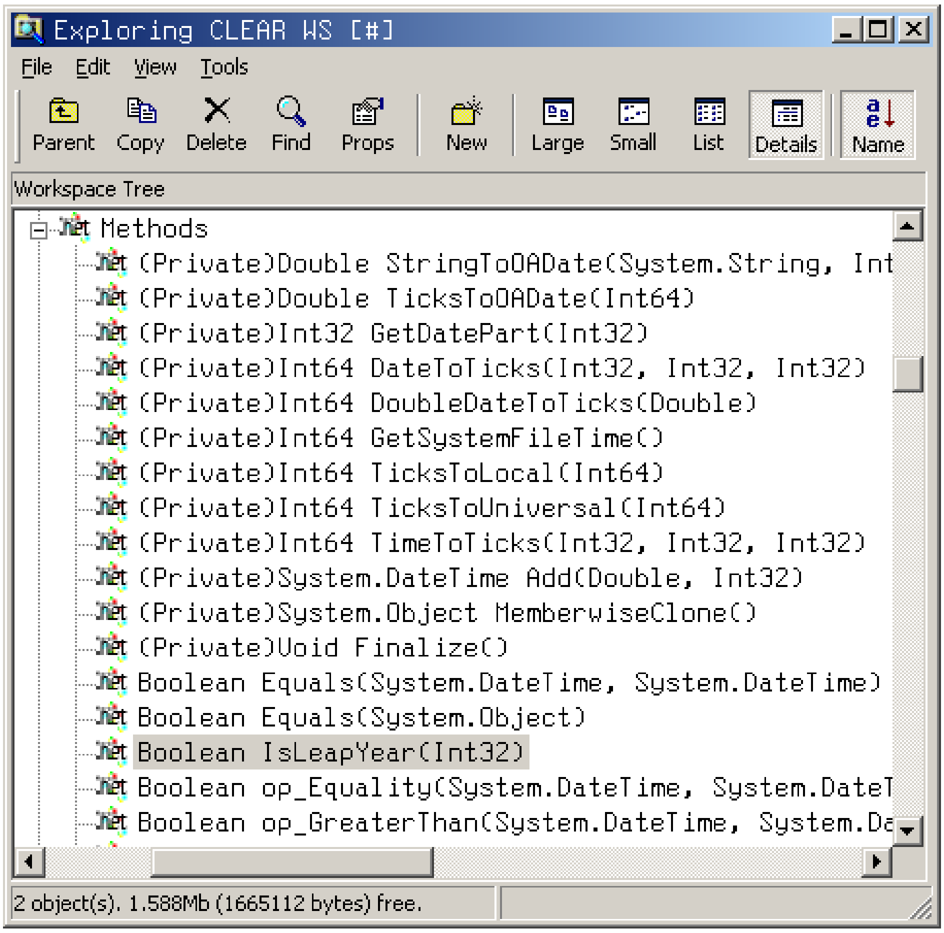
```
      mydt.DayOfWeek
1
```

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
```



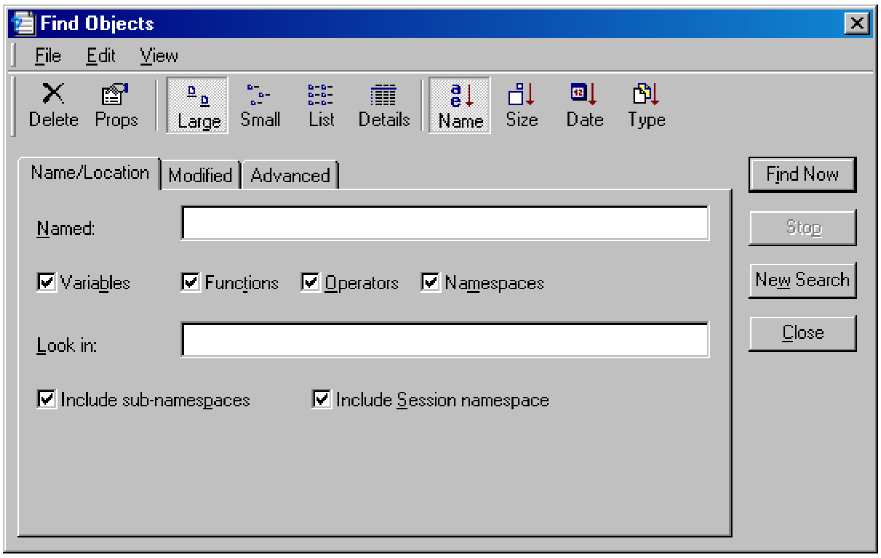
Methods
(Private)Double StringToOADate(System.String, Int
(Private)Double TicksToOADate(Int64)
(Private)Int32 GetDatePart(Int32)
(Private)Int64 DateToTicks(Int32, Int32, Int32)
(Private)Int64 DoubleDateToTicks(Double)
(Private)Int64 GetSystemFileTime()
(Private)Int64 TicksToLocal(Int64)
(Private)Int64 TicksToUniversal(Int64)
(Private)Int64 TimeToTicks(Int32, Int32, Int32)
(Private)System.DateTime Add(Double, Int32)
(Private)System.Object MemberwiseClone()
(Private)Void Finalize()
Boolean Equals(System.DateTime, System.DateTime)
Boolean Equals(System.Object)
Boolean IsLeapYear(Int32)
Boolean op_Equality(System.DateTime, System.DateT
Boolean op_GreaterThan(System.DateTime, System.Da

2 object(s). 1.588Mb (1665112 bytes) free.

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.



Named:

☑ Variables ☑ Functions ☑ Operators ☑ Namespaces

Look in:

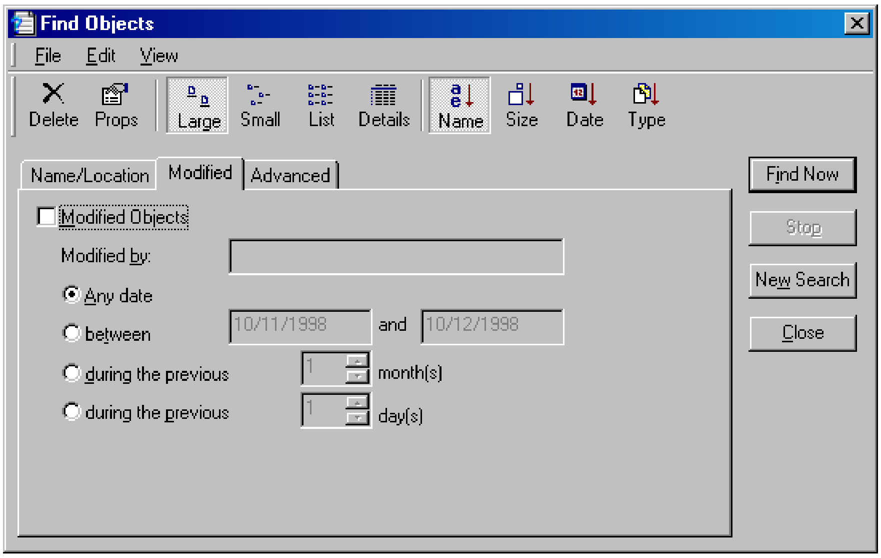
☑ Include sub-namespaces ☑ Include Session namespace

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



☐ Modified Objects

Modified by:

● Any date

○ between 10/11/1998 and 10/12/1998

○ during the previous 1 month(s)

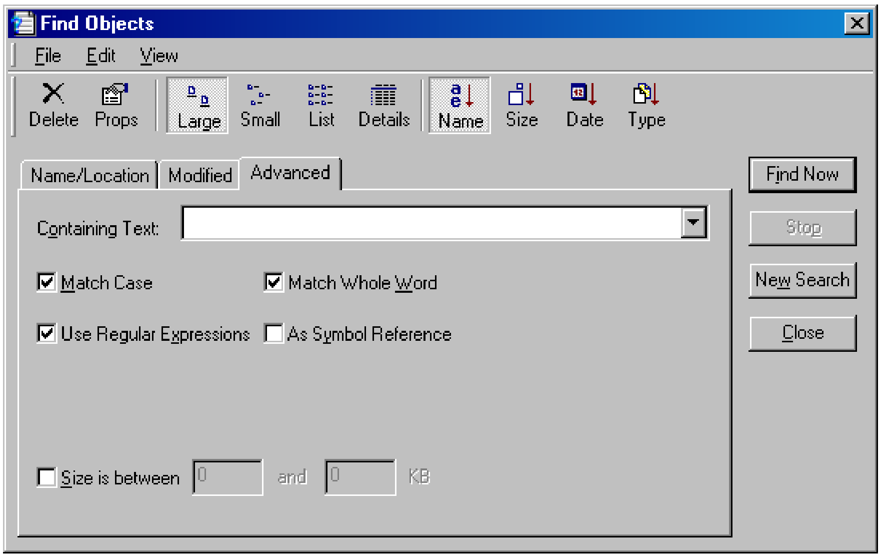
○ during the previous 1 day(s)

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.



Containing Text:

☑ Match Case ☑ Match Whole Word

☑ Use Regular Expressions ☐ As Symbol Reference

☐ Size is between 0 and 0 KB

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.

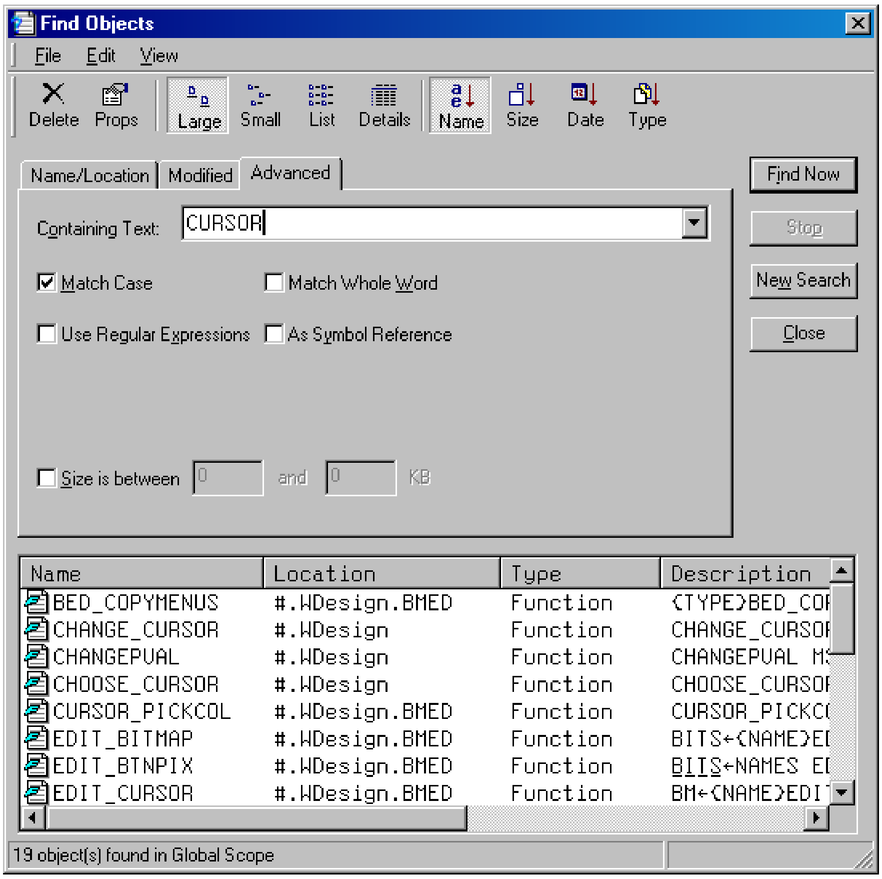
Expand Wildcards specifies whether or not wildcards are applicable. If you enter `FOO*` into the field labelled *Containing Text* and check this box, the system will find objects that contain any text string starting with the 3 characters `FOO`. If this box is not checked, the system will find objects 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`.



Containing Text: CURSOR

☑ Match Case ☐ Match Whole Word

☐ Use Regular Expressions ☐ As Symbol Reference

☐ Size is between 0 and 0 KB

Name	Location	Type	Description
BED_COPYMENUS	#.WDesign.BMED	Function	{TYPE}BED_CO
CHANGE_CURSOR	#.WDesign	Function	CHANGE_CURSO
CHANGEPUAL	#.WDesign	Function	CHANGEPUAL M
CHOOSE_CURSOR	#.WDesign	Function	CHOOSE_CURSO
CURSOR_PICKCOL	#.WDesign.BMED	Function	CURSOR_PICKC
EDIT_BITMAP	#.WDesign.BMED	Function	BITS←{NAME}E
EDIT_BTNPIX	#.WDesign.BMED	Function	BITS←NAMES E
EDIT_CURSOR	#.WDesign.BMED	Function	BM←{NAME}EDI

19 object(s) found in Global Scope

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 copies or moved by dragging from the ListView in the Search tool to the TreeView in the Explorer.

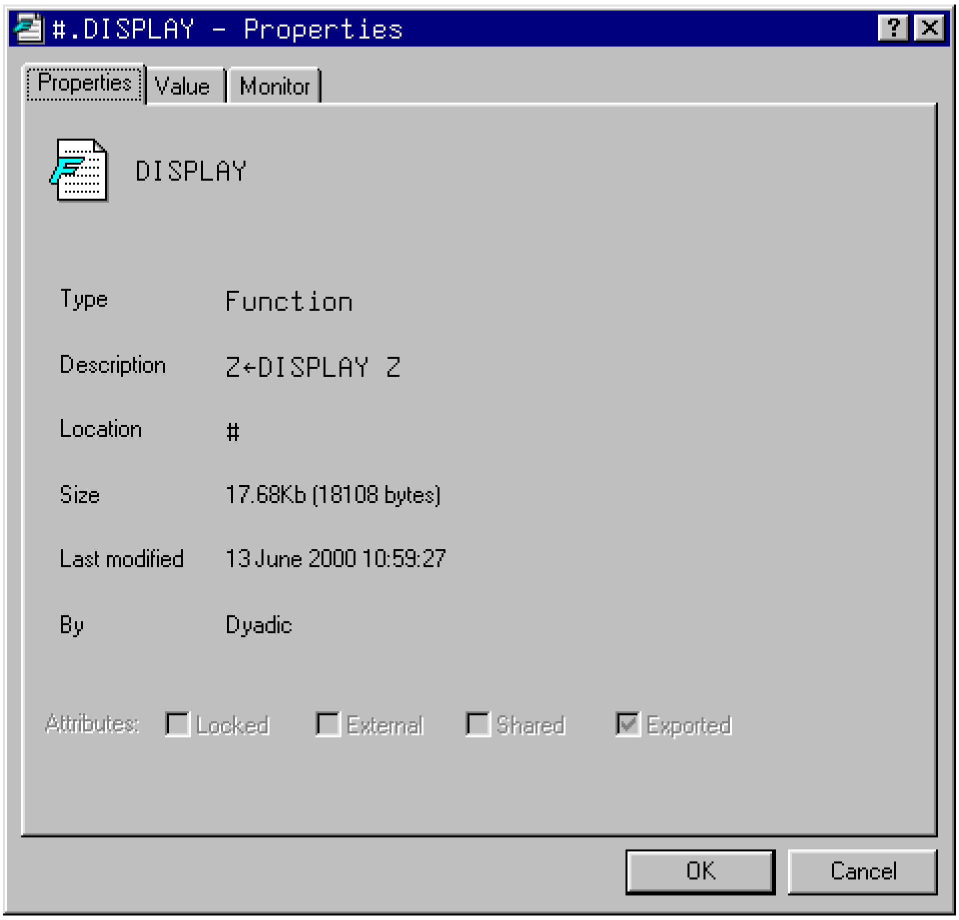
If you wish to specify a completely new set of criteria, press the *New Search* button. This will reset all of the various controls on the 3 pages of the dialog box to their default values.

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.



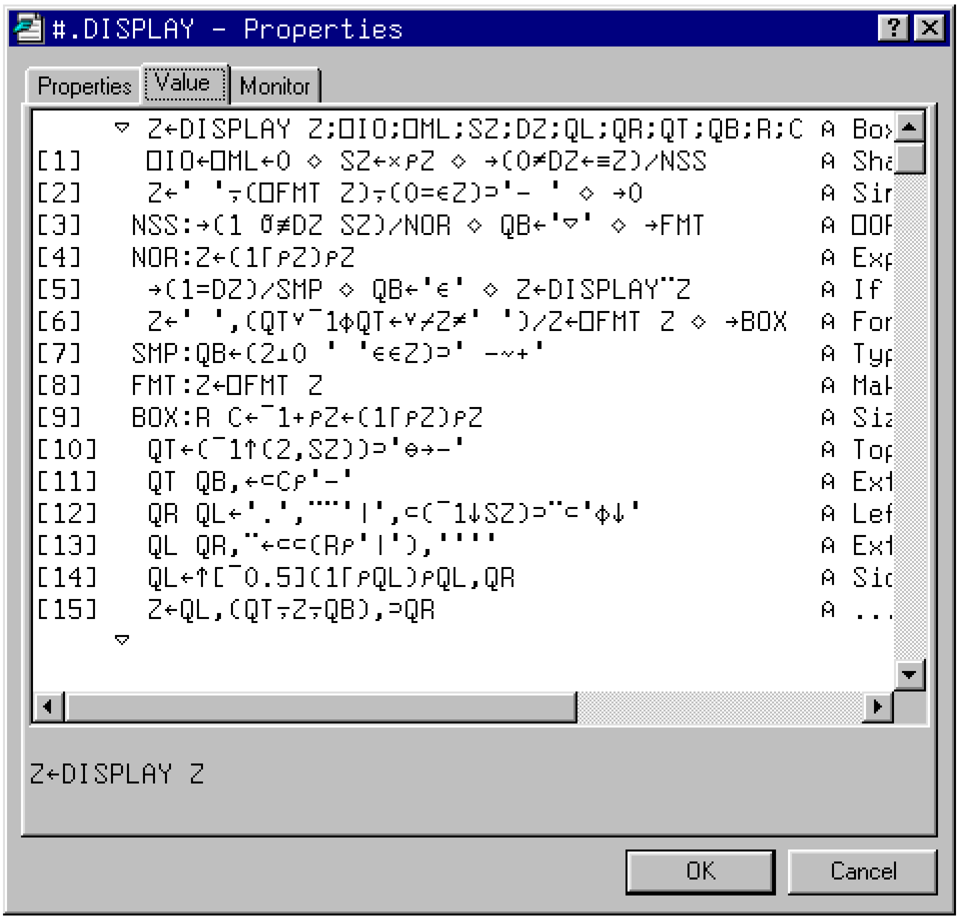
DISPLAY

Type	Function
Description	Z←DISPLAY Z
Location	#
Size	17.68Kb (18108 bytes)
Last modified	13 June 2000 10:59:27
By	Dyadic

Attributes: ☐ Locked ☐ External ☐ Shared ☑ Exported

Value Tab

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

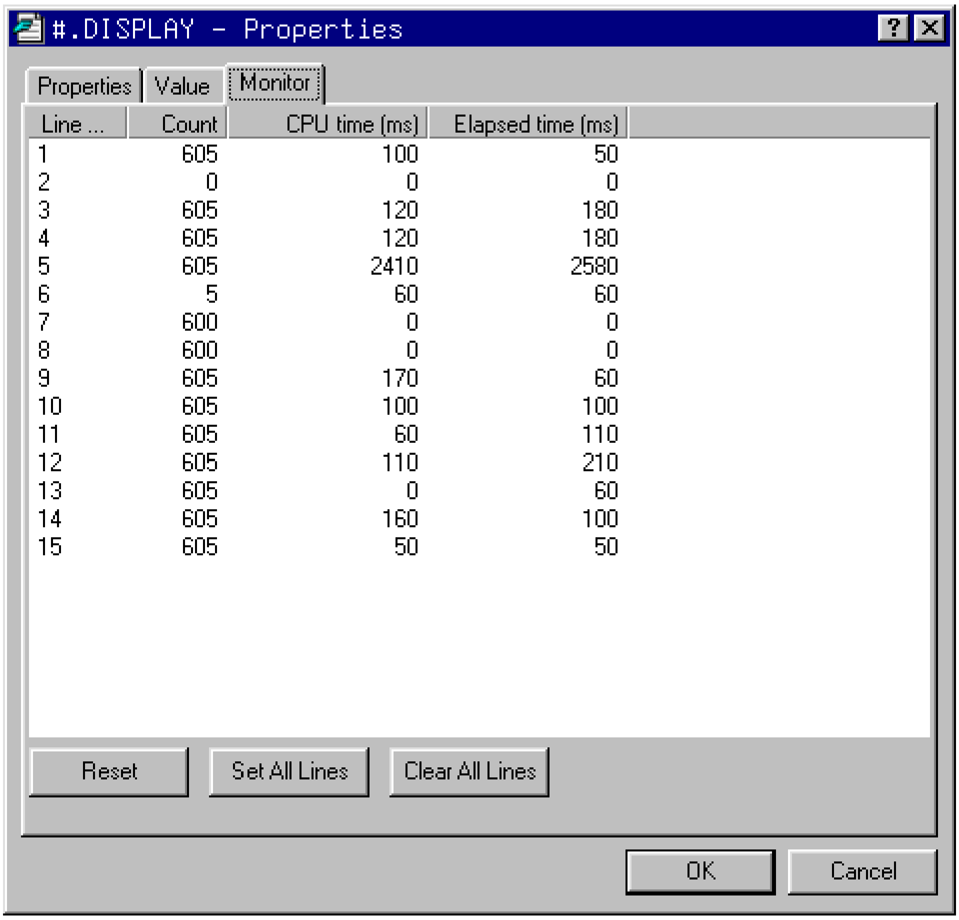


```
      ∇ Z←DISPLAY Z;⎕IO;⎕ML;SZ;DZ;QL;QR;QT;QB;R;C  ⍝ Bo
[1]    ⎕IO←⎕ML←0 ⋄ SZ←×⍴Z ⋄ →(0≠DZ←≡Z)/NSS       ⍝ Sh
[2]    Z←' '⍪(⎕FMT Z)⍪(0=∊Z)⊃'- ' ⋄ →0             ⍝ Si
[3]   NSS:→(1 ⍬≢DZ SZ)/NOR ⋄ QB←'∇' ⋄ →FMT         ⍝ ⎕O
[4]   NOR:Z←(1⌈⍴Z)⍴Z                               ⍝ Ex
[5]    →(1=DZ)/SMP ⋄ QB←'∊' ⋄ Z←DISPLAY¨Z          ⍝ If
[6]    Z←' ',(QT∨¯1⌽QT←∨⌿Z≠' ')/Z←⎕FMT Z ⋄ →BOX    ⍝ Fo
[7]   SMP:QB←(2⊥0 ' '∊∊Z)⊃' -~+'                   ⍝ Ty
[8]   FMT:Z←⎕FMT Z                                 ⍝ Ma
[9]   BOX:R C←¯1+⍴Z←(1⌈⍴Z)⍴Z                       ⍝ Si
[10]   QT←(¯1↑(2,SZ))⊃'⊖→-'                        ⍝ To
[11]   QT QB,←⊂C⍴'-'                               ⍝ Ex
[12]   QR QL←'.',''''|',⊂(¯1↓SZ)⊃¨⊂'⌽↓'            ⍝ Le
[13]   QL QR,¨←⊂⊂(R⍴'|'),''''                      ⍝ Ex
[14]   QL←↑[¯0.5](1⌈⍴QL)⍴QL,QR                     ⍝ Si
[15]   Z←QL,(QT⍪Z⍪QB),⊃QR                          ⍝ ...
      ∇
```

```
Z←DISPLAY Z
```

Monitor Tab

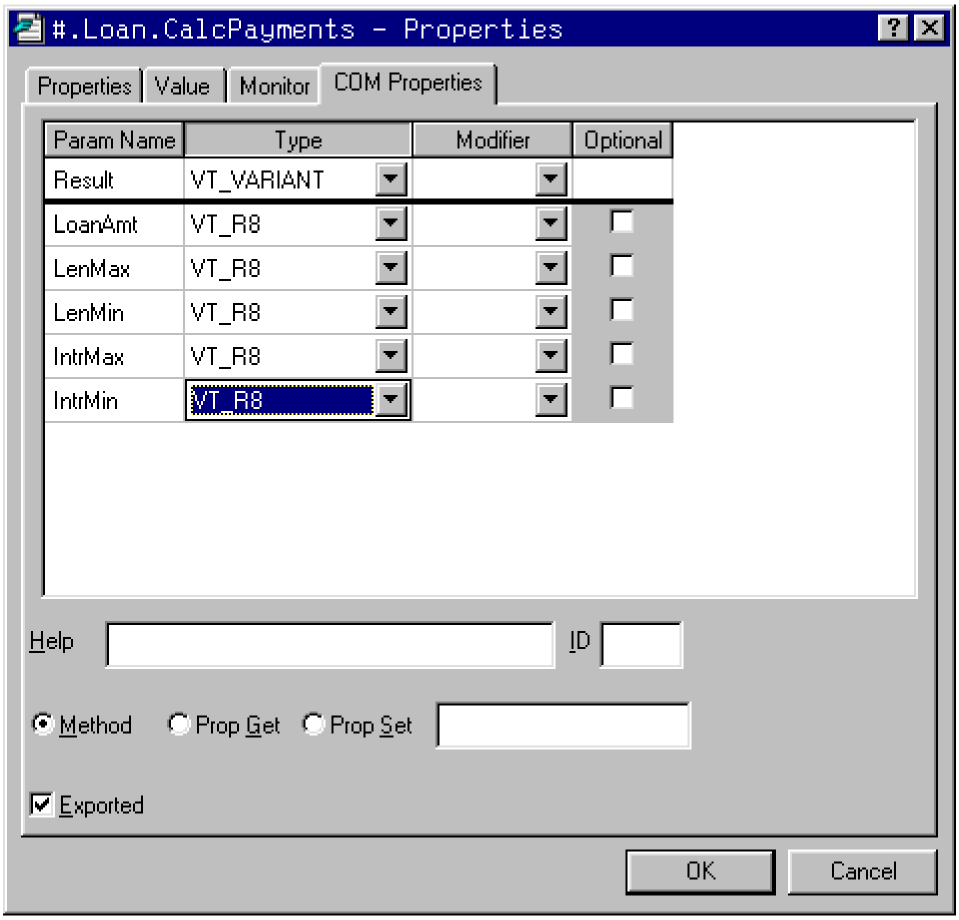
The Monitor tab applies only to a function and displays the result of ⎕MONITOR. The *Reset* button, resets ⎕MONITOR for the lines on which it is currently set. The *Set All Lines* button, sets ⎕MONITOR to monitor all the lines in the function. The Clear All Lines switches ⎕MONITOR off.



Line ...	Count	CPU time (ms)	Elapsed time (ms)
1	605	100	50
2	0	0	0
3	605	120	180
4	605	120	180
5	605	2410	2580
6	5	60	60
7	600	0	0
8	600	0	0
9	605	170	60
10	605	100	100
11	605	60	110
12	605	110	210
13	605	0	60
14	605	160	100
15	605	50	50

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.



Param Name	Type	Modifier	Optional
Result	VT_VARIANT		
LoanAmt	VT_R8		☐
LenMax	VT_R8		☐
LenMin	VT_R8		☐
IntrMax	VT_R8		☐
IntrMin	VT_R8		☐

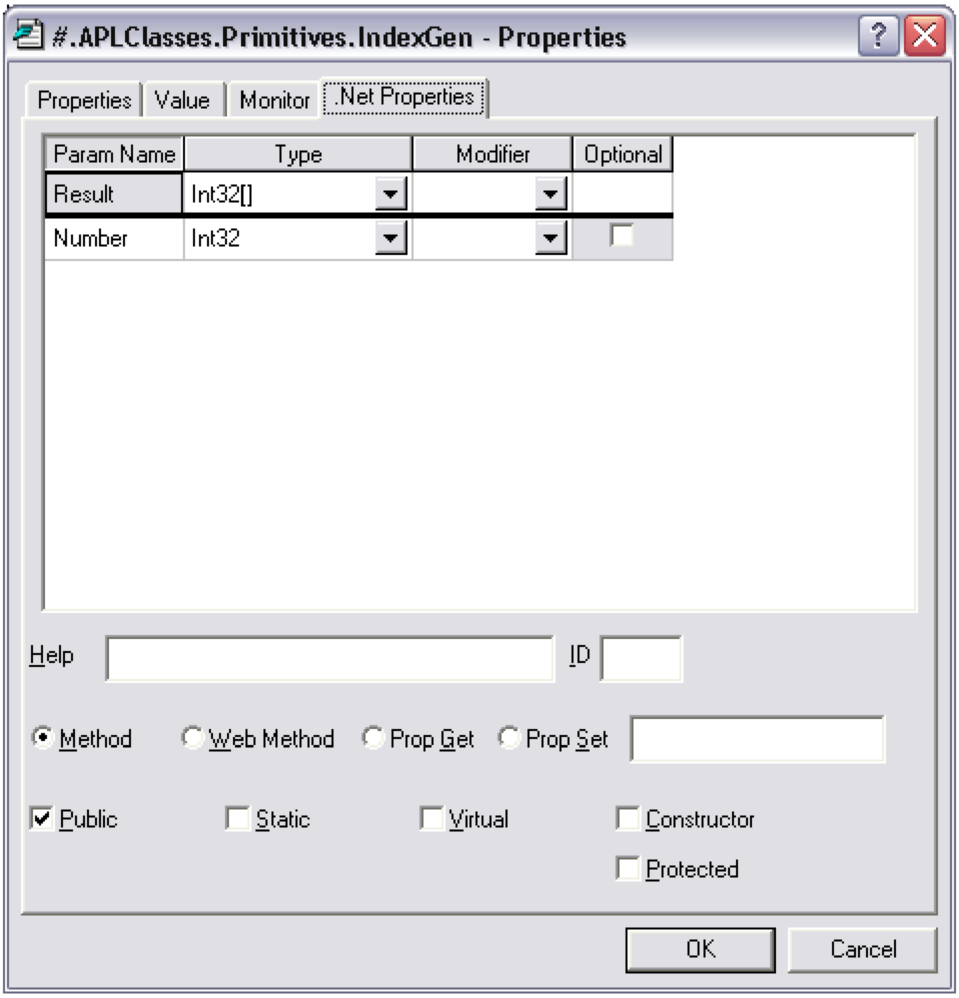
Help
ID

● Method ○ Prop Get ○ Prop Set

☑ Exported

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.



#.APLClasses.Primitives.IndexGen - Properties

Param Name	Type	Modifier	Optional
Result	Int32[]		
Number	Int32		☐

Help

ID

● Method ○ Web Method ○ Prop Get ○ Prop Set

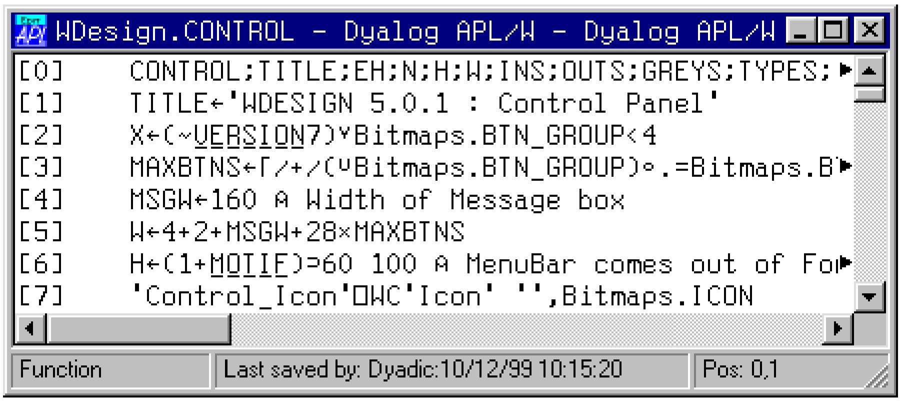
☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

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



```
[0]     CONTROL;TITLE;EH;N;H;W;INS;OUTS;GREYS;TYPES;
[1]     TITLE←'WDESIGN 5.0.1 : Control Panel'
[2]     X←(~VERSION7)∨Bitmaps.BTN_GROUP<4
[3]     MAXBTNS←⌈/+/(∪Bitmaps.BTN_GROUP)∘.=Bitmaps.B
[4]     MSGW←160 ⍝ Width of Message box
[5]     W←4+2+MSGW+28×MAXBTNS
[6]     H←(1+MOTIF)⊃60 100 ⍝ MenuBar comes out of Fo
[7]     'Control_Icon'⎕WC'Icon' '',Bitmaps.ICON
```

Function | Last saved by: Dyadic:10/12/99 10:15:20 | Pos: 0,1

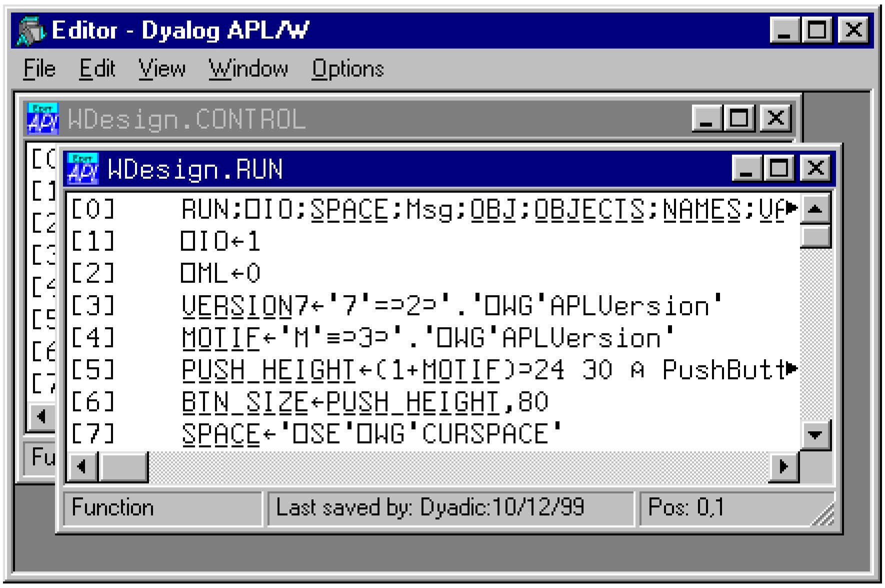
In addition, there is a general *point-and-edit* facility which works in edit and trace windows too. Simply position the input cursor over a name and double-click the left mouse button. Alternatively, you can press Shift+Enter or select Edit from the File menu. The name can appear in the Session, in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

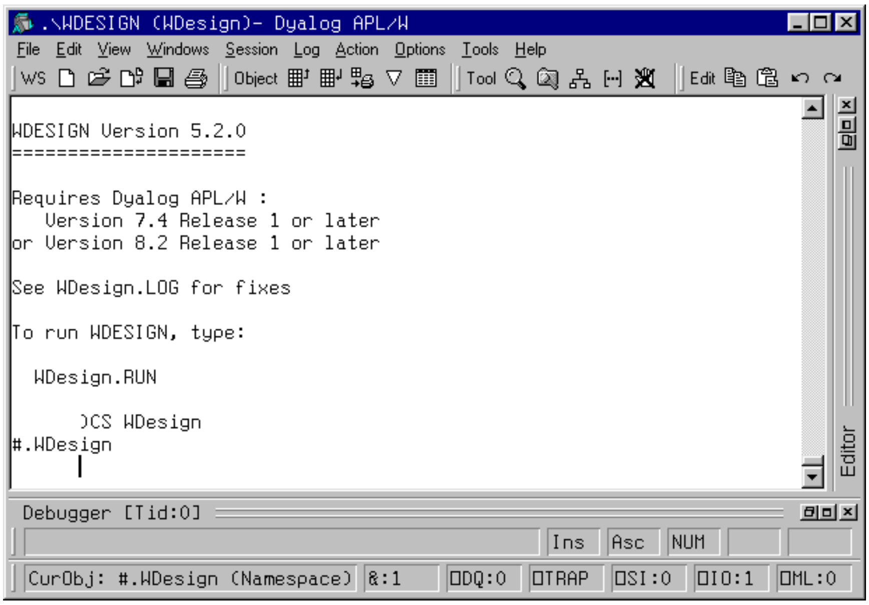


```
[0]     RUN;⎕IO;SPACE;Msg;OBJ;OBJECTS;NAMES;U
[1]     ⎕IO←1
[2]     ⎕ML←0
[3]     VERSION7←'7'=⊃2⊃'.'⎕WG'APLVersion'
[4]     MOTIF←'M'≡⊃3⊃'.'⎕WG'APLVersion'
[5]     PUSH_HEIGHT←(1+MOTIF)⊃24 30 ⍝ PushButt
[6]     BTN_SIZE←PUSH_HEIGHT,80
[7]     SPACE←'⎕SE'⎕WG'CURSPACE'
```

Function | Last saved by: Dyadic:10/12/99 | Pos: 0,1

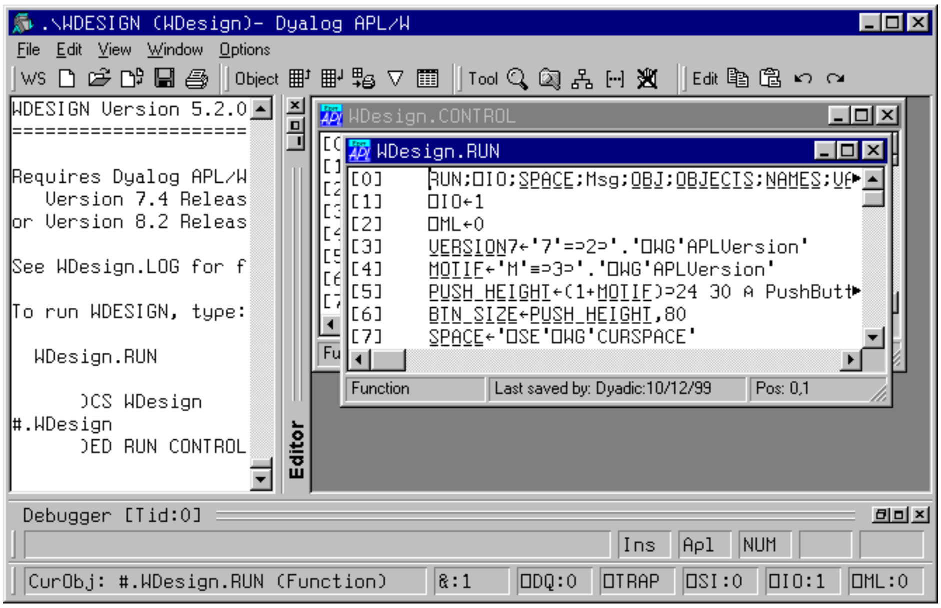
The initial size of an edit window is specified by the **edit_rows** and **edit_cols** parameters. The first edit window is positioned at 0 0. Subsequent ones are staggered according to the values of the **edit_offset_y** and **edit_offset_x** parameters.

By default, the Version 10 Session has the Editor docked along the right edge of the Session window as shown below.


.\WDESIGN (WDesign)- Dyalog APL/W
File Edit View Windows Session Log Action Options Tools Help
WS
Object
Tool
Edit
Editor
Debugger [Tid:0]
Ins
Asc
NUM
CurObj: #.WDesign (Namespace)
&:1
⎕DQ:0
⎕TRAP
⎕SI:0
⎕IO:1
⎕ML:0

```
WDESIGN Version 5.2.0
=====================

Requires Dyalog APL/W :
   Version 7.4 Release 1 or later
or Version 8.2 Release 1 or later

See WDesign.LOG for fixes

To run WDESIGN, type:

  WDesign.RUN

      )CS WDesign
#.WDesign
```

When you edit a function, the Editor window automatically springs into view as illustrated below.


.\WDESIGN (WDesign)- Dyalog APL/W
File Edit View Window Options
WS
Object
Tool
Edit
Editor
WDesign.CONTROL
WDesign.RUN
Function
Last saved by: Dyadic:10/12/99
Pos: 0,1
Debugger [Tid:0]
Ins
Apl
NUM
CurObj: #.WDesign.RUN (Function)
&:1
⎕DQ:0
⎕TRAP
⎕SI:0
⎕IO:1
⎕ML:0

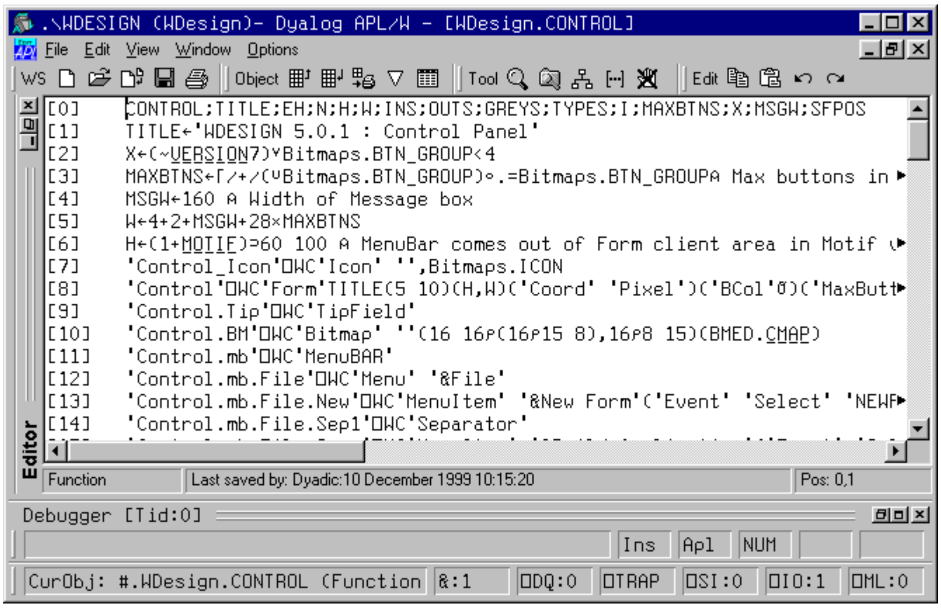
```
WDESIGN Version 5.2.0
=====================

Requires Dyalog APL/W
   Version 7.4 Releas
or Version 8.2 Releas

See WDesign.LOG for f

To run WDESIGN, type:

  WDesign.RUN

      )CS WDesign
#.WDesign
      )ED RUN CONTROL
```

```
RUN;⎕IO;SPACE;Msg;OBJ;OBJECTS;NAMES;VE
⎕IO←1
⎕ML←0
VERSION7←'7'=⊃2⊃'.'⎕WG'APLVersion'
MOTIF←'M'≡⊃3⊃'.'⎕WG'APLVersion'
PUSH_HEIGHT←(1+MOTIF)⊃24 30 ⍝ PushButt
BTN_SIZE←PUSH_HEIGHT,80
SPACE←'⎕SE'⎕WG'CURSPACE'
```

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 *CONTROL* edit window is itself maximised within the Editor too.



```
CONTROL;TITLE;EH;N;H;W;INS;OUTS;GREYS;TYPES;I;MAXBTNS;X;MSGW;SFPOS
TITLE←'WDESIGN 5.0.1 : Control Panel'
X←(~VERSION7)∨Bitmaps.BTN_GROUP<4
MAXBTNS←⌈/+/(∪Bitmaps.BTN_GROUP)∘.=Bitmaps.BTN_GROUP⍝ Max buttons in ▸
MSGW←160 ⍝ Width of Message box
W←4+2+MSGW+28×MAXBTNS
H←(1+MOTIF)⊃60 100 ⍝ MenuBar comes out of Form client area in Motif (▸
'Control_Icon'⎕WC'Icon' '',Bitmaps.ICON
'Control'⎕WC'Form'TITLE(5 10)(H,W)('Coord' 'Pixel')('BCol'⍬)('MaxButt▸
'Control.Tip'⎕WC'TipField'
'Control.BM'⎕WC'Bitmap' ''(16 16⍴(16⍴15 8),16⍴8 15)(BMED.CMAP)
'Control.mb'⎕WC'MenuBAR'
'Control.mb.File'⎕WC'Menu' '&File'
'Control.mb.File.New'⎕WC'MenuItem' '&New Form'('Event' 'Select' 'NEWF▸
'Control.mb.File.Sep1'⎕WC'Separator'
```

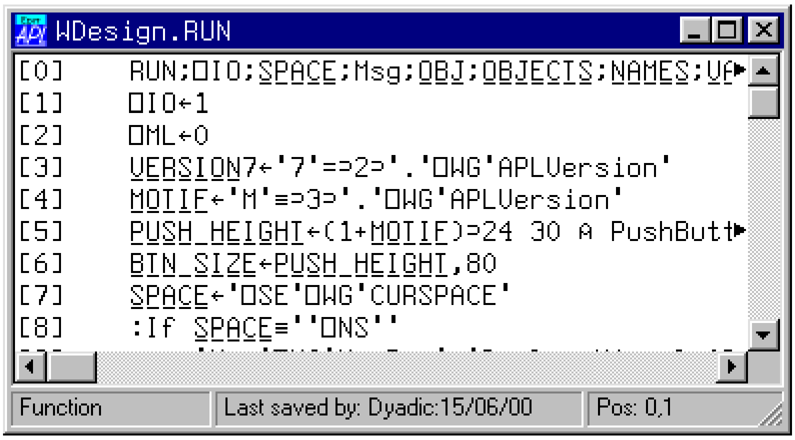
```
Function | Last saved by: Dyadic:10 December 1999 10:15:20 | Pos: 0,1
Debugger [Tid:0]
Ins | Apl | NUM
CurObj: #.WDesign.CONTROL (Function | &:1 | ⎕DQ:0 | ⎕TRAP | ⎕SI:0 | ⎕IO:1 | ⎕ML:0
```

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

Window Management (Classic Dyalog mode)

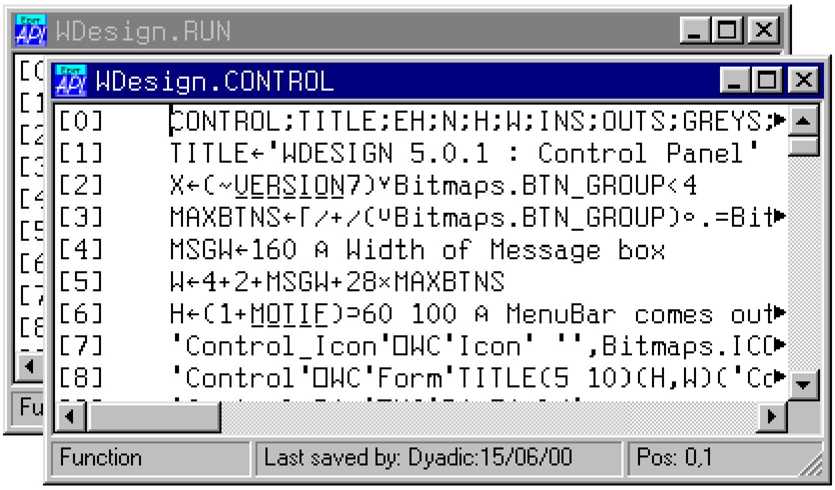
If *Classic Dyalog mode* is selected (*Options/Configure/Trace/Edit*) each Edit window is a top-level window created as a child of the Session window. This means that Edit windows always appear on top of the Session.

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.



```
[0]     RUN;⎕IO;SPACE;Msg;OBJ;OBJECTS;NAMES;VA
[1]     ⎕IO←1
[2]     ⎕ML←0
[3]     VERSION7←'7'=⊃2⊃'.'⎕WG'APLVersion'
[4]     MOTIF←'M'≡⊃3⊃'.'⎕WG'APLVersion'
[5]     PUSH_HEIGHT←(1+MOTIF)⊃24 30 ⍝ PushButt
[6]     BTN_SIZE←PUSH_HEIGHT,80
[7]     SPACE←'⎕SE'⎕WG'CURSPACE'
[8]     :If SPACE≡''⎕NS''
```

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



```
[0]     CONTROL;TITLE;EH;N;H;W;INS;OUTS;GREYS;
[1]     TITLE←'WDESIGN 5.0.1 : Control Panel'
[2]     X←(~VERSION7)∨Bitmaps.BTN_GROUP<4
[3]     MAXBTNS←⌈/+/(∪Bitmaps.BTN_GROUP)∘.=Bit
[4]     MSGW←160 ⍝ Width of Message box
[5]     W←4+2+MSGW+28×MAXBTNS
[6]     H←(1+MOTIF)⊃60 100 ⍝ MenuBar comes out
[7]     'Control_Icon'⎕WC'Icon' '',Bitmaps.ICO
[8]     'Control'⎕WC'Form'TITLE(5 10)(H,W)('Co
```

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.

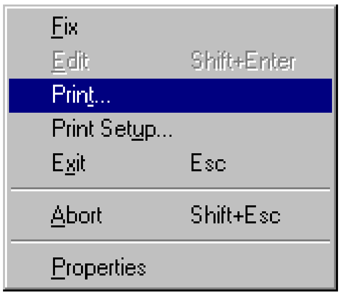
Closing an edit window

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.

The File Menu



The File Menu

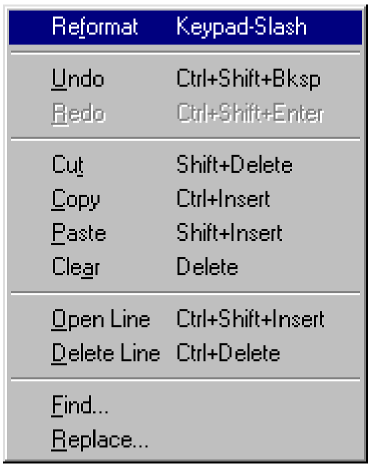
The File menu illustrated above provides the following options.

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	Inactive (see note).
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.

Note: for consistency, the edit window has the same File menu as the Trace Window, but the *Edit* option (which is applicable in the Tracer) is inactive.

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

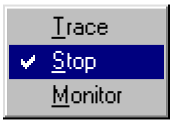
Reformat	Reformats the function body in the edit window, indenting control structures as appropriate.
Undo	Undoes the last change made to the object. Repeated use of this command sequentially undoes each change made since the edit window was opened.
Redo	Re-applies the previous undone change. Repeated 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.
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.
Find	Displays the *Find* dialog box.
Replace	Displays the *Replace* dialog box.

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 View Menu

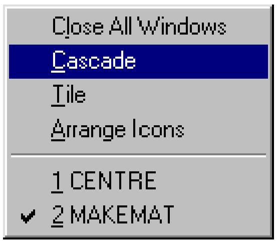
The View menu allows you to display and toggle `⎕TRACE`, `⎕STOP` and `⎕MONITOR` settings The View menu and the actions it provides are described below.



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

The Windows Menu

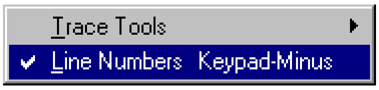
The Windows menu provides a means to control the display of the various edit windows. The Windows menu and the actions it provides are described below.



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.
Object name	Selects the edit window corresponding to the named object.

The Options Menu

The Options menu is used to toggle line numbers, and to control the display of the Trace Tools.



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.

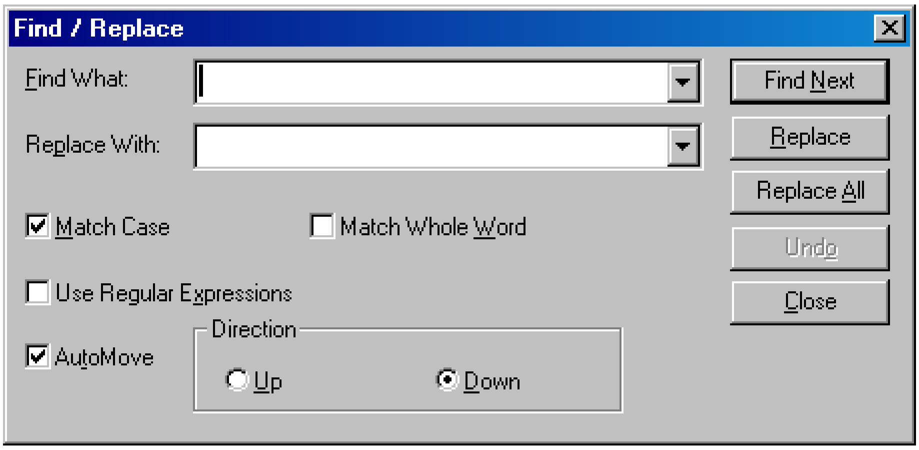
To split a line: select Insert mode; position the cursor at the place you want it split; press Return.

Toggling Localisation

The TL command (which by default is mapped to Ctrl+Alt+l) toggles the localisation of the name under the cursor. If the name is currently global, pressing Ctrl+Alt+l 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.

Find and Replace Dialogs

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

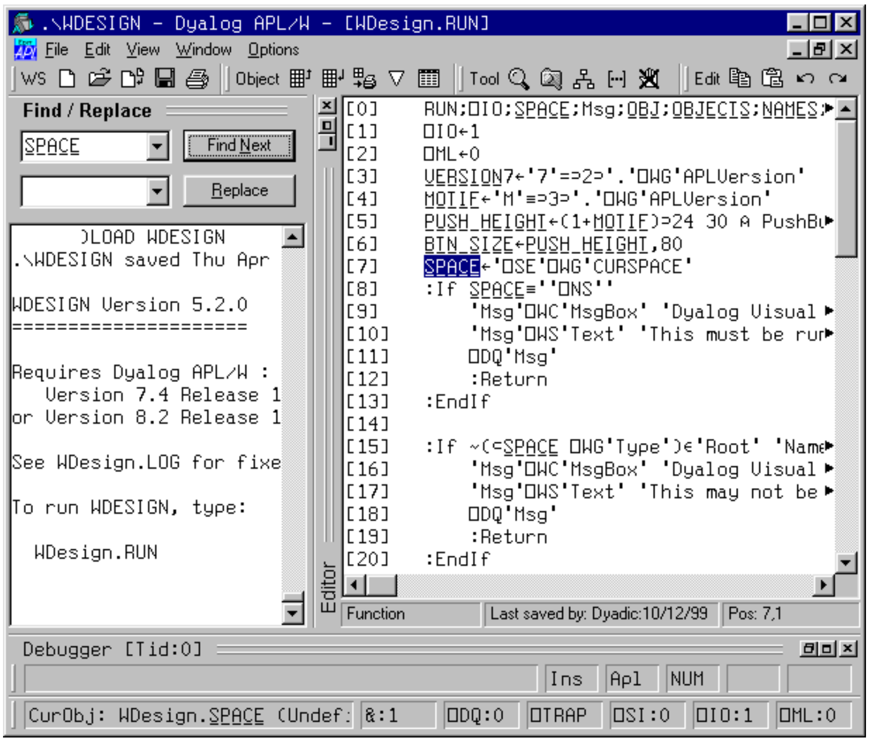


☑ Match Case
☐ Match Whole Word
☐ Use Regular Expressions
☑ AutoMove
○ Up
● Down

Find What	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 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.
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.

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.



Find / Replace: SPACE

```
      )LOAD WDESIGN
.\WDESIGN saved Thu Apr

WDESIGN Version 5.2.0
=====================

Requires Dyalog APL/W :
   Version 7.4 Release 1
or Version 8.2 Release 1

See WDesign.LOG for fixe

To run WDESIGN, type:

  WDesign.RUN
```

```
[0]     RUN;⎕IO;SPACE;Msg;OBJ;OBJECTS;NAMES;►
[1]     ⎕IO←1
[2]     ⎕ML←0
[3]     VERSION7←'7'=⊃2⊃'.'⎕WG'APLVersion'
[4]     MOTIF←'M'≡⊃3⊃'.'⎕WG'APLVersion'
[5]     PUSH_HEIGHT←(1+MOTIF)⊃24 30 ⍝ PushBu►
[6]     BTN_SIZE←PUSH_HEIGHT,80
[7]     SPACE←'⎕SE'⎕WG'CURSPACE'
[8]     :If SPACE≡''⎕NS''
[9]         'Msg'⎕WC'MsgBox' 'Dyalog Visual ►
[10]        'Msg'⎕WS'Text' 'This must be rur►
[11]        ⎕DQ'Msg'
[12]        :Return
[13]    :EndIf
[14]
[15]    :If ~(⊂SPACE ⎕WG'Type')∊'Root' 'Name►
[16]        'Msg'⎕WC'MsgBox' 'Dyalog Visual ►
[17]        'Msg'⎕WS'Text' 'This may not be ►
[18]        ⎕DQ'Msg'
[19]        :Return
[20]    :EndIf
```

Function | Last saved by: Dyadic:10/12/99 | Pos: 7,1

CurObj: WDesign.SPACE (Undef: | &:1 | ⎕DQ:0 | ⎕TRAP | ⎕SI:0 | ⎕IO:1 | ⎕ML:0

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 window. 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. Secondly, when an error or stop occurs and execution halts, you can press Ctrl+Enter (or select the *Trace* menuitem) from the (empty) input line. This operation (termed a *Naked Trace)* starts the Tracer at the point where the error has occurred and is described later in this section.

Tracer Options

In Version 10, 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 Version 10 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:

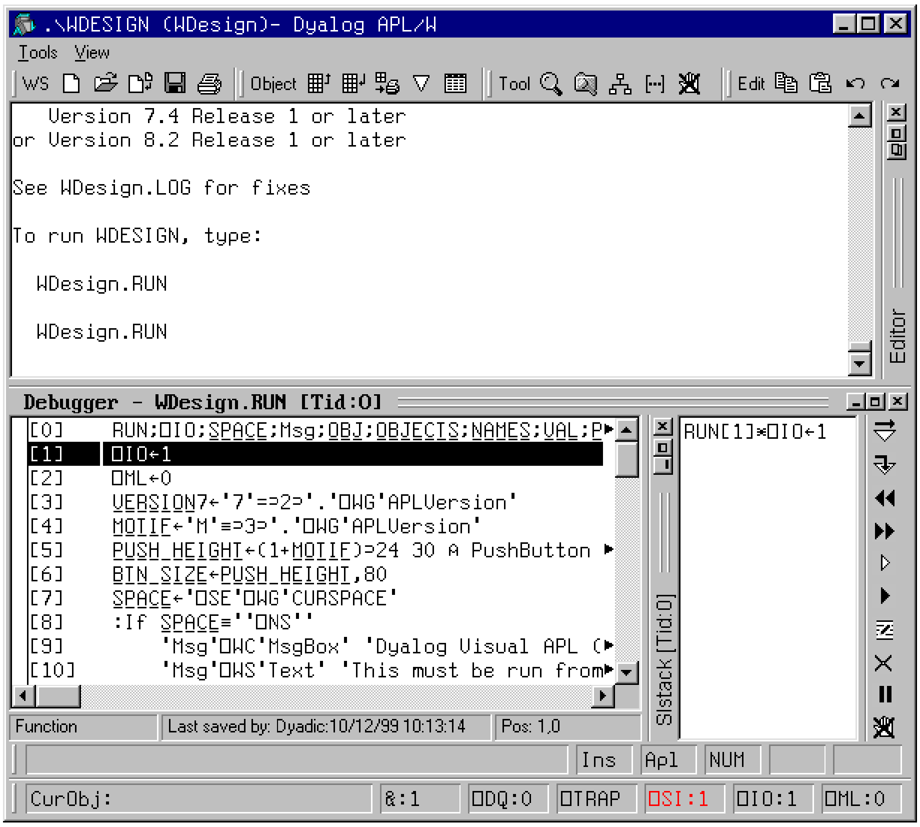
- to have the Tracer operate in multiple windows or in a single window
- to have the Trace window(s) dependant or independent of the Session window.

These alternatives are discussed later in this Chapter.

The Version 10 Tracer

The Version 10 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. Finally, the Trace Tools is implemented as a floating toolbar that may be docked, or remain free floating, as you wish.

In the default Version 10 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 *WDesign.RUN*, the top-level function in the WDESIGN workspace.


.\WDESIGN (WDesign)- Dyalog APL/W
Tools
View
WS
Object
Tool
Edit
Editor
Debugger - WDesign.RUN [Tid:0]
SIstack [Tid:0]
Function
Ins
Apl
NUM

```
   Version 7.4 Release 1 or later
or Version 8.2 Release 1 or later

See WDesign.LOG for fixes

To run WDESIGN, type:

  WDesign.RUN

  WDesign.RUN
```

```
[0]     RUN;⎕IO;SPACE;Msg;OBJ;OBJECTS;NAMES;VAL;P▸
[1]     ⎕IO←1
[2]     ⎕ML←0
[3]     VERSION7←'7'=⊃2⊃'.'⎕WG'APLVersion'
[4]     MOTIF←'M'≡⊃3⊃'.'⎕WG'APLVersion'
[5]     PUSH_HEIGHT←(1+MOTIF)⊃24 30 ⍝ PushButton ▸
[6]     BTN_SIZE←PUSH_HEIGHT,80
[7]     SPACE←'⎕SE'⎕WG'CURSPACE'
[8]     :If SPACE≡''⎕NS''
[9]         'Msg'⎕WC'MsgBox' 'Dyalog Visual APL (▸
[10]        'Msg'⎕WS'Text' 'This must be run from▸
```

```
RUN[1]*⎕IO←1
```

Last saved by: Dyadic:10/12/99 10:13:14

Pos: 1,0

CurObj: &:1 ⎕DQ:0 ⎕TRAP ⎕SI:1 ⎕IO:1 ⎕ML:0

In the default layout, the SIStack and Trace Tools windows are docked together along the right edge of the Trace window itself. The SIStack window starts minimised. The Threads window is not visible, but may be displayed from the Tracer's Tools menu.

Trace Tools

The Tracer may be controlled from the keyboard, or by using the *Trace Tools*.

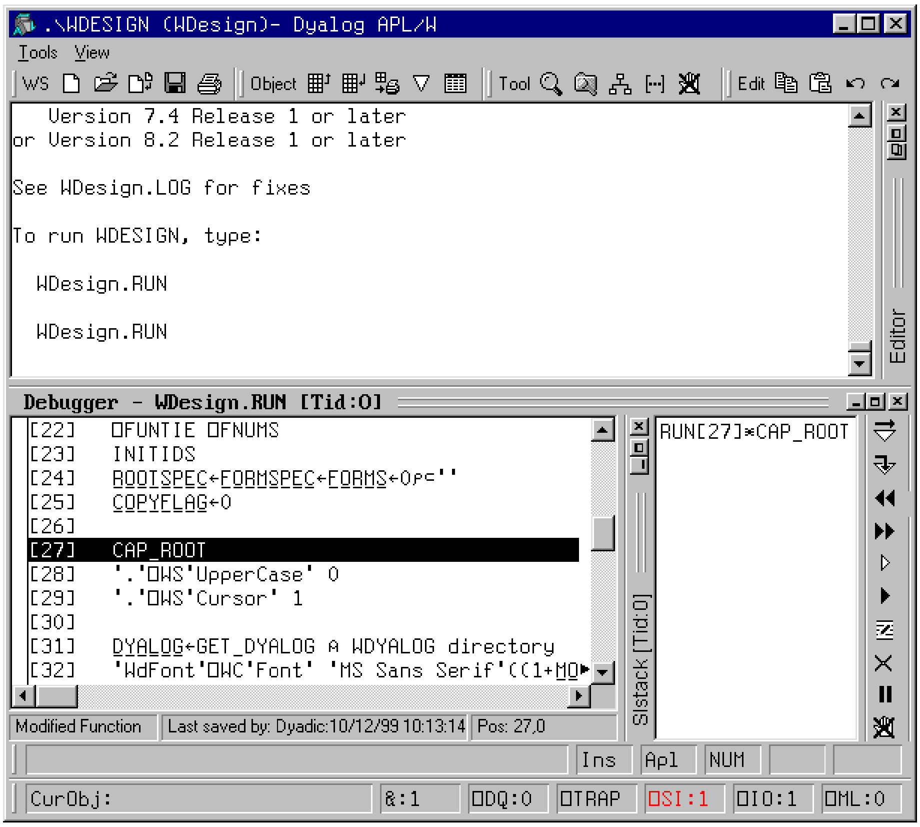
The various Trace Tools buttons are described below. The button names are solely for reference purposes in the description that follows.

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
▶	Resume	RM	→⎕LC	Resumes execution, closes all Trace windows
▷	Continue	BH		Continues execution, leaving Trace windows displayed
✎	Edit	ED	Shift+Enter	Invokes the Editor
✕	Exit	EP	Esc	Closes the Trace window, exits the current function
⏸	Intr		Ctrl+Pause	Interrupts execution
⌧	Reset	CS		Clears all break-points (resets ⎕STOP on every function)

If the Trace Tools window is docked, as in the standard Session files, it is permanently available whenever the Tracer is in use. If the Trace Tools window is not docked, the manner in which it is displayed is configurable through the *Options* menu on the session window. In the *Configuration* Dialog box, you can choose whether or not the Trace Tools are automatic. If so, they will pop-up whenever you initiate a Trace, and disappear at the end. If you switch this option off, you must explicitly select *Show* from the *Trace Tools* sub-menu when you want to use them.

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* 26 times to reach `WDesign.RUN[27]`.


.\WDESIGN (WDesign)- Dyalog APL/W
Tools
View
WS
Object
Tool
Edit
Editor
Debugger - WDesign.RUN [Tid:0]
SIstack [Tid:0]

```
  Version 7.4 Release 1 or later
or Version 8.2 Release 1 or later

See WDesign.LOG for fixes

To run WDESIGN, type:

  WDesign.RUN

  WDesign.RUN
```

```
[22]   ⎕FUNTIE ⎕FNUMS
[23]   INITIDS
[24]   ROOTSPEC←FORMSPEC←FORMS←0⍴⊂''
[25]   COPYFLAG←0
[26]
[27]   CAP_ROOT
[28]   '.'⎕WS'UpperCase' 0
[29]   '.'⎕WS'Cursor' 1
[30]
[31]   DYALOG←GET_DYALOG ⍝ WDYALOG directory
[32]   'WdFont'⎕WC'Font' 'MS Sans Serif'((1+MO
```

```
RUN[27]*CAP_ROOT
```

Modified Function | Last saved by: Dyadic:10/12/99 10:13:14 | Pos: 27,0

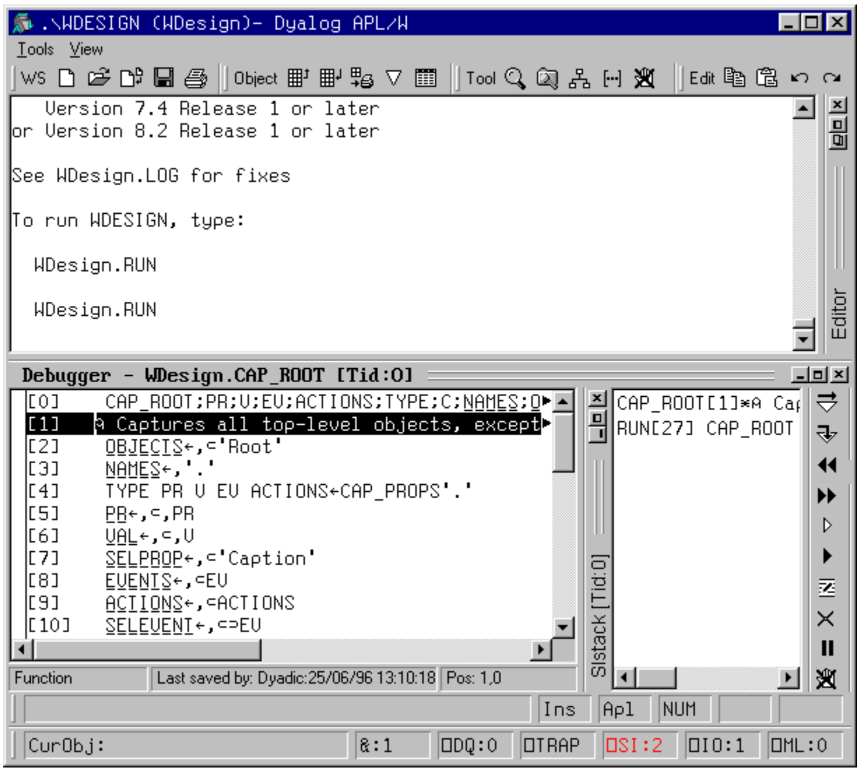
Ins | Apl | NUM

CurObj: | &:1 | ⎕DQ:0 | ⎕TRAP | ⎕SI:1 | ⎕IO:1 | ⎕ML:0

Execution Reached `WDesign.RUN[27]`

The illustration below shows the result of clicking *Trace* at this point. This caused the system to **trace into** `WDesign.CAP_ROOT`, the function called from `WDesign.RUN[27]`.

Notice how the function calling stack is represented in the SIStack window.


.\WDESIGN (WDesign)- Dyalog APL/W
Tools
View
WS
Object
Tool
Edit
Editor
Debugger - WDesign.CAP_ROOT [Tid:0]
SIstack [Tid:0]

```
  Version 7.4 Release 1 or later
or Version 8.2 Release 1 or later

See WDesign.LOG for fixes

To run WDESIGN, type:

  WDesign.RUN

  WDesign.RUN
```

```
[0]     CAP_ROOT;PR;V;EV;ACTIONS;TYPE;C;NAMES;O
[1]     ⍝ Captures all top-level objects, except
[2]     OBJECTS←,⊂'Root'
[3]     NAMES←,'.'
[4]     TYPE PR V EV ACTIONS←CAP_PROPS'.'
[5]     PR←,⊂,PR
[6]     VAL←,⊂,V
[7]     SELPROP←,⊂'Caption'
[8]     EVENTS←,⊂EV
[9]     ACTIONS←,⊂ACTIONS
[10]    SELEVENT←,⊂⊃EV
```

```
CAP_ROOT[1]*⍝ Ca
RUN[27] CAP_ROOT
```

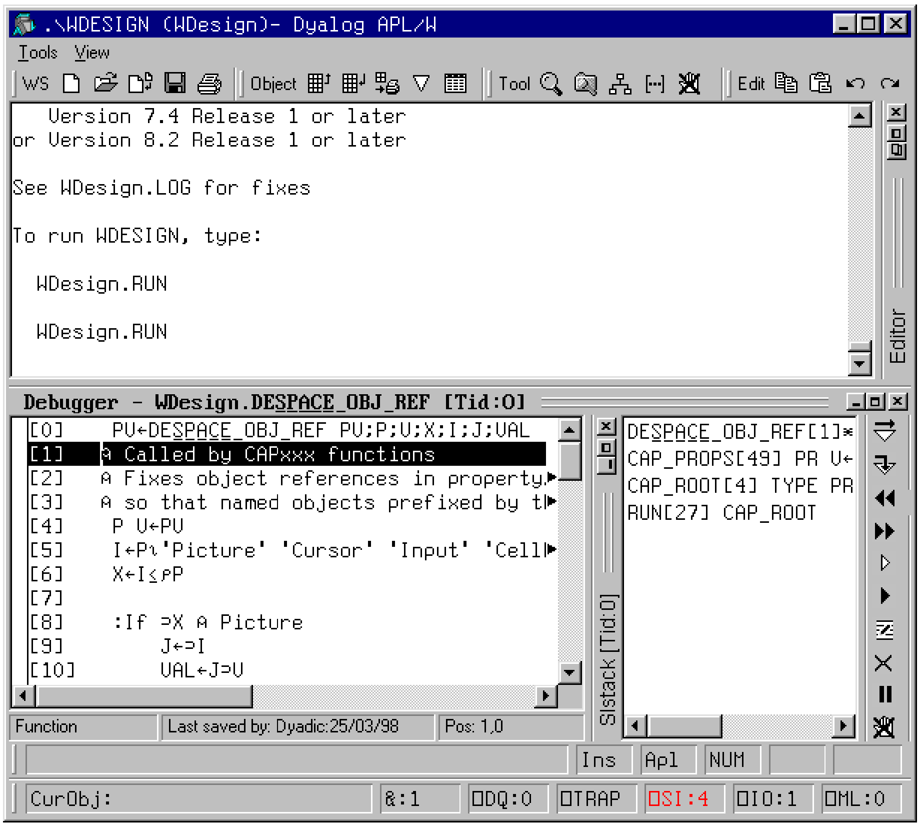
Function | Last saved by: Dyadic:25/06/96 13:10:18 | Pos: 1,0

Ins | Apl | NUM

CurObj: | &:1 | ⎕DQ:0 | ⎕TRAP | ⎕SI:2 | ⎕IO:1 | ⎕ML:0

Execution Reached `WDesign.CAP_ROOT[1]`

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


.\WDESIGN (WDesign)- Dyalog APL/W
Tools
View
WS
Object
Tool
Edit
Editor
Debugger - WDesign.DESPACE_OBJ_REF [Tid:0]
SIstack [Tid:0]

```
  Version 7.4 Release 1 or later
or Version 8.2 Release 1 or later

See WDesign.LOG for fixes

To run WDESIGN, type:

  WDesign.RUN

  WDesign.RUN
```

```
[0]     PV←DESPACE_OBJ_REF PV;P;V;X;I;J;VAL
[1]    ⍝ Called by CAPxxx functions
[2]    ⍝ Fixes object references in property▸
[3]    ⍝ so that named objects prefixed by t▸
[4]     P V←PV
[5]     I←P⍳'Picture' 'Cursor' 'Input' 'Cell▸
[6]     X←I≤⍴P
[7]
[8]     :If ⊃X ⍝ Picture
[9]         J←⊃I
[10]        VAL←J⊃V
```

```
DESPACE_OBJ_REF[1]*
CAP_PROPS[49] PR V←
CAP_ROOT[4] TYPE PR
RUN[27] CAP_ROOT
```

Function | Last saved by: Dyadic:25/03/98 | Pos: 1,0

Ins | Apl | NUM

CurObj: | &:1 | ⎕DQ:0 | ⎕TRAP | ⎕SI:4 | ⎕IO:1 | ⎕ML:0

Execution reached four levels deep

At this stage, the State Indicator is as follows:

```
        )SI
WDesign.DESPACE_OBJ_REF[1]*
WDesign.CAP_PROPS[49]
WDesign.CAP_ROOT[4]
WDesign.RUN[27]
```

Here, as at any point, the APL *stack* is displayed in the SIStack window.

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 <CS> 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.

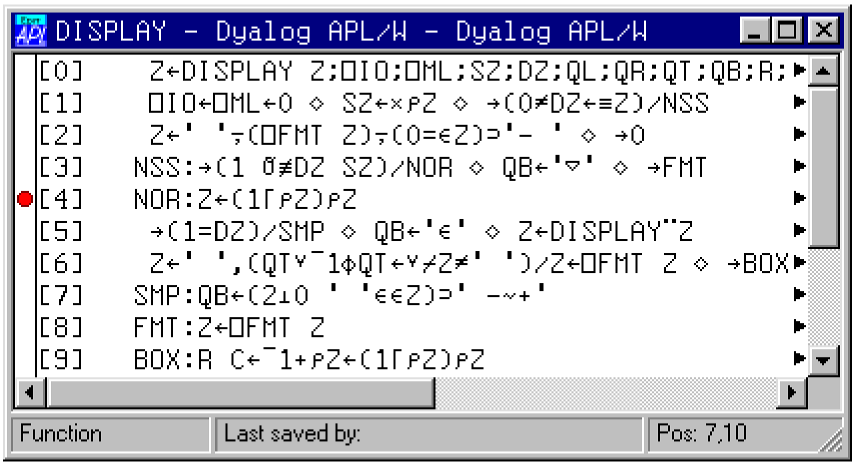
You may also open a new edit window from within the Tracer using point-and-edit.

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 lines `[4]`.



```
 [0]     Z←DISPLAY Z;⎕IO;⎕ML;SZ;DZ;QL;QR;QT;QB;R;
 [1]     ⎕IO←⎕ML←0 ⋄ SZ←×⍴Z ⋄ →(0≠DZ←≡Z)/NSS
 [2]     Z←' '⍪(⎕FMT Z)⍪(0=∊Z)⊃'- ' ⋄ →0
 [3]    NSS:→(1 0≢DZ SZ)/NOR ⋄ QB←'∇' ⋄ →FMT
●[4]    NOR:Z←(1⌈⍴Z)⍴Z
 [5]     →(1=DZ)/SMP ⋄ QB←'∊' ⋄ Z←DISPLAY¨Z
 [6]     Z←' ',(QT∨¯1⌽QT←∨⌿Z≠' ')/Z←⎕FMT Z ⋄ →BOX
 [7]    SMP:QB←(2⊥0 ' '∊∊Z)⊃' -~+'
 [8]    FMT:Z←⎕FMT Z
 [9]    BOX:R C←¯1+⍴Z←(1⌈⍴Z)⍴Z
```

`⎕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 and the subsidiary SIStack and Threads windows are not available.

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 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.

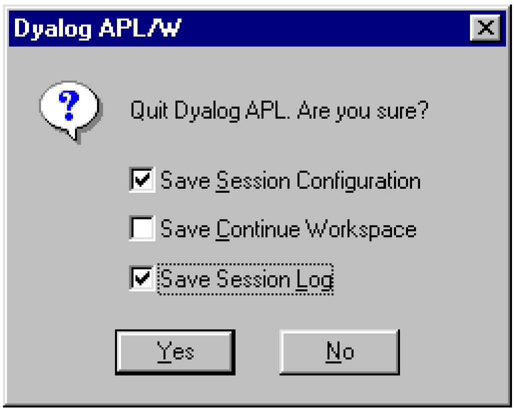
Independent Trace Stack

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.

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.


Dyalog APL/W
Quit Dyalog APL. Are you sure?
☑ Save Session Configuration
☐ Save Continue Workspace
☑ Save Session Log
Yes
No

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, Class, ToolBar, StatusBar, TipField, TabBar, ImageList, PropertySheet, 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, IconPosn, MethodList, ChildList, EventList, PropList
Events	Close, Create, FontOK, FontCancel
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.

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 `'Dyalog APL/W'`
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. Version 8 only.

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. They 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

Dyalog APL 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 unattractive at low resolutions and TrueType fonts are noticeably slower than bitmap fonts, especially when scrolling large amounts of output. 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.

The APL session font is defined by the Font property of `⎕SE`. To change the font permanently, you should set the Font property to something else, and save your Session. If you want to be able to change the font dynamically, you can provide various ways of doing this to suit your taste. The final example in this section illustrates how you may do this using Combo boxes on the tool bar.

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, the name 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.

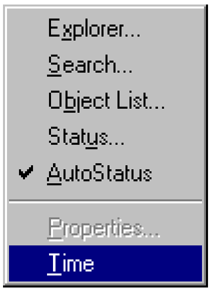
Once again, we will start by changing space into the Tools menu itself

```
      )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:

```
1998 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<ρFN
[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<ρREFS
[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.

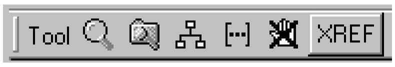
To get `FN` from CurObj, all we need to do is to change the header and lines 1-2 to:

```
[0]   XREF;FN;REFS
[1]   :If 0<ρFN←⎕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

Adding Combo Boxes to the ToolControl

This example shows how you can add Combo boxes to the Session tool bars. These can be used for a variety of purposes, such as selecting a variable. This example uses Combo boxes to change the session font dynamically.

We need to add 2 Combo objects to the tool bar; one to select font name and the other to select font size.

First, we will change space into the ToolControl namespace:

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

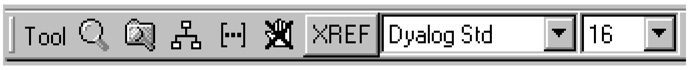
Next we will make a Combo called `f` which displays a choice of 2 fonts, namely Dyalog Std and Dyalog Std TT. When you select one or other of these, the function `⎕SE.CHG_FONT` is run.

```
      FONTS←'Dyalog Std' 'Dyalog Std TT'
      'f' ⎕WC 'Combo' FONTS ('Size' ⍬ 100)('SelItems' 1 0)
                         ('Event' 'Select' '⎕SE.CHG_FONT')
```

Then we make the second Combo called `s` which displays a choice of sizes. In this case, it offers the choice of 16, 22, 32, 44, and 48 pixels which are appropriate for the Dyalog Std font.. When you select one of these choices, the function `⎕SE.CHG_SIZE` is run.

```
      SIZES←'I2'⎕FMT 16 22 32 44 48
      's' ⎕WC 'Combo' SIZES ('Size' ⍬ 50)('SelItems' (5↑1))
                          ('Event' 'Select' '⎕SE.CHG_SIZE')
```

The ToolControl now appears as shown below:



Adding Combo boxes

The function `CHG_FONT`, which is run when you change the selected font name, is as follows:

```
      ∇CHG_FONT MSG;FNAME;SEL;SIZES;SIZE;F
[1]    FNAME SEL←(⊃MSG)⎕WG'Text' 'SelItems'
[2]    ⎕SE.FontObj[1]←⊂FNAME
[3]    SIZES←⊃SEL/(16 22 32 44 48)((9+⍳14),(24+2×⍳12))
[4]    SIZE←2⊃⎕SE.FontObj
[5]    SIZES←SIZES∪SIZE
[6]    SIZES←SIZES[⍋SIZES]
[7]    SEL←SIZES∊SIZE
[8]    SIZES←'I2'⎕FMT SIZES
[9]    'cbtop.bandtb3.tb.s'⎕WS('Items'SIZES)('SelItems'SEL)
      ∇
```

`CHG_FONT[1]` obtains the name of the font selected and a selection vector (`SEL`) which identifies which of the two was chosen.

The next two lines change the name of the font currently in use to the one that has been selected. `CHG_FONT[3]` sets `SIZES` to a set of sizes appropriate for the font selected (Dyalog Std is available only in multiples of 16 and 22 pixels).

Lines `[4-7]` ensures that the current font size realised when the font was changed) is included in the `SIZES` list.

Line `[9]` finally sets the Items and SelItems properties of the `⎕SE.tb.s` Combo box to the appropriate sizes and to reflect the current choice.

The function `⎕SE.CHG_SIZE`, which is run when you change the font size, is simpler:

```
      ∇CHG_SIZE MSG;F
[1]    ⎕SE.FontObj[2]←⍎(⊃MSG).Text
      ∇
```

As a means of changing the Session font, this example is not altogether satisfactory. It does not allow you to select Normal or Bold; nor does it check that the font actually obtained is the one requested. Nevertheless, it illustrates the principles involved.

CHAPTER 3

Input and Output Tables

Introduction

Dyalog APL supports a variety of computing environments, many of which have only a *character* interface as opposed to the *graphical* user interface provided by Microsoft Windows. For example, on Unix systems, Dyalog APL has to be capable of driving 7-bit asynchronous dumb terminals and printers. To cater for such requirements, Dyalog APL uses a system of Input and Output tables which is illustrated in Figure 3.1. The system was originally designed to cater for a multitude of different terminal and printer types; a problem that does not occur in the Windows environment. Under Windows there is just one keyboard and only one generic printer. However, the table-driven design provides complete freedom in defining keyboard layouts and has the flexibility to cater for all National language requirements. It is therefore retained under MS-Windows, even though certain aspects of the design (especially the capability to drive character mode printers) is irrelevant in this environment.

All character input and output is performed via Input and Output tables. These are simple ASCII files that define the correspondence between `⎕AV` and input/output.

The system is entirely flexible, and provides the following facilities :

- A user-definable keyboard layout.
- A means for you to map convenient keys on your keyboard to special functions and commands to be performed by Dyalog APL.
- A means to include additional National Language characters in `⎕AV`.

Overview

Dyalog APL's device support is based upon two sets of files.

The first is a collection of *Input Tables*. These provide the mapping between the codes that Dyalog APL receives as input (i.e. from your keyboard) and characters in `⎕AV`. The Input Tables also tell Dyalog APL which keystrokes are allocated to special functions and commands.

The second database is a corresponding set of *Output Tables*. These provide the information on how characters in `⎕AV` can be generated on various devices, and on how to control video attributes and colour.

Figure 3.1 illustrates how terminal input/output is controlled in Dyalog APL.

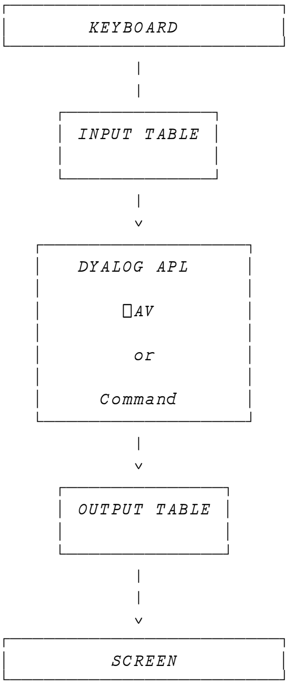


Figure 3.1 Terminal Input/Output in Dyalog APL

Input Tables

The following Input Tables are supplied with Dyalog APL/W as files in the `APLKEYS` sub-directory. Additional National Language tables may also be included.

`APL_US.DIN`	APL/ASCII keyboard, US layout
`APL_UK.DIN`	APL/ASCII keyboard, UK layout
`UNIFY_US.DIN`	Unified keyboard, US layout
`UNIFY_UK.DIN`	Unified keyboard, UK layout
`COMBO_US.DIN`	Combined APL/ASCII/Unified, US layout
`COMBO_UK.DIN`	Combined APL/ASCII/Unified, UK layout
`ASCII.DIN`	For file input using NFILES

Output Tables

The following Output Tables are supplied with Dyalog APL/W as files in the `APLTRANS` sub-directory. Only `WIN.DOT` is relevant to Dyalog APL/W. The other tables are provided only for compatibility with other implementations of Dyalog APL.

`WIN.DOT`	For the PC screen
`ASCII.DOT`	For file output using the NFILES AP
`IBM5202.DOT`	For IBM Quietwriters.
`EPSONFX.DOT`	For Epson FX printers.
`HPLJPLUS.DOT`	For HP Laser Jet Plus printers.
`PROPRINT.DOT`	For IBM Proprinters.
`TOSH351.DOT`	For Toshiba P351 printer.
`ASC_APL.DOT`	For ASCII/APL printers

Modes & Character Sets

In general very few input devices are capable of generating unique codes for all of the 256 characters and over 40 functions and commands used by Dyalog APL. Likewise, it is unusual for a printer to provide a single font containing all of the symbols you want to use. It is therefore necessary to support the concept of *modes*.

You can think of modes as corresponding to different character sets, but the concept is in fact wider than this.

The Input and Output Tables support user-definable modes (theoretically up to 255). Each mode is selected by a *mode select* code or sequence of codes which is itself defined in the Table.

Different **input** modes allow a particular keystroke to be interpreted as a different character depending upon the current mode.

For example, the default keyboard Input Table (`APL_US.DIN`) is a 2-mode keyboard (APL and ASCII) and the default mode is APL. If you type the key labelled “J” (unshifted) it will be interpreted as `⎕AV[74]` (upper-case `J`). Pressing the MODE key (+ on the Numeric Keypad) switches it to ASCII mode. Now if you type the same key labelled “J” (unshifted) it will be interpreted as `⎕AV[26]` (lower-case `j`). Pressing the MODE key again switches back to APL. You can also select APL mode by pressing Ctrl+n and ASCII mode with Ctrl+o.

Different **output** modes allow you to switch character sets on printers and other output devices. For example, an output table may specify that to output `⎕AV[97]` ('A̲') on your printer, Dyalog APL must transmit an SO code (Ctrl+n) to switch the printer to APL mode (if necessary), followed by the 3 characters '`a`', <backspace>, '_'. This facility is largely irrelevant under MS-Windows.

Customising Your Input/Output Tables

Dyalog APL/W provides input tables for your PC that support US and UK APL/ASCII and Unified keyboard layouts. It provides output tables for the screen, and (for compatibility with other implementations) for a range of printers.

Using an editor, you will have no difficulty in modifying your keyboard layout, adding additional National Language characters, or otherwise changing the way that the supplied tables work. The tables are liberally commented, and it is suggested that you use them as a starting point.

The tables provide you with the means to completely redefine `⎕AV`. It is inadvisable to remove or alter the APL symbols if you intend to write APL programs on your PC. If however you have an application that doesn't require APL symbols, there is no reason why you cannot replace the APL symbols in `⎕AV` with characters that will be of more use to you, for example an extended line-drawing set.

The Input Table

Overview

The input table is a character file organised like a matrix. The rows represent Input Items and the columns the ASCII codes or keystrokes to be used to generate them. Each column of codes represents a separate mode.

Lines that are entirely blank or that begin with the symbol '+' are treated as comments and are ignored. With a few exceptions which are noted below the general syntax is as follows :

```
{,}code{=label}: string, string, ...+ comment
```

The text to the left of the colon (`:`) describes the Input Item. It is made up of a *code* and an optional *label* which is preceded by an equals sign (=). The code uniquely identifies the item. If the item is an element of `⎕AV` it is a number representing its 0-origin index in `⎕AV`. If not, it is a special 2-character code that identifies a command. If present, the *label* defines a character string to be returned by `⎕KL` for that Input Item. If the command appears anywhere in a window menu, the *label* will also be shown against it.

After the colon (`:`) there may be one or more strings of numbers separated by commas (`,`). In general there may be one string for each of the modes in the table. Each string defines the input (keystroke) that is to generate the Input Item in that mode.

The colon (`:`) and comma (`,`) separators are mandatory, but blanks, tabs, etc. are allowed almost anywhere.

A comma (`,`) in column 1 indicates that the definition of the Input Item is in addition to a previous definition. This allows you to use several alternatives to mean the same thing. See *Alternative Definitions* later in this Chapter.

If a particular input item is defined for one mode only, you should leave the other mode columns empty. If a particular item is independent of mode you can either repeat the string across all columns, or enter a trailing asterisk (`*`).

Figure 3.2 summarises the different Input Items that are defined in the Input Table, with the exception of the commands which each have their own codes. The commands themselves are listed in Figure 2.5 in Chapter 2.

Code	**Description**
M0 - M255	How to select mode 0, mode 1, etc.
0 - 255	How to enter `⎕AV[0]`, `⎕AV[1],... ⎕AV[255]`
F0 - F255	Logical function keys 0-255
D1 - D3	Mouse button 1 -3 (down)
U1 - U3	Mouse button 1-3 (up)
PROG	Initial input mode (programmer)
USER	Initial input mode (user)

Figure 3.2 Input Items

The Keyboard

The Input Table is a **general** mechanism designed to describe any type of input device. On your PC there is only one such device, namely your keyboard. This section describes how this is supported, and how the corresponding Input Tables are coded. This information will enable you to modify the tables, if necessary, to suit your own requirements.

MS-Windows recognises two types of keys; normal keys and *modifiers*. The modifiers are Shift and Ctrl and are assigned (internal) codes of 1 and 2 respectively by Dyalog APL/W. The Alt key is reserved by MS-Windows, and is not accessible from APL.

When you press a normal key MS-Windows generates a KeyPress event which is detected by Dyalog APL. Windows supplies the *keycode* associated with the key you pressed, and which (if any) *modifiers* were active.

If there were no modifiers active, the APL keyboard driver simply passes on the keycode for interpretation by the Input Table. If one or more modifiers were active, the driver passes 2 bytes on, with the first byte containing the sum of the codes associated with each active modifier.

Thus if you press Shift+Ctrl+a, the first byte would contain 3 (1+2) and the second 65, which is the keycode associated with "a". The keystroke is therefore defined in the Input Table as the string 3 65.

The Mouse

Mouse button press/release **within ⎕SM fields** are treated as special keystrokes which are recognised by ⎕SR. This is implemented by allocating special codes `D1`, `D2` and `D3` to mouse button Down, and `U1`, `U2` and `U3` to mouse button Up. When the user presses or releases a mouse button in a ⎕SM field, APL inserts a 2-byte string into the keyboard buffer. The first byte has the value 41, the second byte identifies the button and whether it was a press or release. These codes are listed in Figure 3.3.

Button	Down	Up
1	41 32	41 192
2	41 64	41 160
3	41 128	41 96

Figure 3.3 Mouse button codes

Caps Lock

The relationship between lower-case and upper-case characters is not 'hard-coded' into the APL program, but is defined in the Input Table. It is therefore possible to support different National language conventions by editing the Table. To do so, it is necessary to understand how the mechanism works.

If a character has an upper-case equivalent, the relationship is coded in the Input Table by adding a minus sign followed by the `⎕AV` position of the upper-case character, after that of the lower-case one. The correspondence is always specified lower-upper and is best illustrated by an example.

`⎕AV[17]` is "`a`" (lower-case) and `⎕AV[65]` is "`A`" (upper-case). The definition of `⎕AV[17]` is coded as :

```
    17-65: ...
```

The "`-`" between the 17 and 65 indicates that `⎕AV[65]` is the upper-case equivalent of `⎕AV[17]`.

Having resolved an input string to a position in `⎕AV`, APL checks the state of the Caps Lock key. If it is OFF, APL takes no action. If it is ON and that position in `⎕AV` is paired with another, APL converts to the other one.

Suppose that you press "a" (keycode 65) with Caps Lock ON. First, APL translates the keycode 65 into `⎕AV[17]`. Then, because Caps Lock is ON and `⎕AV[17]` is paired with `⎕AV[65]`, it results in `⎕AV[65]` which is upper-case "A".

Suppose that you press Shift + "a" with Caps Lock ON. First, APL prefixes the 2-byte string 1 65 as a result of the Shift key. This is then translated by the Input Table into `⎕AV[65]`. Finally, because Caps Lock is on and `⎕AV[65]` is paired with `⎕AV[17]`, the keystroke resolves to `⎕AV[17]` which is lower-case "A".

Thus Caps+a gives "A" and Caps+Shift+a gives "a".

Notice that the above does not apply if you are using the standard APL/ASCII keyboard layout, and the keyboard is in APL mode. Lower-upper case pairings are defined only for the alphabetic ASCII characters. However, you could also define such a relationship for APL symbols if you wished to do so.

The APL_US Input Table

The following examples explain how the various input items are mapped to keystrokes by the `APL_US.DIN` Input Table. Please note that the keycodes generated on National Language keyboards vary from one to another. All the examples assume a US layout.

Example 1

```
M0=Apl: 2 78 *   + Ctrl+n        Apl
M1=Asc: 2 79 *   + Ctrl+o        Ascii
```

These two lines define the mode switches. Remember, the label on the left is what you get; the strings on the right are how you get them. Thus the first line means ...

- whatever mode the keyboard is, receipt of codes 2 78 (Ctrl+n) will select mode 0.

The second line means ...

- whatever mode the keyboard is in, receipt of codes 2 79 (Ctrl+o) will select mode 1.

Example 2

This example shows the entries for 'lower-case a' and 'lower-case b'. The fact that there is no string defined for mode 0 (APL) means that these characters cannot be entered in this mode. Note that anything after the comment symbol (+) is ignored.

```
17-65:  ,     65   + Lower-a
18-66:  ,     66   + Lower-b
```

These entries mean ...

- if APL receives keycode 65 (A) while in mode 1 (ASCII) it is to be interpreted as `⎕AV[17]`. If Caps Lock is on at the same time, it is instead to be interpreted as `⎕AV[65]`.

- if keycode 66 (B) is received while in mode 1 (ASCII) it is to be interpreted as `⎕AV[18]`. If Caps Lock is on at the same time, it is instead to be interpreted as `⎕AV[66]`.

Note that the case relationship between `⎕AV[17]` and `⎕AV[65]`, and between `⎕AV[18]` and `⎕AV[66]` should only be defined in one direction (lower-upper) as above.

Example 3

This example shows the entries for *Upper-Case* characters which can be input both in APL mode and in ASCII mode, although the keystroke used is different in each mode.

```
65:      65,   1 65   + Upper-case A
66:      66,   1 66   + Upper-case B
```

The first entry means ...

- if APL receives keycode 65 (A) while in mode 0 (APL) it is to be interpreted as `⎕AV[65]`.

- if APL receives Shift + keycode 65 (Shift+A) while in ASCII mode it is also to be interpreted as `⎕AV[65]`.

Similarly, `⎕AV[66]` is the result of typing unshifted key "B" in mode 0 or shifted key "B" in mode 1.

Example 4

This example illustrates definitions of some APL symbols. Note that the character '+' can be entered in either mode 0 (APL) or mode 1 (ASCII), but that '÷' can only be entered in mode 0 (APL).

```
169:      189,   1 187  + Plus
170:    1 187,          + Divide
```

The first entry means ...

- if APL receives keycode 189 (which on a US keyboard is engraved with the ASCII minus and underscore symbols) in Mode 0 (APL) it is to be interpreted as `⎕AV[169]`, which is "+".

- if APL receives Shift + keycode 187 (the ASCII "+") in mode 1 (ASCII) it is also to be interpreted as `⎕AV[169]`.

The second entry maps Shift + keycode 187 in Mode 0 (APL) to divide (÷). There is no coding for Mode 1 so divide cannot be input in ASCII mode.

Example 5

This example shows how some special functions and commands are defined.

```
IN:            45     *     + Insert
PT:          1 45     *     + Paste
CP:          2 45     *     + Copy
OP:          3 45     *     + Open
```

These entries mean ...

- if APL receives keycode 45 (the *Insert* key), it is to be interpreted as the INSERT command (IN).
- if APL receives Shift + keycode 45 it is to be interpreted as the PASTE command (PT).
- if APL receives Ctrl + keycode 45 it is to be interpreted as the COPY command (CP).
- if APL receives Ctrl + Shift + keycode 45 it is to be interpreted as the OPEN command (OP).

Example 6

This example illustrates the use of the optional label which can be interrogated by ⎕KL and then displayed in your application.

```
EP=Esc:                       27 *   + EXIT
QT=Shift+Esc:               1 27 *   + QUIT
```

Here The Escape key (keycode 27) is mapped to the EXIT command. The legend on the key *Esc* is also defined. A French Input Table might have the text "Echap" here instead.

Similarly Shift+Esc (keycode 27) is mapped to the QUIT command. There is no actual legend on this key, but this text is made available through ⎕KL so that you can give a *friendly* prompt to your users.

The idea behind this is to avoid having hard-coded descriptions of keys in *help* screens and so forth, as this would destroy much of the advantage of a user-configurable keyboard layout.

The UNIFY_US Input Table

The following examples explain how the various input items are mapped to keystrokes by the `UNIFY_US.DIN` Input Table. Please note that the keycodes generated on National Language keyboards vary from one to another. All the examples assume a US layout.

Example 1

This example shows the entries for 'lower-case a' and 'lower-case b'. Note that anything after the comment symbol (+) is ignored.

```
17-65:  65   + Lower-a
18-66:  66   + Lower-b
```

These entries mean ...

- if APL receives keycode 65 (a) it is to be interpreted as `⎕AV[17]`. If Caps Lock is on at the same time, it is instead to be interpreted as `⎕AV[65]`.

- if APL receives keycode 66 (a) it is to be interpreted as `⎕AV[18]`. If Caps Lock is on at the same time, it is instead to be interpreted as `⎕AV[66]`.

- Note that the case relationship between `⎕AV[17]` and `⎕AV[65]`, and between `⎕AV[18]` and `⎕AV[66]` should only be defined in one direction (lower-upper) as above.

Example 2

This example shows the entries for *Upper-Case* characters

```
65:   1 65   + Upper-case A
66:   1 66   + Upper-case B
```

These entries mean :

- if APL receives Shift(1) + keycode 65 (a) it is to be interpreted as `⎕AV[65]`.

- if APL receives Shift(1) + keycode 66 (b) it is to be interpreted as `⎕AV[66]`.

Example 3

This example illustrates how APL symbols are coded.

```
178:   2 73  + Iota (Index)
105:   3 73  + Upper-I underbar
199:   3 54  + Circle Backslash (Transpose)
```

These entries mean ...

- if APL receives Ctrl(2) + keycode 73 (i) it is to be interpreted as ⎕AV[178] (⍳).
- if APL receives Ctrl(2) + Shift(1) + keycode 73 (I) it is to be interpreted as ⎕AV[105] (I̲).
- if APL receives Ctrl(2) + Shift(1) + keycode 54 (6) it is to be interpreted as ⎕AV[199] (⍉).

Example 4

This example illustrates the definition of ⎕AV[172] (the question mark or roll/deal symbol) that can be entered in one of two ways.

```
172:    1 191 + Query (Roll/Deal)
,172:   2  81 + Query (Roll/Deal)
```

The first entry means ...

- if APL receives Shift(1) + keycode 191 (which on a US keyboard is engraved with "?" above "/"), the keystroke is to be interpreted as ⎕AV[172], which is "?".

The second entry means ...

- if APL receives Ctrl(2) + keycode 81 (q) it is ALSO (indicated by the comma preceding the 172) to be interpreted as ⎕AV[172].

Example 5

This example shows how some special functions and commands are defined.

```
IN:            45           + Insert
PT:          1 45           + Paste
CP:          2 45           + Copy
OP:          3 45           + Open
```

These entries mean ...

- if APL receives keycode 45 (the *Insert* key), it is to be interpreted as the INSERT command (IN).
- if APL receives Shift(1) + keycode 45 it is to be interpreted as the PASTE command (PT).
- if APL receives Ctrl(2) + keycode 45 it is to be interpreted as the COPY command (CP).
- if APL receives Ctrl(2) + Shift(1) + keycode 45 it is to be interpreted as the OPEN command (OP).

Example 6

This example illustrates the use of the optional label which can be interrogated by `⎕KL` and then displayed in your application.

```
EP=Esc:                       27 *   + EXIT
QT=Shift+Esc:               1 27 *   + QUIT
```

Here The Escape key (keycode 27) is mapped to the EXIT command. The legend on the key *Esc* is also defined. A French Input Table might have the text "Echap" here instead.

Similarly Shift+Esc (keycode 27) is mapped to the QUIT command. There is no actual legend on this key, but this text is made available through `⎕KL` so that you can give a *friendly* prompt to your users.

The idea behind this is to avoid having hard-coded descriptions of keys in *help* screens and so forth, as this would destroy much of the advantage of a user-configurable keyboard layout.

Composite Characters & Overstrikes

Composite APL characters which are the result of one symbol overstruck with another are resolved automatically by Dyalog APL. This is because overstrikes can be produced in unpredictable ways by cursor keys, and cannot be predefined in the input table. For example, you can get `C̲AT` by typing `<C><A><T><←><←><←><_>`.

All composite characters are mapped to Ctrl+Shift key combinations. However you are also able to produce them by overstriking in *Replace* mode, because of the automatic overstrike resolution in Dyalog APL.

Initial Modes

For a multi-mode Input Table, the codes PROG and USER define the initial keyboard mode. These special codes have their own simple syntax, eg.

```
PROG: 0      + Use mode 0 (APL) for programming
USER: 1      + Use mode 1 (ASCII) for user input
```

The Session Manager, Editor, and Tracer initialise the keyboard mode to that defined as PROG. In the `APL_XX.DIN` tables this is defined to be mode 0 (APL).

`⎕DQ`, `⎕SR` and the AP124 emulator (prefect) initialise the keyboard mode to that defined as USER. In the `APL_XX.DIN` tables this is defined to be mode 1 (ASCII).

Initial modes are not applicable to a single-mode table such as the `UNIFY_XX.DIN` tables.

Alternative Definitions

It is possible to provide alternative definitions for the same Input Code.

An alternative definition is defined by an input code preceded by a comma (,).

For example:

```
48:  48 *      + 0 Zero
,48: 96 *      + or 0 on Numeric Keypad (Num Lock on)
```

Single-Shifts

Normally, mode switches change the current keyboard input mode. Effectively the new mode becomes permanent until another mode switch is encountered. It is sometimes convenient to use a temporary mode change that affects only the next single keystroke. This is called a 'Single Shift'.

To define a single-shift, you must insert a minus sign (-) before the mode.

Example:

```
M0:          ,     2 78,   2 78
M1:      2 79,         ,   2 79
-M2:     2 71,     2 71,   2 71

43:         ,       ,      1 51     + UK Pound sign
```

This defines Ctrl(2) + g (keycode 71) to be a single-shift. If you type Ctrl(2) + g followed by the Shift(1) + 3 (keycode 51) it will be interpreted as a UK pound.

The advantage of the single-shift is that the keyboard automatically reverts to its preceding mode. Thus you get a UK pound with just 2 keystrokes whether you are currently in ASCII or APL. Note that you cannot achieve the same thing with the following definition:

```
43:   2 71 1 51,  2 71 1 51  + UK pound in either mode
```

This will not work unless you are able to type the keys with a negligible delay between them, because the keys Ctrl+g and Shift+3 will be interpreted as 2 separate keystrokes.

The Output Table

Overview

When you output characters from Dyalog APL, there is an implicit translation between internal representation (⎕AV) and the string of ASCII codes that is needed to display or print the information correctly. The Output Translate Table provides this mapping.

In addition, the Output Translate Table contains colour definitions that allow you to customise your APL environment.

The output table is a character file organised like a matrix. The rows represent Output Items and the columns the information that must be sent to the device to achieve them. Each column of information represents a separate mode.

Lines that are entirely blank or that begin with the symbol '+' are treated as comments and are ignored. With a few exceptions which are noted below the general syntax is as follows :

```
code:   string, string, ...+ comment
```

The text to the left of the colon (`:`) is a *code* that uniquely identifies the Output Item. If the item is an element of ⎕AV, it is a number representing its index in ⎕AV. If not, it is a special code that identifies a video attribute, foreground colour, or background colour. These codes are of the form Vnnn, Cnnn and Bnnn respectively.

After the colon there may be one or more strings of numbers separated by commas (`,`). In general there may be one string for each of the modes in the table. Each string defines what is required to generate the Output Item in that mode.

The colon (`:`) and comma (`,`) separators are mandatory, but blanks, tabs, etc. are allowed almost anywhere.

If a particular output item is defined for one mode only, you should leave the other mode columns empty. If a particular item is independent of mode you can either repeat the string across all columns, or enter a trailing asterisk (`*`).

Figure 3.4 summarises the different types of Output Item.

Code	**Description**
M0 - M255	How to select mode (character set) 0, mode 1, etc.
0 - 255	How to output `⎕AV[0]`,`⎕AV[1]`,`... ⎕AV[255]`
V0 - V255	RGB definition of video attributes 0-255
B0 - B255	RGB definition of background colour 0-255
C0 - C255	RGB definition of foreground colour 0-255
INIT	How to initialise the output device
RESET	How to reset the output device
CC	How to clear a character

Figure 3.4 Output Items

Note that the `INIT`, `RESET`, and `CC` output codes are applicable to printers, but are not used for the MS-Windows display.

The WIN.DOT Output Table

The Character Set

The mapping between `⎕AV` and the fonts supplied with Dyalog APL/W is simply one-to-one, although the positions of the symbols in the font and the position of the characters in `⎕AV` are not the same. This means that `WIN.DOT` is a simple **single-mode** table.

The following entries in the `WIN.DOT` Output Table map `⎕AV[17]` (`a`) to font position 97, and `⎕AV[96]` (`⍙`) to font position 143.

```
17:  97      + Lower-a
96:  143     + Delta underbar
```

Instructions for customising your fonts are given later in this chapter.

Colours & Video Attributes

In Dyalog APL, the appearance of a field in `⎕SM` and **`prefect`** is specified by a single number which is an encoding of video attribute (`V`), background colour (`B`), and foreground colour (`C`), i.e.

```
      appearance ← 256 ⊥ C B V
```

`V`, `B` and `C` are numbers in the range 0-255, and are mapped to physical video attributes and colours through the Output Translate Table.

Although it is theoretically possible to use all three components to specify appearance, it is normal practice to use either video (`V`) alone, or background colour (`B`) and foreground colour (`C`) together.

Originally it was intended that video (`V`) be used only for monochrome ASCII terminals. However suppose that you want to write a `⎕SM` application that may be run on a monochrome ASCII terminal, an 8- or 16-colour display, **and** under MS-Windows. In this case, the use of video (`V`) to specify appearance is a convenient means to achieve suitable results on different devices **without** changing your program.

In Dyalog APL/W, foreground and background colours are defined in terms of the intensities (0-255) of their Red, Green and Blue components. Thus for example a shade of magenta (full red and blue intensity - no green) may be represented by the 3-element vector 255 0 255. Video codes are NOT used in the standard output table `WIN.DOT`. However, if you want to use them for compatibility with other Dyalog APL implementations, they are defined by a 6-element vector whose first 3 elements define the foreground colour, and whose second 3 elements define the background colour. For example :

```
V1:  255 255 255 0 0 255  + White on Blue
```

Although it is possible to define 256 different colours for both foreground and background, the table (as distributed) defines only 16. These represent the 16 **solid** colours that are provided as standard on a VGA under MS-Windows.

Figure 3.5 lists the definitions for the 16 foreground colours. The background ones are the same, but with black and white switched around. This is necessary because the default colour *black on white* must map to 0 (foreground 0 on background 0).

Code	Red, Green, Blue components	Colour
C0	`192 192 192`	White (dim)
C1	`  0   0 128`	Dark Blue
C2	`  0 128   0`	Dark Green
C3	`  0 128 128`	Dark Cyan
C4	`128   0   0`	Dark Red
C5	`128   0 128`	Dark Magenta
C6	`128 128   0`	Dark Yellow
C7	`  0   0   0`	Black
C8	`255 255 255`	White (intense)
C9	`  0   0 255`	Blue
C10	`  0 255   0`	Green
C11	`  0 255 255`	Cyan
C12	`255   0   0`	Red
C13	`255   0 255`	Magenta
C14	`255 255   0`	Yellow
C15	`130 130 130`	Grey

Figure 3.5 Foreground colours

Output Tables for APL/ASCII Printers

The following examples are based on the Output Table **ASC_APL** which is a useful starting point for APL/ASCII printers that switch character sets on receipt of Ctrl+n and Ctrl+o.

Example 1:

```
M0:       ,    15    + Ascii
M1:    14,       ,   + Apl
```

These two lines define the mode switches. Remember, the label on the left represents what is desired; the strings on the right defines how you get it when the device is in the various output modes. This example is for a 2-mode device. Mode 0 is ASCII; Mode 1 is APL.

The first line means ...

To put the device into Mode 0 (ASCII) :

- if it is already in Mode 0, do nothing
- if it is in Mode 1 (APL), send 15 (Ctrl+O)

The second line means ...

To put the device into Mode 1 (APL) :

- if it in Mode 0 (ASCII), send 14 (Ctrl+n)
- if it is already in Mode 1, do nothing

Example 2:

This example shows how to define entries to get *lower-case* a and *lower-case* b. On a standard APL/ASCII printer we can only get lower-case characters when the printer is in ASCII mode. There is therefore no output string for mode 1.

```
17:    97,        + Lower-case a
18:    98,        + Lower-case b
```

To output `⎕AV[17]`, the device must be in mode 0, then we must output ASCII code 97. Similarly for `⎕AV[18]` we must output ASCII code 98. If the device is not already in mode 0 (ASCII), Dyalog APL will automatically select it by outputting the appropriate mode switch string.

Example 3:

This example shows how to define entries for characters that can be output in different modes.

```
65:     65,     97    + Upper-case A
66:     66,     98    + Upper-case B
```

To output `⎕AV[65]` we must send ASCII code 65 if the device is in mode 0, or ASCII code 97 if the device is in mode 1. As the character can be displayed in either, there is no need to switch modes. Similarly, for `⎕AV[66]` (`B`).

Example 4:

This example illustrates definitions of some APL symbols.

```
169:    43,     45   + Plus
170:      ,     43   + Divide
```

Note that the character '+' can be output in either mode 0 (ASCII) or mode 1 (APL), but that '÷' can only be output in mode 1 (APL).

Initialising & Resetting the Device

Two special output strings are defined. `INIT` contains a string that is used to initialise the device; `RESET` contains a string to reset it after use by Dyalog APL.

IMPORTANT: It is assumed that `INIT` places the output device in mode 0.

Printer Support

Introduction

This section describes how to drive printers in character mode using the PRT auxiliary processor. This method of driving printers has been superseded by the Printer object and this section is provided **only** to provide compatibility with other implementations of Dyalog APL. See Printer object in Object Reference for details.

To drive a printer, Dyalog APL needs to know:

- Which output translation table to use.
- Which font, if any, needs to be downloaded.
- The destination of the print.

Dyalog APL is supplied with the necessary support for the following printers:

- Standard ASCII printers
- Standard APL/ASCII printers
- Toshiba 351P with APL font cartridge
- IBM 4201 Proprinter
- IBM 5202 Quietwriter with APL font cartridge
- Epson FX 9-pin printers
- Hewlett-Packard LaserJet Plus for which a selection of fonts is supplied
- Postscript printers (see *`POSTSCRIPT`* workspace)

The PRT Auxiliary Processor

The `prt` Auxiliary Processor is used to route text from Dyalog APL to a printer port or file. This AP provides four external functions; `prtargs`, `prt`, `arbout` and `prtoff`. These functions are not usually called directly; they are called by the appropriate APL cover functions held in the workspace `PRT`.

The function `prtargs` takes a single argument which is a 3-element vector of text vectors, containing the names of the translation table, font and destination. The destination parameter is a character string containing the name of the printer port, or the name of a file to which the output is to be written.

`prtargs` initialises the print task by reading the requested font file and writing its contents to the destination, then selecting the requested translate table for any subsequent print calls. If the supplied translation table name is null, no translation is performed. If the supplied font name is null, no font is downloaded. If the supplied destination name is null, output will be sent to LPT1.

The function `prt` takes a single argument of a simple character matrix and returns no result. Output from `prt` is sent to the chosen destination, and any translation performed depends upon the translation table selected by the call to `prtargs`.

The function `arbout` takes a numeric vector and sends it untranslated to the chosen destination. It works in a similar way to `⎕ARBOUT`, and may be used to insert control codes such as BEL or ESC into the text.

The function `prtoff` terminates the Auxiliary Processor.

Example:

```
      'PRT' ⎕SH 'PRT'                ⍝ Start the AP
      prtargs 'ASCII' '' 'LPT1'      ⍝ Use ASCII.DOT
                                     ⍝ Output Table
                                     ⍝ and write
                                     ⍝ to port LPT1.
      prt 'HELLO'                    ⍝ Send text to AP
      arbout 12                      ⍝ Send page throw
      prt 'GOODBYE'                  ⍝ Send text to AP
      prtoff                         ⍝ Terminate AP
```

The Workspace PRT

The workspace `PRT` contains setup functions for several different types of printers. A setup function defines the translation table (if any) to be used, the font file (if any) to be downloaded, and the destination printer port (default LPT1). The following setup functions are defined:

`PRTON∆ASCII`	Standard ASCII printer. Translation required is "ASCII". Requires no font file. Destination defined as the default.
`PRTON∆STD`	Standard ASCII/APL printer. Translation required is "`ASC_APL`". Requires no font file. Destination defined as the default.
`PRTON∆EPSON`	Epson FX printers. Translation required is "`EPSONFX`". Font file required is "`EPSONFX`". Destination defined as the default.
`PRTON∆TOSHIBA`	Toshiba 351P with APL font cartridge. Translation required is "`TOSH351`". Requires no font file. Destination defined as the default.
`PRTON∆PROPRINTER`	IBM Proprinter. Translation required is "`PROPRINT`". Font file required is "`PROPRINT`". Destination defined as the default.
`PRTON∆5202`	IBM 5202 Quietwriter with APL font. Translation required is "`IBM5202`". Requires no font file. Destination defined as the default.
`PRTON∆HPLJ`	HP LaserJet Plus. Translation required is "`HPLJPLUS`". Font file required is "`HPC12P`". (Note that 3 other font files are provided) Destination defined as the default.

These functions are very simple; they call the general setup function `PRTON`, passing the above information as an argument. This in turn starts the Auxiliary Processor **`prt`** and passes the information on to the external function `prtargs`. The AP initialises the print task by reading the requested font file and writing it to the given destination, then selects the requested translate table for any subsequent print calls.

The function `PRT` is used to pass the text to be printed by the AP. `PRT` converts its argument (which may be any APL object) to a printable form and routes it to the AP using `prt`.

The function `PRTOFF` is used to terminate the AP using `prtoff`.

Example:

```
PRTON∆PROPRINTER ⍝ Setup for IBM Proprinter;
                 ⍝ proprinter font is sent to
                 ⍝ the printer, translate table
                 ⍝ proprinter is selected, and
                 ⍝ all subsequent print calls
                 ⍝ will be sent to "LPT1".
PRT MY_REPORT    ⍝ Send text matrix MY_REPORT
                 ⍝ to the printer
PRTOFF           ⍝ Finish printing
```

If you want to override the defaults, the function `PRTON` can be called directly.

Example:

```
PRTON 'EPSONFX' '' 'TEMP'   ⍝ Set translation to
                            ⍝ epsonFX, don't
                            ⍝ download a font,
                            ⍝ and send text to a
                            ⍝ file called "TEMP"

PRT PAGE1                   ⍝ Print first page
arbout 12                   ⍝ Throw a page
PRT PAGE2                   ⍝ Print next page
arbout 12                   ⍝ Throw a page
PRTOFF                      ⍝ Finish printing

⎕CMD 'PRINT TEMP'           ⍝ Print file
```

Tailoring the supplied functions

The supplied functions act as templates, and can be amended to suit your particular requirements.

Destination Task

If there is more than one printer connected to your system, you will need to add the port of the particular printer that you want to use.

Translation Tables

You may want certain characters to be translated according to your own requirements. For example, you may want `"¯"` to be translated to `"-"` when printing financial reports. To achieve this, amend the relevant translation table.

You can also use the translate table to access the special features of your printer. Suppose your printer switches to *enhanced print mode* on receipt of the string "Esc E Esc G". You could set up the translate table for that printer such that whenever you send "!", it is translated into "27 69 27 71". Hence, the string "!Hello World" would be printed in the enhanced mode. This is sometimes a neater way to drive the printer than making direct use of *`arbout`*.

Unsupported Printers

Your printer may not currently be supported by the standard Dyalog APL product, and may require a translation table or an APL font. It is possible for you to set these up yourself, although you will need a manual for your printer if you are going to attempt it. Please ask your distributor for help and advice.

Note that by default *`prtargs`* expects to find the specified font file in the `FONTS` sub-directory. If you want to use a font located elsewhere on your system, you must specify its full path name.

HP LaserJet Plus Fonts

Four APL fonts for the HP LaserJet Plus are provided. Each font contains only the special APL symbols and is downloaded into font number 1301. This is then selected as the secondary font. All other symbols are obtained from a corresponding Roman 8 ASCII font which is selected as the primary font. The names of the APL fonts indicate which ASCII font they are used with, and are as follows :

`HPE10P` : Elite 10-point, portrait
`HPE10L` : Elite 10-point, landscape
`HPC12P` : Courier 12-point, portrait
`HPC12L` : Courier 12-point, landscape

The function `PRTON∆HPLJ` uses the larger portrait font `HPC12P` in conjunction with Courier 12 Roman 8 characters (the default LaserJet font). To use another one you can either edit the function, or call `PRTON` directly; eg. to use the smaller landscape font `HPE10L`:

```
      PRTON 'HPLJPLUS' 'HPE10L' 'LPT2'
```

All four fonts use the same Translate Table "`HPLJPLUS`". This table maps `⎕AV` characters to APL symbols in the four supplied fonts, and to ASCII symbols in the corresponding Roman 8 font provided by the printer.

To obtain National Language or special characters you must edit this table to select the required symbols from the Roman 8 font.

POSTSCRIPT Workspace

Postscript printers operate in an entirely different manner from other types of printer and cannot be supported using the standard method described above. Instead, a special `POSTSCRIPT` workspace is provided. Full details are supplied in this workspace.

CHAPTER 4

APL Files

Introduction

Most languages store programs and data separately. APL is unusual in that it stores 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 two types of file systems available to you which preserve the idea that your data consists of APL objects; hence you can only access these types of files from within APL

The two types of file systems discussed here are **External Variables** and **Component Files**. The first is very simple to use, since familiar APL expressions are used to access the file. The second has an associated 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.

Read both sections before you decide on the type of file system to use. Although both are actually implemented in the same way internally, each is good in particular circumstances.

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:

```
      DISP 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

```
      DISP 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 look at the two ways that we can write this APL variable data out to disk.

External Variables

Overview

The system function `⎕XT` associates an APL variable with a file. Whenever you reference the variable, data is read from the file. Whenever you assign to the variable, data is written to the file. (See *Language Reference* for more details).

Let's make our database into an external variable.

First, we'll associate a variable `X` with a new file called `personnel` using `⎕XT`:

```
      'personnel' ⎕XT 'X'
```

What's in `X`?

```
      X
VALUE ERROR
      X
      ^
```

`X` has no value, since there is nothing in the file.

Now we'll assign our `DATA` variable to `X`, thus writing our data to disk:

```
       X ← DATA
```

Now, what's the shape of `X`?

```
      ⍴X
124
```

Let's erase `X`, and reassociate the file with our variable `DATA`:

```
      ⎕EX 'X'

      'personnel' ⎕XT 'DATA'
```

We can use the same APL expressions as before to access our database, even though it's now on disk, not in our workspace:

Example 1:

Show record 2

```
           DISP 2⊃DATA
┌───────┬──┬──────────┐
│Pauline│16│Isleworth│
└───────┴──┴──────────┘
```

Note that if the size of the variable is greater than 8Kb and it is nested, indexing only accesses the part that it needs. Hence, in this example, only record 2 is read into the workspace, NOT the whole database.

Example 2:

How many people in the database?

```
           ⍴DATA
   124
```

Example 3:

Correct Pauline's age:

```
           (2 2⊃DATA)←21
```

Example 4:

Add a new record to the database:

```
           DATA ,← ⊂'Geoff' 41 'Oxford'
```

Note that references to the whole variable read or write the ENTIRE file.

The following tables shows how external variable usages correspond to standard file operations, and non-standard ones.

File Operation	External Variable Expression
Create a new file	`'NEWFILE' ⎕XT 'VAR'`
Open existing file	`'OLDFILE' ⎕XT 'VAR'`
Read record from file	`REC ← N⊃VAR`
Write record to file	`(N⊃VAR) ← REC`
Append record to file	`VAR ← VAR , ⊂REC`
Which file is open	`XT 'VAR'`
Close the file	`⎕EX 'VAR'`

External Variables and Standard File Operations

File Operation	External Variable Expression
Reverse the file	`VAR ← ⌽ VAR`
Sort file on age	`VAR ← VAR[⍋2⊃¨VAR]`
Drop first record	`VAR ← 1 ↓ VAR`

External Variables and Non-Standard File Operations

But remember, references to the whole variable read or write the ENTIRE file, and that although it seems simple to sort an entire file, it's going to take quite a while to do it if the file consists of 10,000 records!

Sharing External Variables

If you are working in a network, you may want to make your database available to other users in the system.

External variables may be EXCLUSIVE or SHARED. An exclusive variable can only be accessed by the owner of the file. A shared external variable may be accessed (concurrently) by other users. Access to an exclusive variable is faster than to a shared one because APL does not have to flush back or refresh disk buffers between file accesses. An external variable is always created as EXCLUSIVE. You can change the access control using a Dyalog APL utility program XVAR.EXE.

Controlling Multi-User Access

Dyalog APL contains mechanisms that prevent data getting mixed up if two users update an external variable at the same time. However, it is the programmer's responsibility to control the logic of multi-user updates. Both types of file system use the same facility, `⎕FHOLD`, to achieve this

Be careful when you make an association with a variable. Remember that the variable takes the value of the file on association, not the other way around. Consider the example below:

```
      DATA ← MAKE_DATABASE    ⍝ Complicated program that
                              ⍝ takes a long time to run

      'newfile' ⎕XT 'DATA'    ⍝ Associate a new file
                              ⍝ with the DATA variable

      ⍴DATA                   ⍝ DATA has taken on the
VALUE ERROR                   ⍝ value of this file !
      ⍴DATA
      ^
```

If you want to write existing data to a file, use a temporary variable to make the association:

```
      DATA ← MAKE_DATABASE    ⍝ Complicated program that
                              ⍝ takes a long time to run

      'newfile' ⎕XT 'TEMP'    ⍝ Associate a new file
                              ⍝ with temporary variable

      TEMP ← DATA             ⍝ Write data to file
```

Component Files

Introduction

The APL Component File System is a more formal file system than External Variables. You may already be familiar with Component File Systems offered with other versions of APL; this version is compatible with APL*PLUS, with the exception of the *slippery tie* facility, and is very similar to the version offered by SHARP APL.

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:

```
 14 13 12 11 10 9 8 7 6 5 4 3 2 1    Bit No.
┌──┬──┬──┬──┬──┬─┬─┬─┬─┬─┬─┬─┬─┬─┐
│  │  │  │  │  │ │ │ │ │ │ │ │ │ │
└──┴──┴──┴──┴──┴─┴─┴─┴─┴─┴─┴─┴─┴─┘     File       Access
  ↑  ↑  ↑  ↑  ↑   ↑   ↑ ↑ ↑ ↑ ↑ ↑    Operation    Code
  │  │  │  │  │   │   │ │ │ │ │ │
  │  │  │  │  │   │   │ │ │ │ │ └──── ⎕FREAD          1
  │  │  │  │  │   │   │ │ │ │ └────── ⎕FTIE           2
  │  │  │  │  │   │   │ │ │ └──────── ⎕FERASE         4
  │  │  │  │  │   │   │ │ └────────── ⎕FAPPEND        8
  │  │  │  │  │   │   │ └──────────── ⎕FREPLACE      16
  │  │  │  │  │   │   └────────────── ⎕FDROP         32
  │  │  │  │  │   │
  │  │  │  │  │   └────────────────── ⎕FRENAME      128
  │  │  │  │  │
  │  │  │  │  └────────────────────── ⎕FRDCI        512
  │  │  │  └───────────────────────── ⎕FRESIZE     1024
  │  │  └──────────────────────────── ⎕FHOLD       2048
  │  └─────────────────────────────── ⎕FRDAC       4096
  └────────────────────────────────── ⎕FSTAC       8192
```

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 are 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` gives 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)
⎕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
⎕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)
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's build a component file to hold our personnel database.

First, we must make sure that the component file system is available to us. Unless you are using the FSCB control mechanism, there should never be any need to issue this command (see Component File Control Mechanisms for more details):

```
        ⎕FAVAIL          ⍝ Returns 1 if all OK,
1                        ⍝ else returns 0
```

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

```
      DISP ⎕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×ι(⊂'Geoff')∊NAMES
[6]   DATA←DATA,⊂'Geoff' 41 'Hounslow'
[7]  END: ⎕FHOLD ι0
     ∇

     ∇ UPDATE;DATA;NAMES
[1]  ⍝ Using the component file
[2]   'PERSONNEL' ⎕FSTIE 1
[3]   ⎕FHOLD 1
[4]   NAMES←⊃∘⎕FREAD ¨ 1,¨ι¯1+2⊃⎕FSIZE 1
[5]   →END×ι(⊂'Geoff')∊NAMES
[6]   ('Geoff' 41 'Hounslow')⎕FAPPEND 1
[7]  END:⎕FUNTIE 1 ⋄ ⎕FHOLD ι0
     ∇
```

Successive `⎕FHOLD`s on a file are queued by Dyalog APL; once the first `⎕FHOLD` is released, the next on the queue holds the file. `⎕FHOLD`s 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 `⎕FHOLD`s. 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 `⎕FHOLD`s 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 ⍬ ⋄ →ASK
[17]   ⍝ Ok to use - flag in-use and release hold
[18]   FREE:CV[CMP]←1 ⋄ CV ⎕FREPLACE 1 1⋄ ⎕FHOLD ⍬
[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 ⍬
[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 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:

```
┌─┐    ┌─┐    ┌─┐
│1│    │2│    │3│
├─┴────┼─┴────┼─┴──────┐
│ One  │ Two  │ Three  │
└──────┴──────┴────────┘
```

Replace the second component with something the same size:

```
      'Six' ⎕FREPLACE 1 2
```

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

```
┌─┐    ┌─┐    ┌─┐
│1│    │2│    │3│
├─┴────┼─┴────┼─┴──────┐
│ One  │ Six  │ Three  │
└──────┴──────┴────────┘
```

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│
┌─────┼─┴───┬────────────┼─┴──────────┼─┴────────┐
│□□□□□│ Two │□□□□□□□□□□□□│ 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.

```
            ┌─┐          ┌─┐          ┌─┐
            │2│          │1│          │3│
┌───────────┼─┴────┬─────┼─┴──────────┼─┴────────┐
│□□□□□□□□□□□│BigTwo│□□□□□│ Bigger One │ BigThree │
└───────────┴──────┴─────┴────────────┴──────────┘
```

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%.

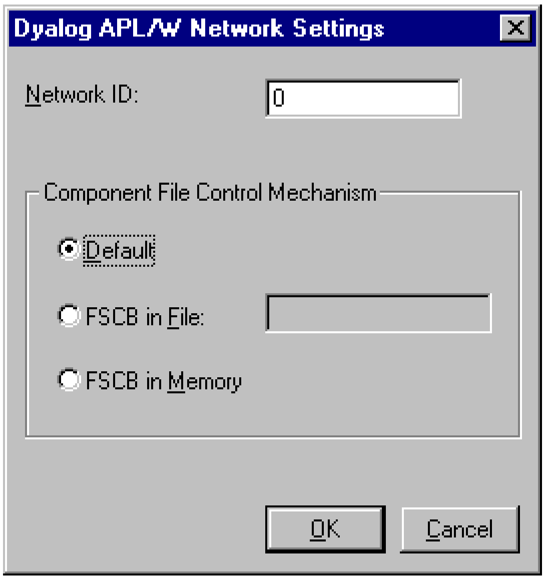
Whenever you issue a monadic `⎕FRESIZE` command on a component file, Dyalog APL COMPACTS the file; that is, it restructures it by reordering the components and by amalgamating the free areas at the end of the file. It then truncates the file and releases the disk space back to the operating system (note 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.

There is no equivalent command to compact an external variable.

Component File Control Mechanisms

Introduction

Three different component file control mechanisms are provided. You may choose which of these is to be used from the *Network* section of the Configuration Dialog box as shown below. However, if you intend to share files with other users it is essential that all users choose the same mechanism. Failure to do so will result in damaged files and loss of data.



Dyalog APL/W Network Settings

Network ID: 0

Component File Control Mechanism

● Default

○ FSCB in File:

○ FSCB in Memory

Component File options

Default

The *default* control mechanism employs standard DOS file facilities and is the recommended option for use under all versions of Microsoft Windows. This mechanism is applied if the **File_Control** parameter is set to 2.

FSCB in file

This option is provided to allow you to share component files with users running earlier Versions of Dyalog APL. It may also be applicable in networks where the standard DOS file facilities to not apply or are unreliable. For example, at the time of writing, a fault in the Microsoft Novell client software for Windows NT prevents the use of the default control mechanism (the equivalent client from Novell operates correctly).

If you choose this option, the File System Control Block (FSCB) is a single control file which normally resides on a network server. The name of the file, which must be accessible by all users for read and write operations, is defined by the *aplfscb* parameter. The FSCB file records information about system-wide component file ties and holds and is dynamically updated whenever any APL application uses *⎕FCREATE*, *⎕FERASE*, *⎕FTIE*, *⎕FSTIE*, *⎕FHOLD*, *⎕FUNTIE*. The FSCB file is also used by *⎕XT* to administer access to External Variables. This mechanism is applied if the **File_Control** parameter is set to 1.

The use of this option is discussed in detail below. Under normal circumstances, it meets all design criteria. However, it has the disadvantage that it does not recover automatically when an APL session that has component files tied terminates abnormally. If this happens, it is necessary to reset the FSCB file manually.

FSCB in memory

This option is suitable for use if you are certain that you will never need to share files with other users nor between two APL sessions on your PC. The mechanism is essentially the same as the FSCB in file, except that it is implemented in memory and provides the best performance of the three control mechanisms available. This mechanism is applied if the **File_Control** parameter is set to 0.

The FSCB File

How it Works

Consider the multi-user environment shown below:

```
 ┌────────────────┐               ┌────────────────┐
 │ APL PROCESS 1  │               │ APL PROCESS 2  │
 └────────────────┘               └────────────────┘
    |           |                    |           |
    ↓           ↓                    ↓           ↓
  ⎕FTIE       ⎕FSTIE               ⎕FSTIE      ⎕FSTIE
    |         ⎕FHOLD                 |           |
    ↓           ↓                    ↓           ↓
┌─────┐   ┌──────────────────────────────┐  ┌────────┐
│INDEX│   │          PERSONNEL           │  │SALARIES│
└─────┘   └──────────────────────────────┘  └────────┘
```

Here there are two APL processes running. APL1 has exclusively tied the INDEX file, and has share-tied the PERSONNEL file, then issued a ⎕FHOLD. APL2 has share-tied the PERSONNEL file and the SALARIES file. In this case, the FSCB would contain entries similar to those shown below:

	APL1	APL2
INDEX	tie	
PERSONNEL	share/hold	share
SALARIES		share

This table is amended every time one of the file functions shown below are used:

```
⎕FCREATE       ⎕FERASE       ⎕FHOLD
⎕FSTIE         ⎕FTIE         ⎕FUNTIE
⎕XT
```

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 capcity 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 128 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 5120 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:

```
┌──────────────────┐
│ Operating System │   ┌────────┐    ┌─────────┐
│ instruction to   │-->│ BUFFER │--->│ File on │
│ write large data │   └────────┘    │  disk   │
│ object to a file │                 └─────────┘
└──────────────────┘
```

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.

APL File Integrity Check

qfsck is an auxiliary processor that checks the internal structure of an APL file.

qfsck defines two external functions, `qfsck` and `qfem`. The function `qfsck` performs the integrity check. If the check fails for some reason, an error code is signalled, which may be trapped. The function `qfem` returns the relevant error message that corresponds to a given error number.

The function `qfsck` is monadic, and takes as an argument a character vector specifying the pathname of the file to be checked. Note that under Windows, the file extension .DCF is not assumed and must be supplied. `qfsck` returns its argument as a shy result, but signals an error and exits if any fault is detected in the structure of the component file.

The function `qfem` is monadic, and takes as a single numeric argument, specifying an event code. `qfem` returns a character vector containing the event message corresponding to the given event code. If an invalid event code is supplied to `qfem`, then `qfem` returns the character vector 'Unknown event code'.

Error Reports

If a problem is encountered by `qfsck`, one of the event codes below will be signalled. If this event code is supplied to `qfem`, the associated error report will be returned.

Error Code	**Error Report**
260	Space not accounted for
261	Error or unexpected EOF reading file
262	Unable to open file
263	Invalid magic number
264	Incorrect index tree depth
265	Block overlap
266	Pointer out of range
267	Component count wrong
268	Too large to check
269	Address tree not ordered
270	Span tree not ordered
271	Address tree not balanced
272	Span tree not balanced
273	Address tree/Span tree totals differ
274	Address tree/Span tree contents differ

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.

However, you must be aware of the possible effects of manipulating APL files outside APL. Please note that these are not only applicable to APL; any system expects its files to be accessed only by co-operating tasks.

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

Error Messages

There is a set of APL error messages associated with the APL file system. These are fully documented in the *Language Reference*. Most of the messages are self-explanatory, but some of those that relate to external variables can be confusing:

```
VALUE ERROR
```

You have associated a variable with a new file that as yet has no value.

```
DOMAIN ERROR
```

You have tried to associate a variable with an invalid file name.

```
FILE TIED
```

You have already associated this file with a variable.

```
FILE ACCESS ERROR
```

You do not have access to this file. Ask the owner to give you access permission, using the appropriate operating system command.

CHAPTER 5

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 (see Figure 1). 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.

Error Codes

Code	Description
0	Any error in the range 1-999
1	WS FULL
2	SYNTAX ERROR
3	INDEX ERROR
4	RANK ERROR
5	LENGTH ERROR
6	VALUE ERROR
7	FORMAT ERROR
10	LIMIT ERROR
11	DOMAIN ERROR
12	HOLD ERROR
16	NONCE ERROR
18	FILE TIE ERROR
19	FILE ACCESS ERROR
20	FILE INDEX ERROR
21	FILE FULL
22	FILE NAME ERROR
23	FILE DAMAGED
24	FILE TIED
25	FILE TIED REMOTELY
26	FILE SYSTEM ERROR
28	FILE SYSTEM NOT AVAILABLE
30	FILE SYSTEM TIES USED UP
31	FILE TIE QUOTA USED UP
32	FILE NAME QUOTA USED UP
34	FILE SYSTEM NO SPACE
35	FILE ACCESS ERROR - CONVERTING FILE
38	FILE COMPONENT DAMAGED

Figure 1 Error codes

52	FIELD CONTENTS RANK ERROR
53	FIELD CONTENTS TOO MANY COLUMNS
54	FIELD POSITION ERROR
55	FIELD SIZE ERROR
56	FIELD CONTENTS/TYPE MISMATCH
57	FIELD TYPE/BEHAVIOUR UNRECOGNISED
58	FIELD ATTRIBUTES RANK ERROR
59	FIELD ATTRIBUTES LENGTH ERROR
60	FULL-SCREEN ERROR
61	KEY CODE UNRECOGNISED
62	KEY CODE RANK ERROR
63	KEY CODE TYPE ERROR
70	FORMAT FILE ERROR
72	NO PIPES
76	PROCESSOR TABLE FULL
84	TRAP ERROR
200-499	Reserved for distributed auxiliary processors
500-999	User-defined errors
1000	Any error in range 1001-1006
1001	Stop vector
1002	Weak interrupt
1003	INTERRUPT
1005	EOF INTERRUPT
1006	TIMEOUT
1007	RESIZE (⎕SM window)
1008	DEADLOCK

Figure 1 Error codes (continued)

Last Error number and Diagnostic Message

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

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

Error Number for last occurring error:

```
      ⎕EN
11
```

Error Message associated with code 11:

```
      ⎕EM 11
DOMAIN ERROR
```

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

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

Use function `DISPLAY` to show structure:

```
      DISPLAY ⎕DM
.→---------------------------------.
|.→-----------..→---------..→-----.|
||DOMAIN ERROR||      10÷0||     ^||
|'------------''----------''------'|
'∊---------------------------------'
```

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

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

Error Trapping Control Structure

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

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

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

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

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

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

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

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

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

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

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

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

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 of 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:

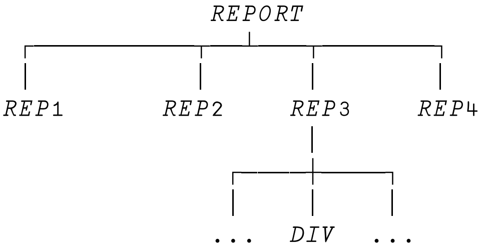
```
     ∇ R←TIE∆FILE FILE;FILES
[1]    ⍝ Tie file FILE with next available tie number
[2]    ⍝
[3]    ⍝ All files in my directory
[4]      FILES←⎕FLIB 'mydir'
[5]    ⍝ Remove trailing blanks
[6]      FILES←dbr¨↓FILES
[7]    ⍝ Required file in list?
[8]      →ERR×⍳~(⊂FILE)∊FILES
[9]    ⍝ Tie file with 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×ι0=ρOPTION←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.

CHAPTER 6

Utilities, Workspaces & Tutorials

XUTILS AP (C Utility Functions)

Introduction

The **XUTILS** Auxiliary Processor provides a set of **fast** utility functions written in C. Starting the Auxiliary Processor causes the following External Functions to be defined in your workspace. Each of these is described fully in the following sections.

`avx` : Returns the zero-origin index in `⎕AV` of a character array.

`box` : Converts matrices to vectors and vice-versa.

`dbr` : Delimited blank removal.

`hex` : Returns the internal hexadecimal representation of an array.

`ltom` : Converts a character linelist to a matrix.

`ltov` : Converts a character array to a vector of vectors.

`mtol` : Converts a character array to a linelist.

`ss` : String search and replacement.

`vtol` : Converts an array of character vectors to a line-list.

Using XUTILS

The Auxiliary Processor is invoked by:

```
      'XUTILS' ⎕CMD ''
```

The left argument is a simple character vector containing the name of the file (`XUTILS`) containing the **XUTILS** auxiliary processor.

The right argument is relevant only in the UNIX environment and is ignored in Dyalog APL/W.

avx R←avx Y

Returns the origin zero index of a character array in `⎕AV`.

`Y` must be a simple character array. The result has the same rank and shape as `Y` but with each character replaced by its zero origin index in `⎕AV`.

Example:

```
      ⎕←A←3 4ρ'ABCDEFabcdef'
ABCD
EFab
cdef

      avx A
65 66 67 68
69 70 17 18
19 20 21 22
```

`box` `R←{X}box Y`

Converts matrices to vectors and vice versa.

`Y` must be a simple matrix, vector or scalar. `X`, if present, must be a simple scalar or 1 or 2 element vector, of the same type as `Y`.

`X` defines the delimiter and fill element in `Y`. If it is a scalar or 1 element vector it specifies the delimiter. If it is a 2 element vector the first element specifies the delimiter and the second specifies the fill element. The default value for the delimiter and fill element is the prototype of `Y`.

If `Y` is a vector it is taken to be a number of subvectors separated by the delimiter. The result in this case will be a matrix with one row more than the number of delimiters in `Y`, and with as many columns as `Y`'s longest subvector. The rows are left justified and padded with fill elements as necessary. If `Y` is a matrix it is taken to contain one subvector per row.

The result is a vector. Each row, from the left to the last non-fill element is moved to the result, with all but the last row being terminated by a delimiter.

Examples:

```
box 1 2 3 0 1 2 3 4 0 1 2 0 1
1 2 3 0
1 2 3 4
1 2 0 0
1 0 0 0

      ','box'Curtin Adam,Brand Pauline,Scholes John'
Curtin Adam
Brand Pauline
Scholes John

      +A←box 'TEXT   FOR  DESPACING'
TEXT


FOR

DESPACING

      box A
TEXT FOR DESPACING
```

dbr R←dbr Y

Delimited blank removal. Removes leading and trailing and excess spaces from a character vector.

`Y` must be a simple character vector or scalar. The result is the same as `Y` but with all leading and trailing spaces removed, and with multiple blanks replaced by single blanks.

Example:

```
      dbr '    The    cat sat on       the mat        '
The cat sat on the mat
```

hex R←hex Y

Returns the internal hexadecimal representation of an array.

`Y` is an array. The result is a character vector containing the internal hexadecimal representation of the array. If `Y` is nested, the result is a linelist (character vector delimited by <NEWLINE> characters).

Examples:

```
      hex 1 2 3
00000004 F1200000 00000003 010203FF

      hex 'ABC' (2 3ρι6)
00000005 71600000 00000002 0080184C 00801860
00000004 F1000000 00000003 414243FF
00000006 F2200000 00000002 00000003 01020304 0506FFFF
```

ltom R←{X}ltom Y

Converts a character linelist to a character matrix.

`Y` must be a simple character array. `X`, if present, must be a character scalar or 1 element vector. `X` specifies the delimiter by which `Y` is to be split. The default is <NEWLINE>.

If any dimension of `Y` is 0, the result is the array `0 0ρ''`. Otherwise `Y` is first ravelled; then if the last character is not equal to `X` one is appended; and the result is a character matrix formed by splitting `Y` at each occurrence of the delimiter `X`. The number of rows in the result is equal to the number of delimiters.

Examples:

```
      NAMES←'PETER',⎕TC[3],'JOE'
      NAMES
PETER
JOE
      ltom NAMES
PETER
JOE

      ρ⎕←',' ltom 'PETER,JOE,HARRY,MARY'
PETER
JOE
HARRY
MARY
4 5
```

ltov R←{X}ltov Y

Converts a character linelist to a vector of character vectors.

`Y` must be a simple character array. `X`, if present, must be a character scalar or 1 element vector. `X` specifies the delimiter by which `Y` is to be split. The default is <NEWLINE>.

If any dimension of `Y` is 0, the result is the array `0ρ⊂''`. Otherwise, `Y` is first ravelled; then if the last character is not equal to `X` one is appended; and the result is a vector of character vectors formed by splitting `Y` at each occurrence of the delimiter `X`. The shape of the result is equal to the number of delimiters.

Examples:

```
      NAMES←'PETER',⎕TC[3],'JOE'
      NAMES
PETER
JOE
      ltov NAMES
 PETER  JOE

      ⍴⎕←',' ltov 'PETER,JOE,HARRY,MARY'
 PETER  JOE  HARRY  MARY
4
```

mtol R←{X}mtol Y

Converts a character array into character linelist. *Y* must be a simple character array. *X*, if present, must be a character scalar or 1 element vector. *X* specifies the delimiter with which each row of the last dimension of *Y* is to be separated from its neighbours. The default is <NEWLINE>.

The result is formed by taking each row of the last dimension of *Y*, removing trailing blanks, appending a delimiter, and catenating them together into a simple character vector.

Example:

```
      ⎕←A←4 6⍴'ABC   DEFG  HI    JKLMNO'
ABC
DEFG
HI
JKLMNO

      ⍴⎕←mtol A
ABC

HI
JKLMNO
19

      ⍴⎕←','mtol A
ABC,DEFG,HI,JKLMNO,
19
```

ss R←{X}ss Y

`ss` is a string search and replacement function which matches `⎕AV` strings using a type of matching known as "regular expressions". A "regular expression" provides a method of matching strings directly and by using patterns (see below).

`Y` must be a 2 or 3 element nested vector of character vectors. `X`, if present, must be a simple boolean scalar or 1 or 2 element vector.

`X` defines the mode of search. If it is a scalar or one element vector it specifies whether case is significant. If it is a two element vector the first element specifies whether case is significant and the second specifies the type of result. These options are summarised in the table below:

	0 (default)	1
`x[1]`	Case is significant in searches	Case is not significant in searches
`x[2]`	Result is an integer vector of the 1-origin indices of the start of each occurrence of the search string	Result is a boolean vector with 1's indicating the start of each occurrence of the search string

If `Y` is a 2 element vector, a string search is performed. The result is the instances of the regular expression `Y[2]` in `Y[1]`, represented as specified in `X`.

If `Y` is a 3 element vector, a string replacement is performed. The result is `Y[1]`, with all matches of the regular expression `Y[2]` replaced by `Y[3]`.

Regular Expressions:

The simplest regular expression is any character that is not a special character, which matches only itself:

```
      ss 'This is a string' 'i'
3 6 14
```

More complex regular expressions can be formed by concatenating several expressions:

```
      ss 'This is a string' 'is'
3 6
```

The following characters have special meaning:

```
∧ $ . * ( ) [ ] \
```

and are used to form more complicated patterns. The following is a summary of the special characters:

`.`	Matches any single character
`∧`	Forces the regular expression following it to match the string that starts the vector. Note that this is the APL "and" symbol, not the ASCII "caret".
`$`	Forces the regular expression preceding the $ to match the string that ends the vector.
`[abc]`	This matches the set of all characters found between the "`[`" and "`]`". If there is a "`-`" in the expression (as in `[a-z]`), then the entire range of characters is matched. If the first character after the "`[`" is "`∧`", then any character BUT those in the set are matched. Other than this, regular expression characters lose their special significance inside square brackets. To match a "`∧`" in this case, it must appear anywhere but as the first character of the class. To match a"`]`", it should appear as the first character of the class. A "`-`" character loses its special significance if it is the first or last character of the class.
`\`	Used to escape the meaning of any following special character. For example, "`\$`" is needed to match a literal "`$`", and "`\n`" is used to match carriage return.
`(re)`	Parenthesis are used to group regular expressions.
`re*`	Match any number (including zero) of occurrences of the regular expression.

Errors:

In addition to the standard APL error messages, `ss` also returns error codes for an ill-formed regular expression, as follows:

ERROR 240 : Bad number.

ERROR 241 : \digit out of range.

ERROR 242 : Illegal or missing delimiter

ERROR 243 : No remembered search string

ERROR 244 : \(\) imbalance.

ERROR 245 : More than 2 numbers given between \{ \}.

ERROR 246 : } expected after \.

ERROR 247 : First number exceeds second in \{\}

ERROR 248 : [ ] imbalance.

Examples:

```
      A←'HERE IS A VECTOR to be searched'

⍝ Find start position of VECTOR

      ss A 'VECTOR'
11

⍝ Replace VECTOR with STRING

      ,A←ss A 'VECTOR' 'STRING'
HERE IS A STRING to be searched

⍝ Match E or e, followed by space

      ss A '[Ee] '
4 22

⍝ Find all occurrences the characters AEIOU, not
⍝ followed by a character from the same set ...

⍝ ... Case sensitive, return positions of matches

      ss A'[AEIOU][^AEIOU]'
2 4 6 9 14

⍝ ... Not case sensitive, return positions of matches

      1 ss A'[AEIOU][^AEIOU]'
2 4 6 9 14 19 22 26 30

⍝ Extract all characters from the set AEIOU,
⍝ disregarding case

      (1 1 ss A'[AEIOU]')/A
EEIAIoeeae

⍝ Replace all characters AEIOU with null

      ss A'[AEIOU]' ''
HR S  STRNG to be searched
```

vtol R←{X}vtol Y

Converts an array of character vectors into a character linelist.

Y must be an array of character vectors. *X*, if present, must be a character scalar or 1 element vector. *X* specifies the delimiter with which each element of *Y* is to be separated from its neighbours. The default is <NEWLINE>.

The result is a simple character vector formed by appending the delimiter to each element of *Y*, and catenating them to form a vector.

Example:

```
      A←2 2ρ'ABC' 'DEFG' 'HI' 'JKLMNO'
      A
 ABC  DEFG
 HI   JKLMNO

      ρ⎕←',' vtol A
ABC,DEFG,HI,JKLMNO,
19
```

DOSUTILS Workspace & DLL

Introduction

The **DOSUTILS** workspace contains a set of APL cover functions that perform a selection of commonly used DOS tasks **without** using ⎕CMD. These tasks are actually performed by a companion Dynamic Link Library DOS_U32.DLL which is written in C. The DLL is installed in your Windows directory by the Dyalog APL/W installation program **setup**. The APL functions access the routines in the DLL using ⎕NA.

The DOSUTILS functions are compatible with all previous versions of Dyalog APL, including those for Windows 3.x and can therefore be used in applications that must support both environments. An additional workspace called NTUTILS provides similar but extended facilities by making direct calls on the WIN32 API via ⎕NA. However NTUTILS is not compatible with Windows 3.x versions of Dyalog APL.

In all of the DOSUTILS functions, the names by which the DLL routines are referenced are **localised** and the routines are therefore reloaded each time the functions are used. If you wish to perform a task repeatedly, you may wish to modify the cover functions to avoid this small overhead.

The DOSUTILS software was conceived and written by Insight Systems ApS from whom the rights to include it with Dyalog APL/W were obtained.

The APL cover functions are as follows:

DosCopy : Copies the contents of one file into another

DosDir : Returns a list of file names

DosDirX : Returns a list of file names with extended information

DosErase : Erases a file

DosRename : Renames a file

DosCopy R←DosCopy Y

Copies the contents of one native DOS file into another.

`Y` must be a 2-element vector of character vectors containing the names of the source and destination files respectively. The source file must exist; the destination file need not exist and will be created if necessary. If the operation succeeds, the result is 0.

Example:

```
      DosCopy 'c:\DYALOG74\apl.ini' 'temp'
```

The following errors may be signalled (error number 11):

```
      source not found
      invalid target pathname
      error reading source file
      error writing target file
      unknown error: nnn
```

DosDir R←DosDir Y

Obtains a list of file names.

`Y` must be a simple character vector containing the specification of the file(s) to be listed. If the operation succeeds, the result `R` is a 14-column matrix containing file names.

Example:

```
      DosDir'C:\DYALOG74\XFLIB\*.*'
    NFILES.EXE
       PRT.EXE
     QFSCK.EXE
    WSCOPY.EXE
    XUTILS.EXE
   PREFECT.EXE
```

If the operation fails, the function signals the following error (error number 11):

```
      error in dir: nnn
```

DosDirX R←DosDirX Y

Obtains a list of file names with extended information.

`Y` must be a simple character vector containing the specification of the file(s) to be listed. If the operation succeeds, R is a 44-column matrix containing file names and following information.

Columns	**Description**
1-8	File Name
10-12	File extension
14	A=Archive Bit Set
15	H=Hidden Bit Set
16	R=Read Only
17	S=System
18	D=Directory
20-28	File size in bytes
30-37	Date (CCYYMMDD)
39-44	Time (HHMMSS)

Example:

```
      DosDirX'C:\DYALOG74\XFLIB\*.*'
       .         D         0 19940301 092758
      ..         D         0 19940301 092758
  NFILES EXE A        157768 19931111 115314
     PRT EXE A        153528 19931111 115406
   QFSCK EXE A        157456 19931111 133910
  WSCOPY EXE A        152996 19931111 115458
  XUTILS EXE A        156106 19931111 115642
 PREFECT EXE A        180144 19931111 134052
```

If the operation fails, the function signals the following error (error number 11):

```
      error in dir: nnn
```

DosErase R←DosErase Y

Erases a (native DOS)file

`Y` must be a simple character vector containing the name of the file to be erased. If the operation succeeds, the result is the file handle. If not, the function signals the following error (error number 11):

```
      erase failed: nnn
```

Example

```
      DosErase'JUNK2'
5
       DosErase'JUNK2'
erase failed: ¯1
      DosErase'JUNK2'
      ^
```

DosRename R←DosRename Y

Renames a (native DOS)file

`Y` must be a 2-element vector of character vectors containing the existing and new names of the file respectively. The file must exist; the new name must not. If the operation succeeds, the result is 0.

```
      DosRename'FILE1' 'FILE2'
0
```

If the operation fails, the function signals an error (error number 11) as follows:

```
      rename failed:nnn
```

The Display Workspace

The *DISPLAY* workspace contains a single function called *DISPLAY*. It produces a pictorial representation of an array, and is compatible with the function of the same name which is supplied with IBM's APL2. The *DISPLAY* function in the *UTILS* workspace is very similar, but employs line-drawing characters. A third form of presentation is provided by the *DISP* function which is also in the *UTILS* workspace.

As there is nothing else in the *DISPLAY* workspace (the description is stored in its *⎕LX* rather than in a variable) the function can conveniently be obtained by typing :

```
      )COPY DISPLAY
```

DISPLAY is monadic. Its result is a character matrix showing the array with a series of boxes bordering each sub-array. Characters embedded in the border indicate rank and type information. The top and left border contain symbols that indicate rank. A symbol in the lower border indicates type. The symbols are defined as follows:-

→	Vector.
↓	Matrix or higher rank array.
⊖	Empty along last axis.
⌽	Empty along other than last axis.
∊	Nested array.
~	Numeric data.
-	Character data.
+	Mixed character and numeric data.
∇	*⎕OR* object.

Example:

```
DISPLAY 'ABC' (1 4⍴1 2 3 4)
.→-----------------.
| .→--.  .→------. |
| |ABC|  ↓1 2 3 4| |
| '---'  '~------' |
'∊-----------------'
```

Example:

```
      AREAS←'West' 'Central' 'East'

      PRODUCTS←'Biscuits' 'Cakes' 'Rolls' 'Buns'

      SALES←?4 3ρ100 ◊ SALES[3;2]←⊂'No Sales'

      DISPLAY ' ' PRODUCTS⍪.,AREAS SALES
.--------------------------------------------.
| .→---------------------------------------. |
| ↓             .→---.  .→------.   .→---. | |
| |             |West|  |Central|   |East| | |
| | -           '----'  '-------'   '----' | |
| | .→-------.                             | |
| | |Biscuits|  14      76          46     | |
| | '--------'                             | |
| | .→----.                                | |
| | |Cakes|     54      22          5      | |
| | '-----'                                | |
| | .→----.             .→-------.         | |
| | |Rolls|     68      |No Sales|  94     | |
| | '-----'             '--------'         | |
| | .→---.                                 | |
| | |Buns|      39      52          84     | |
| | '----'                                 | |
| '∊---------------------------------------' |
'∊-------------------------------------------'
```

Example:

```
      ⎕SM←↑('PAULINE' 10 10)(21 10 20)('FARNHAM' 10 25)

      DISPLAY ⎕SM
.→----------------.
↓ .→------.       |
| |PAULINE| 10 10 |
| '-------'       |
|                 |
| 21        10 20 |
|                 |
| .→------.       |
| |FARNHAM| 10 25 |
| '-------'       |
'∊----------------'
```

UTIL Workspace (APL Utility Functions)

The *UTIL* workspace contains the APL utility functions which are briefly described below. For further details `)LOAD UTIL` and type `SHOW HELP`.

APLVERSION	Identifies the version of Dyalog APL you are running. The result is a 3-element vector of character vectors. The first element identifies the system type. The second contains the version number. The third element is either empty or is the character `'X'` (X Window System) or `'W'` (Microsoft Windows).
BMVIEW	Bitmap viewer
CENTRE	Centres text within a field.
CODES	This displays the decimal, ASCII and hex codes for every key pressed. This is very useful if you need to set up your own keyboard files.
DETRAIL	Removes trailing blanks
DISP	Puts boxes around nested arrays to show structure and depth. *DISP* produces a more compact result than *DISPLAY*.

```
      DISP 'ABC' (1 2 3)
┌───┬─────┐
│ABC│1 2 3│
└───┴─────┘
```

DISPLAY	Puts boxes around nested arrays to show structure and depth. The result is similar to that produced by the IBM APL2 *DISPLAY* function.

```
      DISPLAY 'ABC' (1 2 3)
┌→──────────────┐
│ ┌→──┐ ┌→────┐ │
│ │ABC│ │1 2 3│ │
│ └───┘ └~────┘ │
└∊──────────────┘
```

ECHO	Returns the value of a given environment variable
FNGREP	Searches the functions named on the left (or the complete workspace if the left argument is omitted) for matches of the regular expression on the right. Useful for locating in which functions certain variables are used and set.
FNREPL	Similar to FNGREP, but provides string replacement.
LJUST	Left-justifies text within a field.
MAKEMAT	Convert delimited vector to a matrix.
MATRIX	Make 1-row matrix from scalar or vector.
PROP	Display given property value for each node in a tree of GUI objects.
PROPS	Display all property values for a given GUI object.
RJUST	Right-justify text within field.
SET	Equivalent to DOS SET command, e.g.

```
      ↑SET
COMSPEC    C:\COMMAND.COM
PATH       c:\dos;c:\dyalog
PROMPT     $p$g
```

SETMON, *VIEWMON*	These functions help to analyse performance. *SETMON* sets ⎕*MONITOR* on all functions in the workspace, or on all functions used by that named in its argument. After running the system, *VIEWMON* is used to browse a report showing cpu usage.
SM_TS, *TS_SM*	Converts dates between ⎕*SM* and ⎕*TS* formats.
TREE	Displays a tree of GUI objects.
WSPACK	Conserves workspace by sharing identical arrays.

GROUPS Workspace & WSCOPY AP

The workspace `GROUPS` contains functions which emulate the system commands available in some other implementations of APL. These functions allow APL objects to be grouped together. The group can then be copied as a whole from the saved workspace, using the **WSCOPY** Auxiliary Processor. This AP may also be used in stand-alone mode if required.

The APL functions held in the WS are as follows:

`GROUP` : forms a new group or expands or disperses an existing group.

`GRPS` : lists the names of groups defined in the workspace.

`GRP` : displays the membership of a particular group.

`COPY` : copies objects including groups from a saved workspace.

The workspace contains `HOW` variables which describe their use in detail.

The external functions defined by the AP are as follows:

`wsnl` : returns a list of items of a given name class from a named saved WS.

`wsval` : copies the value of a named object from a named saved WS.

COPY `COPY WSNAME OBJ1 OBJ2 OBJ3 ...`

Copies objects `OBJ1`, `OBJ2`, `OBJ3` ... from the saved workspace `WSNAME`. The argument is a name list i.e. a simple text scalar or vector containing names separated by blanks, a text matrix with one name per row, or a vector of text vectors. If not, `DOMAIN ERROR` is reported. If a named object is a group, its members (if present) are also copied.

Errors:

a) No pendant or suspended functions (`⎕LC` is empty)

If `WSNAME` does not exist or is inaccessible, `ws not found` is reported. If any named objects are not present in the saved workspace, the message `not copied` followed by a list of missing names is printed.

b) `COPY` called from another function (`⎕LC` not empty):

If `WSNAME` does not exist or is inaccessible, `DOMAIN ERROR` is signalled.

Examples:

```
      ⍝ Copy X,Y and group GRAPHICS from MYWS
      COPY 'MYWS' 'X' 'Y' 'GRAPHICS'

      ⍝ Ditto, but Z not present.
      COPY 'MYWS X Y Z GRAPHICS'
Z not found
```

GROUP

`GROUP GNAME OBJ1 OBJ2 OBJ3 ...`

Forms the group `GNAME` containing objects `OBJ1`, `OBJ2`, `OBJ3` etc. Objects which themselves are groups are expanded into their constituent members. If no objects are specified, the group `GNAME` is dispersed. The argument is a namelist i.e. a simple text scalar or vector containing names separated by blanks, a simple text matrix with one name per row, or a vector of text vectors. If not, `DOMAIN ERROR` is reported.

Examples:

```
      ⍝ Form the group GRAPHICS
      GROUP 'GRAPHICS' 'DRAW' 'PLOT' 'TEXT'

      ⍝ Add SCALE to group GRAPHICS
      GROUP 'GRAPHICS GRAPHICS SCALE'

      ⍝ Form the group UTILITIES
      GROUP 3 9 ⍴'UTILITIESSCREEN  GRAPHICS '
```

GRP

`GRP GNAME`

Displays the membership of a particular group. `GNAME` must be a simple text scalar or vector, otherwise `DOMAIN ERROR` is reported.

Example:

```
      GRP 'GRAPHICS'
DRAW
PLOT
TEXT
SCALE
```

GRPS

`GRPS`

Lists the names of groups defined in the workspace.

Example:

```
      GRPS
GRAPHICS
UTILITIES
```

TUTOR Workspace (APL Tutorial)

Many of our users have several years programming experience using first generation APLs, but no experience of nested arrays. Others have used other second generation APLs, and need to know the differences between Dyalog APL and the APLs they are more familiar with.

TUTOR is an interactive self-teaching system which highlights the extensions introduced to the APL language by Dyalog APL.

TUTOR presents you with a list of subjects and you choose those of interest, after which you are lead through the relevant screens of information, a step at a time. Facilities exist which allow you to skip around these screens as you wish.

Contents of the Tutorial Book

1 : Nested Arrays

What is a nested array
Building nested arrays
Display of arrays
Storage requirements
Benefits and penalties

2 : Functions & Nested Arrays

Extensions to existing functions
New functions for nested arrays
Indexing

3 : Primitive Operators

Definitions
Operators and derived functions
New operators
Scope of operators

4 : Defined Operators

5 : Selective Specification

6 : Function Assignment

7 : External Variables

Why external variables
What is an external variable
Comparison with component files

8 : External Tasks

Communicating with the Operating System from APL
Auxiliary processors

WINTRO Workspace (GUI Introduction)

The `WINTRO` workspace contains a tutorial introduction to the GUI features in Dyalog APL/W. It is intended to convey the general principles of how the system works, rather than providing specific information. A more detailed set of tutorials are provided in the `WTUTOR` workspace.

The tutorial consists of an executable sequence of lessons with instructions and commentary.

WTUTOR Workspace (GUI Tutorial)

The `WTUTOR` workspace contains a set of tutorials designed to help you explore different aspects of the GUI support.

When you `)LOAD` the workspace, you will see a list of topics. Choose a topic and press *OK* or double-click the topic you require.

Each topic consists of a series of lessons which are presented in the same manner as described above for the `WINTRO` workspace.

WTUTOR95 Workspace

The `WTUTOR95` workspace contains an additional set of tutorials.

WDESIGN Workspace (GUI Design Tool)

Introduction

`WDESIGN` is an interactive tool to assist the Dyalog APL programmer in developing GUI applications. It allows you to do the following tasks :

1) Design the lay-out of your user-interface in terms of one or more Forms. A Form is a window to which you can attach pulldown Menus and various other controls including Buttons, Edit fields, Combo boxes, and so forth. Forms, Menus and controls are all referred to as Objects.

2) Define the appearance and behaviour of your OBJECTS by assigning values to PROPERTIES.

3) Attach CALLBACK FUNCTIONS to EVENTS in OBJECTS. These functions perform tasks which are appropriate to your application.

4) Edit your `MAIN` program and your callback functions as an integrated part of the `WDESIGN` process.

5) Test your application, change it, re-test it, etc. within the `WDESIGN` context.

6) Save your application as a workspace.

7) Maintain and enhance an existing application, whether or not it was originally created using the `WDESIGN` toolset.

Building a New Application

To build a new application, you start by `)LOAD`ing the `WDESIGN` workspace. This contains a single namespace called `WDesign`. Its main function `WDesign.RUN` is executed by the `⎕LX`.

Your first Form (`FORM1`) is created by default. You can create others by clicking on *New Form* in the *File* menu.

The Toolbar is used to add objects to Forms. The Object Inspector is used to assign values to properties. These determine the appearance and behaviour of your objects.

The Object Inspector is also used to associate callback functions with particular events in particular objects, and to define a `MAIN` function that runs your application.

The functions are edited using the Dyalog APL editor, which is invoked via the Object Inspector.

Once you have carried out these steps, you can test your application and then return to "design mode" for further work.

When you save the application, the `WDesign` namespace is erased from the workspace, leaving only your `MAIN` and callback functions.

The GUI objects can either be saved with the workspace, or you can instead have `WDESIGN` build APL functions which in turn will create your objects when they are run.

Maintaining an Existing Application

To change an existing application, you start by `)LOAD`ing the application workspace.

If your workspace has the GUI objects `)SAVE`d, you simply `)COPY WDESIGN`, and type `WDesign.RUN`.

If your application uses functions to build the objects, you must run these functions first before `)COPY`ing `WDESIGN` in.

`WDESIGN` captures existing objects from the workspace and presents them for editing.

Objects may be added or deleted, `MAIN` and callback functions can be edited, and the application tested and saved, in the same way as if you were creating a new application from scratch.

Other Utility Workspaces

ARACHNID Workspace

This is a card game that demonstrates various features of Dyalog APL/W including the use of the Bitmap and Image objects.

BUILDSE Workspace

This workspace was used to build the default APL session. To configure the session differently, you may edit the functions and rebuild and save the session.

BMED Workspace

This workspace contains functions for editing bitmaps and for creating picture buttons.

DCOMREG Workspace

This workspace contains functions that may be used to register an OLE Server, written in Dyalog APL, for DCOM.

EXCEL Workspace

This workspace contains some functions to explore the use of the DDE interface (shared variables) to communicate with Microsoft Excel. Instructions for using the workspace are provided in *Interface Guide*. A small spreadsheet and command macro, which are also required, are supplied in the EXCEL sub-directory in DYALOG74.

FTP Workspace

This contains a collection of file transfer utilities implemented using FTP.

GRAPHS Workspace

This contains some "Business Graphics" utility functions and a graphics demonstration.

NTUTILS Workspace

This workspace contains some file and directory utility functions, similar to those provided in DOSUTILS, but which are specific to Windows 95 and Windows NT.

OCXBROWS Workspace

This workspace allows you to browse any OLE Controls that may be installed on your system.

OPS Workspace

This workspace contains a collection of defined operators that serve both as examples and as useful utilities.

POSTSCRIPT Workspace

This workspace is provided only for compatibility with other implementations of Dyalog APL. It is otherwise irrelevant to Dyalog APL/W.

PREFECT Workspace

This contains *PDESIGN*, a panel design utility, and useful cover functions for the **prefect** Auxiliary Processor (AP124 emulation).

PREDEMO Workspace

This demonstrates some of the facilities provided by the **prefect** Auxiliary Processor (AP124 emulation).

PRT Workspace

This workspace is provided only for compatibility with other implementations of Dyalog APL. It is otherwise irrelevant to Dyalog APL/W.

QUADNA Workspace

This workspace contains examples that illustrate how to call Windows API functions using ⎕*NA*.

SQAPL and SQATEST Workspaces

This workspace contains the APL cover functions for the OBDC interface. See *Interface Guide*.

Sample Workspaces

The following workspaces are provided in the sub-directory dyalog10\samples. These workspaces are provided for illustration alone and are not intended for production use.

ActiveX Samples

DUALBASE and DUALFNS Workspaces

These workspaces illustrate how to write an ActiveX Control in Dyalog APL.

Dfns Samples

DFNS Workspace

This workspace contains some examples of dynamic functions and operators, including some useful utilities.

MIN & MAX Workspaces

These workspaces contains some more examples of dynamic functions and operators.

OLE Samples

LOAN Workspace

This workspace illustrates an OLE Server using a Loan sheet example. Visual Basic and Excel client samples are included.

CFILES Workspace

This workspace illustrates an OLE Server that allows you to read Dyalog APL component files into Excel.

OLEAUTO Workspace

This workspace illustrates how you can access OLE Servers such as Microsoft Access and Microsoft Excel.

OLEASYNC Workspace

This workspace illustrates how an OLE Server written in Dyalog APL may be called so that it executes in parallel (asynchronously), possibly on a different computer.

SHORTCUT Workspace

This workspace illustrates how you may call OLE objects via non-standard interfaces. This example creates a shortcut on your desktop.

TCPIP Samples

CHAT Workspace

This workspace illustrates how the TCP/IP interface can be used to *chat* between two or more APL workspaces.

QFILES Workspace

This workspace illustrates how the TCP/IP interface may be used to implement a client/server component file system.

REXEC Workspace

This workspace illustrates how the TCP/IP interface may be used to implement client/server (remote execution) operations.

WWW Workspace

This workspace contains basic functions to illustrate the principles of internet access using Dyalog APL.

Third-Party Workspaces

A collection of third-party workspaces is provided in the `OUTPRODS\TOOLS` subdirectory. Although these are supplied on an as-is and unsupported basis, they are generally robust and useful. Dyadic is grateful to the respective authors for permission to include these workspaces with Dyalog APL/W.

CHAPTER 7

Workspace Transfer

Introduction

It is often necessary to either transfer Dyalog APL workspaces from one machine to another, or to transfer workspaces created by another version of APL to Dyalog APL.

Since the internal structure of an APL workspace is dependent upon the architecture of the machine on which it was created, and the version of APL that was used to create it, it is not possible to transfer a workspace directly. It must first be transformed into a format that is common to both source and target environments. This can then be transferred, and used to create a workspace on the target machine.

The following sections describe the steps that are involved in moving workspaces between machines and different versions of APL. APL component files are transferred in a similar way to workspaces; each component is read into an APL variable which is then transferred.

The subsequent sections describe specific examples of this process. The basic technique is the same in most cases. Doubtless better techniques can be used, but the following examples are useful as a basis from which to work.

The problems of conversion from one version of APL to Dyalog APL are not discussed in this section. Nor are the procedures for transferring from Dyalog APL to other APLs. However, using the techniques discussed in this section, it is a simple (if tedious) matter to produce software that will transfer in the opposite sections.

Where possible, the tools required to transfer to Dyalog APL are supplied with the product; where this is impractical, listings of the appropriate software is given.

General Techniques

Create a Script File

If we can produce a text file on the source machine that contains all of the statements necessary to recreate the workspace, we could execute each line of this file on the target machine using the Dyalog APL, thereby recreating the workspace in the correct format.

How do we transform a workspace into a text format? Functions are easy; we can just produce a listing of them. Variables are more difficult; we must produce lines of text that when executed, recreate the variables with the correct contents, type, shape and depth.

Any APL programmer would be capable of writing the simple APL system required to perform both of these tasks; however, a workspace called `DWSOUT` (Dyalog WorkSpace OUT) is supplied for certain versions of APL (Dyalog APL, APL*PLUS), and where the workspace cannot be supplied (VSAPL), appropriate listings are given.

Transfer to Target Machine

Once this file has been created, it should be transferred from the source to the target machine. This transfer may be done using diskettes or tapes (first ensure that the source media can be read by the target machine), or the file may be sent across some kind of network using file transfer software.

However the transfer is effected, the file should arrive on the target machine EXACTLY as it left the source machine; i.e. the transfer mechanism should not perform any conversion or translation. Note that the file produced by `DWSOUT` is a raw text file, and may contain any characters in the range 0 to 255; some network transfer packages react badly to characters in the range 0 to 32 unless you ask that the file is transferred in transparent mode.

Create Dyalog APL Workspace

Each line of the file must read and executed by Dyalog APL. Again, it would be a simple task for any APL programmer to write the required functions; however, a workspace called `DWSIN` (Dyalog WorkSpace IN) is supplied which can process files produced by Dyalog APL, APL*PLUS and VSAPL. It could easily be amended to cope with files produced by other APLs.

Exception

The exception to the above technique is the transfer of IBM APL2 workspaces. APL2 provides the user with a system command `)OUT` which produces a text file representing the workspace. This file can be transferred to the target machine, but must then be decoded in a special manner. See the relevant section for details.

DWSOUT Workspace

Introduction

Since different versions of APL have different mechanisms for creating and appending to native files, each version of APL requires a different version of the workspace *DWSOUT*. Versions currently exist for transferring from Dyalog APL, APL*PLUS and VSAPL. However, it would be a fairly simple matter for any of these workspaces to be amended to suit any version of APL.

Each workspace *DWSOUT* contains a function *DWSOUT* that is effectively a very simple workspace lister, which lists the contents of variables as well as listing functions. Where appropriate, a function *DCFOUT* is also supplied; this lists the contents of component files, by reading each component into a variable, and listing the variable using the same techniques as *DWSOUT*.

Each version of *DWSOUT* is essentially the same. Hence, the description of the use of *DWSOUT* given below is valid for all versions. The particular restrictions of each version is discussed in the relevant sections.

Transferring Workspaces

The function *DWSOUT* is used to transfer entire workspaces, or named objects from a workspace. The full syntax of the function call is:

WS Out *{names} ⍙⍙DWSOUT wsname*

wsname is a simple character vector containing the name under which the workspace is to be *)SAVE*d on the target system. *names*, if present, is a character matrix containing the names of the objects to be transferred. The default is the entire workspace.

A file called **wsname.DXF** is created, and listings of the requested objects, system variables and constants, and some control statements are appended to the file.

Restrictions

The following restrictions apply to all versions of `DWSOUT`:

- The state indicator must be empty.
- The names of objects to be transferred may not begin with `⍙⍙`.
- Locked functions cannot be transferred.

Example

Create a text file called NEWWS.DXF containing a complete listing of a workspace called `MYWS`.

```
      )LOAD MYWS
saved ....

      )FNS
FOO GOO HOO

      )VARS
A B C

      )COPY DWSOUT
saved ...

      ⍙⍙DWSOUT 'NEWWS'
Functions ...
FOO, GOO, HOO,
Variables ...
A, B, C,
System Variables ...
Finished
```

Transferring Component Files

Where applicable, the function `DCFOUT` is used to transfer entire component files. The full syntax of the function call is:

File Out `{options} ⍙⍙DCFOUT filename`

`filename` is a simple character vector containing the name of the file to be created on the target system.

`options`, if present, is either a simple text vector containing the name of the file to be transferred, or a two element vector whose first element contains the name of the file, and whose second contains a passnumber. If `options` is omitted, the file name is taken from `filename` with a passnumber of 0.

A file called `filename.DXF` is created. Each component in the file is read and assigned to a variable; this variable is then transferred using the same techniques as `DWSOUT`. The access matrix is read and transferred in the same manner.

Example

Create a text file called `PJB.DXF` containing a complete listing of a component file called PJB.

```
      )LOAD DWSOUT
saved ...

      ⍙⍙DCFOUT 'PJB'

Components ...
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Finished
```

DWSIN Workspace

The workspace DWSIN contains a function DWSIN that can process script files produced by the DWSOUT workspaces appropriate for several different APLs.

DWSIN is used to read the text files produced by DWSOUT and DCFOUT. Each line of the file is read and executed, thus recreating the workspace or component file. The full syntax of the function call is:

WS In {*source*} ∆∆*DWSIN filename*

filename is a simple character vector containing the name of the file to be processed. This file must have been produced by a previous call of *DWSOUT*. Note that the suffix "*.DXF*" is automatically appended to the given file name.

source, if present, is a simple text vector containing the name of the source APL, taken from the set *'DYALOG'* , *'APLPLUS'*, *'STSCMF'* or *'VSAPL'*. If *source* is omitted, the default is *'DYALOG'*.

DWSIN reads the file, applying the relevant translation from the source APL to Dyalog APL, and executes each line, thus recreating the objects.

If an object cannot be recreated because of badly formed lines in the file, then that object is ignored and a warning message printed. The names of such objects are held in a variable *WontFix*.

Example

Create a Dyalog APL workspace from the file `NEWWS.DXF`, which was created by Dyalog APL on another machine.

```
      )LOAD DWSIN
saved ...

      ⍙⍙DWSIN 'NEWWS'
Processing script ...
Functions & Operators ...
FOO, GOO, HOO,
Variables ...
System variables ...
*****************************
**** Workspace name is NEWWS
**** REMEMBER TO )SAVE IT !!!
*****************************
```

Example

Create a Dyalog APL component file from the file `PJB.DXF`, which was created by APL*PLUS/PC.

```
      )LOAD DWSIN
saved ...

      'APLPLUS' ⍙⍙DWSIN 'PJB'
Processing script ...
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Finished.
```

Transferring Dyalog APL Workspaces

Dyalog APL workspaces and component files are binary compatible across machines with similar architectures. For example, workspaces created under Dyalog APL for DOS can be used immediately by Dyalog APL under Xenix, as long as both environments are running the same version of Dyalog APL. However, if you are not sure if the source and target machines are compatible, it is best to transfer workspaces using `DWSOUT` and `DWSIN`.

The `DWSOUT` workspace for Dyalog APL copes with nested arrays, defined functions and operators, assigned functions, `⎕OR` objects and component files. However, the following restrictions apply:

- Arrays containing the `⎕OR` of an assigned function cannot be transferred.
- Locked functions cannot be transferred. However, a matrix of the names of the locked functions is created and this transferred. This variable is called `LockedFns`.
- Functions such as `SUM←+/` cannot be transferred. A matrix of the names of such functions, together with a *best estimate* of their contents, is created in the target workspace. This variable is called `AssignedFns`.
- Any workspace name supplied to `DWSOUT` or file name supplied to `DCFOUT` must not exceed 10 characters.

Importing APL*PLUS Workspaces

The `DWSOUT` workspaces for both APL*PLUS/PC and APL*PLUS II are available on request from your Dyalog APL distributor.

The following restrictions apply to their use:

- Functions containing references in their header line to APL*PLUS specific system variables cannot be fixed by Dyalog APL. It is recommended that such functions are edited on the source machine before transfer. The following system variables cause a problem if they occur in a header line:

  ```
  ⎕ALX, ⎕CURSOR, ⎕ELX, ⎕HNAMES, ⎕HTOPICS,
  ⎕KEYW, ⎕SA, ⎕SEG, ⎕WINDOW
  ```

- No attempt is made to convert APL*PLUS specific code to the Dyalog APL equivalent.

- Any workspace name supplied to `DWSOUT` or file name supplied to `DCFOUT` must not exceed 8 characters.

- All graphic characters are transferred, but to their single line equivalents.

- Any character not in Dyalog APL's character set is translated as `⎕AV[⎕IO+127]`.

- `⎕AV[⎕IO+255]` (alternative space) is translated to `⎕AV[⎕IO+32]` (space) before transfer.

- Component files are assumed to start at component one.

Importing IBM VS APL Workspaces

A listing of an appropriate *DWSOUT* is supplied. You must amend the functions *⍙⍙FILE_CREATE*, *⍙⍙FILE_WRITE* and *⍙⍙FILE_CLOSE* to suit your particular operating system.

You need only use the leading characters ⍙⍙ for all function and variable names if you wish to avoid any name clashes.

This version of *DWSOUT* is subject to the following restrictions (which you can code around if you so wish):

- No error checking is performed.
- No system variables are transferred.
- All objects in the workspace are transferred; there is no option for transferring a subset.
- Functions are transferred in *⎕CR* form, and trailing blanks are not removed.

```
Workspace listing for : ./DWSOUT            ⍙⍙DOUBLE
______________________________________________________

⍙⍙DOUBLE
========

     ∇ ⍙⍙A←⍙⍙B ⍙⍙DOUBLE ⍙⍙C;⍙⍙D;⍙⍙E;⍙⍙F;⎕IO
[1]    ⍝ REPLACE CHAR ⍙⍙B WITH (2⍴⍙⍙B) IN STRING ⍙⍙C
[2]     ⎕IO←0
[3]     ⍙⍙F←(⍙⍙B=⍙⍙C)/⍳⍙⍙D←⍴⍙⍙C←,⍙⍙C
[4]     ⍙⍙E←(⍙⍙D+⍴⍙⍙F)⍴1
[5]     ⍙⍙E[⍙⍙F←⍙⍙F+⍳⍴⍙⍙F]←0
[6]     ⍙⍙A←⍙⍙E\⍙⍙C
[7]     ⍙⍙A[⍙⍙F]←⍙⍙B
     ∇


⍙⍙DWSOUT
========

     ∇ ⍙⍙DWSOUT ⍙⍙FILENAME;⎕IO;⎕PP;⎕PW;⍙⍙EOL;⍙⍙NEWNAME
[1]    ⍝ PRODUCES A SCRIPT FROM AN IBM VSAPL WORKSPACE
[2]     ⎕IO←1
[3]     ⎕PP←16
[4]     ⎕PW←255
[5]    ⍝ DATA FILE NAME (MUST END IN .DXF ON TARGET MA
        CHINE)
[6]     ⍙⍙NEWNAME←(⍙⍙FILENAME≠' ')/⍙⍙FILENAME
[7]     ⍙⍙FILENAME←⍙⍙NEWNAME,'.DXF'
[8]    ⍝ SET END OF LINE
[9]     ⍙⍙EOL←⎕AV[256]
[10]   ⍝ OPEN DATA SET
[11]    ⍙⍙FILE_CREATE
[12]   ⍝ SEND CONTROL STATEMENTS
[13]    ⍙⍙OUT'⎕WSID←''',⍙⍙NEWNAME,''''
[14]   ⍝ TRANSFER FUNCTIONS
[15]    ⍙⍙OUTFNS
[16]   ⍝ TRANSFER VARIABLES
[17]    ⍙⍙OUTVARS
[18]   ⍝ AND FINISH
[19]    ⍙⍙FILE_CLOSE
[20]    ⍙⍙TOSCREEN ⎕TC[2],'FINISHED'
     ∇
```

```
____________________________________________________________________
27th August 1994                                          Page :   1
```

```
Workspace listing for : ./DWSOUT          ⍙⍙FILE_CLOSE
_____________________________________________________

⍙⍙FILE_CLOSE
============

     ∇ ⍙⍙FILE_CLOSE
[1]    ⍝ CLOSE DATA FILE ⍙⍙FILENAME - SYSTEM DEPENDENT
     ∇


⍙⍙FILE_CREATE
=============

     ∇ ⍙⍙FILE_CREATE
[1]    ⍝ CREATE FILE ⍙⍙FILENAME - SYSTEM DEPENDENT
     ∇


⍙⍙FILE_WRITE
============

     ∇ ⍙⍙FILE_WRITE ⍙⍙TEXT
[1]    ⍝ WRITE ⍙⍙TEXT TO FILE ⍙⍙FILENAME - SYSTEM DEPE
         NDENT
     ∇


⍙⍙OUT
=====

     ∇ ⍙⍙OUT ⍙⍙LINE;⍙⍙N;⍙⍙R
[1]    ⍝ PASS LINE OF TEXT TO FILE, FINISHING WITH AN
         END OF LINE CHAR
[2]     ⍙⍙FILE_WRITE ⍙⍙LINE,⍙⍙EOL
     ∇
```

```
________________________________________________________________
27th August 1994                                      Page :   2
```

```
Workspace listing for : ./DWSOUT             ⍙⍙OUTFN

⍙⍙OUTFN
=======

      ∇ ⍙⍙OUTFN ⍙⍙NAME;⍙⍙
[1]     ⍝ OUTPUT FUNCTION AS A VARIABLE, THEN FIX
[2]      →⍙⍙GO×⍳0≠1↑⍴⍙⍙←⎕CR ⍙⍙NAME
[3]      ⍙⍙TOSCREEN' (LOCKED)'
[4]      →0
[5]     ⍙⍙GO:⍙⍙OUTVAR'⍙⍙'
[6]      ⍙⍙OUT'⎕FX ⍙⍙'
      ∇


⍙⍙OUTFNS
========

      ∇ ⍙⍙OUTFNS;⍙⍙FNS;⍙⍙I
[1]     ⍝ OUTPUT ALL NON-XFER FUNCTIONS
[2]      ⍙⍙TOSCREEN ⎕TC[2],'FUNCTIONS '
[3]      ⍙⍙FNS←(⍙⍙FNS[;1 2]∨.≠'⍙⍙')⌿⍙⍙FNS←⎕NL 3
[4]      ⍙⍙I←0
[5]     ⍙⍙NEXTI:→((1↑⍴⍙⍙FNS)<⍙⍙I←⍙⍙I+1)/⍙⍙END
[6]      ⍙⍙TOSCREEN'... ',⍙⍙FNS[⍙⍙I;]
[7]      ⍙⍙OUT'⍝',⍙⍙FNS[⍙⍙I;]
[8]      ⍙⍙OUTFN ⍙⍙FNS[⍙⍙I;]
[9]      →⍙⍙NEXTI
[10]    ⍙⍙END:
      ∇
```

```
____________________________________________________________
27th August 1994                                Page :    3
```

```
Workspace listing for : ./DWSOUT            ⍙⍙OUTVAR

⍙⍙OUTVAR
========

     ∇ ⍙⍙OUTVAR ⍙⍙NAME;⍙⍙SHAPE;⍙⍙VALUE;⍙⍙N;⍙⍙M;⍙⍙X;⍙⍙
        Q;⍙⍙I
[1]     ⍝ SPLITS VARIABLES INTO MANAGEABLE CHUNKS
[2]      ⍙⍙VALUE←⍎⍙⍙NAME
[3]      ⍙⍙SHAPE←⍴⍙⍙VALUE
[4]      ⍙⍙VALUE←,⍙⍙VALUE
[5]     ⍝ TREAT NUMERIC AND TEST DIFFERENTLY
[6]      →(⍙⍙NUM,⍙⍙TXT)[1+0≠1↑0⍴⍙⍙VALUE]
[7]     ⍝ NUMERIC
[8]     ⍙⍙NUM:⍙⍙OUT'⍙⍙←',(⍕×/⍙⍙SHAPE),'⍴0'
[9]      ⍙⍙N←10
[10]     ⍙⍙I←0
[11]    ⍙⍙L1:→(0=⍴⍙⍙VALUE)/⍙⍙EXIT
[12]     ⍙⍙OUT'⍙⍙[',(⍕⍙⍙I),'+⍳',(⍕⍙⍙M),']←',⍕(⍙⍙M←⍙⍙N⌊
        ⍴⍙⍙VALUE)↑⍙⍙VALUE
[13]     ⍙⍙VALUE←⍙⍙M↓⍙⍙VALUE
[14]     ⍙⍙I←⍙⍙I+⍙⍙M
[15]     →⍙⍙L1
[16]    ⍝ TEXT
[17]    ⍙⍙TXT:⍙⍙OUT'⍙⍙←',(⍕×/⍙⍙SHAPE),'⍴'' '''
[18]     ⍙⍙Q←''''
[19]     ⍙⍙N←60
[20]     ⍙⍙I←0
[21]    ⍙⍙L2:→(0=⍴⍙⍙VALUE)/⍙⍙EXIT
[22]     ⍙⍙X←(⍙⍙M←(⍴⍙⍙VALUE)⌊⍙⍙N-+/⍙⍙Q=⍙⍙N↑⍙⍙VALUE)↑⍙⍙
        VALUE
[23]     ⍙⍙OUT'⍙⍙[',(⍕⍙⍙I),'+⍳',(⍕⍙⍙M),']←''',(⍙⍙Q ⍙⍙DO
        UBLE ⍙⍙X),⍙⍙Q
[24]     ⍙⍙VALUE←⍙⍙M↓⍙⍙VALUE
[25]     ⍙⍙I←⍙⍙I+⍙⍙M
[26]     →⍙⍙L2
[27]    ⍝ ISSUE CODE TO RESHAPE VARIABLE AND ASSIGN TO
        VARIABLE NAME
[28]    ⍙⍙EXIT:⍙⍙OUT'⍙⍙←',((4×0≠⍴⍙⍙SHAPE)↓'(⍳0)',⍕⍙⍙SHA
        PE),'⍴⍙⍙'
[29]     ⍙⍙OUT ⍙⍙NAME,'←⍙⍙'
     ∇
```

```
27th August 1994                                    Page :   4
```

```
Workspace listing for : ./DWSOUT          ⍙⍙OUTVARS
____________________________________________________

⍙⍙OUTVARS
=========

     ∇ ⍙⍙OUTVARS;⍙⍙VARS;⍙⍙I
[1]    ⍝ PRODUCES SCRIPT TO RECONSTITUTE VARIABLES
[2]     ⍙⍙TOSCREEN ⎕TC[2],'⍙⍙VARIABLES '
[3]     ⍙⍙VARS←(⍙⍙VARS[;⍳2]∨.≠'⍙⍙')⌿⍙⍙VARS←⎕NL 2
[4]     ⍙⍙I←0
[5]    ⍙⍙L:→((1↑⍴⍙⍙VARS)<⍙⍙I←⍙⍙I+1)/⍙⍙END
[6]     ⍙⍙TOSCREEN'... ',⍙⍙VARS[⍙⍙I;]
[7]     ⍙⍙OUT'⍝',⍙⍙VARS[⍙⍙I;]
[8]     ⍙⍙OUTVAR ⍙⍙VARS[⍙⍙I;]
[9]     →⍙⍙L
[10]   ⍙⍙END:
     ∇


⍙⍙TOSCREEN
==========

     ∇ ⍙⍙TOSCREEN ⍙⍙MSG
[1]    ⍝ DISPLAY MESSAGE AT TERMINAL
[2]     ⍙←⍙⍙MSG
     ∇
```

```
27th August 1994                             Page :   5
```

Importing IBM APL2 Workspaces

IBM APL2 has a system function `)OUT` which produces a text file that represents the workspace. This file must be transferred and decoded line by line on the target machine. This decoding is performed by the supplied workspaces `APL2IN` and `APL2PCIN`.

`APL2IN` is suitable only for importing workspaces from mainframe APL2; `APL2PCIN` performs the same task for workspaces exported from APL2/PC.

Create a script file

Load your workspace, and use `)OUT` to create a file.

Transfer to target machine

Move file to target machine; no translation should be performed.

Create Dyalog APL workspace

Use the workspace `APL2IN` or `APL2PCIN` to process the script file. This workspace contains a function called `⍙⍙APL2IN`, which takes as its right argument the name of the file that has been transferred from APL2. Once the file has been processed, you must save the active workspace.

Example

a) On mainframe:

```
      )LOAD MYWS
saved ...

      )OUT
```

b) Transfer file to target machine, with NO translation

c) On target machine:

```
      )LOAD APL2IN
saved ...

      ⍙⍙APL2IN 'filename'

      )SAVE
```

CHAPTER 8

Character-Mode Screen Manager

1. Introduction

The Dyalog APL Screen Manager was developed for character-mode PCs and terminals. It has since been implemented and extended for true "window" systems including the X Window System and Microsoft Windows.

As a character-mode interface, ⎕SM has largely been superceded by the GUI support that is described in *Interface Guide*. Nevertheless, ⎕SM continues to be supported under MS-Windows, so that existing Dyalog APL character mode applications will run unchanged. Furthermore, by making use of the SM object which is part of the GUI support, the developer has a means to extend existing applications to add GUI features, and need not rewrite them from scratch.

In this chapter, certain terms, in particular **screen** and **window** may cause confusion, because in Dyalog APL/W, ⎕SM is allocated its own separate Window on the screen, or is displayed in a GUI object.

To avoid confusion between true "Windows" and ⎕SM's own "windows" which are internal to a form, the former are described in this chapter using a capital "W".

Furthermore, the word **screen** should be interpreted as referring to to the ⎕SM Window and not the entire display.

1.1 Overview

⎕SM is a system variable that defines a **form** to be displayed on your screen. In general when you assign a value to ⎕SM or change it in any way, you will see a change to your display. ⎕SR is a system function that allows the user to enter data into a form, or simply move about it using the cursor and other movement keys. Any changes thus made by the user are reflected in new values in ⎕SM.

⎕SM is a nested matrix whose rows represent **fields**, and whose columns represent **attributes**. Fields have contents (character data, numbers, and dates) which are formatted according to field type and displayed in a window. If the contents of the field are larger than the window, the data may be scrolled both vertically and horizontally by the user. Furthermore, related fields may be grouped into vertical and horizontal scrolling groups, so that when the data in one field is scrolled, the data in related fields scrolls with it. Fields may also possess attributes which define various different behaviours as well as video attributes to define colour, intensity, and so on.

Together ⎕SM and ⎕SR provide a powerful mechanism for developing user interfaces for your applications.

1.2 Window Management

In Dyalog APL/W, ⎕SM is (by default) displayed in a separate Window, which provides the interface between an application and the user. If there is an SM object defined as part of the GUI, ⎕SM uses its Window instead. The SM object permits ⎕SM to be used as part of a Form, and integrated with other objects such as menus, scrollbars, buttons and so forth.

Unless there is an SM object, the ⎕SM Window is displayed ONLY when a non-empty value is assigned to ⎕SM. If ⎕SM is empty (the default value is `0 13⍴0`) its Window does not appear.

A reference to ⎕SR causes the cursor to move from the APL Session Window to the ⎕SM Window. Note that it is necessary however for the user to **focus** on the ⎕SM Window before the cursor is actually displayed and the Window will accept input. During the execution of ⎕SR only the ⎕SM Window is active and no other APL Window will respond to input.

If after exiting from ⎕SR, the system returns to "desk-calculator" mode, the cursor will automatically revert to the Session window, although again the user must focus on it before input will be accepted.

Like any other system variable, `⎕SM` can be localised, so you can have several different values of `⎕SM` at different levels in the APL stack. Each localised definition of `⎕SM` **overlays** rather than replaces the forms defined by other values of `⎕SM` further down the APL stack. Naturally the form defined by a localised `⎕SM` disappears when the function in which it is localised terminates. The `⎕SM` Window reverts to its previous state before that function was called. Thus localising `⎕SM` is a simple and natural way to produce pop-up menus, pop-up help, etc.

1.3 General Principles - ⎕SM

`⎕SM` is a nested matrix which defines a **form**. Its rows represent fields, and its columns attributes. `⎕SM` may contain up to 65534 rows (fields) and between 3 and 13 columns (attributes) whose meaning is as follows:

Column	*Definition*	*Default*
1	*Field Contents*	*MANDATORY*
2	*Field Position - Top Row*	*MANDATORY*
3	*Field Position - Left Column*	*MANDATORY*
4	*Window Size - Rows*	0
5	*Window Size - Columns*	0
6	*Field Type*	0
7	*Behaviour*	0
8	*Video Attributes*	0
9	*Active Video Attributes*	0
10	*Home Element - Row*	1
11	*Home Element - Column*	1
12	*Scrolling Group - Vertical*	0
13	*Scrolling Group - Horizontal*	0

The contents of a field may be characters, numbers, or another form.

Character data may be a simple scalar, a vector, or a matrix. A vector is treated as a 1-row matrix. The text is displayed left-justified in the window defined by `⎕SM[;2 3 4 5]`.

Numeric data may be a simple scalar, vector, or a 1-column matrix. It is displayed 1 number per row (a vector is treated as a 1-column matrix) in the window defined by `⎕SM[;2 3 4 5]`, and formatted according to the field type. A special class of numeric data is **dates**. Dates are represented in `⎕SM` as numbers (the number of days since Jan 1st 1900), but are displayed as character strings using special format rules. Date formats are described in detail in Section 3.5.

If the field contents are larger than the window (in either direction) the data may be scrolled by the user. Character data is scrolled one character at a time. Numeric data, including dates, is scrolled one number at a time. Different fields can be linked together into vertical and horizontal scrolling groups. If the data is scrolled vertically in one field, the data in all fields with the same vertical scrolling group is scrolled in parallel.

A field may also contain another form, i.e. a nested matrix with the same structure as `⎕SM`. Clearly this definition is recursive, so it is possible to define forms within forms within forms and so forth. This feature has two main uses. Firstly it provides a convenient way to represent more than one form, which although displayed together are logically separate and perhaps handled by separate APL functions. Secondly, nested forms provide the mechanism for scrolling at a higher level than a single character or number. If a field contains another form, it effectively defines a window onto a set of **sub-fields** which scroll one **sub-field** at a time. If the field is nested to a second level, it defines a window onto a set of **forms** which scroll one **form** at a time. Further levels of nesting can be used to provide scrolling of sets of forms, and so on.

1.4 General Principles - ⎕SR

`⎕SR` is a system function that allows the user to interact with the form defined by `⎕SM`. The optional left argument provides 3 syntaxes :

```
CONTEXT ←                           ⎕SR FIELDS
CONTEXT ← (⊂KEYS)                   ⎕SR FIELDS
CONTEXT ←  KEYS  INITIAL_CONTEXT ⎕SR FIELDS
```

The right argument is a list of fields that the user may visit. If the fields defined in `⎕SM` are all simple, the list of fields is a simple integer vector of `⎕SM` row numbers. For example `⎕SR 1 2 3` lets the user visit fields 1, 2 and 3. If the fields defined in `⎕SM` are nested and contain forms, the list may specify sub-fields to whatever level is required. For example `⎕SR (1 2) (2 3)` lets the user visit field 1 sub-field 2 and field 2 sub-field 3. See Section 5.2 for further details.

In Dyalog APL/W, `⎕SR` has been extended to allow the user to concurrently interact with the form defined by `⎕SM`, and other GUI objects. This is done by specifying the names of these objects in the right argument, in addition to fields of `⎕SM`. If this is done, `⎕SR` processes user-interaction with `⎕SM` fields as described in this Chapter, and processes events generated by the GUI objects in the same way as `⎕DQ`. For further details, see the description of the SM object in Object Reference.

The optional left argument is either a scalar containing a vector of exit key codes, or a 2-element vector whose second element specifies initial context information.

Exit keys define character keys, control keys, or mouse buttons which cause ⎕SR to terminate user interaction and return a result. Keys that are mapped to characters in ⎕AV are identified using those characters.

Control keys are identified using the mnemonic codes defined in the Input Translate Table (Chapter 1). For example, the list `('A' 'TB' 'F1')` would mean exit only if the user hits the TAB key or the F1 key, or enters the character A. Note that (if using the standard Input Translate Tables) "A" can be entered as <Shift a> in ASCII mode, or as unshifted "a" in APL mode.

The default exit code list is `('QT' 'ER' 'EP')` which maps to Shift+Esc, Enter and Esc respectively. If KEYS is an empty character vector `('')`, ⎕SR will exit as soon as the user presses **any** key.

The result of ⎕SR is a 6 or 9-element nested vector. The first 6 elements contain the cursor position (field, row, column), the exit key, event code and modification flags. If the exit from ⎕SR was caused by the user pressing a mouse button (in versions that support a mouse), the result contains 3 additional elements indicating the pointer position (field, row, column). See Section 7 for details.

The initial context information is a vector which specifies the initial cursor position, initial keystroke, event, and modification flags. Only as many elements as are required need be specified; those that are omitted take default values. See Section 5.2 for further information.

1.5 Associated Facilities - ⎕SD & ⎕KL

⎕SD (Screen Dimensions) and ⎕KL (Key Labels) provide information about the current display and keyboard, so that applications can be written in a device- independent manner.

⎕SD is a 2-element integer vector containing the current number of rows and columns in the ⎕SM Window. If the user changes the size of the ⎕SM window, the new size is reflected in ⎕SD. In non-window implementations of Dyalog APL, ⎕SD returns the size of the screen,

⎕KL is a system function that returns the labels associated with the specified keys. These are taken from your Input Translate Table and are normally defined to be the legends engraved on your keys.

2. Field Attributes

2.1 Field Contents : ⎕SM[;1]

Column 1 of ⎕SM contains the data associated with each field. The data may be simple or nested. Simple data may be character, numbers, or dates. Normally nested data represents another form, and must therefore have the same general structure as ⎕SM itself. However, a special case exists for handling blanks and invalid data in free-format numeric fields. Table 2.1.1 at the end of this section summarises what data is valid in which type of field. See Section 8.7 for further details.

If you have already defined a form, assigning a new value to ⎕SM[;1] will immediately display that value on the screen.

2.2 Field Position : ⎕SM[;2 3]

Columns 2 and 3 of ⎕SM contains the row and column positions respectively of each field. At the top level these represent physical positions on the screen or Window, with (1,1) meaning the top left corner, and ⎕SD the bottom right corner. A position less than (1,1) or greater than ⎕SD is invalid and causes *FIELD POSITION ERROR*.

In a nested field, the position of a sub-field is defined relative to the start of the enclosing window. The position of a sub-field is not restricted by the size of the screen, except that no element may be located at a position greater than row or column 32767.

When the cursor is moved to a sub-field which is located off the screen, the form is scrolled so that the field comes into view.

Figure 2.2.1 illustrates the positioning of a simple field. The first character position of the field is at row 3, column 12.

```
⎕SM←1 3⍴'Hello World' 3 12
```

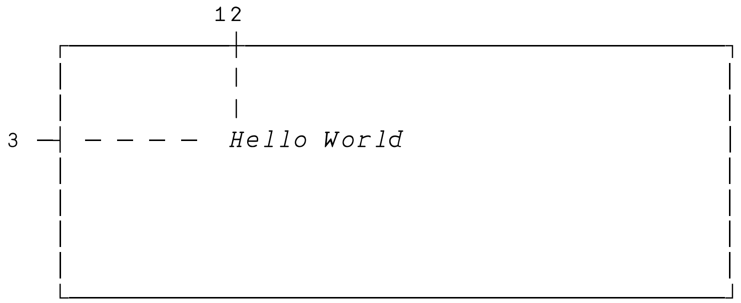


Fig 2.2.1 Positioning of a Simple Field

Figure 2.2.2 illustrates the positioning of nested fields. In this example, `⎕SM` contains 2 fields each of which is a form. The first field (`FORM1`) is located at screen position (2,5). The second field (`FORM2`) is located at (5,35). `FORM1` contains 2 sub-fields which are located at (4,12) and (3,24) respectively relative to the start of its window (2,5). The resultant screen position of the first sub-field is (5,16) and that of the second is (4,28).

```
      FORM1←↑('Hello World' 4 12) (42 3 25)
      FORM2←↑((⍳3) 1 3) ((5 4⍴'ABC') 1 7)
      ⎕SM←↑(FORM1 2 5)(FORM2 5 35)
```

(Note that the positions of `FORM1` and `FORM2` on the screen are shown by dots. In practice these would be blanks)

```
              1    1    2    2    3    3
     1   5    0    5    0    5    0    5
  ┌┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴─────────────┐
1 ┤                                                   │
2 ┤    .........................                      │
3 ┤    .........................                      │
4 ┤    .......................42                      │
5 ┤    ...........Hello World...     ..1...ABCA       │
6 ┤                                  ..2...BCAB       │
7 ┤                                  ..3...CABC       │
8 ┤                                  ......ABCA       │
9 ┤                                  ......BCAB       │
  │                                                   │
  │                                                   │
  └───────────────────────────────────────────────────┘
```

Fig 2.2.2 Positioning of Nested Fields

There are no particular restrictions on overlapping fields, and several fields may occupy the same position on the screen. See Section 8.1 for further details.

Once you have defined a form, you can change the position of the fields by assigning new values to `⎕SM[;2 3]`. This can be used to produce animated effects.

2.3 Window Size : ⎕SM[;4 5]

Columns 4 and 5 specify the size of window (number of screen rows and columns) associated with each field. If unspecified or zero, the size of the window is decided by the field contents. If the field is simple, the window is the number of rows and columns occupied by the formatted data. If the field is nested, it is the rectangle bounding the windows required for its sub-fields.

If specified, the window must lie between screen position (1,1) and (32767,32767). If the window is smaller than the space required by the contents of the field, the data may be scrolled by the user.

The window defines the area on the screen which is occupied by a field. In particular the window will be displayed using the field's specified background colour (default 0), and will totally or partially obliterate any other window defined by an earlier row in `⎕SM`. See Section 8.1 for details.

Once you have defined a form, you can change the size of the window associated with any field by assigning new values to `⎕SM[;4 5]`. This can be used to provide a zoom/un-zoom feature for user help.

Table 2.4.1 Valid/invalid Field Contents

	Field Type/behaviour				
Field Contents	*Def*	*Char*	*Num*	*Num Calc*	*Date*
Character Scalar	*YES*	*YES*	*YES*	*NO*	*YES (i)*
Vector	*YES*	*YES*	*YES*	*NO*	*YES*
Matrix	*YES*	*YES*	*NO*	*NO*	*NO*
Numeric Scalar	*YES*	*NO*	*YES*	*YES*	*YES*
Vector	*YES*	*NO*	*YES*	*YES*	*YES*
(1 col) Matrix	*YES*	*NO*	*YES*	*YES*	*YES*
Nested Vector of Char Vectors (and) Numeric scalars	*YES*	*NO*	*YES*	*NO*	*YES*
Another form (⎕SM definition)	*(field type ignored)*				

Note (i): It is theoretically possible to have a single character date field with the format "D*", although it would be of no practical value.

2.4 Field Type : `⎕SM[;6]`

Column 6 of `⎕SM` specifies the field type. This attribute is applicable only to simple fields, and is ignored if the field is nested. The various field types provided are described in detail in Section 3. If the field type is unspecified or zero, it is derived from the field contents.

The following combinations of field type and field contents are not allowed and cause the error `FIELD TYPE/CONTENTS MISMATCH` :

a) numeric data in a character field

b) character data in a pocket-calculator numeric field

You may however assign character data to a normal numeric field; for example to initialise a numeric field to blanks. For further details, see Table 2.4.1 and Section 8.7.

You can of course change field type dynamically. For example, you could highlight a piece of text by changing the field type from 1 (character) to 3 (uppercase). You could also allow the user to select the format of numeric or date fields by changing the field type in response to a function key.

2.5 Field Behaviour : `⎕SM[;7]`

Column 7 of `⎕SM` specifies the field behaviour. This is defined as the sum of a set of permissible behaviour codes, which are themselves powers of 2. Certain behaviour codes apply only to simple fields. See Section 4 for details. If unspecified the default is zero.

You can dynamically change field behaviour by assigning new values to `⎕SM[;7]`.

2.6 Video Attributes : ⎕SM[;8]

Column 8 of ⎕SM specifies the field video attributes. This is an encoding of foreground colour, background colour, and video, ie.

```
VID ← VIDEO
    + BACKGROUND COLOUR × 256
    + FOREGROUND COLOUR × 256 × 256
```

or

```
VID ← 256 ⊥ F B V
```

The corresponding video, background colour, and foreground colour indices are mapped via the Output Translate Table to actual screen appearance depending upon the screen in use. The default is 0. Note that foreground colour 0 is normally white and background colour 0 is normally black.

In a simple field, all three video attributes are significant. In a nested field, the window is displayed in the specified background colour, but the other two attributes are ignored in favour of the attributes defined for the sub-fields.

For a particular field, the video attribute can either be a single number, or an array of numbers with the same dimensions as the field contents. If so, it specifies the video attribute on a per-element basis.

You can dynamically change the colour of a field simply by assigning a new value into `⎕SM[;8]`. This is useful for highlighting fields in error.

2.7 Active Video Attributes : `⎕SM[;9]`

Column 9 of `⎕SM` specifies the 'active' field video attributes. Like `⎕SM[;8]` this is an encoding of foreground colour, background colour, and video which is mapped to a particular screen appearance via the Output Translate Table.

The Active Video Attribute specifies the appearance of the field or sub-field when it contains the cursor.

Unlike `⎕SM[;8]` the active video attribute is always a single number. You cannot have different active video attributes on a per-element basis.

By default if the field is a simple field, its active video defines the appearance of the whole field when the cursor is in it. However if the field is "row-significant" (see Section 4.12) only the row that contains the cursor is displayed using the active video.

The active video of a nested field defines the appearance of the field when it is addressed by the cursor as an element of another nested field. This means that the active video of the top-level field in a nested structure is never used. Unlike `⎕SM[;8]` in which the background colour alone is significant, the active video of a nested field overrides the video attributes of its sub-fields.

If unspecified, active video defaults to 0 (normal video. No active video is specified explicitly by assigning a value of `¯1`.

2.8 Home Elements : `⎕SM[;10 11]`

Columns 10 and 11 of `⎕SM` specify the home elements. These represent the position of the field contents relative to the window, and are applicable only to scrollable fields.

If unspecified, the default value of `⎕SM[;10 11]` is (1,1) which means that the data is positioned in the top left corner of the window. You may position the data otherwise by specifying a different pair of values to `⎕SM[;10 11]`.

After user interaction you can derive the new position of the data relative to the window from the new value of `⎕SM[;10 11]`. Remember too that the row and column position returned by `⎕SR` are relative to the data, not the window. If you need to know which row and column of the window the cursor was on when `⎕SR` terminated you must subtract the home element from the result of `⎕SR` and add 1.

Figure 2.8.1 illustrates how the values of the home elements are affected by scrolling.

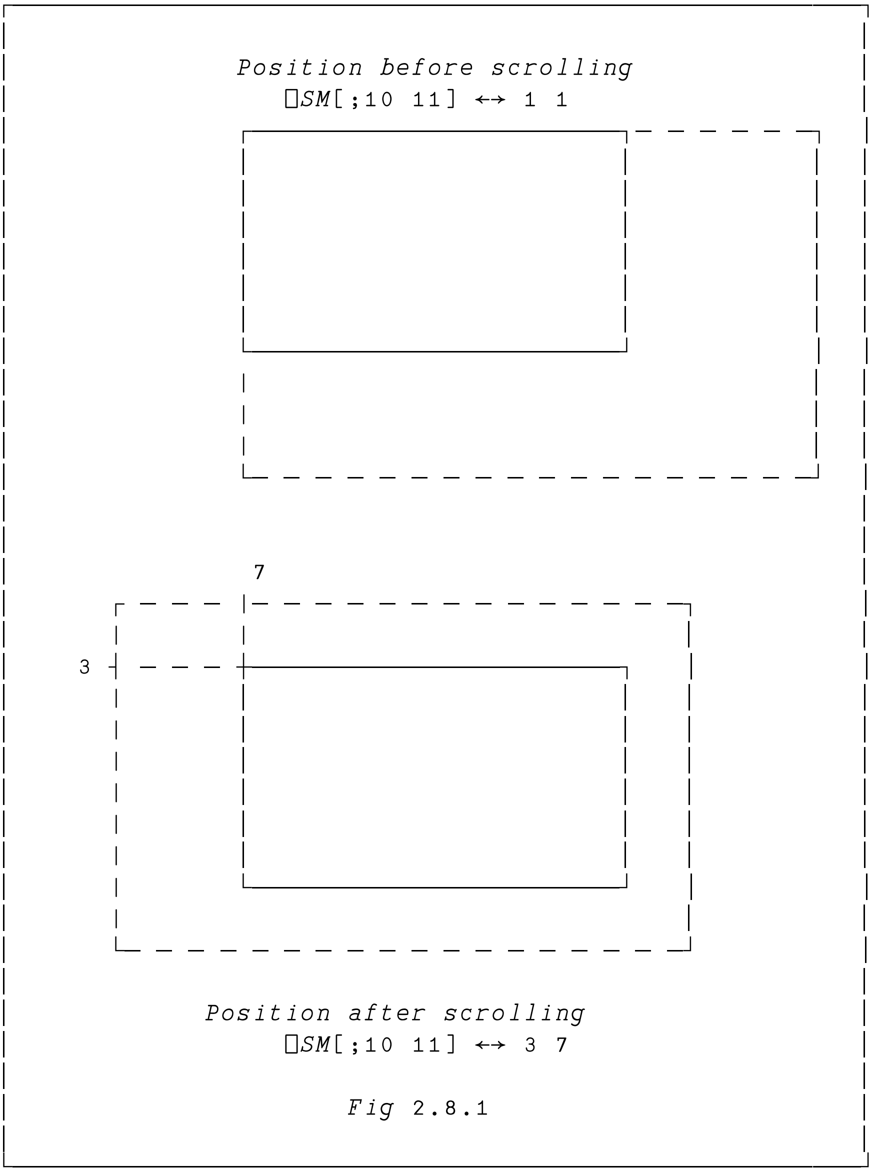


Fig 2.8.1

2.9 Scrolling Groups : `⎕SM[;12 13]`

Columns 12 and 13 of `⎕SM` specify vertical and horizontal scrolling groups respectively. A scrolling group is an arbitrary integer that links together a number of fields for scrolling purposes. Figure 2.9.1 illustrates the way fields may be linked in scrolling groups for spreadsheet applications.

All the fields with the same value of `⎕SM[;12]` scroll vertically together. When the user scrolls the data in one field using Cursor Up, Cursor Down, etc., the data in all the fields of the same group scrolls up or down 1 element in parallel. Although fields in the same vertical scrolling group are usually arranged side by side so that they occupy the same rows of the screen, this is not a requirement. Vertical scrolling is often combined with row-significant behaviour (See Section 4.12).

All the fields with the same value of `⎕SM[;13]` scroll sideways together. When the user scrolls the data in one field using Cursor Left, Cursor Right, etc., the data in all fields of the same group scrolls sideways one element in parallel. Although fields in the same horizontal scrolling group are usually arranged above one another so that they occupy the same columns of the screen, this is not a requirement. It is possible to include a numeric or date field in a horizontal scrolling group. However, as the element of such a field is the entire number or date, the effect is not useful.

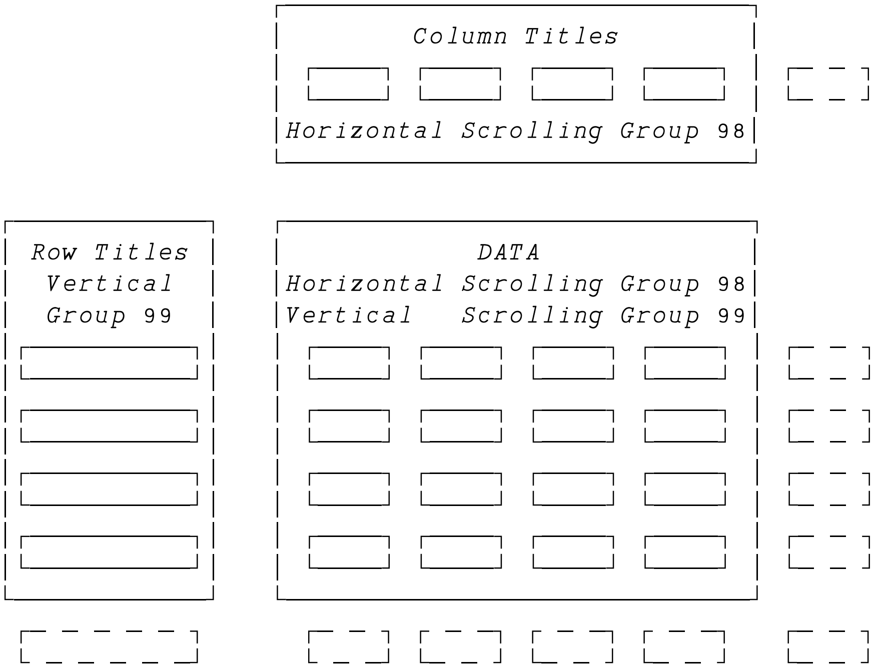

Column Titles
Horizontal Scrolling Group 98
Row Titles
Vertical
Group 99
DATA
Horizontal Scrolling Group 98
Vertical Scrolling Group 99

Fig. 2.9.1 Scrolling Groups for a Spreadsheet

3. Field Types

3.1 Introduction

Column 6 of ⎕*SM* specifies field type. It is relevant only to simple fields, ie fields which contain simple text or numbers, and is ignored if the field is nested.

The following field types are valid:

Type	*Description*	*Example*
0	*Default*	
1	*Character*	*Hello World*
3	*Character, uppercase*	*HELLO WORLD*
4	*Character, lowercase*	*hello world*
5	*Numeric, 'point plain' format*	12345.678
5*x*	*As type* 5 *but with x decimal places*	12345.68
6	*Numeric 'point triple' format*	12,345.678
6*x*	*As type* 6 *but with x decimal places*	12,345.68
7	*Numeric 'comma plain' format*	12345,678
7*x*	*As type* 7 *but with x decimal places*	12345,68
8	*Numeric 'comma triple' format*	12.345,678
8*x*	*As type* 8 *but with x decimal places*	12.345,68
9*x*	*Date, format x*	30 *Apr* 1949

3.2 Default Fields

If you specify 0, the field type is defined by the field contents. If the field contains characters, the field type will be 1; if the field contains numbers the field type will be 5. Note that if your program assigns numbers to a field with type 0 that previously contained characters, the field type changes to numeric.

3.3 Character Fields

Field contents may not be numeric. If they are, the assignment to `⎕SM` will cause error 56 (`FIELD CONTENTS/TYPE MISMATCH`). Character data will be displayed top and left-justified in the window. A character vector will be treated as a 1-row matrix. The data may contain any of the characters in `⎕AV`, but terminal control characters (`⎕AV[1 2 3]` in origin 0) will be replaced by blanks. If the field type is 3, all lowercase alphabetic characters will be translated to uppercase characters before being displayed. If the field type is 4, the reverse translation (uppercase to lowercase) will be applied. For behaviour during `⎕SR` input, see Section 6.1. Note however, that if the field type is 3 or 4, the appropriate uppercase/lowercase translation is applied. This conversion assumes that the lowercase and uppercase alphabets occupy `⎕AV[17-42]` and `⎕AV[65-92]` in origin 0 respectively.

3.4 Numeric Fields

If the field contains numbers, the data is displayed top and right-justified in the window with 1 number per row. A numeric vector will be treated as a 1-column matrix.

If the field is a pocket-calculator field with behaviour code 8192 `(C̲A̲L̲C̲)` the field contents must be numeric. Assigning character data will cause error 56 (`FIELD TYPE/CONTENTS MISMATCH`).

If the field is a normal numeric field it may initially contain text. If so, the text is displayed "as is" but is right-justified and truncated if necessary. In a multi-row field an enclosed text vector may be substituted for a number.

If the field type is 5, 6, 7, or 8, a column of numbers is displayed in a similar way to standard APL output. Each number is formatted independently to the appropriate number of decimal places (affected by `⎕PP`); and then the numbers are aligned so that the last digit of their integer portion occupies the same column on the screen. Note that integers are shown with no decimal point, and '`E`' format is not provided.

If the field type is 5x, 6x, 7x, or 8x the numbers are formatted to "x" decimal places, right-justified, and aligned vertically so that their decimal points occupy the same column position.

If the field type is 5, 5x, 6 or 6x, numbers are displayed using a "`.`" as the decimal point. If the field type is 7, 7x, 8, or 8x numbers are displayed using a "`,`" as the decimal point.

If the field type is 6 or 6x, a "`,`" separator is inserted between thousands. If the field type is 8 or 8x, a "`.`" separator is used instead.

Negative numbers are displayed using the ASCII minus sign "`-`".

If when fully formatted a number cannot be displayed in the number of columns specified by `⎕SM[;3]` it is displayed as a string of asterisks.

Numeric fields may be input using normal free-format input or pocket-calculator input. See Section 6 for further information.

3.5 Date Fields

The field may contain numeric or character data. Characters are inserted "as is" but are right-justified and truncated if necessary.

Numbers are taken to be dates specified in terms of the number of days since 1st January 1900. When displayed they are converted to character strings according to a date format specification, and may be edited in this form by the user. Valid dates obtained by referencing `⎕SM` are numeric. Figure 3.5.1 illustrates the principle:

Value assigned to ⎕SM	18017
Date displayed as	30/04/49
User changes to	13/05/89
Value returned from ⎕SM	32640

Fig. 3.5.1
Date conversion by ⎕SM

The date format is specified by the last digit of the field type. For example, 93 refers to date field, format 3. Date formats are defined in the APL Format file. See Section 8.8 for further details.

Figure 3.5.2 illustrates the part of this file which defines date formats. The system works as follows :

First the days of the week are defined. This is done in a series of lines prefixed by the code "`D:`". The first 7 words (separated by blanks) define the names. Next the names of the months are declared by the first 12 words contained in lines beginning with the code "M:". Finally, various date formats are defined by lines beginning with the code "`Fx`", where "`x`" specifies the format number (to be used for field type "`9x`").

```
D:Sunday Monday Tuesday Wednesday
D:Thursday Friday Saturday

M:January       February        March
M:April         May             June
M:July          August          September
M:October       November        December

F0=  /  /  :DD/MM/YY          + 09/04/49
F1:DDMMYY                     + 090449
F2:YYMMDD                     + 490409
F3=  /  /  :MM/DD/YY          + 04/09/49
F4:MMDDYY                     + 040949
F5:DD MON YY                  + 09 Apr 49
F6:DAY* D* MON* YYYY          + Saturday 9 April 1949
F7:DAY, D* of MON* YYYY       + Sat, 9 of April 1949
F8:D*:M*:YYYY                 + 9:4:1949
F9:D*/M*/YY                   + 9/4/49
```

Fig. 3.5.2
Specification of Date Formats

The general form of the date format specification is :

`Fx=zerodate:datespec`

where **`zerodate`** optionally defines a string of characters to be displayed when the corresponding value in the field is 0, and **`datespec`** defines the format of the date otherwise.

`datespec` consists of one or more special date format codes and other characters. When a date is displayed, the format codes are replaced by corresponding strings and/or values. Other characters are output unchanged. For example, **`F7:DAY, D* of MON* YYYY`** would display the value 18017 as "`Sat, 9 of April 1949`". The code "`DAY`" is replaced by "`Sat`", "`D*`" by "`9`", and so forth. The comma and spaces in the format specification are treated as punctuation and displayed unchanged. Punctuation may contain any of any characters in `⎕AV`, including National language characters and APL symbols. See Section 8.8.1 for further information.

Figure 3.5.3 describes the coding system in detail.

Code	*Meaning*	*Example*
YY	*Year number (see y2k note)*	89
YYYY	*Year + century*	1989
*MON**	*Month name*	*November*
MON	1*st* 3 *letters of month name*	*Nov*
MM	2 *digit month number*	01
*M**	*Month number*	1
*DAY**	*Day name*	*Mardi*
DAY	1*st* 3 *letters of day name*	*Mar*
DD	2 *digit day number*	05
*D**	*Day number*	5

Fig. 3.5.3
Date Formatting Codes

Note that codes containing a "*" define dates which are of variable length; those which do not define fixed format fields. During input, any punctuation in fixed-length date fields is automatically inserted.

If you assign the number 0 to a date field, the corresponding **`zerodate`** string is displayed. If such a string is not defined, the **`datespec`** string is displayed. For example, if you were using the standard format file, placing a 0 in a field of type 90 would display as " / / ". Placing a 0 in a field of type 91 would display as "*DDMMYY*". Note that the user can reset the field to zero by pressing the Cut key.

Year-2000 Compliance

As stated previously, dates are by default interpreted relative to 1st January 1900. If you use 2-digit year format, this will mean that dates after 1999 will be interpreted wrongly. You may avoid this problem using the yy_window parameter. This allows you to define fixed or sliding windows for the interpretation of 2-digit years.

4. Field Behaviour

Column 7 of *⎕SM* is used to define certain field characteristics. These are expressed as the sum of a set of behaviour codes which are themselves powers of 2. Table 4.1 summarises these codes which are described in detail below.

4.1 Behaviour Code 1 (*E̲X̲I̲T̲*)

This behaviour code causes *⎕SR* to terminate when the user attempts to skip to another field. The 4th element of the result of *⎕SR* will contain the key used to leave the field, and the 5th element will indicate that event code 1 has occurred. This behaviour can be used if you want to validate the field unconditionally, or if you want to reset a help message associated with the field. Notice that if the user presses an exit key it will not trigger this behaviour. Only a key that would otherwise cause a skip to another field will do so.

4.2 Behaviour Code 2 (*M̲O̲D̲I̲F̲I̲E̲D̲*)

This behaviour code causes *⎕SR* to terminate when the user attempts to skip to another field after having changed the current field. The 4th element of the result of *⎕SR* will contain the key used to leave the field, and the 5th element will indicate that event code 2 has occurred. This behaviour can be used if you want to validate the field only if it has been changed. Notice that if the user presses an exit key it will not trigger this behaviour. Only a key that would otherwise cause a skip to another field will do so.

Note that the field is deemed to have been changed if the user has typed in it. Thus if the user overtypes a "0" with a "0", the field is deemed to have been modified, and an attempt to leave it will trigger this event. You can tell if the value of the field has actually changed by comparing the corresponding element of *⎕SM* before and after the *⎕SR*.

Code	Name	Description
1	<u>EXIT</u>	*Terminates ⎕SR with event code 1 if user presses a key that would cause the cursor to leave field.*
2	<u>MODIFIED</u>	*Terminates ⎕SR with event code 2 if user types in field then presses a key that would cause the cursor to leave the field.*
4	<u>BADFIELD</u>	*Terminates ⎕SR with event code 4 if user tries to leave a field which is invalid.*
8	<u>NOFIELD</u>	*Terminates ⎕SR with event code 8 if user presses a movement key that would generate a skip, and there is no adjacent field,*
16	<u>ENTER</u>	*Terminates ⎕SR with event code 16 if cursor enters field.*
64	<u>BUTTON</u>	*Terminates ⎕SR with event code 64 if mouse button pressed in field.*
128	<u>STICK</u>	*Cursor Keys : disables skip to another field*
256	<u>START</u>	*Cursor Keys : cursor enters field at start of data or row.*
512	<u>PROTECT</u>	*Protected field (input ignored)*
1024	<u>ACTVID</u>	*Leave active video on*
2048	<u>NOCURSOR</u>	*Invisible cursor*
4096	<u>ROWSIG</u>	*Row-significant field*
8192	<u>CALC</u>	*Calculator-style numeric input*
16384	<u>AUTOTAB</u>	*Auto-tab on filling field*

32768	*DISCARD*	*Discard trailing characters*
65536	*CLEAR*	*Clear row/field on first char.*

Table 4.1 Field Behaviours

4.3 Behaviour Code 4 (*BADFIELD*)

This behaviour code causes ⎕*SR* to terminate if the user attempts to skip to another field when the contents of the current row of the current field are invalid. The 4th element of the result of ⎕*SR* will contain the key used to attempt to leave the row, and the 5th element will indicate that event code 4 has occurred. This behaviour is only useful for fields that can be invalid, ie. date fields and numeric fields with free-format input. Numeric fields with behaviour code 8192 (*CALC*) are guaranteed to always contain a valid number, so there is no point in setting this behaviour code. Numeric fields with free-format input are guaranteed to reject invalid characters, but the user may enter things like "`1.3.4-3`" which can only be validated on exit.

Note that the behaviour applies to individual rows, so you will get an exit if the user enters an invalid number in a column of numbers and tries to Cursor Up or Down to another number in the same field.

4.4 Behaviour Code 8 (*NOFIELD*)

This behaviour code causes ⎕*SR* to terminate under the following conditions :

a) If a Cursor Key is used to attempt to skip to another field, and there is no field in the chosen direction.

b) If the user presses Tab whilst in the last field in the list of fields being processed by ⎕*SR*; or in the last row of the last field if the field is "row-significant".

c) If the user presses Shift+Tab whilst in the first field in the list of fields being processed by ⎕*SR*; or in the first row of the first field if the field if the field is "row-significant ".

The 4th element of the result of ⎕*SR* will contain the key used to attempt to skip from the field, and the 5th element will indicate that event code 8 has occurred.

This behaviour can be used to create a "wrap-around" effect where the Cursor Keys are used to skip between fields such as in spreadsheet applications and horizontal menus. It is also useful if you want to process groups of fields separately, as it prevents the cursor from circulating back to the beginning at the end of the field list.

Notice that if the user presses an exit key it will not trigger this behaviour. Only a key that would otherwise cause a skip to another field will do so.

4.5 Behaviour Code 16 (*ENTER*)

This behaviour code causes ⎕*SR* to terminate if the user attempts to enter the field. The 4th element of the result of ⎕*SR* will contain the key used to attempt to enter the field, and the 5th element will indicate that event code 16 has occurred. The first 3 elements of the result will contain the number of the field being entered, and the cursor position (1,1).

This behaviour can be used to display a help message or a pop-up menu giving a list of choices for this field. It can also be used to trap arrival in the first of a group of fields that you may want to process in some special way. It is also useful for trapping entry to dummy fields to control the positioning of a form during scrolling. See Section 8.5 for details.

4.6 Behaviour Code 64 (*BUTTON*)

This behaviour code makes the field sensitive to mouse buttons. It is only applicable to those versions of Dyalog APL with mouse support. A *BUTTON* field cannot be entered with the cursor. However if the user points to it with the mouse pointer and presses a button, ⎕*SR* terminates. The position of the cursor remains unaltered, and is returned in the first 3 elements of the result. Information concerning the *BUTTON* field is returned in elements 7,8 & 9.

4.7 Behaviour Code 128 (*STICK*)

This behaviour code disables the "skip" function of the Cursor Keys so that they stick (and the terminal beeps) when they reach the edge of the data. This feature is useful if you wish to standardise on Tab and Shift+Tab keys for moving between fields.

4.8 Behaviour Code 256 (*START*)

This behaviour code causes the cursor to enter a field or a row of a "row-significant" field at the beginning of the data or at the beginning of the row rather than geographically. It only applies to Cursor Key movement; skipping with Tab and Shift+Tab is unaffected. This behaviour means that Cursor Up and Cursor Down can be used to skip between fields, or between rows of "row significant" fields without the need to move the cursor sideways back to the beginning. This is particularly useful if you are entering data into a series of blank fields for the first time. It is less useful if the user is likely to want to edit data at random, in which case the default geographical movement is probably more helpful.

Note that this behaviour is a property of the field being entered, not of the one being left.

4.9 Behaviour Code 512 (*PROTECT*)

This behaviour code prevents the user from editing or changing data in the field. Any keystroke other than a movement key is ignored and the terminal beeps. This behaviour is useful for menus and for fields in which you want the user to be able to position the cursor, but not change the data.

4.10 Behaviour Code 1024 (*ACTVID*)

By default when *⎕SR* terminates the active video of the current field is reset to normal in the same way as it is when the cursor skips to another field. This behaviour code overrides the default and causes the active video to remain set. The active video will remain set until your program issues another *⎕SR* **on the same form**. This is important, if your program pops up another form (by localising *⎕SM*), the appearance of the first form, including the highlighting of the last active field, is unaffected.

A common use of this feature is to present a stack of menus in which the item selected at one level remains highlighted while the user chooses from a sub-menu. It can also be used to indicate that validation (triggered by *⎕SR* terminating on an event) is in progress.

4.11 Behaviour Code 2048 (*NOCURSOR*)

This behaviour code temporarily disables the cursor. It is useful for menus and perhaps for "pocket-calculator" fields.

4.12 Behaviour Code 4096 (*ROWSIG*)

This behaviour code defines a field to be "row-significant". It is applicable only to simple multi-row fields, and is ignored if the field contains only a single row or is nested. This behaviour is intended to be used for a group of related fields that are processed together as a table by a single *⎕SR*. It is of little value when used with a single field. A group of row-significant fields have the following properties:

4.12.1 Active video applies to the row of the field in which the cursor resides; not to the field as a whole.

4.12.2 The default behaviour of the Tab key is modified so that it skips along the same row between fields, but skips down one row when moving from the last field in the field list to the first. Shift+Tab works in the same way but in the reverse direction. This motion is illustrated in Fig. 5.5.2 in the section describing the Tab and Shift+Tab keys.

4.12.3 Behaviour codes 1 (*EXIT*), 2 (*MODIFIED*), and 16 (*ENTER*) operate on individual rows in the field rather than on the field as a whole. This means that if you define a field with behaviour codes 2 and 4096, you will get an exit from ⎕*SR* on each row that the user changes.

4.12.4 If behaviour code 256 (*START*) is also defined, the Cursor Keys will skip to the beginning of the row of data rather than to the beginning of the field. For example, Cursor Down in a row-significant field will position the cursor at the beginning of the next row rather than on the same column. The exception to this rule is a Pocket Calculator field (see below) in which the cursor is normally positioned on the "units" column.

4.13 Behaviour Code 8192 (*CALC*)

This behaviour code defines a field to have "Pocket-Calculator" style of input. It is applicable only to simple numeric fields, and is ignored if the field is of type character, date, or is nested.

In a Pocket-Calculator field, the cursor in general remains fixed and data enters from the right. Cursor Left and Cursor Right keys may skip to adjacent fields, but do not move the cursor within the field. Input behaves like a pocket-calculator, ie digits enter from the right and cause existing digits to shift to the left.

New input automatically clears the existing value in the field on the first keystroke.

This behaviour is extremely useful because it guarantees that a field contains only valid numbers, and depending upon your taste, is a convenient way to enter data. It is especially useful if the user is entering new data or typically overwriting existing numbers. It is less helpful when the user needs to edit the odd digit in a large number. For this situation, the default "free-format" style of input may be preferable.

An important property of pocket-calculator fields is that Cursor Left and Cursor Right do not move the cursor around **within** the field. Instead they make the cursor skip to an adjacent one. This makes pocket-calculator fields very suitable for spreadsheet applications in which this type of cursor behaviour is expected.

For further details, see Sections 6.4 and 5.9.1.

4.14 Behaviour Code 16384 (*AUTOTAB*)

This behaviour code causes the automatic generation of a Tab after a character is entered in the last position of a field. In a "Pocket-Calculator" field (where input always occurs in the last position) a Tab is generated when the field is filled. Note that this is EXACTLY equivalent to the user filling a field without this behaviour, then pressing Tab. If an event occurs as a result of the autotab, the key returned in the 4th element of the result of ⎕*SR* is "TB".

4.15 Behaviour Code 32768 (*DISCARD*)

This behaviour code affects the way that existing characters are handled when new data is entered with INSERT mode ON. By default, characters to the right of the cursor are shifted to the right as each new character is entered. When a non-blank character hits the right edge of the data (which is not necessarily the edge of the window if the field scrolls) further entry is ignored and the terminal beeps. If this behaviour is set, the default is overridden, and displaced characters fall off the right end of the field and are discarded.

In "Pocket-Calculator" fields this behaviour is ignored.

4.16 Behaviour Code 65536 (*CLEAR*)

If the **first** key the user presses in a field is a printable character, this behaviour causes the field to be cleared to blanks from the current position of the cursor to the end of the current row. In the case of a multi-row character field (type 1,2 or 3) which does not have behaviour code 4096 (*ROWSIG*) the field is cleared to the end of the field, not just to the end of the current row.

If the first character the user presses is a movement or other key, this behaviour is disabled, and the data is available for editing as normal.

This behaviour is useful when you expect the user to want to **replace** rather than **modify** existing information on the screen, eg a default value.

This behaviour is assumed automatically in pocket-calculator fields and need not be specified explicitly.

5. Moving around the Form

5.1 Addressing Fields Using ⎕SR

The right argument of `⎕SR` specifies a list of fields that the user may visit. Until `⎕SR` terminates the user may position the cursor in any of these fields and move freely between them. The list of fields is a scalar or vector. Elements of the list are either integer scalars or integer vectors.

A scalar specifies a top-level field, and relates directly to a row in `⎕SM`. For example, the expression `⎕SR 2 4 27` allows the user to visit the fields defined by rows 2, 4, and 27 of `⎕SM`.

A vector element specifies a sub-field. For example, the statement `⎕SR (1 3)(2 1 5)` allows the user to visit the third sub-field of field 1 (ie the field defined by the 3rd row of the matrix contained in `⎕SM[1;1]`) and the 5th sub-field of the first sub-field of field 2 (ie the field defined by the 5th row of the matrix contained in row 1, column 1 of the matrix contained in `⎕SM[2;1]`).

A nested field can be accessed at varying levels according to the need. The principles are :

a) The objects addressed by elements of the right argument of `⎕SR` are referred to as **fields** and may be visited by the user. The Tab and Shift+Tab keys can be used to skip between these items. Under certain circumstances the Cursor Keys may also be used to skip between fields.

b) Within each of the fields addressed in this way the user may move the cursor from one **element** to another using the Cursor Keys. An element of a simple field is a single character or number, depending upon the field's type and behaviour. An element of a nested field may be a simple sub-field, or another nested field.

c) The user may not type in a field unless it is addressed as a simple field. To allow the user to modify data in a nested field you must address the sub-fields themselves.

5.2 Initialisation

5.2.1 Default

By default the cursor is positioned on the first element of the data in the first field (excluding `BUTTON` behaviour fields) in the list of fields to be visited.

In a simple character, default numeric, or date field, the first element means row 1, column 1 of the data. However, in a pocket-calculator field the cursor is normally positioned in the 'units' column of the field. See Section 6.4 for details.

In a nested field, the first element is the sub-field located at the smallest row number, and then at the smallest column number, of the enclosing window. If the first element is itself a nested field, the cursor may not necessarily coincide with any data.

5.2.2 Initial Cursor Position Specified

If the optional left argument to `⎕SR` specifies an initial cursor position the cursor is placed there instead. The initial cursor position is specified in terms of the field, then (optionally) the row and column.

The field specification must exactly match one of the fields in the right argument or it will cause an `INDEX ERROR`. A `NONCE ERROR` will be generated if the field specification refers to a field with `BUTTON` behaviour, which cannot accept the cursor.

If specified, the row and column must represent a valid cursor position within the field. If not it will cause a `NONCE ERROR`. The exception to this is that if the field is a Pocket-Calculator field the column is ignored and the cursor is placed in the "units" column of the field.

If the row and column are omitted, the cursor is placed in the first element of the field.

Consider the following examples. Note that `KEYS` refers to exit keys.

a) Let user visit fields 1, 2 & 5 but position cursor initially on row 3, column 4 of field 2 :

```
      KEYS (2 3 4) ⎕SR 1 2 5
```

b) Let user visit sub-field 2 of field 1 and sub-field 3 of field 2. Position cursor initially at row 2, column 4 relative to the top-left corner of the window associated with field 1, sub-field 2.

```
      KEYS ((1 2) 2 4) ⎕SR (1 2) (2 3)
```

c) An error ... the specified initial cursor position is invalid. In this case you cannot put the cursor on row 2 because the field only has 1 row.

```
      ⎕SM←'HELLO' 10 10
      KEYS (1 2 1) ⎕SR 1
NONCE ERROR
```

d) An error ... the field specified in the left argument is not a member of the right argument.

```
      KEYS (1 2 4) ⎕SR (1 2) (2 3)
INDEX ERROR
```

e) An error ... the initial cursor position is invalid. In this case there is no element at row 4 13 in field 1. Note that field 1 contains 2 elements, one at (4,12); the other at (3,25). These are the only valid cursor positions in the field.

```
      FORM1←↑('Hello World' 4 12) (42 3 25)
      FORM2←↑((⍳3) 1 3) ((5 4⍴'ABC') 1 7)
      ⎕SM←↑(FORM1 2 5)(FORM2 5 35)
      KEYS (1 4 13) ⎕SR 1 2
NONCE ERROR
```

5.2.3 Initial Keystroke

The 4th element of the initial context information specifies an initial keystroke. If unspecified the default is `''` which means no initial key.

If the initial keystroke is an exit key it is ignored. This allows your program to re-use the result of a previous `⎕SR` without modification, to cause screen processing to restart where it left off.

If the initial keystroke is an input key or a movement key it is actioned. If it is another key defined for the Input Translate Table but one which is neither an input key nor a movement key (eg F1), it is ignored. An invalid code will cause the error `KEY CODE UNRECOGNISED`.

The initial keystroke is actioned exactly as if it had been pressed by the user; except that if an event were to occur as a result of the key being pressed, the event is ignored. The reason for this is to allow your program to process an event without having to calculate where to position the cursor next.

For example, suppose that you want to trap user exit from a certain field so you assign it a behaviour code 1 (*EXIT*). When ⎕SR exits you check the cause. If it is because the user has left the field in question, you want to take some action and then continue. However, you do not want to have to calculate where the cursor should go next, which depends upon the key the user pressed to skip out of the field. ⎕SR therefore accepts that key as an initial keystroke, but does not action the event that, coupled with the field's behaviour, it would otherwise cause.

The code sample shown in Figure 5.2.3.1 illustrates this principle.

```
  EXITKEYS←'ER' 'F1' 'F2'   ⍝ Exit keys

  FIELDS←⍳10                ⍝ Fields to be input by
                            ⍝ user
  ⎕SM[5;7]←1                ⍝ ⎕SR will exit when user
                            ⍝ leaves field 5
  CONTEXT ← 1 1 1 ''        ⍝ Initial CONTEXT

Again: CONTEXT← (EXITKEYS CONTEXT) ⎕SR FIELDS

  →(∧/5 1=CONTEXT[1 5])/PROC5
  ...

PROC5:   ⍝ Process according to contents of field 5


  →Again
```

Fig. 5.2.3.1

5.3 Basic Rules of Cursor Movement

Cursor movement is governed by a simple algorithm that has been found to produce effective behaviour in a wide range of different forms.

When the user presses a key that causes a skip to another field or to another element in the same field, the following steps are performed.

First the 'target element' is identified. This is the element of the field to which the cursor must be moved. The particular field and element depends upon a number of things including the movement key that was pressed, the current position of the cursor, and field attributes.

If the target element is already visible on the screen, the cursor simply moves there.

Otherwise, `⎕SR` searches through those elements of fields specified in its right argument **which are currently visible on the screen** for the element which is geographically nearest to the target element, and moves the cursor there.

Finally, the form is scrolled to bring the target element under the cursor with the refinement that the corresponding field's home element is not allowed to become negative.

This principle is illustrated in Figure 5.3.1. In this example, the cursor is at position C and the user presses Tab.

First, the target element is identified as D, which is not currently visible on the screen. Next `⎕SR` considers the visible elements (B,C,E, & F) and chooses the one that is geographically nearest to D, ie. E, and moves the cursor there. Finally the entire form is scrolled so that D moves to the position under the cursor.

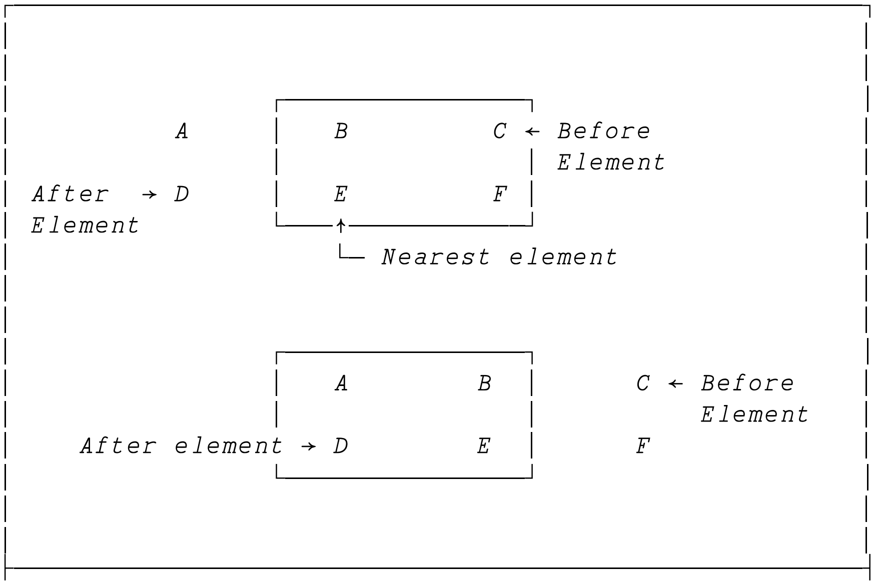


Fig. 5.3.1

```
│                  Cursor Movement                   │
└────────────────────────────────────────────────────┘
```

5.4 Movement Keys

Tab and Shift+Tab (BackTab) move the cursor between the fields specified in the right argument to `⎕SR`, whatever the structure of these fields may be. Tab moves to the next field in the list and Shift+Tab to the previous one. By default, the fields are treated in a circular manner so that Tab from the last goes to the first, and Shift+Tab from the first goes to the last. If the form is larger than the screen, skipping from one field to another causes it to scroll.

The Cursor Keys have three functions - move, scroll, and skip; and they operate geographically.

a) Within a field they move the cursor from one element to another.

b) If the contents of the current field are larger than the window they cause the data to scroll one element in the chosen direction.

c) They cause the cursor to skip to a geographically adjacent field.

There are two sets of "Jump" keys which move the cursor by larger increments within a field (a and b above). One set moves a "window's worth" at a time. The second moves to the end of the data in the chosen direction. These keys **do not** skip to an adjacent field.

The Ctrl+Home key moves the cursor to the beginning of the data in the current field.

The Enter key moves the cursor from its current position to the beginning of the next row in a multi-row field.

Unless specified as exit keys, the MOUSE BUTTONS can also be used to move the cursor. Within a field the user can freely reposition the cursor by moving the mouse pointer and clicking the **left** button. Movement between two fields is also possible, but is subject to other behaviours (`MODIFIED` and `EXIT` in the current field; `ENTER` and `BUTTON` in the pointer field).

Code	Keystroke	Action
TB	Tab	Next field (in the right argument to ⎕SR)
BT	Shift+Tab	Previous field (in the right argument to ⎕SR)
HO	Ctrl+Home	Start of data
ER	Enter	Next row (in a multi-row field).
LC	←	Move left 1 element, scroll left 1 element, or skip left to adjacent field.
RC	→	As LC but move rightwards.
DC	↓	As LC but move downwards.
UC	↑	As LC but move upwards.
LS	Ctrl+PgDn	Move (scroll if necessary) 1 "window's worth" left
RS	Ctrl+PgUp	As LS but move rightwards.
DS	PgDn	As LS but move downwards.
US	PgUp	As LS but move upwards.
LL	Home	Move (scroll if necessary) to leftmost column of data
RL	End	As LL but move to rightmost.
DL	Ctrl+Shift+↓	As LL but move to bottom of data.
UL	Ctrl+Shift+↑	As LL but move to top of data.
D1	BUTTON 1	Move cursor to pointer
to		
D5	BUTTON 5	Move cursor to pointer

Table 5.4.1 : *Movement Keys*

5.5 Tab and Shift+Tab Keys (TB, BT)

Tab and Shift+Tab cause the cursor to skip from the current field to a **target element** in a **target field**.

When the user presses Tab, the target **field** is the next field in the list defined by the right argument of `⎕SR`. If the current field is the last one, the list is processed again from the beginning, and the target field is taken to be the first.

When the user presses Shift+Tab, the target field is the previous field in the list. If the current field is the first one, the target field is taken to be the last.

In both cases, the choice of target **element** depends upon whether or not the field has behaviour 4096 (`ROWSIG`)

5.5.1 Normal Case

By default, the target element is the **first** element of data in the target field. This means that if the contents of the target field have been scrolled, they will be unscrolled horizontally and/or vertically back to the beginning. Note that if the field is a nested field the **first element** is identified as the element located at the lowest row number, and then if there is more than one element on that row, the one at the lowest column number.

5.5.2 `ROWSIG` case

If the target field is a simple multi-row field with behaviour code 4096 (`ROWSIG`), the target element is calculated differently.

Tab : If the target field is any field but the first one in the list, the target element is the first element of data in the **current row**.

If the target field is the first in the list, the target element is the the first element of data in the **next row**. If there is no next row, it is set to 1, and the cursor enters the field at the beginning.

Shift+Tab : If the target field is any field but the last one in the list, the target element is the first element of data in the **current row**.

If the target field is the last in the list, the target element is the first element of data in the **previous row**. If there is no previous row, it is set to the last row of the data.

Consider the effect of successive Tabs while the system executes the expression `⎕SR 1 2 3` on 3 fields laid out as shown below. Figure. 5.5.1 illustrates the default cursor movement; Figure. 5.5.2 shows what happens if the fields are row-significant.

Notice that in the latter case tabbing from field 1 to field 2 and from 2 to 3 preserves the current row. However, tabbing from field 3 to field 1 increments the row number by 1. Finally tabbing from the last row in field 3 causes the cursor to go to row 1 of field 1.

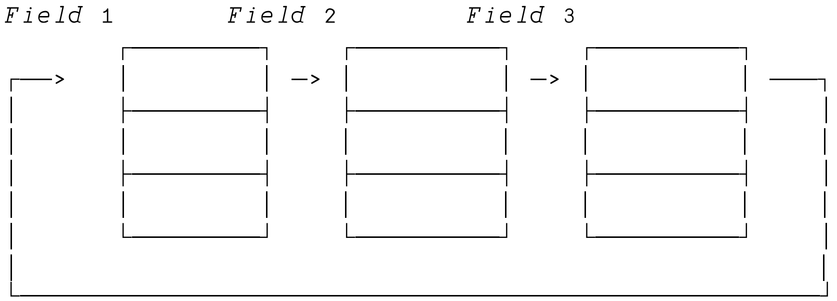


Fig 5.5.1 Default behaviour of Tab key

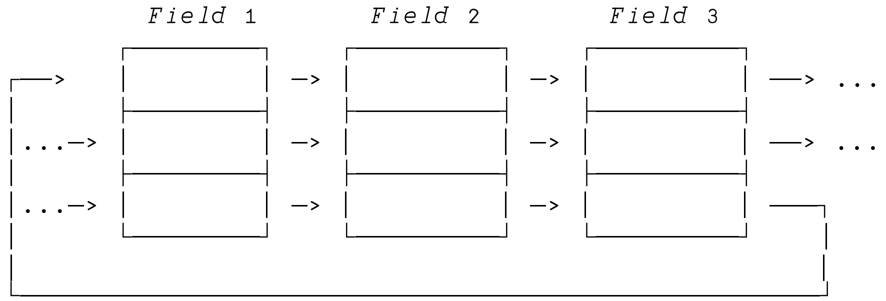


Fig 5.5.2 Behaviour of Tab key in row-significant fields

Notice that the row is relative to the data, not to the window. Thus if the contents of fields 1, 2, and 3 were 10 rows each, and the fields were linked in the same vertical scrolling group, tabbing from field 3 at the bottom of the window would cause the fields to scroll up. Furthermore, tabbing from the last row of the data in field 3 would unscroll all 3 fields and position the cursor in the first row of the data in field 1.

The logic to implement this behaviour depends ONLY on the target field being row-significant. For it to work properly it is ESSENTIAL that a set of related row-significant fields are processed together in a single call to `⎕SR`. The inclusion of unrelated fields in the list will cause unexpected results, as will the processing of fields containing differing numbers of rows. For further details on row-significant fields, see Section 4.12.

5.6 Ctrl+Home Key (HO)

Pressing Ctrl+Home moves the cursor to the first element of data in the current field. This means that if the contents of the field have been scrolled, they will be unscrolled horizontally and/or vertically back to the beginning of the data. Note that if the field is a nested field the **first element** is identified as the element located at the lowest row number, and then, if there is more than one element on that row, the one at the lowest column number.

5.7 Enter Key (ER)

If the current field is a simple multi-row field and the cursor is not in the last row of data, the Enter key causes the cursor to skip. The **target field** is the current field, and the **target element** is the first element of data in the next row. If the data in the current field has been scrolled sideways, it will be unscrolled sideways so that the cursor is at the beginning.

The Enter key is only applicable to simple multi-row fields and is otherwise ignored.

Note that Enter is a default exit key, in which case this behaviour does not apply.

5.8 Jump Keys (LS, LL, etc.)

The Left/Right/Up/Down Screen keys (LS, RS, US, DS) move the cursor a "window's worth" of data within the current field. The corresponding "limit" keys (LL, RL, UL, DL) move the cursor to the end of the data in the chosen direction. If necessary the data is scrolled.

5.9 Cursor Keys (LC, RC, UC, DC)

This section describes the behaviour of the Cursor Keys in terms of Cursor Right and Cursor Down. Cursor Left and Cursor Up operate in a similar way but in different directions. Notice that there are certain differences between the way that the horizontal and vertical Cursor Keys operate. In particular, the horizontal Cursor Keys do not operate at all within pocket-calculator fields.

Note that movement of the mouse when a button is held down is interpreted in terms of the cursor keys. This is particularly useful for scrolling.

5.9.1 Cursor Right (Left)

If the user presses Cursor Right, the **target field** is initially taken to be the current field, and the **target element** is the next element to the right on the same row. In a nested field, this means the element located at the same row number, and at the smallest column number greater than that of the current column. If such an element exists and it is already visible, the cursor moves there. If the element exists but is not currently visible, it is scrolled into view. If there is no such element to the right of the current one, AND the current field has behaviour code 128 (*STICK*) the key is ignored and the terminal beeps. Otherwise, ⎕*SR* attempts to identify a new **target field** to the right of the current one and move there. At first, only those fields which follow the current one in the right argument to ⎕*SR* are considered. A simple field is acceptable as the target field if its window occupies the current row. A nested field is acceptable if it is **located** (ie has its origin) on the current row. From the fields that satisfy these criteria, the one with the smallest column number greater than that of the current column is selected.

If no such field is found, those fields which precede the current field in the right argument to ⎕*SR* are then considered in the same way. If even then no field is selected, the key is ignored and the terminal beeps. Note that this algorithm can sometimes cause an apparently illogical skip if your fields are not accessed in geographical order. It does have the benefit however of improving performance in cases where a large number of fields are being processed, such as in a spreadsheet.

Having identified the target field, the selection of target **element** depends upon behaviour codes 256 (*START*) and 4096 (*ROWSIG*).

- By default Cursor Right selects the first element, and Cursor Left the last element in the **current row**.

- If the target field has behaviour code 256 (*START*), both keys select the **first** element of data. If the data is simple, this is row 1, column 1.

- If the target field has behaviour code 256 (*START*) **and** 4096 (*ROWSIG*) Cursor Left selects the first element of data in the current row, not the last.

- Note that if the field is a nested field, the target element is selected by virtue of its position in the window.

5.9.2 Cursor Down (Up)

If the user presses Cursor Down, the **target field** is initially taken to be the current field, and the **target element** is the next element below the current one on the same column. In a nested field, this means the element located at the same column number, and at the smallest row number greater than that of the current one. If such an element exists and it is already visible, the cursor moves there. If however, the field is a simple field with both behaviour codes 128 (*START*) and 4096 (*ROWSIG*), the cursor moves to the first element on the next row rather than moving down the current column.

If there is no such element below the current one, AND the current field has behaviour code 128 (*STICK*) the key is ignored and the terminal beeps.

Otherwise, ⎕*SR* attempts to identify a new **target field** below the current one and move there. At first, only those fields which follow the current one in the right argument to ⎕*SR* are considered. A simple field is acceptable as the target field if its window occupies the current column. A nested field is acceptable if it is **located** (ie has its origin) on the current column. From the fields that satisfy these criteria, the one with the smallest row number greater than that of the current row is selected. If no such field is found, those fields which precede the currentfield in the right argument to ⎕*SR* are then considered in the same way. If even then no field is selected, the key is ignored and the terminal beeps. Note that this algorithm can sometimes cause an apparently illogical skip if your fields are not accessed in geographical order. It does have the benefit however of improving performance in cases where a large number of fields are being processed, such as in a spreadsheet.

Having identified the target field, the selection of target **element** depends upon behaviour code 256 (*START*).

- By default Cursor Down selects the element on the first row, current column in the target field. Cursor Up selects the element on the last row, current column.

- If the target field has behaviour code 256 (*START*), both keys select the **first** element of data. If the data is simple, this is row 1, column 1.

- Note that if the field is a nested field, the target element is selected by virtue of its position in the window.

5.10 Scrolling

5.10.1 Introduction

Scrolling occurs whenever ⎕SR attempts to position the cursor on an element of a field that is not visible on the screen. This can be caused by moving from one element to another within a field, or by skipping between fields.

Whenever data is scrolled, other data which is related to it in some way will be scrolled in parallel.

The first instance of this is data that is related geographically. An example that we take for granted is that when we scroll up in a character matrix, the data in every column scrolls in unison. This is because each element of the matrix is related geographically to the others. The same principle applies in a nested field. If it is necessary to scroll an invisible sub-field into view, all sub-fields in the same form are scrolled so that they maintain their positions relative to one another. This feature can be used to implement an input form that is physically larger than the screen.

The second way to achieve related scrolling is to explicitly link fields together into scrolling groups. If scrolling occurs within a field, ⎕SR checks the field's horizontal or vertical scrolling group (⎕SM[;12] or ⎕SM[;13]) according to the chosen direction. If non-zero, the contents (ie elements) of any other fields in the same group are scrolled 1 element in the same direction. This feature can be used to link simple related fields together for parallel scrolling, and to link nested fields together for spreadsheets.

5.10.2 Scrolling Within Simple Fields

Scrolling in a simple field occurs if the window allocated to the field is smaller than the number of rows and columns required by the field contents after formatting. Scrolling is caused by pressing the appropriate Cursor Key when the cursor is at the edge of the window.

If the cursor is on the rightmost column of the window and there is more data to the right, pressing Cursor Right will shift all the data one character position to the left in the window. The characters that were displayed in column 1 will disappear, and a new column of data will scroll into view. Cursor Left operates in the same way but in the reverse direction.

Note that sideways scrolling in a simple field applies only to character fields, it is not supported within numeric and date fields.

If the cursor is on the bottom row of the window, and there is more data below, pressing Cursor Down will shift all the data up one row in the window. The previous contents of the first row will disappear, and a new row will scroll into view at the bottom. Cursor Up operates in the same way but in the reverse direction. Vertical scrolling works in all types of simple fields.

If the window in which the data is being scrolled is not itself entirely visible on the screen, the effective window in which the data is scrolled is reduced to the part of it that is visible.

5.10.3 Scrolling in a Nested Field

Scrolling within a nested field operates in the same way as in a simple field except that the field element is not a simple character, but is a sub-field or a collection of sub-fields.

If the cursor is on the rightmost visible element of a nested field, and there is another element located on the same row but at a higher column number which is currently not visible, pressing Cursor Right will cause the field contents to scroll left.

If the current element is one of a set of elements that are positioned in a horizontally regular fashion in the window, i.e. they all occupy the same number of columns and have the same horizontal spacing, the data will shift 1 element to the left. The left-most element will disappear, and new data will scroll into view from the right. If not, the data will shift sufficiently to the left to bring the new element into view whilst attempting to keep the cursor stationary. This may cause an element to be clipped by the left edge of the window, i.e. part of the element is visible, part invisible.

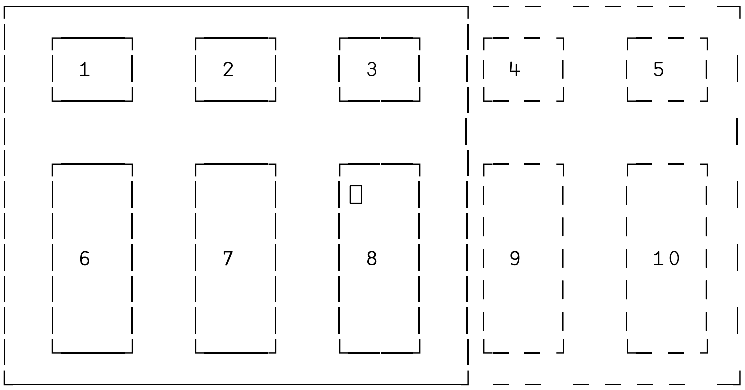
Elements arranged above and below the current one will also scroll in parallel so as to maintain their relative positions. Notice however that unlike a simple field the structure need not be rectangular nor need these other elements be arranged in the same horizontally regular fashion as those in the current row. This could also result in clipping.

5.10.4 Scrolling Between Fields

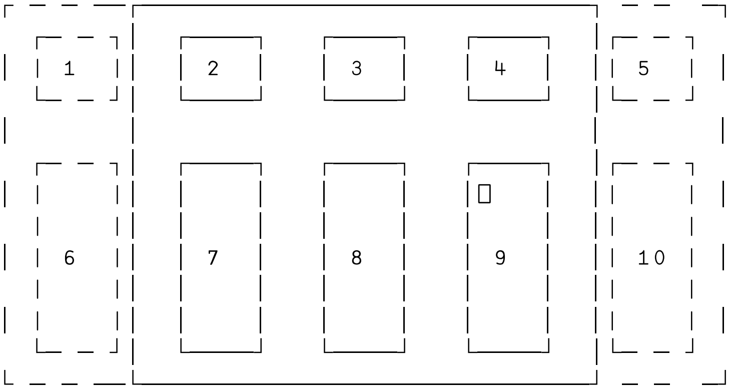
Scrolling will occur if the user skips to a field that is currently not visible on the screen. If skipping is a result of using the Cursor Keys, the behaviour is very similar to that described in the previous section because the cursor keys only move geographically. If Tab or Shift+Tab is used the behaviour is less specific and is best illustrated by example.

Figure. 5.10.4.1 illustrates scrolling of fields which are arranged in a regular fashion. Figure. 5.10.4.2 illustrates scrolling with fields arranged at random.

Fig 5.10.4.1 Scrolling of regular fields

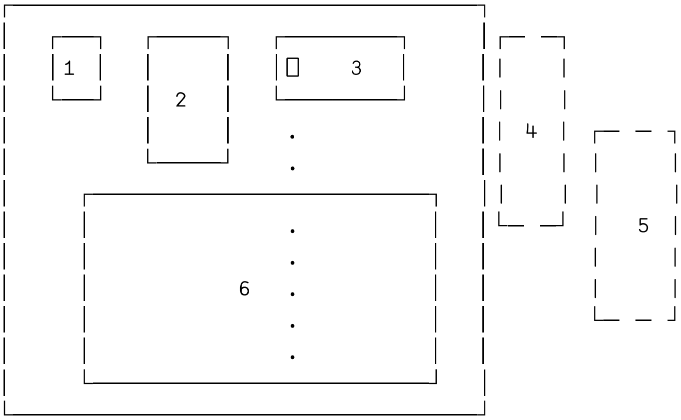


a) Position before pressing Tab

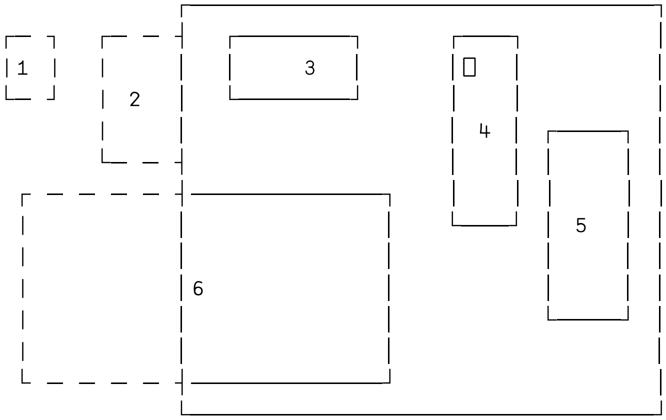


b) Position after pressing Tab

Fig. 5.10.4.2 Scrolling of irregular fields



a) Position before pressing Tab



b) Position after pressing Tab

6. Input

Introduction

User input is only allowed in simple fields . If a field contains sub-fields the user may move the cursor around, but not enter or change data. To allow the user to input into sub-fields of a nested field, you must address the sub-fields directly in the right argument of `⎕SR`.

`⎕SR` provides two sorts of field input; free format, and pocket-calculator. Free format input is used for character fields, default numeric fields, and date fields. Pocket-calculator input is an option for numeric fields only.

Like in the Session Manager, all data entry is governed by the Input and Output Translate tables.

Using the standard 2-mode APL/ASCII Input Table, `⎕SR` initially puts the keyboard in USER mode (ASCII). APL mode may be selected by the user pressing Ctrl+n or using the MODE key. If you want to prohibit the input of APL characters you should use a single-mode ASCII-only table.

Note that whatever the mode, overstrike resolution is applied. For example, "`[`" overstruck with "`∘`" will resolve to "`{`". Overstrike resolution is hard-coded and cannot be changed using the Input/Output Translation Tables.

6.1 General Input Rules

Except for input into Pocket Calculator fields (See Section 6.4) characters are entered in a similar manner to the way they are handled by the Session Manager. The main difference is in the way that Insert Mode works.

Normally if the user types a character the symbol will be displayed and the cursor advanced one character position to the right.

If the field has behaviour code 65384 (`C̲L̲E̲A̲R̲`) **and** the first character typed is a printable character, the field is cleared from the current cursor position to the end of the current row or to the end of the field, before the character is displayed.

If a character is typed on top of another in the rightmost character position in the cell, it normally replaces the original one and the cursor remains stationary at that position. Subsequent characters merely replace the one by which they were preceded. If however behaviour code 16384 (`AUTO`) is set, typing a character in the last character position is exactly the same as if the user followed the key by pressing Tab.

During input the following keys have a special effect. The codes indicated in parentheses are the codes by which these keys are identified in the Input Translate Table.

6.1.1 Insert key (IN)

The Insert key (IN) toggles Insert Mode on and off.

If Insert Mode is on the cursor remains stationary and the characters to the right of it are shifted. If the current row of the field is full (ie if the rightmost character is non-blank) there are two possibilities :

a) By default inserted characters are ignored and the terminal beeps.

b) If the field has behaviour code 32768 (`DISCARD`) inserted characters are accepted, and trailing characters on the end of the row are discarded.

6.1.2 Cursor Keys (LC, RC, UC, DC)

During input, the Cursor Keys move the cursor one character position in the chosen direction. If the cursor is already positioned on the edge of the field, the default action is to move the cursor to the nearest field in the direction indicated. However, the cursor will not move if the field has behaviour code 128 (`STICK`) or there is no adjacent field in the chosen direction. Note that movement of the mouse when a button is held down is interpreted in terms of the cursor keys.

6.1.3 Jump Keys (LS, LL etc.)

These keys are analogous to the cursor keys except that (LS, RS, US, DS) move the cursor one window's worth in the corresponding direction, and (LL, RL, UL, DL) move the cursor to the limit of the data in the corresponding direction.

6.1.4 Delete (DI)

Deletes the character under the cursor. Characters to the right (if any) are shifted left one character position, and the current row is padded with a blank.

6.1.5 Shift+Delete (CT)

In all fields, except pocket-calculator and date fields, this key deletes all characters under and to the right of the cursor on the current row, and replaces them with blanks. The deleted string is stored in the paste buffer, replacing its previous contents. If the string represents a valid number, the value of the number is also stored.

For the action of this key in date and pocket- calculator fields, see Sections 6.3.2 and 6.4.2 respectively.

6.1.6 Shift+Insert (PT)

In a character field, the paste key copies the character string in the paste buffer into the current field. If the paste buffer is empty the key is ignored and the terminal beeps.

The string in the paste buffer replaces the characters at and to the right of the current cursor position. If the string in the paste buffer is larger than the space available to the right of the cursor on the current row, the excess characters on the end of the string are ignored. If the string in the paste buffer is smaller than the space available, the current row is padded with blanks.

For the action of this key in other fields, see Sections 6.2.2, 6.3.3 and 6.4.3.

6.1.7 Destructive Space and Backspace (DP, DB)

These keys replace the character under the cursor with a blank, then move the cursor in the corresponding direction. If the cursor cannot move (see above) the key is ignored and the terminal beeps.

6.2 Numeric Fields

If the field is numeric and does not have behaviour code 8192 (`CALC`), input follows the same rules as for character fields, with the following exceptions:

6.2.1 Accepted Characters

Only the following character are accepted in numeric fields; all others are ignored and cause the terminal to beep.

0-9	*Digits* 0-9	⎕*AV*[49-58]
.	*Decimal point*	⎕*AV*[47] (*i*)
,	*Triple Separator*	⎕*AV*[195] (*i*)
-	*Minus*	⎕*AV*[169]
¯	*APL Minus*	⎕*AV*[46]
	Space	⎕*AV*[32]

Table 6.2.1
Accepted Characters

(*i*) *dependent upon field type*

6.2.2 Shift+Insert (PT)

The action of the Paste Key in standard numeric fields differs from its action in character fields only in that invalid characters are not allowed. If the character string in the paste buffer contains invalid characters the entire string is ignored and the terminal beeps.

6.2.3 Validation and Formatting on Exit

When the user attempts to leave the current row in a numeric field using a movement key or an exit key, the contents are validated and reformatted according to the field type. For the data to be considered valid, it must be a well-formed number and contain no more digits than the current value of `⎕PP`. If the number includes a decimal point there must be only one of them. Triple separators are ignored so "`1,23,456.78`" in a field of type 62 is acceptable and is converted to the value `123456.78`. An entirely blank entry is considered to be invalid.

If the current row of the field is valid, the key is honoured. Before the cursor leaves the field it is reformatted according to the field type and right-justified in the row. If the data is invalid, there are two possibilities :

a) By default, the key is ignored and the terminal beeps.

or

b) If the field has behaviour code 4 (*B̲A̲D̲F̲I̲E̲L̲D̲*), `⎕SR` terminates with event code 4.

If the number contains more digits than the current value of `⎕PP` it is considered invalid and the terminal beeps. The number is reformatted showing the first `⎕PP` significant digits followed by as many underscores as there are excessive digits. For example if you entered "`123456.7890123`" and `⎕PP` was 10, the screen would display "`123456.7890___`" and beep when you attempted to leave that row of the field.

6.3 Date Fields

If the field is a date field, input follows the same rules as for character fields, with the following exceptions:

6.3.1 Punctuation

The punctuation in fixed-length date fields is automatically inserted during input. For example, if the date is "`DD/MM/YY`" entry of the date "`30/04/49`" is achieved by typing "`300449`". Punctuation is also inserted when characters are shifted as a result of using the Delete Key or of being in Insert Mode.

This feature only applies to fixed-length date fields. If the format is "`D*/M*/YY`" the punctuation is not in a fixed position on the field, so it cannot be inserted automatically.

6.3.2 Shift+Delete (CT)

If the field is a date field, this key resets the value (in `⎕SM`) in the current row of the field to 0. Note that a value of 0 in a date field displays as the **`zerodate`** string (if specified) or as the **`datespec`** string that defines the date format (See 3.5). For example, if the field type is 95 and the standard format file is in use, it will display as "`DD MON YY`". If the date is a valid date, the corresponding value is stored in the paste buffer, together with the deleted character string. If the row currently contains an invalid date, only the character string is stored.

6.3.3 Shift+Insert (PT)

If the field is a date field, the contents of the paste buffer replace the existing contents of the current row of the field. If the paste buffer contains a valid number or date, the corresponding **value** is copied in and formatted according to the field type exactly as if it has been assigned to *⎕SM*. Note that this value could have previously been cut from any type of field. Otherwise, the character string in the paste buffer is used and is copied in starting at the leftmost character position in the field. In this case, if the string is larger than the field width, the excess characters on the end of the string are ignored. If the string is smaller than the field width, the field is padded with blanks.

6.3.4 Validation on Exit

When the user attempts to leave the current row in a date field using a movement key or an exit key, the contents are validated. For the data to be considered valid, it must be a well-formed date as defined for the date format associated with the field type. See Section 3.5 for further details.

If the current row of the field is valid, the key is honoured. If not there are two possibilities. By default the key is ignored and the terminal beeps. However, if the field has behaviour code 4 (*B̲A̲D̲F̲I̲E̲L̲D̲*), *⎕SR* terminates with event 4.

6.4 Pocket-Calculator Fields

Pocket-calculator input is an option for numeric fields, and is selected using behaviour code 8192 (*C̲A̲L̲C̲*). Neither the user nor the application program can put anything other than a valid number in such a field.

Data in a pocket-calculator field is initially formatted in the same way as for a standard numeric field. However, during input the data is constantly reformatted if required so that the appearance of the field is always consistent with the values it contains.

When the cursor enters a pocket-calculator field or skips to a new row in the field, the cursor is placed in the "units" column. This is the right-most column before the decimal place. If the field contains integers, it is the right-most column in the field. If the field is of type 63 (3 decimal places) this is the fifth column from the right edge of the field.

The first valid character that you enter resets the value of the field to zero, then honours the character. At this stage the following characters are valid :

0-9	*Digits* 0-9	⎕*AV*[49-58]
. *or* ,	*Decimal point*	⎕*AV*[47] *or* ⎕*AV*[195]
-	*Minus*	⎕*AV*[169]
¯	*APL Minus*	⎕*AV*[46]

Table 6.4
Valid Characters

For example, if the field contained "`123`" and you typed "`4`", the new value in the field would be 4. If you typed "`-`", the new value in the field would be `¯0`.

Once this first character has been typed, only digits and the decimal point ("`.`" or "`,`" according to field type) will be accepted. Plus, Minus, and the APL Minus are now regarded as invalid characters.

Subsequent digits enter the field from the right, ie. the digits under the cursor and to the left move towards the left, out of the way. If the field type is 7, 7x, 8 or 8x the appropriate triple separator is inserted for you automatically. There is no need to type it; in fact it is invalid to do so.

If you type a decimal point ("`.`" or "`,`" according to the field type) the decimal point enters the field under the cursor, shifting the exiting digits to the left. Henceforth, only digits will be accepted, and a subsequent decimal point will be treated as an invalid character.

Digits continue to enter the field from the right. If the cursor is already at the rightmost position in the field, existing characters are shifted to the left as before. If however the cursor is not on the right-most column of the field, it moves to the right as each new digit is entered until it reaches the right edge. Once at the right edge, new digits cause existing data to move left as before.

Note that when you enter a decimal place or a digit after the decimal place, it may cause all the numbers in the column to shift to the left, not just the one in the current row.

`⎕SR` will not allow the entry of a number with more digits than the current value of `⎕PP`. If the user tries to enter more they are ignored and the terminal beeps.

During input, only a small number of special keys are available. All others are treated as invalid.

6.4.1 Delete Key

This key erases the digit or decimal point under the cursor. In general, the field contents shift to the right to take up the position of the deleted character.

6.4.2 Shift+Delete (CT)

This key resets the value (in ⎕SM) in the current row of the field to 0. This is displayed according to the format specified by the field type. For example, if the field type is 52 it will be displayed as "`0.00`". The value of the number is stored in the paste buffer together with the corresponding character string, and the previous contents of the buffer are lost.

6.4.3 Shift+Insert (PT)

If the paste buffer contains a numeric or date value, that value replaces the existing value in the current row of the field and is formatted according to the field type exactly as if it has been assigned to ⎕SM. This value could have previously come from a numeric field or a date field. If the paste buffer is empty or contains only a character string, the key is ignored and the terminal beeps.

6.4.4 Advantages and Disadvantages

The advantages of pocket-calculator fields are :

a) A column of numbers is always formatted in a consistent, uniform, and easy to read manner.

b) It is **impossible** for the user to enter an invalid number.

The disadvantage is that it may require a lot of keystrokes to change a digit somewhere in the middle of a large number. In such a case, use of the pocket- calculator field behaviour is probably inappropriate.

One of the important properties of pocket-calculator fields is that Cursor Left and Cursor Right do not move the cursor within the field. This is because in such fields, the cursor remains stationary and data is entered from the right; just like the way a pocket-calculator works. This means that in fields with this type of behaviour, the **only** function of Cursor Left and Cursor Right is to skip to another field.

This means that pocket-calculator fields are very useful for spreadsheet applications where it is normal to use the Cursor Keys to move between cells.

7. Interpreting the Result of Screen Read

7.1 Summary

⎕*SR* allows the user to visit the fields and to interact with the GUI objects specified in its right argument. All cursor positioning, editing, validation, etc. is handled within ⎕*SR* until one of three things happen :

a) The user presses an exit key.

b) A ⎕*SR* event occurs as a result of a specified field behaviour.

c) A weak or strong interrupt occurs (see 7.9).

d) A *TIMEOUT* event occurs (see 7.11).

e) A GUI EVENT occurs that does not have an associated callback function.

f) The ⎕*SM* window is resized (see 7.10).

At this point, ⎕*SR* terminates and returns a 6 or 9 element vector summarised in Table 7.1.1. The result contains 6 elements unless termination was caused by the user pressing a mouse button in a ⎕*SM* field, in which case it contains 9. The latter can occur only in those versions of Dyalog APL that have mouse support.

7.2 Exit Field

This is a numeric scalar or vector indicating the field or sub-field in which the exit occurred. Note that if ⎕*SR* terminated due to an *ENTER* event, this refers to the field with the *ENTER* behaviour.

7.3 Row and Column

Elements 2 and 3 of the result of ⎕*SR* contain the row and column number of the cursor when the exit occurred. These numbers represent offsets relative to the start of the data, not the window. If the data has been scrolled, you can calculate the cursor position relative to the window by subtracting the value of the home elements and adding 1.

Element		*Type*	*Description*
1	*Field*	*Numeric Scalar or Vector*	*Identifies field or sub-field from which exit occurred.*
2	*Row*	*Numeric Scalar*	*Row number of cursor relative to the start of the enclosing window from which exit occurred.*
3	*Column*	*Numeric Scalar*	*Column number of cursor relative to the start of the enclosing window from which exit occurred.*
4	*Key Code*	*Character Vector*	*Last key struck by user prior to exit.*
5	*Event Code*	*Numeric Scalar*	*Identifies reason for exit.*
6	*Changed Flags*	*Boolean Vector*	*Identifies fields that have been changed.*
7	*Field (ptr)*	*Numeric Scalar or Vector*	*Identifies field or sub-field on which mouse button was clicked causing exit.*
8	*Row (ptr)*	*Numeric Scalar*	*Row number of cursor relative to the start of the enclosing window on which mouse button was clicked.*
9	*Column (ptr)*	*Numeric Scalar*	*Column number of cursor relative to the start of the enclosing window on which*

			mouse button was clicked.

Table 7.1.1 : The result of ⎕SR

7.4 Exit Key

The fourth element of the result of `⎕SR` contains the code for the last key the user pressed before exit occurred. The only exception to this is that if a field has behaviour code 16384 (`A̲U̲T̲O̲`) coupled with code 1 (`E̲X̲I̲T̲`) or 2 (`M̲O̲D̲I̲F̲I̲E̲D̲`), pressing a key in the last character of the field will cause `⎕SR` to terminate; but the key reported will be "TB" and not the key actually pressed. This is because a TAB is generated automatically when the user presses a key in the last position of an "autotab" field.

7.5 Event Code

The fifth element of the result of `⎕SR` is an integer containing the **sum** of the event codes that caused `⎕SR` to terminate. These event codes are listed in Table 7.5.1. For example, if you allocate behaviour codes 2 **and** 4 to a particular numeric field, and the user enters some invalid data then tries to leave the field, the event number returned will be 6 (4+2). If you were to continue from the same point and this time the user simply tried to leave the field (without modifying the existing invalid data) the event number returned would be 4.

Behaviour code 16 (`E̲N̲T̲E̲R̲`) is the only behaviour code which generates a unique event, as no other events can occur at the same point.

To check if a particular event has occurred, you can use the following function `EVENT`.

```
      ∇R←E EVENT CODE
[1]   ⍝ Checks CODE is in event E returned from ⎕SR.
[2]   ⍝ Note that ⎕SR returns the sum of all the events
[3]    R←(1 2 4 8 16 32 64⍳CODE)⊃⌽(16⍴2)⊤E
      ∇
      6 EVENT 2
1
```

Code	Event
0	User pressed an exit key.
1	User attempted to exit field. This event will only occur if behaviour code 1 (EXIT) is set for the field in question.
2	User attempted to exit field after having changed it. This event will only occur if behaviour code 2 (MODIFIED) is set for the field in question.
4	User attempted to exit an invalid field. This event will only occur if behaviour code 4 (BADFIELD) is set and the field type is 9x (date), or numeric.
8	User pressed a cursor key and there is no adjacent field in the indicated direction, OR user pressed Tab in the last field OR Shift+Tab in the first field of the list specified by the right argument to ⎕SR. In the case of a row-significant field (behaviour code 4096) this applies to the last row of the last field or the first row of the first field only.
16	User attempted to enter field. This event will only occur if behaviour code 16 (ENTER) is set for the field in question.
32	User has not pressed an exit key (and no other event has occurred) in ⎕RTL seconds since the call to ⎕SR. See 7.11 for details.
64	User has pressed a mouse button in (another) field which has behaviour BUTTON.
2*17	A GUI event occurred which does not have an associated callback function

Table 7.5.1 : *Event Codes*

7.6 Changed Fields

The sixth element of the result of ⎕SR is a boolean vector indicating which of the fields specified in the right argument have changed. You may supply an original set of change flags in the Initial Context information in the left argument of ⎕SR. If so, the resulting change flags are calculated by OR-ing the flags for the fields changed this time with those you supplied. This allows you to accumulate the changes if you are going in and out of ⎕SR to process various events.

7.7 Pointer Field

This is a numeric scalar or vector indicating the field or sub-field which the user pointed to with the mouse pointer and pressed a mouse button. To cause an exit from ⎕SR this field must have behaviour B̲U̲T̲T̲O̲N̲.

7.8 Pointer Row and Column

Elements 8 and 9 of the result of ⎕SR contain the row and column number over which the user positioned the mouse pointer before pressing a button.

7.9 Interrupt

Weak interrupt (Ctrl+Break) causes ⎕SR to exit *gracefully* and return a result, but the exit key is empty and the event code is zero. Unless the weak interrupt is trapped (error 1002) execution is suspended at the start of the next executable line.

7.10 Processing Window Resize Events

If there is no *SM* GUI object defined, *⎕SM* occupies its own special Window on the screen and the user may alter the size of the *⎕SM* Window during execution of *⎕SR*. If this happens, *⎕SR* exits *gracefully* and returns a result, but the exit key is empty and the event code is zero. An APL event (1007) is generated. Unless this event is trapped, execution is suspended at the start of the next executable line. In general, your program should catch the resize using *⎕TRAP* and then reformat the contents of the *⎕SM* Window in accordance with its new size, which will be reported by *⎕SD*. If this is not possible, you may have to ask your user to make the Window larger.

If there is an *SM* GUI object defined, *⎕SM* is displayed within it. The user cannot directly resize the *SM* object, but he **can** (by default) resize the Form that contains it. This causes a Configure (resize) event on the *SM* object. There are two ways to process this. If the Configure event is not enabled, *⎕SR* will terminate as described above. However, if the Configure event is associated with a callback function, this function will be invoked to handle the resize. If you wish to prevent a resize, you can do so by setting the Sizeable property of the parent Form to 0.

The *BUDGET* function in the SMDEMO workspace provides an example of using *⎕SM* in conjunction with other GUI objects, and illustrates how resize can be handled.

7.11 Processing TIMEOUTS

TIMEOUT (Event Code 32) is supported by *⎕SR* so that you can :

a) Take some special action if the user fails to respond

b) Periodically check the status of some external source of input (e.g. get the latest stock prices)

c) Display a digital clock

... etc.

TIMEOUT is handled by *⎕SR* in a rather special way for two reasons. Firstly, unlike other events *TIMEOUT* can occur during user input in the middle of a field. At this stage the data in the field is not necessarily complete or even valid. Secondly, once your application has processed the *TIMEOUT* you want to return the user to EXACTLY the same point as before. Indeed you may not even want your user to notice that a *TIMEOUT* has occurred.

The principles of processing a *TIMEOUT* in *⎕SR* are as follows :

1. Set *⎕RTL* to the number of seconds before the *⎕SR* is to *TIMEOUT*.

2. *Give* the *⎕SM* form to the user by calling *⎕SR*.

3. When *⎕SR* exits, check the event code. If the exit is due to a *TIMEOUT*, the result will contain 32 plus the sum of any other "events", for example *BADFIELD* (4). Save the entire result from *⎕SR* (*CONTEXT*). Note that the 4th element of the result (exit key) is empty.

4. Take whatever action you want.

5. Call (re-enter) *⎕SR* again, giving the previous result (*CONTEXT*) as the second element of the left argument. The act of "feeding back" the event code 32 is taken to mean "continue where you left off". The normal initialisation that occurs at the start of a call to *⎕SR* is not performed.

6. Unless you reset *⎕RTL* to zero, the process is automatically repeated until the user exits.

NOTE: When *⎕SR* exits with a *TIMEOUT*, normal field validation is bypassed. At the point of *TIMEOUT*, the data in a numeric or date field may be invalid. If so, the corresponding element of *⎕SM* will contain character data (the invalid field contents) and not a number. Data in a numeric field with *CALC* behaviour is always valid and is always represented in *⎕SM* as a number.

8. Miscellaneous

8.1 Overlapping Fields

⎕SM allows you to define overlapping fields. This can be very useful because you can define the background to a form as a single field occupying the entire ⎕SM Window, then overlay your input fields on top.

When a form is displayed, the fields in ⎕SM are drawn in the order in which they are defined in ⎕SM. This means that a field defined by `⎕SM[I;]` potentially obliterates all or part of fields defined in `⎕SM[⍳I-1;]` if they overlap.

If you allow the user to visit a field which is behind and therefore obliterated by another, ⎕SR will behave exactly as if the field were visible except that the hidden field's contents will not be shown. This feature can be used for password fields if your terminal does not support the invisible video attribute. Beware however that skipping between overlapping fields using the Cursor Keys may produce strange effects. To avoid this problem, it is suggested that you use behaviour code 128 (STICK) for such fields.

In many instances it is possible to produce the same effect of localising ⎕SM by catenating new rows onto it, and then later dropping them off. This technique relies on the fact that fields obliterate any that are defined earlier in ⎕SM that they overlap.

8.2 Localising ⎕SM

The localisation of ⎕SM is one of the most important and useful features of the Screen Management facilities.

When you assign a new value to a localised ⎕SM, the previous display remains unchanged, and the new form is overlaid on top. When the function with the local ⎕SM terminates, the display reverts to its original state, before the function was called. If the localised ⎕SM occupies a space that is smaller than the ⎕SM Window, the effect is of a pop-up sub-window.

A further consideration is a Dyalog APL feature known as 'pass-through' localisation. In several other APL implementations, a reference to a localised system variable before it is assigned a value is an error. In Dyalog APL, a localised system variable takes its initial value from the calling environment (⎕TRAP excepted). This means that if you call a function in which ⎕SM is localised, it inherits the value it had before the function was called until you assign a new value to it. This is one reason the Window does not immediately clear when you localise ⎕SM.

8.3 Blanking the Window

Section 8.2 described how the effect of localising `⎕SM` is to overlay a new form on top of the one (or ones) already displayed in the `⎕SM` Window. This means that parts of the old form "show through" the parts of the new one that are unoccupied by fields. This is often undesirable.

A practical solution to this problem is to define a single field located at (1,1) with a window size of `⎕SD`, containing a single space character. This field must appear as the **first** row of the localised `⎕SM`.

8.4 Vector Assignment to ⎕SM

In general, `⎕SM` is a matrix with between 3 and 13 columns. In early trials it was discovered that the most common cause of errors when using `⎕SM` was in forgetting this. `⎕SM` will therefore accept a suitable vector which is assigned to it. When referenced it will always yield a matrix.

Example :

```
      ⎕SM←'Hello World' 10 12
      ⍴⎕SM
1 3
      ⎕SM⍪←42 10 30
      DISP ⎕SM

┌───────────┬──┬──┐
│Hello World│10│12│
├───────────┼──┼──┤
│    42     │10│30│
└───────────┴──┴──┘
```

Notice however that this does not apply when a field is nested as there would be a confusion with the case of a default numeric field which contains text in its first row.

8.5 Oversize Windows

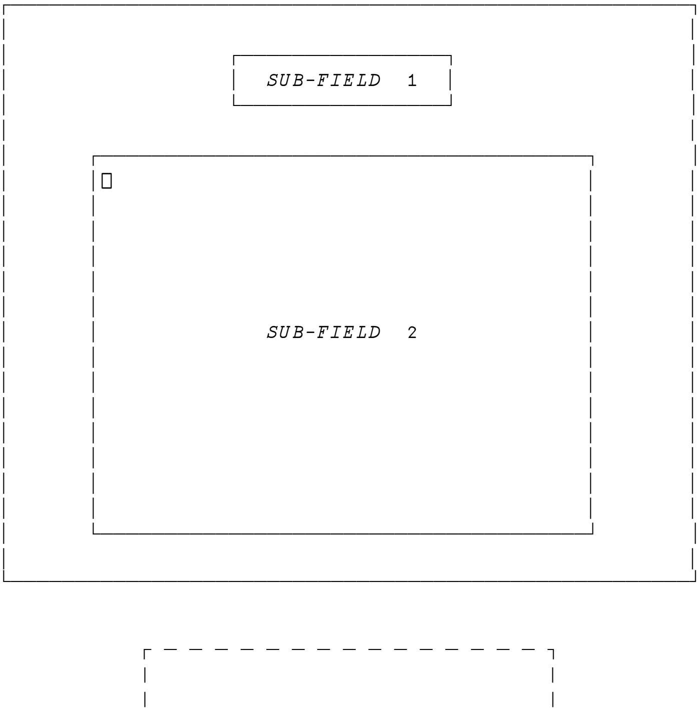
If a window is larger than the space required for the field contents, `⎕SM` does not extend the data with blanks or zeros. The use of oversize windows can lead to unexpected effects. For example, `⎕SR` will not allow the user to overtype in the area between the data and the window. The use of oversize windows is not recommended; you can use `↑` or Reshape (`⍴`) to make your data fit the window as you see fit.

8.6 Scrolling Whole Forms

The scrolling algorithm discussed in Section 5.4 works well for regular forms, i.e. forms in which the sub-fields are laid out in a uniform manner. It also produces good behaviour in non-uniform forms, although clipping of some fields is inevitable. However, if the first **input** sub-field is not positioned at (1,1) in the window, scrolling back to the beginning of the form may not have the desired effect. Consider the function *BIGFORM*1. Sub-field 1 contains a title, so the ⎕*SR* only asks the user to visit sub-fields 2 and 3. Figure. 8.6.1 illustrates the initial display. Figure. 8.6.2 shows the result of pressing TAB twice. As you can see, ⎕*SR* has scrolled sub-field 3 to the top of the Window leaving sub-field 2 out of view.

```
     ∇ BIGFORM1;⎕SM;SM
[1]    SM←1 7⍴'FORM TITLE' 2 35 0 0 0 0
[2]    SM⍪←(20 60⍴⎕A)5 10 0 0 0 0
[3]    SM⍪←(10 40⍴⎕D)33 20 0 0 0 0
[4]    ⎕SM←SM 1 1
[5]    ⎕SR(1 2)(1 3)
     ∇
```

Fig. 8.6.1 Initial Position



```
|          SUB-FIELD  3            |
|                                  |
|                                  |
|                                  |
└ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _┘
```

Fig. 8.6.2 Position after TAB

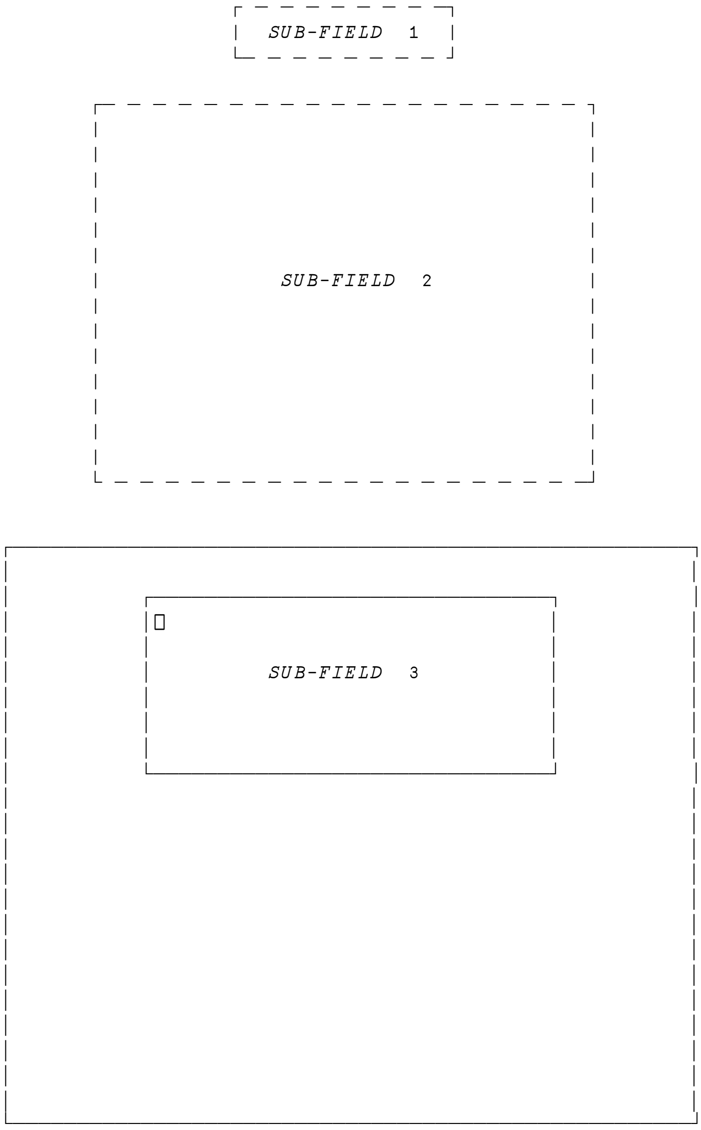
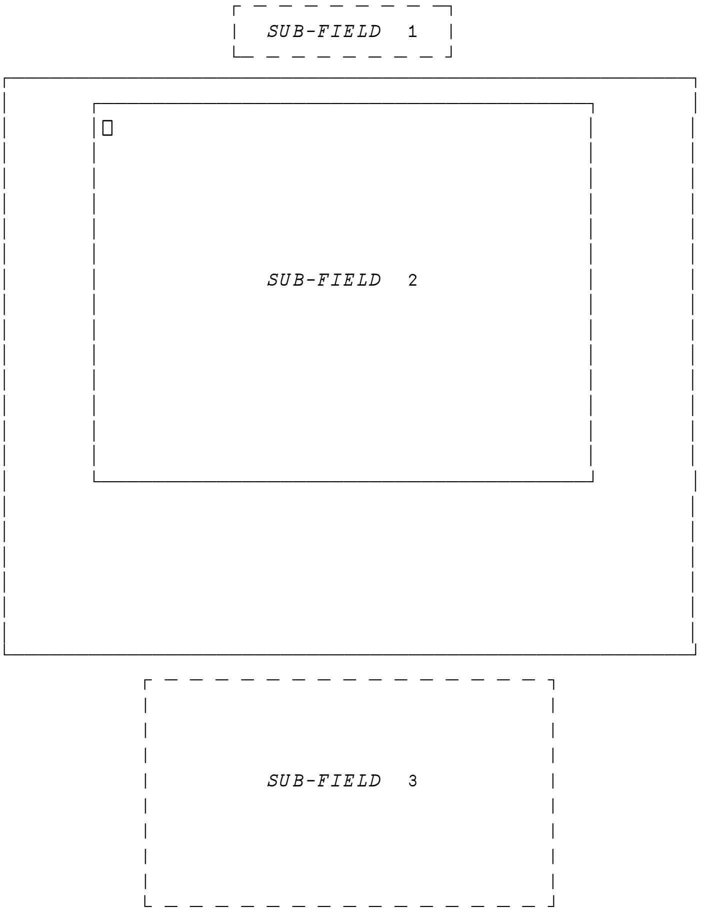


Fig. 8.6.3 Position after Second TAB



As you can see, when the user pressed TAB in sub-field 3, ⎕*SR* scrolled sub-field 2 (the next field to visit) into view. However the scrolling algorithm has left the sub-field out of view. In this case such behaviour is undesirable.

The solution to this problem is to include a 'dummy read' on sub-field 1. This will ensure that it is scrolled back onto the form when the user presses TAB in sub-field 3. As we don't actually want the user to access sub-field 1, entry to this field is trapped using behaviour 16 (*ENTER*) and the cursor re-positioned on sub-field 2.

The function `BIGFORM2` makes the form scroll as we want. The initial position and the position after a TAB is the same as before (Figures. 8.6.1 and 8.6.2). However, a second TAB causes the Window to revert to the initial position (Figure. 8.6.1) with sub-field 1 displayed in the Window as we want.

```
      ∇ BIGFORM2;⎕SM;SM;Z;KEYS;CURSOR;E̲N̲T̲E̲R̲
[1]     E̲N̲T̲E̲R̲←16
[2]     SM←1 7⍴'FORM TITLE' 2 35 0 0 0 E̲N̲T̲E̲R̲
[3]     SM⍪←(20 60⍴⎕A)5 10 0 0 0 0
[4]     SM⍪←(10 40⍴⎕D)33 20 0 0 0 0
[5]     ⎕SM←SM 1 1
[6]     KEYS←'ER' 'EP'
[7]     CURSOR←(1 2)1 1
[8]    Again:Z←(KEYS CURSOR)⎕SR(1 2)(1 3)(1 1)
[9]     →(Z[5]=E̲N̲T̲E̲R̲)/Again
      ∇
```

The same sort of procedure may be required whenever the first input sub-field in a scrolling window is not positioned at the origin. If you do not have a conveniently positioned output sub-field to use, it is a good idea to define a dummy one at (1,1).

8.7 Blanks & Invalid Numerics

Free-format numeric fields are slightly unusual in that they may contain character strings rather than numbers. The reasons for this are twofold.

Firstly, it is desirable to have a numeric field that may initially be blank. One solution is to have a behaviour "display 0 as blank". However, it is also useful to distinguish between the 0 which has been typed by the user and the 0 displayed as blanks by the application program. As `⎕SR` is re-entrant, it would be difficult if not impossible to do this if the "display 0 as blanks" solution had been adopted.

The second requirement stems from the fact that it is possible for the user to enter an invalid number, for example "`1.2.3`". Of course, your program can rely on the default behaviour which is to beep and not allow the user to leave that field. However, you may prefer to issue your own error message, and you certainly do not want the invalid contents of the field lost. It is essential therefore that `⎕SM` accurately reflects the contents of the field at all times, even when the contents cannot be converted into a number.

There are only two ways in which you can get character strings into numeric fields. One way is for your program to put them there; the other is by using the `B̲A̲D̲F̲I̲E̲L̲D̲` behaviour. Both are under your control. If you choose not to put blanks into numeric fields and you take the default behaviour which prevents the user from leaving an invalid field, you will not have to allow for character strings in numeric fields in your programs.

If you choose to allow characters in numeric fields, it is ESSENTIAL that you explicitly define the field type to ⎕*SM*. If you use field type 0 with data of mixed type, the results will be unpredictable.

8.7.1 Blanking a Numeric or Date Field

Each element of a numeric or date field can either be a number or an enclosed character scalar/vector.

To set a date field to blanks it is recommended that you use the **`zerodate`** facility.

To a particular row of a numeric field to blanks, you simply assign a space character to the appropriate element.

Example :

```
⍝ 4th row of field is blank
⎕SM←(42.5 0 ' ' 99) 10 10 0 0 52

⍝ 4-row numeric field, 5 blank cols
⎕SM←(4⍴⊂5⍴' ') 10 10 0 0 60
```

Note that elements of the field contents are assigned 1 per row, thus ...

```
⍝ numeric field containing 4 blank elements
⍝ shown in a 2-row window.
⎕SM←(4⍴' ') 10 10 2 8 5
```

Unless the width of the window is defined explicitly, its width will be determined by the widest element.

Note also that an entirely blank row in a numeric field is regarded as invalid. Thus if the user moves into a field that you have filled with blanks, he will by default have to enter valid numbers before leaving the form. An entirely blank row in a date field will also be invalid unless the corresponding **`zerodate`** string is also blank.

8.7.2 Handling Invalid Data

By default, the user will remain stuck in a numeric or date field until it contains a valid number or date. If the user tries to skip to another field or presses an exit key, the terminal will beep and the key will be ignored. However, if the field has behaviour code 4 (*B̲A̲D̲F̲I̲E̲L̲D̲*) ⎕*SR* will exit when the user attempts to leave it. At this point ⎕*SM* contains the invalid data.

Suppose that you have a field (field 3, say) containing a single number, and the user has entered "`1.2.3`", then pressed TAB. Assuming that you have given the field behaviour code 4 (*B̲A̲D̲F̲I̲E̲L̲D̲*), `⎕SR` will exit returning in the first 4 elements of its result:

```
      3 1 6 'TB' 4
```

From this information, your program can deduce that field 3 is invalid. Furthermore `⎕SM[3;1]` contains the invalid text, in this case "`1.2.3`". You can therefore issue an error message and call `⎕SR` again, e.g.

```
      (⊃⎕SM[3;1]),' is not a number - Please re-enter'
```

Suppose that the field contained 3 rows. This time `⎕SR` might return ...

```
      3 2 6 'DC' 4
```

... indicating that the second number is invalid. The invalid text is obtained by the expression :

```
      2⊃⎕SR[3;1]
```

8.8 National Language Characters in Dates

The file `DEFAULT.DFT` in the `APLFMT` sub-directory contains user-definable specifications for date formats. Interpretation of this file does not use the Input Translate Table mechanism. Instead, you can code `⎕AV` positions directly into the file by prefixing the `⎕AV` index with a backslash. Note that the `⎕AV` index is expressed as a 3-digit decimal number in origin 0, ie "`\091`" means `⎕AV[91]`.

Suppose that é is `⎕AV[148]`. To get the é into Février, you would code the first 2 month names as follows:

```
M:Janvier   F\148vrier
```

The colon character ("`:`") is used as a separator in the date coding scheme. You cannot therefore use a colon within the **`zerodate`** string (see 6-20). A colon may instead be specified by its position in `⎕AV` as "`\240`".

9. Support Workspaces

9.1 SMTUTOR

SMTUTOR contains an on-line tutorial designed to introduce you to the Screen Manager.

It is organised into a series of "BOOKS" that should be read in sequence.

BOOK 1 : Introduction to the Dyalog APL Screen Manager

An overview of the Screen Manager, describing the basic concepts, and discussing a simple example of its use.

BOOK 2 : Defining the Screen Map

Covers the full definition of *⎕SM*, and the different ways in which it can be used.

BOOK 3 : Using and tailoring the Screen Read

Covers the full definition of *⎕SR*, and the different ways in which it can be used.

BOOK 4 : Cursor Movement

Covers the default behaviour of the cursor movement keys, that is, ↑, ↓, ←, →, TAB and BACKTAB.

Each book in the series contains a set of lessons designed to introduce you to the Dyalog APL Screen Manager. These lessons are best run on a colour screen.

The lesson consist of a sequence of pages containing sample Dyalog APL expressions together with suitable instructions and commentary.

To read a given book, type *BOOK n*, where *n* is the appropriate number, and the first lesson will be printed. To run the next lesson, type *NEXT* or press F1. To recall the previous lesson, type *LAST* or press F2. To review a particular lesson, type *LESSON n*, where *n* is the lesson number.

Each lesson consists of a screenful of text. In general, instructions and commentary are in lower case; expressions to be executed are in upper case and indented 6 spaces. You don't have to re-type the expressions provided. Instead you can use the Dyalog APL Session Manager.

To execute an expression, point at it with the cursor, and press the Enter key. This can be done using the mouse by putting the pointer on the expression, clicking the left button (to position the cursor) and then clicking the middle button.

The expression is copied down to the current line and executed. Once you have gained experience, you can experiment by editing the expression prior to submission. Expressions must be executed in the order provided.

To begin ...

```
      )LOAD SMTUTOR

      BOOK 1

      BEGIN
```

9.2 SMDEMO

`SMDEMO` contains some examples which illustrate the use of the Dyalog APL Screen Manager (`⎕SM` and `⎕SR`). The examples consist of a sequence of lessons containing sample Dyalog APL expressions together with suitable instructions and commentary. The examples are best run on a colour screen.

To start, type `BEGIN`. To run the next lesson, type `NEXT` or press F1. To recall the previous lesson, type `LAST` or press F2. To review a particular lesson, type `LESSON n`, where `n` is the lesson number.

Each lesson consists of a screenful of text. In general, instructions and commentary are in lower case; expressions to be executed are in upper case and indented 6 spaces. You don’t have to re-type the expressions provided. Instead you can use the Dyalog APL Session Manager.

To execute an expression, point at it with the cursor, and press the Enter key. The This can be done using the mouse by putting the pointer on the expression, clicking the left button (to position the cursor) and then clicking the middle button.

The expression is copied down to the current line and executed. Once you have gained experience, you can experiment by editing the expression prior to submission. Expressions must be executed in the order provided.

9.3 SMDESIGN

This is a workspace to design simple `⎕SM` screens. The main function is `SMDESIGN` whose argument is either a `⎕SM` definition matrix or an empty vector. The result is the `⎕SM` definition matrix for the new screen.

To design a new screen :

```
      SM ← SMDESIGN ''
```

To modify an existing design :

```
      SM ← SMDESIGN SM
```

The facilities provided which are described in detail below are as follows :

Key	Description
F1	Create a new field
F2	Move/Resize Current Field
F3	Delete Current Field
F4	Edit Contents of Current Field
F5	Change Attributes of Current Field
F6	Change Default Field Attributes
F7	Toggle Field Outlines
F8	Toggle Help Message

Table 9.3
SMDESIGN Facilities

9.3.1 Selecting and Locating Fields

`SMDESIGN` is intended to help you to design simple forms; the type of forms used for data entry. Forms of this type typically contain overlapping fields. For example, a data-entry form might contain 1 field for the *background* text, with several other input fields overlaid on top.

To cater for overlapping fields, *`SMDESIGN`* uses boxes to locate the fields as you move around the screen.

Initially, *`SMDESIGN`* displays your screen as it would appear in use, but with two modifications. Firstly, the CURRENT field (ie. the field containing the cursor) is identified by a pink outline. Note that the outline hides the contents of the first and last row and the first and last column of the field. If the field contains only 1 row or column, all you see is the pink line. Secondly, a 2-line HELP message is overlaid at the bottom of the display.

If you press F7, ALL the fields in your form will be shown in outline (using the colours allocated to the field). Press F7 again to return to the normal display. In either case, the CURRENT field is outlined in pink.

As you move the cursor around the screen, the CURRENT field changes according to its position. Note that using the CTRL key in conjunction with the arrow keys let you move the cursor in larger increments.

9.3.2 Creating New Fields

To create a new field, move the cursor to where you want the field to start, and press F1. The new field appears with its top-left corner at the cursor position. The field is outlined in yellow, and initially has the default size (normally 1 row, 1 column - but you can change this by changing the default field attributes).

The arrow keys are initially used to change the new field's size. Ctrl with the arrow keys changes the size by larger increments, and can be used to quickly set up a large field. If you press F1, the arrow keys move the field rather than resize it. Again, Ctrl can be used to move in larger increments. Pressing F1 again sets the arrow keys back to *resize mode*.

When you have positioned and sized the field as you want it, press Esc. If you decide to abandon the create, press Ctrl-D.

If you exit and save the new field (Esc) it becomes the CURRENT field, and is outlined in pink.

9.3.3 Moving and Resizing a Field

To move and/or resize the CURRENT field (ie the one outlined in pink) press F2. The colour of the outline changes to yellow. You move or resize the field by pressing the cursor keys. As you do so, the yellow outline indicates the new field size and position. The pink outline shows its original location.

The arrow keys are initially used to change the new field's size. Ctrl with the arrow keys changes the size by larger increments, and can be used to quickly set up a large field. If you press F1, the arrow keys move the field rather than resize it. Again, Ctrl can be used to move in larger increments. Pressing F1 again sets the arrow keys back to "resize mode".

When you have re-positioned and resized the field as you want it, press Esc. The yellow outline indicating the new field location will disappear, and be replaced by the pink one. If you decide to abandon the change by pressing ^D the original location and size will remain unaltered.

9.3.4 Deleting a Field

To delete the CURRENT field (ie the one outlined in pink) press F3. You will be asked to confirm or cancel the deletion. If you confirm, the field will be deleted. If the deleted field was originally superimpose an another field, the background will become the CURRENT field and will be outlined in pink. If not, there will be no CURRENT field.

9.3.5 Editing Field Contents

To edit the contents of the CURRENT field (the one outlined in pink), press F4. The pink outline will disappear and the field will be displayed using its ACTIVE VIDEO (if defined). The cursor will remain where it was.

You may now type data into the field. However the data must be valid. If the field is of type NUMERIC or DATE you may enter only valid numbers or dates. (If you want to put character data into a numeric or date field, you may do so by first defining the field type to be CHARACTER, then entering the data, then changing the field type to NUMERIC or DATE.)

9.3.6 Field Attributes

To change the attributes of the CURRENT field (the one outlined in pink), press F5. A pop-up form appears showing the current attributes of the field.

The window size and position, home elements and scrolling groups, are shown as numbers. To change one of these items simply type in a new value. For example, you can move the field to a new location by typing new values for "Top Row" and "Left Column".

The field type, behaviour, video, and active video are shown not as numbers, but by more helpful descriptive text. For example, if the field is type 62 the description against "Type" will be "Numeric 9,999.99"; indicating that the field is numeric and is displayed with 2 decimal places and with a "," between triples.

To change field type, behaviour, video, or active video, place the cursor on the appropriate row of the pop-up form and press Enter. This will bring up a set of pop-up menus from which you make your selection.

Selecting Field Type

The first menu offers you the choice of DEFAULT, CHARACTER, NUMERIC or DATE field types. To choose, move the cursor to the item you want, and press Enter.

If you select CHARACTER, you will get a second menu asking you to choose between NO TRANSLATION, UPPERCASE TRANSLATION, or LOWERCASE TRANSLATION. Pick UPPERCASE TRANSLATION if you always want characters in the field to appear in uppercase. Pick LOWERCASE TRANSLATION if you always want lowercase. These field types are useful if you are going to do a table look-up with the user's input, but you don't want to force entry in a particular case.

If you select NUMERIC, you will get a second menu asking you to choose the number of decimal places. Select ANY DECIMALS if you want to allow the user to be able to input any number of decimal places, or if you want output values shown in full precision. If not, move the cursor to the second row of the menu and enter the number of decimals you require. If you want the field to contain integers enter "0" decimals. Finally, a third menu will appear for you to choose which format you want.

If you select DATE, you will get a second menu listing the various date formats defined in your current APL FORMAT file. Move the cursor to the one you want and press Enter. See GETTING DATE FORMATS below.

Selecting Field Behaviour

The field behaviour menu offers you a list of the different behaviours that are available. Behaviours are additive. Those that have been selected for your field are highlighted in red. You can turn a behaviour on or off by positioning the cursor on it and pressing Enter.

When you have made your selection, press Esc.

Selecting Field Video

The first menu asks you to choose foreground colour. The cursor is initially positioned on the colour currently selected. Position the cursor on the colour of your choice and press Enter. If you quit from this menu the currently selected colours will be unaltered.

After choosing a foreground colour, you will get a menu offering a choice of background colours. Again, the cursor will initially be positioned on the currently selected background colour. If you quit from this menu, you will return to the Foreground Colour menu.

Selecting Active Video

First you will be offered the choice of having no active video (i.e. the field appearance remains unaltered when the cursor enters the field) or of defining an active video. If you select the latter you will be prompted to define the foreground and background colours in the manner described above.

9.3.7 Default Field Attributes

When you create a new field, it initially assumes the default field attributes. These are stored in the workspace in a global variable `def_field_attr`.

You can change the default attributes by pressing F6. This gives you a pop-up menu similar to the one used for changing the attributes of the CURRENT field, except that it does not contain field position. The position of a newly created field is determined by the position of the cursor.

With this one exception, the procedure for changing default field attributes is the same as for changing the attributes of individual fields, which is described in Section 9.3.6.

9.3.8 Field Outlines

By default the form is displayed exactly as it would be viewed by the user, with only the CURRENT field identified by a pink outline. If you have overlapping fields and/or empty input fields with the same background colour as the rest of the form, it is difficult to see where the fields are located.

If you press F7, all the fields on the form are shown in outline; their contents are not visible. The colour of the outline is the same as that defined for the field. If you press F7 again, the outlines are removed.

The choice of how you want the fields to be displayed is yours; the facilities described above (move/resize, delete, attributes, etc.) work in the same way in both cases.

9.3.9 Toggling the Help Message

At all stages of the design process, a short 'help' message, indicating which keys perform which actions, is displayed at the bottom of the screen. This message may hide or partially hide some of your fields. Press F8 to obtain a menu allowing you to position the help message at the **Bottom** of the screen, the **Top** of the screen, or to turn it **off**.

You may only toggle the help message at the top-level *Design* screen.

CHAPTER 9

The AP124 Emulator

Introduction

The Full Screen Data Manager provides facilities for the design and management of user interface panels. It can be used to implement menus, forms, spreadsheets and editors. It is compatible with IBM's AP124 and AP126.

The Full Screen Data Manager is implemented as an Auxiliary Processor called *prefect* , which defines the basic functions which drive the screen. An associated workspace *PREFECT* supplies higher level cover functions, plus facilities for panel design.

For new applications, this software has been superceded by ⎕*SM*/⎕*SR* and by the GUI Support described in Interface Guide. It is supplied under MS-Windows to provide compatibility with existing applications, and to provide a rapid conversion path from mainframe APL applications.

Under MS-Windows, the *PREFECT* screen is implemented as a single non-sizeable window.

Overview

Panels

A user interface panel consists of a number of rectangular fields with a variety of attributes. A field may contain data, and can be identified by number or name. Up to 80 panels may be managed concurrently.

A field is defined by the following attributes:

Location	Location on the screen, defined by row and column position of the top left hand corner.
Shape	Defined by number of rows and columns.
Type	Determines the type of data that may be displayed or entered into the field.
Video	Specifies video characteristicsof field.
Protect	Determines whether the field is protected or not. If a field is not protected, then the user may enter data into it, of type determined by the 'type' attribute.
Colour	Specifies foreground and background colours of the field.

Example:

```
                              Panel
                                |
                                v
                   _____________________
                  |   ____   _________  |
                .--->|Name| |         |<--Input field,
                | |  |____| |_________| | red on blue
Output fields  -| |                     |
with background | |   ___    ___        |
text defined    .--->|Age|  |   |<--------Numeric
                  |  |___|  |___|       | input field,
                  |                     | yellow on red
                  |   ________________  |
                  |  |                | |
                  |  |________________|<--Output field
```

```
                                |_____________________| for messages
```

Basic Principles

Prefect works on the principle of deferred screen update. The APL program passes all panel definitions to prefect; these definitions are held in prefect's own internal memory or stack. The user sees no change to his screen until the APL program signals prefect which panel is to be displayed. Prefect displays that panel, and allows the user to enter data into any unprotected fields. The panel image held in prefect's internal memory reflects any changes made to the screen. The user tells prefect when he has finished and prefect in turn tells the APL program. If the APL program wants to know about any changes made to the panel, it must then ask prefect.

The following example is given in pictorial form, to illustrate the flow of control and information between the APL program, prefect and the user's screen.

Step 1:

The APL program passes several panel definitions to prefect. By default, the last panel specified is the 'active panel':

```
                  __________________
Panel 1 -------->|     MENU         |<---Main menu
               ___________________  |    panel
Panel 2 ----->|  Help for Add     |<-----Help panel
            ____________________  | |
           |                    | | |
           |  ____    _______   | | |
           | |Name|  |       |  | | |
           | |____|  |_______|  | | |
Panel 3 -->|  ___     __        |<-------Data entry
(Active    | |Age|   |  |       | |_|    panel
 panel)    | |___|   |__|       |_|
           |  ________________  |
           | |                | |
           | |________________| |
           |____________________|

           ↑‾‾‾‾‾ prefect
           |
           |
   Panel definitions
           ↑
   APL Workspace              User
```

Step 2:

APL program tells prefect to display the active panel to the user:

```
                ________________
               |     MENU         |
             ___________________  |
            |  Help for Add   | |
          ____________________  | |
         |                    | | |
         |  ____    _______   | | |
         | |Name|  |       |  | | |
         | |____|  |_______|  | | |
         |  ___     __        | | |
         | |Age|   |  |       | |_|
         | |___|   |__|       |_|
         |  ________________  |
         | |                | |
         | |________________| |
         |____________________|

         ↑-----  prefect ------|
         |                     |
         |                     ↓
Display active panel        Panel
         ↑                     ↓
    APL Workspace           User
                      _____________________
                     |   ____   _________  |
                     |  |Name| |         | |
                     |  |____| |_________| |
                     |   ___    ___        |
                     |  |Age|  |   |       |
                     |  |___|  |___|       |
                     |   ________________  |
                     |  |                | |
                     |  |________________| |
                     |_____________________|
```

Step 3:

The user enters data into the input fields; all input to the screen is mirrored in prefect's internal picture of the screen. The user may edit his data, then when he is satisfied, he tells prefect that he has finished with his data entry. Prefect in turn tells the APL program, and passes control back again.

```
             ___________________
            |     MENU          |
          ____________________  |
         |  Help for Add      | |
       _____________________  | |
      |                     | | |
      |  ____    _______    | | |
      | |Name|  |Pauline|   | | |
      | |____|  |_______|   | | |
      |  ___     __         | | |
      | |Age|   |21|        | |_|
      | |___|   |__|        |_|
      |  _________________  |
      | |                 | |
      | |_________________| |
      |_____________________|

      ↓----- prefect ------|
      |                     |
      |                     ↑
   finished              finished
      ↓                     ↑
APL Workspace             User
                   ______________________
                  |   ____   __________  |
                  |  |Name| |Pauline   | |
                  |  |____| |__________| |
                  |   ___    ___         |
                  |  |Age|  |21 |        |
                  |  |___|  |___|        |
                  |   _________________  |
                  |  |                 | |
                  |  |_________________| |
                  |______________________|
```

Step 4:

The APL program might then ask prefect for the contents of any changed fields. These are supplied as a nested vector, one element per field requested.

```
            __________________
           |     MENU         |
         ___________________  |
        |  Help for Add      | |
      ____________________   | |
     |                    |  | |
     |  ____    _______   |  | |
     | |Name|  |Pauline|  |  | |
     | |____|  |_______|  |  | |
     |  ___     __        |  | |
     | |Age|   |21|       |  |_|
     | |___|   |__|       |_|
     |  ________________  |
     | |                | |
     | |________________| |
     |____________________|

     ↓¯¯¯¯¯ prefect
     |
Changed fields
     |
APL Workspace
     |
'Pauline' 21            User
                  _____________________
                 |   ____   _________   |
                 |  |Name| |Pauline  |  |
                 |  |____| |_________|  |
                 |   ___    ___         |
                 |  |Age|  |21 |        |
                 |  |___|  |___|        |
                 |   ________________   |
                 |  |                |  |
                 |  |________________|  |
                 |______________________|
```

Step 5:

The APL program validates the input, and issues an appropriate error message if necessary.

```
                  __________________
                 |     MENU         |
               ___________________  |
              |  Help for Add     | |
            ____________________  | |
           |                    | | |
           |  ____    _______   | | |
           | |Name|  |Pauline|  | | |
           | |____|  |_______|  | | |
           |  ___     __        | | |
           | |Age|   |21|       | |_|
           | |___|   |__|       |_|
           |  ________________  |
           | |TOO YOUNG !!    | |
           | |________________| |
           |____________________|

           ↑----- prefect ------|
           |                    |
Put error message in            |
message field and give          |
control of screen back          |
    to the user                 |
           ↑                    |
     APL Workspace            User
Set up error message            ↓
                        _____________________
                       |   ____   _________  |
                       |  |Name| |Pauline  | |
                       |  |____| |_________| |
                       |   ___    ___        |
                       |  |Age|  |21 |       |
                       |  |___|  |___|       |
                       |   ________________  |
                       |  |TOO YOUNG !!    | |
                       |  |________________| |
                       |_____________________|
```

Step 6:

The user does not understand what he has done wrong, and asks prefect for more help. Prefect tells the program, which has to decide what to do next.

```
                __________________
               |     MENU         |
             ____________________ |
            |  Help for Add     | |
          ____________________  | |
         |                    | | |
         |  ____    _______   | | |
         | |Name|  |Pauline|  | | |
         | |____|  |_______|  | | |
         |  ___     __        | | |
         | |Age|   |21|       | |_|
         | |___|   |__|       |_|
         |  _________________ |
         | |TOO YOUNG !!    | |
         | |________________| |
         |____________________|

         ↓----- prefect ------|
         |                    |
User needs help             Help!
         ↓                    ↑
   APL Workspace            User
                      _____________________
                     |   ____   _________  |
                     |  |Name| |Pauline  | |
                     |  |____| |_________| |
                     |   ___    ___        |
                     |  |Age|  |21 |       |
                     |  |___|  |___|       |
                     |   ________________  |
                     |  |TOO YOUNG !!    | |
                     |  |________________| |
                     |_____________________|
```

Step 7:

The program asks prefect to make the HELP panel the active panel, and to give this to the user.

```
             __________________
            |     MENU          |
          ___________________   |
         | Name   Pauline    |  |
       ____________________  |  |
      |                    | |  |
      |    Help for Add    | |  |
      |                    | |  |
      | Age must be in the | |  |
      |   range 30 to 70   | |  |
      |                    | |_|
      |                    |_|
      |                    |
      | Press any function |
      |   key to continue  |
      |____________________|
      ↑----- prefect ------|
      |                    |
 Make help panel           |
 active and give           |
 it to the user            |
      ↑                    ↓
APL Workspace            User
                   _____________________
                  |                     |
                  |    Help for Add     |
                  |                     |
                  | Age must be in the  |
                  |   range 30 to 70    |
                  |                     |
                  |                     |
                  | Press any function  |
                  |   key to continue   |
                  |_____________________|
```

Step 8:

User reads help message, and then continues. Prefect tells program, which in turn asks prefect to make the input panel active again and give it back to the user.

```
             __________________
            |     MENU         |
          ___________________  |
         |  Help for Add     | |
       ____________________  | |
      |                    | | |
      |  ____    _______   | | |
      | |Name|  |Pauline|  | | |
      | |____|  |_______|  | | |
      |  ___     __        | | |
      | |Age|   |21|       | |_|
      | |___|   |__|       |_|
      |  ________________  |
      | |TOO YOUNG !!    | |
      | |________________| |
      |____________________|

      ↑----- prefect ------|
      |                    |
Give input panel           |
  back to user             |
      ↑                    ↓
APL Workspace            User
                   ______________________
                  |   ____   _________  |
                  |  |Name| |Pauline  | |
                  |  |____| |_________| |
                  |   ___    ___        |
                  |  |Age|  |21 |       |
                  |  |___|  |___|       |
                  |   ________________  |
                  |  |TOO YOUNG !!    | |
                  |  |________________| |
                  |_____________________|
                         User
```

Step 9:

Steps 4 and 5 are repeated, until either the data is acceptable, or the user decides to quit.

Updating the Panel

Data Entry

When a panel is displayed on the screen, the user can enter data into any unprotected field. Text fields accept any data; numeric fields accept characters from the set '`0123456789 ¯-,.`'. Protected fields reject data. The bell sounds if the user types whilst the cursor is positioned within a protected field.

The user may move around the screen and edit his data using a variety of commands. The actual keystrokes required to execute a particular command depends on the way in which the keyboard for the terminal has been defined. (See *Prefect and your terminal* for details of the requirements).

Given below is a description of the command, followed by the input code which identifies the command.

Cursor Movement:

Up one line	UC	Up Character
Down one line	DC	Down Character
Left one character	LC	Left Character
Right one character	RC	Right Character
Next unprotected field	TB	Tab
Last unprotected field	BT	Shift+Tab
Previous character	NB	Backspace
First unprotected field	HO	Ctrl+Home
First unprotected field on next line	ER	Enter

Editing:

Delete character	DI	Delete
Delete to end of field	CT	Shift+Delete
Toggle Insert Mode	IN	Insert
APL characters		Ctrl+n
Ascii characters		Ctrl+o

Prefect has some limitations on editing; no automatic wrapping is performed on insert and no automatic scrolling is performed.

Different field types have different rules for input; see section entitled Panel Definitions in Detail for more information.

Screen Control

The user tells prefect that he has finished by the use of function keys. Any function key terminates user input; however, prefect tells the APL program exactly which one was pressed, so that it can take the appropriate action. Up to 256 function keys may be set up for use with prefect; all of these may in turn be moded, thus allowing the possibility of an almost limitless number to be set up. The function keys are defined in your keyboard Input Table.

It is possible to define panels that contain no input fields. Such panels are often used for control or menu screens, to determine the next step that the user wishes to take.

Example:

```
-------------------------
| --------------------- |
| |      Options      | |
| |                   | |
| |     F1: Update    | |
| |     F2: Report    | |
| |     F3: Quit      |<--- One protected field
| |     F4: Select    | |
| |                   | |
| | Use function keys | |
| | to select option. | |
| --------------------- |
-------------------------
```

Panel Definitions in Detail

As described in the overview, a panel is a collection of fields, each of which is described by a collection of attributes. Some of these attributes are device dependent, and details concerning them are taken from the appropriate terminal database. Given below is a detailed description of each attribute; information on how these are set up are given in the next section entitled **Programmer's Interface**.

Location

The location of a field is defined to be the coordinates of the top left hand corner of the field. The coordinates are given as:

Start row
Start column

Size

The size of a field is defined by:

Number of rows
Number of columns

Type

The type of a field determines its format and validation . When data is displayed in a field, it is first formatted according to its type. When data is entered in a field, it is validated according to its type.

The following types are currently available:

Output

- This field type is included for upwards compatibility with previous versions of prefect.
- A field could be defined as output only; that is, no data could be entered.
- This field type has been superseded by the **protection** attribute, although it is still recognised as a valid type.

Text

- No check on entry.
- Input is left-justified.
- No formatting done on output.

Numeric

- Characters from set `0` to `9 . - ¯` space are allowed.
- Input is left-justified.
- No check on entry of validity of number.
- More than one number allowed per row.
- No formatting done on output.

Upper Case

- As for TEXT, but any letters are automatically translated to upper case on entry or output.

Lower Case Translation

- As for TEXT, but any letters are automatically translated to lower case on entry or output.

Numeric Point Plain

- Allows characters from set `0` to `9 . - ¯`
- Decimal point denoted by .
- Input is right-justified.
- No editing is allowed; user must erase and re-enter.
- Allows a prescribed number of decimal places.
- Numerics are right-justified on output, & formatted to prescribed number of decimal places.

Numeric Point Triple

- As for NUMERIC POINT PLAIN, but on input & output, commas are automatically inserted between triples eg. 1234 is displayed as 1,234

Numeric Comma Plain

- Allows characters from set `0` to `9 , - ¯`
- Decimal point denoted by ,
- Validates format of numeric on entry.

- Input is right-justified.
- No editing is allowed; user must erase and re-enter.
- Allows a prescribed number of decimal places.
- Numerics are right-justified on output, & formatted to prescribed number of decimal places.

Numeric Comma Triple

- As for NUMERIC COMMA PLAIN, but on input and output, full-stops are automatically inserted between triples e.g. `1234` is displayed as `1.234`

Video

Video attributes are given as indices which correspond to the video codes set up in the appropriate output table for your terminal. Up to 256 video characteristics are allowed. Code 0 is assumed to be the default video. If the video code is not in the output table, prefect assumes null, and does nothing.

Protect

A field may be defined as protected or unprotected. A protected field may be updated only by the program; the user may not enter data into it. Hence, protected fields are used for titles, prompts, messages etc. An unprotected field may be updated by the program OR the user. Hence, unprotected fields may be used to display default data, and to collect new or modified data from the user.

With unprotected fields, the data allowed depends on the TYPE of field. Data may be validated or translated on entry.

With protected fields, the data is formatted on output. A program can put invalid data in a field eg. text data to a field of type NUMERIC. In this case, the text will be displayed in the numeric field, left-justified. However, on retrieval of numeric fields, all non-numerics are retrieved as zero.

Colour - Foreground & Background

The colour attribute for a field is divided into foreground colour and background colour. Colour attributes are given as indices which correspond to the colour codes set up in the appropriate output table for your terminal. Up to 256 foreground and background colours are allowed. Code 0 is assumed to be the default colour in both cases. If the colour code is not in the output table, prefect assumes null, and does nothing.

Programmer's Interface

The Auxiliary Processor prefect is invoked by

```
      'prefect' ⎕SH ''
```

Twelve external functions are defined, concerned with panel definition and manipulation:

PCURS	Cursor positioning
PDEFN	Panel management
PENTER	Allow enter key to be used as a function key
PFLDS	Field label definitions
PGET	Data retrieval
PGETN	Numeric data retrieval
PKEY	Function key labels
PPUT	Output
PREAD	Input from user
PREDRAW	Refresh the screen
PRTL	Prefect response time limit
PVAL	Data validation

Each function has a variety of uses, depending on its arguments, as shown on the next page.

Each function is described fully in the following sections. The functions are discussed in alphabetical order.

Panel Definition

Define new panel layout	`PDEFN matrix`
Assign field labels	`PDEFN labels`
Place data in fields	`data PPUT fields`
Position cursor	`position PCURS fields`
Sound bell	`PPUT 2`

Manipulation

Select active panel	`PDEFN +panel`
Delete a panel	`PDEFN -panel`
Modify panel layout	`PDEFN matrix`

Input

Accept input to active panel	`PREAD`

Data Retrieval

Validate field contents	`PVAL fields`
Get field contents	`PGET fields`
Get field contents as numeric	`PGETN fields`

Query

Cursor position	`PCURS ι0`
Field numbers	`PFLDS labels`
Active panel number	`1 PDEFN 0`
Active panel definition	`PDEFN ι0`
Screen size	`1↓PDEFN 0`
Changed field numbers	`PGET 0`

Miscellaneous

Flush updates	`PPUT 1`
Suspend full screen	`PPUT 0`
Get function key labels	`PKEY keys`
Refresh screen	`PREDRAW`
Define ENTER as function key 0	`PENTER setting`
Set response time limit	`PRTL seconds`

Cursor Positioning

The function `PCURS` is concerned with cursor positioning or location. The result returned depends on use. `PCURS` is used for the following operations:

Set cursor position	`position PCURS field`
Query cursor position	`PCURS ⍳0`

Set Cursor Position: `{R}←{pos}PCURS field`

Allows the cursor to be positioned in the active panel. The cursor position is defined by field, then by row and column within the field.

The field specification may be by name or number. A field number of zero implies the screen as a whole, and the cursor will be positioned relative to the screen.

The position specification must be supplied as a two element numeric vector, giving row and column number within the specified field. If no position is supplied, then the default of 'start of field' is taken.

The result is a three element numeric vector giving the previous cursor position, as field number, row and column. The result is shy.

Note that the cursor position is considered to be part of the panel definition, and is saved with the panel. Each saved panel may have the cursor positioned in a different place. The default cursor position is the start of the first input field, or the top-left corner (home) if no input fields are defined for the panel.

Errors:

`DOMAIN ERROR` is reported if no panels exist.
`RANK ERROR` is reported if the left argument is not a vector.
`LENGTH ERROR` is reported if the left argument is not of length two.

Example:

```
      2 1 PCURS 3 ⍝ Position cursor in the first column
                  ⍝ of the second row of field three.
```

Query Cursor Position: {R}←PCURS ⍳0

This returns the location of the cursor in the active panel as a three element numeric vector, giving field number, row and column. The result is shy. If the cursor is not in a field, then the field number is zero, and row and column are screen coordinates.

Errors:

`DOMAIN ERROR` is reported if no panels exist.

Example:

```
      +PCURS ⍳0 ⍝ Where is the cursor now?
3 2 1
```

Panel Management

The function *PDEFN* is concerned with panel management. Unless otherwise stated, *PDEFN* returns a shy result which is a three element numeric vector giving:

active panel number (0 if no panels defined)
number of lines on screen
number of columns in screen

The following operations may be performed using the function *PDEFN*:

Define new panel layout	*PDEFN matrix*
Assign field labels	*PDEFN labels*
Select active panel	*PDEFN +panel*
Delete a panel	*PDEFN -panel*
Modify panel layout	*PDEFN matrix*
Active panel number	*PDEFN* 0
Active panel definition	*PDEFN* ⍳0
Screen size	*PDEFN* 0

Define New Panel: *{R}←PDEFN matrix*

This defines, and makes active, a new panel. Each row describes a field. *matrix* has 6 to 9 columns.

Columns:

1. Row number of top left corner

2. Column number of top left corner

3. Number of rows

4. Number of columns

5. Type, encoded as:

   1 = text
   2 = simple numeric
   3 = uppercase translation
   4 = lowercase translation
   5 = numeric 'point plain' format, any number of decimal places
   5n = numeric 'point plain' format, n decimal places
   6 = numeric 'point triple' format
   6n = numeric 'point triple' format, n decimal places
   7 = numeric 'comma plain' format, any number of decimal places
   7n = numeric 'comma plain' format,n decimal places
   8 = numeric 'comma triple' format, any number of decimal places
   8n = numeric 'comma plain' format,n decimal places

6. Video

   Codes given correspond with entries in the Output Table `WIN.DOT`. 0 should be defined as normal or default video.

7. Protect

   Set to 1 for protected ie. no data entry allowed Set to 0 for unprotected ie. data entry allowed

8. Foreground Colour

   Codes given correspond with entries in the Output Table `WIN.DOT`. 0 should be defined as normal or default foreground colour.

9. Background Colour

   Codes given correspond with entries in the Output Table `WIN.DOT`. 0 should be defined as normal or default background colour.

The panel definition is placed on the stack, and assigned the next panel number. Panels are numbered from one. This panel number is used in subsequent operations to identify the panel. Fields and field labels are initialised to blank. The cursor is positioned at the start of the first input field. Fields that extend beyond the screen are truncated.

Errors:

LENGTH ERROR is reported if no fields are defined.

Example with no colour attributes:

```
PDEFN 10 10 1  6 1 1 1 ⍝ text,normal video,protected
PDEFN 10 20 1 10 3 3 0 ⍝ uppercase,video3,unprotected
PDEFN 12 10 1  6 1 1 1 ⍝ text,normal video,protected
PDEFN 12 20 1  2 2 3 0 ⍝ simple num,video3,unprotected
```

Assign Field Labels: {R}←PDEFN labels

This associates text labels with fields in the active panel. Only the first six characters are significant. Labels may be provided as a nested vector of character vectors, or as a linelist (ie. a simple character vector with labels delimited by the newline character). Non-character labels are ignored. Labels for non-existing fields are ignored. The field may then be referenced by number or by label.

Errors:

DOMAIN ERROR is reported if no panels exist.

Example:

```
⍝ Assign labels to fields 2 and 4, leave others
⍝ unlabelled
PDEFN '' 'name' '' 'age'

⍝ Can later retrieve by name or number
PGET 2    ⍝  or
PGET 'name'
```

Select Active Panel: {R}←PDEFN panel_num

This makes the specified panel active. This is the panel shown to the user on the next *PREAD*.

Errors:

DOMAIN ERROR is reported if the panel does not exist.

Example:

```
PDEFN 2    ⍝ Select panel 2 as the active panel
```

Delete Panel: {R}←PDEFN -panel_num

This deletes the specified panel. When the active panel is deleted, the highest numbered remaining panel becomes active. When no panels remain, the user reverts to normal screen mode. No errors are reported if the specified panel does not exist.

Example:

```
      PDEFN -5    ⍝ Delete panel 5
```

Modify Panel: {R}←PDEFN matrix

This modifies the active panel. Each row gives the modification to be made to an existing field definition. The 3 columns are as follows:

1. Number of field to be modified.
2. Column of definition matrix to be modified.
   Note that only field location and attributes (columns 1 2 5 6 7 8 and 9) may be changed. Field size (columns 3 and 4) may not.
3. New value.

Errors:

`DOMAIN ERROR` is reported if no panels have been defined, the field does not exist, or the column number is not in the allowable range.

Examples:

```
      PDEFN 3 1 20     ⍝ Move field 3 to row 20
      PDEFN 7 6  4     ⍝ Change video of field 7
      PDEFN 1 7  1     ⍝ Protect field 1 and
      PDEFN 1 5  3     ⍝ translate input to uppercase
```

Active Panel Number: {R}←PDEFN 0

Returns the active panel number and the screen size.

Example:

```
      +PDEFN 0
3 30 80
```

Active Panel Definition: {R}←PDEFN ⍳0

This returns the definition of the active panel as a nine-column matrix.

Error:

`DOMAIN ERROR` is returned if no panels exist.

Example:

```
      +PDEFN ⍳0
10 10 1  6 0 1 0 0 0
10 20 1 10 1 3 0 0 0
12 10 1  6 0 1 0 0 0
12 20 1  2 2 3 0 0 0
```

Enter Key Used as Function Key

Redefine Enter: {R}←PENTER N

This function allows the effect of the ENTER key to be redefined. `N` may be 0 or 1. The result returned is shy and is set to the value of `N`.

If `N` is 0, the default, then the Enter key moves the cursor to the next input field or the next line in an input field.

If `N` is 1, then the Enter key is redefined as function key 0. Note that the setting for `PENTER` persists throughout the prefect session, until changed.

Field Label Definitions

Query Field Numbers: `R←PFLDS labels`

This function returns the numbers of the fields corresponding to the specified label list.

The label list may be supplied as either a nested array in which each element is a label name, or as a linelist (ie. a simple character vector containing label names separated by newline characters).

Blank or non-existing labels return one plus the number of fields in the panel. If a numeric argument is supplied, it is returned unchanged.

Example:

```
      PFLDS 'name' 'age'
2 4
```

Data Retrieval

These functions are concerned with the retrieval of data from the active panel.

Retrieve data	`PGET FIELDS`
Determine changed fields	`PGET 0`
Retrieve numeric data	`PGETN FIELDS`

Retrieve Data: R←{param}PGET fields

This function returns the contents of the specified fields.

The fields may be specified by number or by name.

The left argument, if supplied, must be a numeric vector, with value zero or one. This determines the shape of the result.

The result is a nested array with one element per specified field. Each element consist of the contents of the field, returned by default, as a matrix. If the left argument to `PGET` is one, then the field contents are returned as vectors.

The contents of text fields (types 1, 3 or 4) and simple numeric fields (type 2) are returned as character matrices or vectors. The contents of formatted numeric fields (types 5,6, 7 or 8) are returned as numbers. The reason for this apparent discrepancy is that type 2 fields are only partially validated on input, and cannot be guaranteed to be numeric.

If the field specification is a scalar, then the result is simple.

Errors:

None

Example:

```
      NAME  AGE ← PGET  'name'  'age'
```

Changed Fields: `R←PGET 0`

This function returns a boolean vector; 1 where a field has been changed since the last `PREAD`, 0 otherwise.

Example:

```
      PDEFN form
      PREAD
      CHANGED_FIELDS←(PGET 0)/⍳1↑PFORM
```

Retrieve Numeric Data: `R←PGETN fields`

This function returns the numeric contents of the specified fields. Fields may be specified by number or by name. The result may be simple or nested, with one element per specified field. The values of single row fields are returned as scalars, otherwise as vectors. A zero is returned for non-existing fields, for fields containing invalid numerics, and for blank fields.

`PGETN` is usually used to retrieve simple numeric fields of type 2, after they have been validated by `PVAL`.

Example:

```
      AGE ← PGETN 'age'
```

Function Key Labels

Query Function Keys: *R←PKEY fn_keys*

This function returns the labels defined for function keys in the terminal input translate table. If labels are not defined, then <ESC> number is returned for function key numbers 0 to 9, <UNDEFINED> otherwise.

Note that this function has been superceded by the system function `⎕KL` (see *Language Reference* for details). Systems that use `PKEY` should be amended to use `⎕KL`.

Example:

```
      ⍝ get labels for function keys 0,1,2,9 and 999,
      ⍝ where only labels 0, 1 and 2 exist.

      PKEY 0 1 2 9 999
 HELP  F1  F2  <ESC>9  <UNDEFINED>
```

Output

This function is concerned with passing information to prefect. The result of `PPUT` is shy and null. The following operations may be performed with `PPUT`.

Place data in fields	`data PPUT fields`
Suspend full screen	`PPUT 0`
Flush update	`PPUT 1`
Sound bell	`PPUT 2`

Put Data in Fields: `{R}←data PPUT fields`

The field specification may be supplied by number or by name. The data must be supplied as a nested vector, one element per field. The data may be numeric or text. Data is ignored for non-existent fields.

If text data is supplied as a two-dimensional matrix, truncation is performed as for the APL 'take' function if the data is too large for the field. If text data is supplied in any other form, it is inserted in ravel order.

If numeric data is supplied, it is inserted one number per row of the field. Numbers are rounded to the nearest decimal. Asterisks are displayed if the number is too large for the field. Negative numbers are preceded by a minus sign. If the field type is 5, 6, 7, 8 or 5n, 6n, 7n, 8n numbers are right-justified and formatted. See the section 'Panel Definitions in Detail' for further information. Otherwise numbers are left-justified in the field.

Errors:

`DOMAIN ERROR` if invalid arguments are given.

Example:

```
      'Sales' SALES 'Margin' 15.5 PPUT 1 2 3 4
```

Suspend Full Screen Mode: *{R}←PPUT* 0

This suspends full screen work. The screen is cleared, and the user is placed in desk calculator mode. There is an automatic redraw of the screen when the next full screen input or output operation is requested.

This suspension of full screen mode is automatic when the last panel is released.

This feature is useful during system development.

Flush Updates: *{R}←PPUT* 1

This flushes any updates to the screen.

When a panel is changed, the screen is not updated immediately; the update is normally deferred until the next user input is requested, using `PREAD`. However, this feature can be used to display change immediately; for example, warning messages can be displayed without making a request for user input.

Example:

```
'Pauline' 21 PPUT 2 4   ⍝ Define data for fields
                        ⍝ 2 and 4
PPUT 1                  ⍝ Show these changes to
                        ⍝ the user but keep
                        ⍝ control of the screen
```

Normally changes made to the active panel by the APL program are not immediately displayed on the screen. Instead, the screen update is deferred until the user is given control by `PREAD`.

`PPUT 1` overrides deferred update and causes all pending changes to be displayed. `PPUT 1` can be used to display warning messages when it is inappropriate to request user input with `PREAD`.

Sound Bell: {R}←PPUT 2

This ensures that the screen bell will sound when the next `PREAD` or `PPUT 1` is performed. It is part of the active panel definition. It is useful for signalling error messages, or for waking the user up.

Example:

```
'Pauline is not 21' PPUT 7⍝ Define error
PPUT 2⍝ Flag bell
PPUT 1⍝ Flush updates
```

Input from User

User Input: `{R}←PREAD`

`PREAD` updates the display with any pending changes, and places the terminal in full-screen mode for data entry.

In updating the display, `PREAD` only refreshes those parts of the screen that the APL program has changed since the last `PREAD` or the `PPUT 1`. If there are no screen updates pending, `PREAD` does not update the display. Secondly, if there is a bell pending, `PREAD` rings the bell. Finally, if the APL program has requested a new cursor position, `PREAD` moves the cursor. The terminal is then placed in full-screen mode, and the user may enter data.

Data entry while under the control of `PREAD` is described fully in previous sections.

Data entry may be terminated in one of 3 ways:

a) The user presses a function key. In this case, the result of `PREAD` is the number of the function key pressed. Note that `PENTER` can be used to make the ENTER or RETURN key act as function key 0, so that ENTER or RETURN can be used to terminate input.

b) `PREAD` times out (see `PRTL` for further details). In this case the result of `PREAD` is `¯1`.

c) The user presses Ctrl+Break, or selects "Interrupt" from the "Action" menu on the Session Window. In this case, `PREAD` does not return a result but signals `ERROR 219` to APL. Unless `ERROR 219` is trapped, APL will stop on and display the line containing the call to `PREAD`, as for any other event.

Refresh the Screen

Refresh Screen: `PREDRAW`

This function allows the program to request that the screen be redrawn.

Prefect Response Time Limit

Response Time Limit: {R}←PRTL N

This function emulates ⎕RTL. If PRTL is set, and no response from a call to PREAD is received within N seconds, then a 'time out' occurs, and PREAD returns a result of ¯1.

As for ⎕RTL, N must be an integer value. A value of 0 indicates that no time out limit is to be applied.

Note that the value of PRTL persists for the remainder of the prefect session, until changed.

PRTL can be used in different ways depending on the active panel definition.

Consider a panel that has been defined to display some information for the user ie. it has no unprotected fields and no data entry is expected. Then if PRTL is set, control can be passed by prefect to the calling function every N seconds so that the information can be updated if necessary.

If PRTL is used with a panel that does allow data entry, then more control must be exercised by the calling program. A common sequence is as follows:

- Set response time limit
- Issue PREAD and give control of the screen to the user
- If a timeout occurs in the PREAD, then the screen is in one of two states:

1. The user has made no changes to the screen. The program can take appropriate action, perhaps sounding the bell to wake the user up, or passing control to another screen that informs the user of facts that may influence his data entry when he does start typing.

2. The user has made changes to the screen, ie. he is entering data. In this case, the PRTL may be retained, in which case, the program gets control of the screen every N seconds and can take appropriate action, OR the PRTL may be reset and control of the screen passed back to the user until he presses a function key in the usual way.

The use of PRTL is best illustrated by example; please see the workspace PREDEMO.

Errors:

DOMAIN ERROR if N is invalid.

Example:

```
      PRTL 5          ⍝  Set response limit to 5 seconds
      KEY←PREAD       ⍝  Give panel to user
                      ⍝  Wait 5 seconds
      KEY             ⍝  Display return codes
¯1
                      ⍝  Program takes action on timeout
```

Data Validation

Validate Numbers: `R←{decimals}PVAL fields`

This function checks for valid numeric data.

The field specification may be supplied by number or label.

The left argument specifies the number of decimal places permitted. If the left argument is supplied as a scalar, then it is taken to apply to all the fields specified in the right argument. If it is given as a vector, then it is expected to match the length of the right argument; elements of the arguments are paired in the usual fashion.

The result is boolean, with one element for each field specified. A value of 1 indicates a valid numeric field, and 0 indicates an invalid numeric field. A field which is entirely blank is deemed valid. If a field has more than one row, then the result is nested, with one element per row.

This validation works on any field type; that is, a field need not be defined as numeric for `PVAL` to be used. However, it is usually used to validate simple numeric fields (type 2), which are not fully validated on input.

Errors:

`DOMAIN ERROR` if field does not exist.

Example:

```
      PDEFN form               ⍝ Define panel
      PREAD                    ⍝ Allow user input
      PVAL 'age'               ⍝ Validate age field
0                              ⍝ Invalid numeric
```

The PREFECT Workspace

The *PREFECT* workspace contains a small system for designing complex panels, plus some useful cover functions.

Designing panels

A panel may always be defined using the basic tools supplied by the prefect Auxiliary Processor, in the manner described in the previous sections. However, the function *PDESIGN* is supplied to help with the design of more complex panels. It uses the AP prefect so that panels may be designed in a full screen manner. It is menu driven, and uses the function keys to select and drive the available options.

Design Panel: *panel←PDESIGN panel*

If the argument supplied is null, then *PDESIGN* assumes a new panel is to be defined. If the argument is a panel definition, then *PDESIGN* assumes that the panel is to be modified.

A panel definition is a three element vector consisting of the following elements:

`1 ⊃ panel`	Matrix defining field layout
`2 ⊃ panel`	Nested vector containing field text
`3 ⊃ panel`	Nested vector of field names

The result of *PDESIGN* may be used as the argument to the function *PDESIGN* to define a new panel.

PDESIGN offers facilities to:

- Set field positions
- Define field text
- Define field attributes - format/type
  - video
  - protection
  - foreground colour
  - background colour
- Define field labels
- Perform general editing

Try using *PDESIGN*, taking advantage of the *HELP* facility.

Cover Functions

Some useful cover functions are available in the *PREFECT* workspace. These functions should be regarded as a starting point only, and should be amended to suit your applications.

Assign Variables: *PASSIGN LABELS*

Get contents of fields with labels *LABELS*, and create variables with the same names as *LABELS* from the data.

Background Colours: *{R}←{N}PBCOLOUR F*

Determines current background colours of fields *F*. Changes background colours to *N* if *N* is supplied.

Set Beep: *PBEEP F*

Make beep pending on active panel.

Changed Fields: *R←PCHANGE*

Returns with 1 if any fields have been changed since last *PREAD*.

Clear Fields: *PCLEAR FLD*

Clear the given field(s).

Define Panel: `{R}←PDEFINE P`

Define nested panel `P`, format as returned by `PDESIGN`

`P[1]` - 6, 7, 8 or 9 column matrix specifying panel layout
`P[2]` - background text to be displayed in fields
`P[3]` - field labels

Edit Text: `R←PEDIT T`

Two dimensional text edit or browse.

Output Error: `{FIELD}PERROR MSG`

Put error `MSG` out to `ERRORFIELD` field and beep. Uses video code 13 for errors.

Foreground Colours: `{R}←{N}PFCOLOUR F`

Determines current foreground colours of fields `F`. Changes foreground colours to `N` if `N` is supplied.

Flush to Screen: `PFLUSH`

Flush changes given to prefect to the screen.

Output Help: `PHELP HELP`

Put out help screen using `HELP` variable.

Get Panel Image: `R←PIMAGE`

Return current image of active panel as text matrix.

Terminate Prefect: *PKILL*

Expunge all prefect functions to kill auxiliary processor.

Display Items: *PLIST ITEMS*

Define a panel displaying *ITEMS* as a matrix. Allow user to browse up and down the panel using function keys 1 and 2.

Display Menu: *PMENU MENU*

Define a panel displaying the given *MENU*. *MENU* is a three element vector of title, function keys and associated options. Uses video codes 17 for title, 0 for text and 18 for message lines.

Output Message: *{MSG}PMESSAGE TXT*

Put message *TXT* out to *MSG* field and beep. If *MSG* field not specified, use global variable *M̲E̲S̲S̲A̲G̲E̲F̲I̲E̲L̲D̲*.

Field Locations: *{R}←{N}PMOVE F*

Determines current location of fields *F*. Moves fields to new locations *N* if *N* is supplied.

Set Options: *{PANEL}←{PANEL}POPTIONS DATA*

Insert key labels for options into *O̲P̲T̲I̲O̲N̲S̲F̲I̲E̲L̲D̲* field in panel. *DATA* is a two element vector of key numbers and associated text.

Protection Settings: {R}←{N}PPROTECT F

Determines current protection settings of fields F. Changes protection settings to *N* if *N* is supplied.

Redefine Attributes: {FLDS}PREDEFINE DEFN

Redefine panel attributes (location, type, video, protection, foreground and background colour) and default background text, using original panel definition *DEFN*. If *FLDS* is specified, only those fields are redefined.

Reset Panels: PRESET

Release all panels.

Field Shapes: R←PSHAPE

Return shape of fields *F*.

Screen Size: R←PSIZE

Return screen dimensions.

Initiate Prefect: PSTART

Start up prefect auxiliary processor if not already started.

Interrupt Prefect: PSTOP

Suspend prefect to permit some line-by-line work.

Field Types: `{R}←{N}PTYPE F`

Determines current types of fields `F`. Changes types to `N` if `N` is supplied.

Undefine Panel: `PUNDEFN`

Undefine current panel.

Video Settings: `{R}←{N}PVIDEO F`

Determines current video settings of fields `F`. Changes video settings to `N` if `N` is supplied.

Example Systems

The workspace `PREDEMO` contains some example functions which illustrate how you can use the Full Screen Data Manager. If you try using them and amending them, we hope that you will begin to see how easy it is to define full screen user interfaces using this product.

Please remember though, that these are examples only, and have not been exhaustively tested. Also, these functions do not show the ONLY way to implement a full screen interface, and you can probably design a much more efficient interface which suits your particular application better.

In order to make the examples as device independent as possible, no explicit use is made of the foreground and background colour attributes, and video codes 0 and 11 to 20 are used to highlight input fields, errors etc. For monochrome screens, these codes have been set up to use combinations of all allowable video attributes eg. reverse, blinking reverse etc. However, because the video attributes for colour screens are severely limited, different colour combinations have been used instead. When you are writing your applications, you will know which devices you will be targeting for, and can use the colour and video attributes together to greater effect.

Example 1 : Simple Data Entry

In this example, we will build up the program statements required to execute Steps 1 through 8 as described in the overview.

First, we can use `PDESIGN` to define two panels, one for the help screen and one for data entry:

```
                                HELP PANEL

                           ------------------
                          |  Help for Add     |
                          |                   |
One unprotected field --> |Age must be in the |
filled with text          |   range 30 to 70  |
                          |                   |
                          | Press any function|
                          | key to continue   |
                           ------------------


                                HELPPANEL ← PDESIGN ''


                                1⊃HELPPANEL

                          5 21 12 29 1 0 1

                                DISPLAY 2⊃HELPPANEL

                          .→------------------------.
                          |  .→--------------------. |
                          |  ↓   Help for Add      | |
                          |  |                     | |
                          |  | Age must be in the  | |
                          |  |   range 30 to 70    | |
                          |  |                     | |
                          |  | Press any function  | |
                          |  |     to continue     | |
                          |  '---------------------' |
                          '∊------------------------'

                                DISPLAY 3⊃HELPPANEL
```

```
. ⊖ .
|   |
' _ '
```

```
                     DATA PANEL

                _____________________
               |   ____   _________  |
Protected    .--->|Name| |         |<-Unprotected field
fields with -| |  |____| |_________| |   Video 1
background   | |   ___    ___        |
text defined '--->|Age|  |   |<-------Unprotected field
               |  |___|  |___|       |   Video 1
               |   ________________  |
               |  |                |<-Protected field
               |  |________________| |   for message
               |_____________________|
```

```
                     DATAPANEL ← PDESIGN ''

                     1⊃DATAPANEL

                5 21 1  4 0 0 1
                5 31 1  9 1 1 0
                8 21 1  3 0 0 1
                8 31 1  2 5 1 0
               12 25 1 78 0 0 1

                     DISPLAY 2⊃DATAPANEL

               .→-------------------------.
               | .→---. .⊖. .→--. .⊖. .⊖. |
               | ↓Name| ⌽ | ↓Age| ⌽ | ⌽ | |
               | '____' '_' '___' '_' '_' |
               '∊-------------------------'

                    DISPLAY 3⊃DATAPANEL

               .→---------------------------.
               | .⊖. .→---. .⊖. .→--. .→--. |
               | | | |name| | | |age| |msg| |
               | '_' '____' '_' '___' '___' |
               '∊---------------------------'
```

Step 1:

Pass the panel definitions to prefect:

```
PDEFINE HELPPANEL ⋄ PDEFINE DATAPANEL
```

Step 2 & Step 3:

Tell prefect to give the active panel to the user. The user enters data into the unprotected fields, and tells prefect that he has finished by pressing a function key. Prefect returns with the number of the function key pressed:

```
KEY ← PREAD
```

Step 4:

Get the data from the panel; remember that fields can be identified by number or label:

```
DATA AGE ← PGET 'name' 'age'
```

Step 5:

Validate the data. If there is an error, put out message and give screen back to the user:

```
→DOIF AGE < 30 ⋄ 'TOO YOUNG!' PPUT 'msg'
```

Step 6:

User asks for help, by pressing the relevant function key (e.g. 2):

```
→LHELP × ⍳ KEY = 2
```

Step 7:

Switch the active panel to the help panel (panel 1), and show it to the user:

```
LHELP: PDEFN 1 ⋄ KEY ← PREAD
```

Step 8:

Switch back to the data panel (panel 2), and repeat the process:

```
PDEFN 2 ⋄ →Step3
```

Based on these steps, we can write the function `UPDATE`, which allows the user to create a new record, or to update an existing one.

```
      ∇ DATA←UPDATE DATA;KEY;NAME;AGE
[1]     ⍝ Example prefect program
[2]     ⍝
[3]     ⍝ Start the prefect AP if its not already started
[4]      PSTART
[5]     ⍝ Define Help screen (panel 1) and Data screen
          (panel 2)
[6]      PDEFINE HELPPANEL ⋄ PDEFINE DATAPANEL
[7]     ⍝ Place default data in data fields
[8]      DATA PPUT'name' 'age'
[9]     ⍝ Give data screen to prefect; take action
         depending on function key
[10]    LREAD:KEY←PREAD ⋄ →(LEND,LHELP,LCONT)[1 2⍳KEY]
[11]    ⍝ Clear message field, in case the error message
          is still there
[12]    LCONT:PCLEAR MESSAGEFIELD
[13]    ⍝ Retrieve new data values
[14]     NAME AGE←PGET'name' 'age'
[15]    ⍝ Data validation - age must be in range 30 to 70
[16]    ⍝ Uses PERROR to put message in field 'msg' and
          to highlight error field
[17]     →DOIF AGE<30 ⋄ 'name'PERROR'TOO YOUNG!' ⋄ →LREAD
[18]     →DOIF AGE>70 ⋄ 'name'PERROR'TOO OLD!' ⋄ →LREAD
[19]    ⍝ Form result and exit
[20]     DATA←NAME AGE ⋄ →LEND
[21]    ⍝ Help - switch to HELP screen, give to user,
          switch back to data
[22]    LHELP:PDEFN 1 ⋄ KEY←PREAD ⋄ PDEFN 2 ⋄ →LREAD
[23]    ⍝ Clear all screens from prefect and exit
[24]    LEND:PRESET
      ∇
```

This function makes use of some of the cover functions available in the `PREFECT` workspace. These functions are meant as guidelines, and can and should be amended to suit your particular applications.

Try using this function, which is saved in the workspace `PREDEMO`. As you can see from this listing, function key 2 is used for *help*, function key 1 for 'quit' and any other function key for continue.

To create a new data record, use `DATA ← UPDATE ''`

To modify an existing data record, use `DATA ← UPDATE DATA`

Example 2: A simple data base system

Consider a system that allows us to add, modify, delete and display the employee records held in the STAFF database. Such a system is contained in the workspace *PREDEMO*. Try running the system, and looking at the functions used in the programs for ideas on how to write your own applications using *PREFECT*.

Start the system by typing *STAFF∆RUN*.

Example 3: A simple dealer system

Consider the kind of system required in the dealing room of a bank. The dealers are on the telephone all day long, discussing prices of shares. Once they decide to buy, they need to record the details of the deal in some way. Also, whilst they are talking on the phone, they need to be told as soon as any information comes in (from, for example, Reuters) that may affect any decisions they make. The simple functions used in this example try to show how you might make use of prefect and *PRTL* to implement this kind of system.

Start the system by typing *DEALER∆RUN*.

CHAPTER 10

Writing Auxiliary Processors

Introduction

Auxiliary Processors (APs) are non-APL programs that provide Dyalog APL users with additional facilities.

Under UNIX, APs run as separate UNIX tasks, and communicate with the Dyalog APL interpreter through UNIX pipes. Under MS-Windows, APs run as separate Windows programs and communicate with APL through a global memory segment.

Typically, APs are used where speed of execution is critical, such as in screen management software, or for utility libraries. Auxiliary Processors may be written in any compiled language, although 'C' is preferred and is directly supported.

When an Auxiliary Processor is invoked from Dyalog APL, one or more **external functions** are fixed in the active workspace. Each external function behaves as if it was a locked defined function, but is in effect an entry point into the Auxiliary Processor. An external function occupies only a negligible amount of workspace.

Where there are simple differences between the way APs work under Windows and UNIX, this is indicated by (Windows) or (UNIX). Unless otherwise specified, all references in this Chapter relate to other sections in this *User Guide*.

Statement of Warranty

Dyadic guarantees that the AP interface works as described herein, and accepts responsibility for the correctness of the various support functions which are distributed as part of the AP Writer's Toolkit. Dyadic reserves the right to change the interface at some future date, but if so undertakes to update the Toolkit so as to minimise the impact on users who have developed their own APs. Users are strongly recommended therefore to use the support routines provided. Dyadic is not responsible for failures in Dyalog APL resulting from errors in user-written APs.

How Auxiliary Processors Work

Starting up an AP

An Auxiliary Processor is invoked using the dyadic form of the *⎕CMD* or *⎕SH* System Function. *⎕CMD* is probably more natural for Windows users, and *⎕SH* for the UNIX user. However, these System Functions are synonymous, and work in identical ways in both environments.

The left argument to *⎕CMD*/*⎕SH* is the name of the executable file containing the AP. This may be a full path name or a simple file name. In the latter case, the search path defined by the `wspath` parameter will be scanned for the file. Under Windows, the left argument may be the full command line for running the AP. This can be used, for example, to run an AP under a debugger.

The right argument to *⎕CMD*/*⎕SH* is either a simple text scalar or vector, or a vector of text scalars or vectors. Under UNIX, the first item in the right argument is the name to be entered in the process table and will appear in a `ps` listing. Subsequent items, if present, are passed to the AP as calling parameters. Under Windows, the right argument is ignored.

On locating the load module specified as the left argument to *⎕CMD*/*⎕SH*, Dyalog APL starts the AP.

Under UNIX, it does so by opening two pipes using the UNIX system command `execv`. One pipe is used for passing information from APL to the AP, the other for data travelling in the opposite direction.

Under Windows, APL starts the AP as a Windows program and uses messages to do task-switching. Data is transferred between APL and the AP through a global memory segment. This segment is viewed as a file by the AP. However, if the standard toolkit functions (**`startup`**, **`define`**, **`getarray`**, **`putarray`** etc.) are used, the communications mechanism is transparent. There is a separate communications segment for each APL session.

There follows an interchange of information which establishes external functions in the user's workspace. First, the AP and APL exchange their process ids. Then the AP sends the total number of its external functions, followed by their names, code numbers and calling syntax. APL responds by entering each external function name in the symbol table together with other pertinent information. APL then continues processing while the AP waits for instructions.

Using an AP

Once established, an AP is used by a reference to one of its external functions. When an external function is referenced, APL passes to the AP its code number followed by its arguments. Under UNIX and Windows, if the external function is defined as being result returning, APL waits. Otherwise, APL continues processing in parallel.

Upon receipt of a code number, the AP reads the arguments which were supplied to the external function from the pipe (UNIX), or from the global memory segment (Windows), and, typically, calls a corresponding subroutine to process the data in the desired manner. If the external function returns a result, the AP transmits it to APL. Having received a result, APL then continues processing while the AP waits for the next external function reference.

Terminating an AP

An AP is terminated normally under the control of APL when all of its external functions are expunged from the active workspace. This could occur with the use of `)CLEAR`, `)LOAD`, `)ERASE`, `⎕EX`, or by the termination of a defined function in which all external functions are localised. When any of these events occur, APL signals to the AP that it should terminate. Under UNIX, this is done by closing the pipe that the AP is reading. Under Windows, APL sends an explicit "shutdown" message to the AP. In both cases, the AP then raises `SIGHUP`. The AP writer can use `signal()` to handle this condition and perform any termination tasks, such as flushing output buffers. The same thing happens when APL itself terminates with `)OFF` or `⎕OFF`.

External Functions

Function Name

The name of an external function is established using the toolkit function **`define`**. The rules for an external function name are the same as for any other Dyalog APL name, which may contain `∆` and `⍙`, upper case, lower case, numeric, underscore and underscored characters. To cater for all possibilities the name is read as if it had been entered at the keyboard starting in ASCII mode, except that the control codes must be specified by an escape sequence. See **`define`** for further details.

Note that there is no direct relationship between the APL name of an external function, and a corresponding function name in the Auxiliary Processor.

Function Syntax

External functions may have any syntax. Specifically, they may be monadic, dyadic or ambivalent and return a result, a shy result or no result.

Mode of Operation

Under UNIX and Windows, a result-returning external function causes APL to wait until it has finished its task. A non result-returning external function allows APL to continue processing in parallel. The former is said to be synchronous, the latter asynchronous. A synchronous or result-returning external function is capable of normal APL error handling. An asynchronous external function is not. An Auxiliary Processor may fix a mixture of synchronous and asynchronous external functions.

Interrupt Handling

Under UNIX, an external function may elect to handle its own interrupts. In this case, `SIGINT` and `SIGQUIT` signals will be passed to the AP during execution of the function. Otherwise, interrupts will be ignored.

Under Windows, there is no mechanism for pre-emptive interrupts, but a Windows AP can receive keyboard messages and react to Ctrl+Break in an appropriate manner if required.

Code Number

Each external function in an AP is assigned a unique code number, which is used by APL to identify it and to notify the AP of the task to be performed. Code numbers have no intrinsic meaning other than as identifiers.

Errors

Dyalog APL will report a *`DOMAIN ERROR`* in the following situations.

- If the left argument to *`⎕CMD`*/*`⎕SH`* is not the name of a file containing a valid Auxiliary Processor.
- If the AP supplies an invalid name to **`define()`**.
- Under UNIX and Windows, an AP may terminate abnormally if it crashes, or is killed by a user. In such cases, APL is unaware of its termination and a subsequent reference to an external function will cause *`DOMAIN ERROR`*.

An Example Auxiliary Processor

Listing 1 below shows outline 'C' source code for a simple Auxiliary Processor under Windows. It contains one external function **`idn_d`** (*`IDN_TO_DATE`*) for converting dates from International Day Number (number of days since 1st January 1900) to character format (eg 18 Jan 94, or 18 Jan 1994), and another external function **`d_idn`** (*`DATE_TO_IDN`*) for converting the other way.

Normally a 'C' program begins execution with the special function **`main()`**. Note however, for a Windows AP, `main()` is replaced by **`ap_main()`**. (The supplied file `example.c` actually contains extra code at the beginning to determine which function to call (`main` or `ap_main`), depending upon the current platform.)

```
#include     <apl.h>
#include     <support.h>

UCH taskid[] = "example";

int ap_main(int argc, UCH *argv[])
{
      ARRAY *rslt, *larg, *rarg,
            *getarray(), *freearray(),
            *idn_d(), *d_idn();
      int   fncode;

      startup(2);
      define("IDN_TO_DATE", RSLT|OPTL, 1);
      define("DATE_TO_IDN", RSLT|MONA, 2);

      larg = rarg = rslt = 0;
      setjmp(errbuff);
```

```
        for(;;)
              {
              larg = freearray(larg);
              rarg = freearray(rarg);
              rslt = freearray(rslt);

              fncode = fromapl();

              switch(fncode)
                    {
                    case -1:
                          toapl(0);
                          fromapl()
                          exit(0);
                    case 1:
                          larg = getarray();
                          rarg = getarray();
                          rslt = idn_d(larg,rarg);
                          break;
                    case 2:
                          rarg = getarray();
                          rslt = d_idn(rarg);
                          break;
                    }
              putarray(rslt);
              }
  }
```

```
ARRAY *idn_d(larg, rarg)      // international day number
                              // to date
ARRAY *larg, *rarg;
{

      ARRAY       *rslt;
      UCH   *cptr, *format;
      int         (*makedint)(), (*initdint())();
      int         year, day, leap, i, numchars;
      unsigned    idn;

      static int day_tab[2][13] =
                  {
      {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
      {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
                  };
      static UCH month[][4] =
                  {
      "Nul", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
            "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
                  };
      static UCH cbuf[12];

      if(larg)
            {
            numchars = 11;
            format = "%02d %s %4d";
            }
      else
            {
            numchars = 9;
            format = "%02d %s %02d";
            }

      if(rarg->type == NESTED || rarg->eltype == APLCHAR)
error(DOMAIN);
      if(rarg->rank > 1) error(RANK);
      if(rarg->rank && arraybound(rarg) != 1)
error(LENGTH);
```

```
        makedint = initdint(rarg);
        idn = (*makedint)((int *)arraystart(rarg),0);

        year = idn/365;
        day = (idn - year*365)
                      - ((year == 0) ? year : year-1)/4;
        year = year - (day <= 0);
        leap = ((year%4 == 0) && (year%100 != 0))
                              || (year%400 == 0);
        day = day + (day <= 0) * (365 + leap);

        for(i = 1; day > day_tab[leap][i]; i++)
             day -= day_tab[leap][i];

        rslt = mkarray(APLCHAR, 1, &numchars);
        cptr = arraystart(rslt);
        sprintf(cbuf, format, day, month[i],
             larg ? 1900+year : year);
        for(i = 0; i < numchars; i++)
             *cptr++ = unixtoav(cbuf[i]);

        return rslt;
}
```

```
ARRAY *d_idn(rarg)            // date to international day
                              // number
ARRAY *rarg;
{
      ARRAY       *rslt;
      UCH   *strcpy(), avdate[11], uxdate[11];
      UCH   mnth[4];
      int         day, year, leap;
      int         i, mno, zero;

      static int day_tab[2][13] =
                  {
      {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
      {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
                  };

      static UCH month[][4] =
                  {
      "Nul", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
            "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
                  };

      if(rarg->rank != 1) error(RANK);
      if(rarg->eltype != APLCHAR) error(DOMAIN);
      if(rarg->shape[0] > 11) error(LENGTH);

      strncpy(avdate, arraystart(rarg), rarg->shape[0]);
      avdate[rarg->shape[0]] = '\0';
      for(i=0; i<11; i++) uxdate[i] = avtounix(avdate[i]);
      sscanf(uxdate, "%d %3s %d", &day, mnth, &year);

      mno = 1;
      while((strcmp(mnth, month[mno]) != 0)
                                    && mno <= 12) ++mno;

      if (mno > 12) error(DOMAIN);
      year = (year > 1900) ? (year-1900) : year;
      if ((year < 0) || year > 99 ) error(DOMAIN);
      leap = ((year%4 == 0) && (year%100 != 0))
                              || (year%400 == 0);
      if ((day < 1) || day > day_tab[leap][mno])
                                    error(DOMAIN);
```

```
        for(i = 1; i < mno; i++) day += day_tab[leap][i];
        day = day + (365 * year) + (year - 1) / 4;

        zero = 0;
        rslt = mkarray(APLLONG, 0, &zero);
        rslt->shape[0] = day;

        return rslt;
}
```

Listing 1: Source file `example.c`

The preliminary section of code in Listing 1specifies inclusion of header files **`apl.h`** and **`support.h`**, which are supplied with Dyalog APL. **`UCH`** is declared to be an `unsigned char` in **`apltypes.h`**. **`apltypes.h`** is included in **`machine.h`** which is included in **`apl.h`** which is itself included in **`example.c`**.

The first program section of `example.c` deals with initialisation. In this example, the AP fixes two external functions called *`IDN_TO_DATE`* and *`DATE_TO_IDN`*, which are associated with corresponding functions in the AP, **`idn_d`** and **`d_idn`**. **`idn_d`** and **`idn_d`**, are declared as functions which returns a result of type **`ARRAY*`** (as is the supplied toolkit function **`getarray`** defined in `support.c` that forms part of **`apl.lib`**). They return structure pointers to APL arrays. The local variables `larg`, `rarg`, and `rslt` are declared likewise. The structure **`ARRAY`** is defined in **`apl.h`** (see extract in Listing 2 below).

```
…
#define ALIGNWD(A) unsigned:sizeof(unsigned)*8-A

typedef struct                // an APL ARRAY
{
      unsigned length;  // tot length (words) incl self

      unsigned type     :4;   // see below
      unsigned rank     :4;   // rank of array
      unsigned eltype   :4;   // type of all elements)
      ALIGNWD(12);            // forces word alignment
      unsigned shape[1];      // rho vector - shape[rank]
} ARRAY;
…
#define     NESTED      0x7

#define     APLCHAR     0           // APL ARRAY eltypes

#define     RANK        4
#define     LENGTH      5
#define     DOMAIN      11
…
```

Listing 2: Extract from include file `apl.h`

Following the declarations, the AP calls **`startup`** to inform APL how many external functions are to be fixed. It then calls **`define`** to inform APL of the name, syntax and code number for each external function. In this example, *`IDN_TO_DATE`* (code number 1) is ambivalent (monadic or dyadic), *`DATE_TO_IDN`* (code number 2) is monadic, and both return results. The local variables `larg`, `rarg`, and `rslt` are initialised to zero (empty) and the standard 'C' library function `setjmp` is invoked to control error returns. **`errbuff`** is defined in **`support.h`**. **`startup`** and **`define`** are supplied toolkit functions.

The control section of the AP is a never ending loop. Space occupied by APL objects is released with **`freearray`** and the AP waits for the next instruction from APL using **`fromapl`**. A value -1 received by `fromapl` means that APL is going to terminate the AP. In this case, there is no special "clearing-up" to do, and the AP simply sends an acknowledgement and awaits shutdown.

When `fromapl` receives a function code (either 1 or 2) it uses **`getarray`** to read the argument(s), and calls either **`idn_d`** or **`d_idn`** to process the data in the desired manner. The result is then sent to APL using **`putarray`**. Functions **`freearray`**, **`fromapl`**, **`getarray`** and **`putarray`** are supplied toolkit functions.

Figure 1 shows the program logic diagrammatically.

```
                      ┌<───────────────────────────┐
                      │                            │
                      │                            │
free space occupied by last set of APL objects     │
                      │                            │
                      │                            │
get instruction (function code) from APL           │
                      │                            │
                      │                            │
       (1)┌───────────┴───────────┐(2)             │
          │                       │                │
          │                       │                │
get right & left args       get right arg          │
          │                       │                │
          │                       │                │
          │                       │                │
call function idn_d      call function d_idn       │
          │                       │                │
          └───────────┬───────────┘                │
                      │                            │
                      │                            │
           send result back to APL                 │
                      │                            │
                      │                            │
                      └───────────────────────────>┘
```

Figure 1: The control section of a simple Auxiliary Processor

The function **`idn_d`** is declared as taking two arguments, **`larg`** and **`rarg`**, which together with the result are pointers to APL arrays. The local variable **`rslt`** is declared likewise. Of the other local declarations, those for `initdint`, a supplied toolkit routine, and for `makedint`, are noteworthy. APL arrays store data in various internal formats depending upon type and value. To relieve the programmer of the task of conversion to a standard 'C' data type, a set of access routines is provided. **`initdint`** is one of these. It returns a pointer to the function (also provided) which will obtain an APL array element as a default integer, whatever the internal format of the array. **`initdint`** is therefore declared as a function that returns a pointer to a function, and **`makedint`** is itself declared as a function pointer.

`idn_d` illustrates the use of ambivalence. If any left argument was specified it will return the result in 'DD MMM YYYY' format. If not, it will return a string in the format 'DD MMM YY'. `larg`, which contains a pointer to the left argument, will be zero if none was specified.

Following the test for the presence of a left argument are four data validity checks on the right argument. **idn_d** requires that it be supplied with a single number, the International Day Number, which is to be converted. First, it checks that the array is not nested; then that the data is numeric. If either test fails, it issues a *DOMAIN ERROR*. Next, it checks that the array is a scalar or vector, and issues a *RANK ERROR* if an array of higher rank is specified. Finally, it checks that if the array is a vector, its length is 1; otherwise, it issues a *LENGTH ERROR*. **type, rank** and **eltype**, which are used to access the header information, are part of the structure definition of **ARRAY** in **apl.h** where the globals `NESTED`, `APLCHAR`, `DOMAIN`, `RANK` and `LENGTH` are also declared (see extract in Listing 2 above). **error** is a toolkit function, which signals an error to APL, after which APL will resume processing and the AP will return to its wait state. **arraybound** gives the number of elements in an APL array.

The next two statements in `example.c` extract the international day number from the APL array as an integer of the default size of the computer. (Some machines default to 2 byte integers, others to 4.) Subsequent statements calculate the year, month and day. Next, **idn_d** uses **mkarray** to build an empty APL array in memory, which is to be a text vector (type `APLCHAR`; rank 1) of length given by **numchars**. Note that the third argument supplied to **mkarray** is an address (`&`), which points to the shape of the array to be initialised.

mkarray initialises the APL array, but does not fill it with any data. The remaining statements in **idn_d** illustrate how this is done. An APL array consists of header, of varying length, followed by data. First, **arraystart** is used to obtain a character pointer to the beginning of the data. Then the date is copied character by character into the array starting at this point. **unixtoav** is used to convert characters from ASCII to *⎕AV* representation. **arraystart** and **unixtoav** are toolkit functions, whose definitions are to be found in `support.c` and `trans.c` respectively. Finally **idn_d** returns a pointer to the APL array.

Hints on Developing APs

Developing APs under Windows

Dyalog APL/W currently supports Auxiliary Processors produced by the Microsoft C/C++ 32-bit Optimizing Compiler Versions 12 and 13 for 80x86. Version 13 is supplied with Microsoft Visual Studio .Net and is used to produce Dyalog APL itself. Version 12 is supplied with Microsoft Visual Studio Version 6. APs may also be produced with compilers such as those from Watcom, Borland, and GNU.

Compiling C Source Code

Consider the simple case of the source file `C:\simple.c` in Listing 3 below.

```
#include <stdio.h>

main()
{
     printf ("This is a C program.\n");
}
```

Listing 3: A very simple C program

In order to compile this file, start a Visual Studio Command Prompt from the Windows Start button. This sets the environment for using the Microsoft Visual Studio .NET tools (by running the Visual Studio batch file `vsvars32.bat,` which specifies directory paths to various executables, libraries and include files such as `stdio.h`).

Change the current directory appropriately, and type

```
C:\> cl.exe simple.c
```

This compiles (and links) the above C program yielding an executable file called `simple.exe` which can then be run *in situ* from the command line.

```
C:\> simple.exe
This is a C program.
```

Details of the command line syntax of the compiler **`cl.exe`**, and the various command line options available to it, can be found in the MSDN library or Visual Studio .NET documentation.

Compiling and Linking an AP

In general, an AP will require links to functions in **`apl.lib`** and other common source files. Environment variables `LIB` and `INCLUDE` may be extended to make all the required files visible to the linker.

For example, to compile the elementary AP file **`example.c`**, which may be found in the directory `C:\dyalog\xfsrc\`, begin by opening a Visual Studio Command Prompt from the Start button and change the current directory to `\xfsrc\`. In order to find the specified include files, **`apl.h`** and **`support.h`**, in the current directory (**`.`**), type

```
C:\dyalog\xfsrc> set INCLUDE=%INCLUDE%;.
```

In order to find the **`apl.lib`** library, which is in subdirectory `\mscv7\`, type

```
C:\dyalog\xfsrc> set LIB=%LIB%;.\msvc7\
```

The following line will compile the program source file **`example.c`** (above) into a binary object code file **`example.obj`**.

```
C:\dyalog\xfsrc> cl.exe /c example.c
```

Command line option `/c` requests compilation *without* linking. Linking is then performed in a separate step using the Microsoft Incremental Linker **`link.exe`**. The linker links object files and libraries, resolves named resources, and creates a 32-bit executable (**`.exe`** file) or a dynamic-link library (**`.dll`** file).

```
C:\dyalog\xfsrc> link /subsystem:windows example apl.lib
```

This command links together object code in `example.obj` with toolkit library object code in `apl.lib`, as required by `example.c`, in order to produce the self-contained executable **`example.exe`**. In general, there may be many object files and library files involved in the `link.exe` command line. The linker option `/subsystem:` tells the operating system how to run the `.exe` file. The various command line options can be found in the MSDN library or Visual Studio .NET documentation.

Running the Example AP

The new example AP can be started immediately, and tested, in a Dyalog APL session.

```
      'C:\dyalog\xfsrc\example'⎕CMD''

      IDN_TO_DATE 34534
20 Jul 94
```

Compare this result with that of the Root method `IDNToDate` whose description is to be found in the Dyalog APL *Object Reference*.

```
      #.IDNToDate 34534
1994 7 20 2
```

Windows Makefiles

A *make file* coordinates the combination of multiple options, source files, library files and other resource files for compilation and linking. Compilation and linking commands, such as those used above, are supplemented with further compiler and linker options and incorporated into the distributed *make script* **example.makefile** listed below.

```
#  EXAMPLE makefile
#  modify this file for your own AP/DLL
#  currently set up for microsoft compiler
# change to DEBUG=1 to build a debug executable
DEBUG=0
# change to DLL=1 to build a dynamic link library, else it's an AP
DLL=0
#location of intermediate files(eg .obj) Set to "."(without quotes)for current dir.
OBJDIR=          msvc7
# location of final executable. Set to "." (without quotes) for current dir.
EXEDIR=          ..\xflib
# binary to create
TARGET=          $(EXEDIR)\example.exe
# sources from which it is built
SOURCES=         example.c
#include files that it uses
INCLUDES=        apl.h            \
                 support.h
#additional libraries
LIBS=            user32.lib       \
                 advapi32.lib
OBJECTS=$(OBJDIR)\$(SOURCES:.c=.obj)

!if $(DEBUG)
DFLAGS=          /Od
LDFLAGS=         /debug /map
!else
DFLAGS=          /Ox
!endif

!if  $(DLL)
DLLFLAGS=        /DLL
!else
APLLIB=          $(OBJDIR)\apl.lib
!endif

COMPILER=        cl.exe
LINKER=          link.exe
RC=              rc.exe
# compiler flags
CFLAGS=/Zi /c /nologo $(DFLAGS) /I. /Fd$(OBJDIR)\$(*B).pdb /Fo$(OBJDIR)\$(*B).obj
# linker flags
LFLAGS=$(LDFLAGS) $(DLLFLAGS) /nologo /INCREMENTAL:NO /SUBSYSTEM:WINDOWS
all: $(TARGET)

$(OBJECTS): $(SOURCES)
        @$(COMPILER) $(CFLAGS) $(*B).c
$(TARGET): $(OBJECTS)
        @$(LINKER) $(LFLAGS) /OUT:$@ $(OBJECTS) $(APLLIB) $(LIBS)
clean:
        -@erase /q  $(TARGET)
        -@erase /q  $(OBJECTS)
```

Listing 4: Compilation script file `example.makefile` to build example AP

Using the distrubuted make script in Listing 4, the source code in `example.c` may be compiled and linked using the Visual Studio nmake tool, **`nmake.exe`**. (Note that `nmake.exe` is the Windows version of the UNIX `make` command.)

```
C:\dyalog\xfsrc> nmake /f example.makefile
```

The `example.makefile` script contains compiler and linker debugging options that furnish debugging information within **`.pdb`** files, whose purpose is to regenerate source code in the debugger from object code, as and when required.

Another distributed **`makefile`** contains the instructions needed to compile (if necessary) and (re-)link all of the APs for which 'C' source code is supplied. Note that, to save space, the `.obj` files are not distributed with Dyalog APL, so the source will be automatically compiled first.

In order to run this script you simply need to type `nmake` (because "makefile" is the default filename used by `nmake`).

```
C:\dyalog\xfsrc> nmake
```

(A `.bat` file called `make.bat` is supplied to initiate the `nmake`.)

In the `makefile` script of Listing 5 below, after a number of variables have been defined, all dependent files ending in `.c` are compiled via the line

```
@$(COMPILER) $(CFLAGS) $(*B).c
```

Then four separate calls to link.exe are made. The first `LINKER` line links `prt.obj` (from compiled `prt.c`) with `apl.lib` and some other standard 'C' libraries to produce the `prt.exe` - the PRT auxiliary processor for driving printers.

```
@$(LINKER) $(LFLAGS) /OUT:$@ $(OBJDIR)\prt.obj $(LIBS)
```

The second `LINKER` line produces the XUTILS AP, and the third produces the NFILES AP.

```
#  AP makefile. Makes all the APS
#  see example.makefile for a makefile to make a single AP.

!IFDEF DEBUG
DFLAGS=/Od
!ELSE
DFLAGS=/Ox
!ENDIF

COMPILER=cl.exe
LINKER=link.exe
CFLAGS=/Zi /c /J /nologo $(DFLAGS) /I.     \
        /Fd$(OBJDIR)\$(*B).pdb /Fo$(OBJDIR)\$(*B).obj
LFLAGS=/debug /nologo /INCREMENTAL:NO /SUBSYSTEM:WINDOWS

EXEDIR=..\xflib
OBJDIR=msvc7
WINLIBS=         user32.lib                \
                 advapi32.lib              \
                 gdi32.lib
INCLUDES=        apl.h                     \
                 apltypes.h                \
                 io_maps.h                 \
                 machine.h                 \
                 quadav.h                  \
                 regexpav.h                \
                 support.h
APLLIB=  $(OBJDIR)\apl.lib
LIBS=            $(APLLIB) $(WINLIBS)
TARGETS=         $(EXEDIR)\nfiles.exe      \
                 $(EXEDIR)\prt.exe         \
                 $(EXEDIR)\xutils.exe      \
                 $(EXEDIR)\dyalog32.dll
EXPORTS=         /export:MEMCPY            \
                 /export:STRNCPY

all: $(TARGETS)
# build example.exe
        @nmake /f example.makefile

.c{$(OBJDIR)}.obj:
        @$(COMPILER) $(CFLAGS) $(*B).c

$(EXEDIR)\prt.exe:         $(OBJDIR)\prt.obj        $(APLLIB)
        @$(LINKER) $(LFLAGS) /OUT:$@ $(OBJDIR)\prt.obj $(LIBS)
$(EXEDIR)\xutils.exe:     $(OBJDIR)\xutils.obj      $(APLLIB)
        @$(LINKER) $(LFLAGS) /OUT:$@ $(OBJDIR)\xutils.obj $(LIBS)
$(EXEDIR)\nfiles.exe:     $(OBJDIR)\nfiles.obj      $(APLLIB)
        @$(LINKER) $(LFLAGS) /OUT:$@ $(OBJDIR)\nfiles.obj $(LIBS)
$(EXEDIR)\dyalog32.dll:   $(OBJDIR)\dyalog32.obj $(APLLIB)
        @$(LINKER) /DLL $(LFLAGS) /OUT:$@  $(OBJDIR)\dyalog32.obj \
                 $(LIBS) $(EXPORTS)
clean:
        -@erase /q  $(EXEDIR)\*.*
        -@erase /q  $(OBJDIR)\*.obj $(OBJDIR)\*.pdb
```

Listing 5: Compilation script file `makefile` to build all distributed APs

Creating a DLL for Use with ⎕NA

The fourth call to link.exe in `makefile` (of Listing 5 above) produces a DLL that contains two utility functions `MEMCPY` and `STRNCPY`. These are exported versions of standard 'C' library functions `memcpy` and `strncpy`. The command line option `/DLL` in the `makefile` line below effects the creation of a dynamic link library rather than of an executable file (the default). The content of **`dyalog32.c`** is listed below.

```
@$(LINKER) /DLL $(LFLAGS) /OUT:$@ $(OBJDIR)\dyalog32.obj\
            $(LIBS) $(EXPORTS)

#include <machine.h>
#include <windows.h>
#include <stdlib.h>
#include <string.h>

typedef void *VP;

void MEMCPY(VP to, VP from, unsigned nbytes)
      {
      memcpy
            (
            to
      ,     from
      ,     nbytes
            );
      }

void  STRNCPY(VP to, VP from, unsigned nbytes)
      {
      strncpy
            (
            to
      ,     from
      ,     nbytes
            );
      }

int APIENTRY DllMain
(HANDLE hdll, DWORD reason, LPVOID  reserved)
{
return 1;
}
```

Listing 6: Source file `dyalog32.c`

An example of the use of these two exported functions, `MEMCPY` and `STRNCPY`, may be found in the distributed workspace, QUADNA.DWS.

Note that the library `apl.lib` is created using Visual Studio program `lib.exe`.

Debugging an AP

Dyalog APL/W Auxiliary Processors can be tested using the standard Microsoft debugging tools. To start an AP using the debugger, the left argument should contain the full command line including the debugger options, rather than just the name of the `.EXE` file.

Assuming that the Visual Studio program `devenv` and the example AP executable are both on the path, then the example AP may be started in debug mode with the APL statement

```
      'devenv /debug example'⎕CMD ''
```

However, by the time the debugger has been initiated, the time expected to start the AP might have been exceeded, causing the AP to expire with a DOMAIN ERROR. The time limitation can be removed by setting a DEBUGAP flag on the Dyalog command line. The value of the flag is not important; merely its existence is checked.

```
C:\DYALOG\DYALOG.EXE  DEBUGAP=1
```

In order to exploit the full power of the debugger, the AP should have been created with debug flags set in the make file. These are set accordingly in the CFLAGS and LFLAGS variables in the make files of Listings 4 and 5 above.

Hint: In order to cause a break in an AP at some particular point of interest in the AP code, it is sometimes helpful to create a simple message box in the code at that point. The debugger will then halt at the point of interest in the C program, or can be started after the AP and attached to the running process.

Developing APs under UNIX

Makefiles in UNIX

The **`makefile`** distributed with Dyalog APL/M is suitable for building the Auxiliary Processors for which source code is supplied. This `makefile` may be used to develop extensions and modifications to the supplied APs or as a model for developing new ones. It is similar to Windows make files described above, but the compiler is (typically) called `cc` rather than `cl.exe`, the linker is called `ld` rather than `link.exe`, and object files have extension `.o` rather than `.obj`.

Debugging an AP in UNIX

Various techniques and software are available for debugging. The **`printf`** statements included in the source code can be used to display diagnostic messages and data values during execution. Alternatively, Dyalog APL/M and the AP can be run under special debugging software such as **`dbx`**. To achieve this, it is necessary to start **`dbx`** under an interactive shell. For example:

```
      '/bin/sh' ⎕SH'sh' '-i'  (starts interactive shell)

# stty sane<LF>               (reset terminal I/O)

# echo <CTRL O>               (select ASCII)

# exec dbx example            (run example AP under dbx)

*                             (dbx prompts for instructions)
```

No `DEBUGAP` flag is necessary in UNIX as APs are not timed out on start-up.

Modifying Distributed APs

Source code for certain Auxiliary Processors is supplied to enable the user to extend and customise the routines that are provided. It is likely, however, that Dyadic will provide its own extensions and developments in later releases. It is recommended, therefore, that users take the following steps to ensure that local changes are not lost when a new release is installed.

1. Retain a copy of the original source code as supplied.

2. When installing a new release, ensure that new AP source code will not overwrite existing customised code by first taking copies, or by installing in a different directory.

Error Handling

APL Errors

Synchronous External Functions (i.e. those that return a result) may generate APL errors using the **`error()`** function call. Asynchronous External Functions (i.e. those that don't return a result) may not.

The effect of calling **`error()`** is very similar to the effect of using *⎕SIGNAL* in a defined function or operator. The only real difference is that **`error()`** only signals an error number, and not an error message. An event generated by **`error()`** may of course be caught by *⎕TRAP*.

The internal mechanism for handling errors between APL and an AP is unimportant. However, you should take note of the following points:

- Your AP must execute the statement **`setjmp(errbuff);`** before it uses **`error()`**.
- Under UNIX all error handling is controlled using a signal defined by `SIGHANDSHAKE`. This is normally 15 but is different on some machines (e.g. it is 31 on Suns). Do not use this signal for your own purposes. This does not apply to Windows.

Interrupts

Under Unix, External Functions may be declared to handle their own interrupts (see **`define()`**). This does not apply under Windows.

Under UNIX this means that APL disables its own interrupt handling (**intr** and **quit**) before calling an External Function, and re-instates it on return. An External Function must then take its own action if the user presses one of these keys.

The AP-Writer’s Toolkit

The AP writer's toolkit is a directory of C source files and object libraries containing a number of useful functions, symbolic constants, and macro substitutions.

The supplied files include the following:

`apl.lib`	compiled module library (Windows only)
`machine.h` **`d_machine.h`**	machine dependent parameters
`apl.h`	array definitions and symbolic constants
`support.c` **`support.h`**	auxiliary processor interface functions
`access.c`	APL to C data conversion functions
`aconvs.c`	APL array conversion functions
`quadav.h`	⎕*AV* tables
`getch.c`	reads the input stream (low-level)
`trans.h` **`trans.c`**	input and output translation functions
`tty.c`	terminal control functions (UNIX)
`resolve.c`	overstrike resolution functions

These source files are provided for information only. Some of them are part of the Dyalog APL interpreter and contain some functions that are of no interest to the AP writer. Such functions are not documented in this section.

The remainder of this Chapter describes some of the functions provided in the **apl.lib** library.

The AP Writer's Toolkit

Function	Description
`aget_a`	copies an APL array
`arraybound`	returns the number of elements in an `ARRAY`
`arraystart`	returns a pointer to the start of the data in an `ARRAY`
`avtoasc`	displays a `⎕AV` character on the current output device
`avtounix`	converts a value from `⎕AV` into its ANSI or nearest ASCII equivalent
`convlen`	determines the size of the buffer needed to contain the `⎕AV` representation of an ASCII string
`convtoav`	converts an ASCII string to `⎕AV`
`data`	accesses the start of data in an `ARRAY`
`define`	defines the name, syntax and code number of an external function
`dget_b`	get `DOUB` array from `BOOL` array
`dget_i`	get `DOUB` array from `INTG` array
`dget_l`	get `DOUB` array from `LONG` array
`dget_s`	get `DOUB` array from `SINT` array
`domain`	signals a DOMAIN ERROR
`error`	signals an error
`freearray`	releases space occupied by an APL `ARRAY` in the AP
`fromapl`	receives data from APL
`getarray`	reads an `ARRAY` into the AP

The AP Writer's Toolkit
continued…

Function	Description
`getd`	accesses an object with eltype `DOUB` as a double
`iget_b`	get `INTG` array from `BOOL` array
`iget_s`	get `INTG` array from `SINT` array
`igetb`	accesses an object with eltype `BOOL` as an int
`igetd`	accesses an object with eltype `DOUB` as an int
`igeti`	accesses an object with eltype `INTG` as an int
`igetl`	accesses an object with eltype `LONG` as an int
`igets`	accesses an object with eltype `SINT` as an int
`initdint`	returns the function needed to access data in an `ARRAY` as type int
`initint`	returns the function needed to access data in an `ARRAY` as type int
`initintg`	returns the function needed to access data in an `ARRAY` as a 16-bit integer
`initlong`	returns the function needed to access data in an `ARRAY` as type long
`inittty`	initialises terminal control parameters (applicable to UNIX only)
`init_out`	initialises an output stream for a block of output
`inkey`	gets a keystroke
`i_init`	gets Input Table and Output Table(s)

The AP Writer's Toolkit continued…	
Function	**Description**
`lget_b`	get `LONG` array from `BOOL` array
`lget_i`	get `LONG` array from `INTG` array
`lget_s`	get `LONG` array from `SINT` array
`lgetb`	accesses an object with eltype `BOOL` as a long
`lgetd`	accesses an object with eltype `DOUB` as a long
`lgeti`	accesses an object with eltype `INTG` as a long
`lgetl`	accesses an object with eltype `LONG` as a long
`lgets`	accesses an object with eltype `SINT` as a long
`mkarray`	sets up an `ARRAY` with a given element type, rank and shape
`o_init`	gets Output Table(s)
`putarray`	returns a result from an external function
`putd`	places a value of type double in an element of an `ARRAY`
`resolve`	resolves APL overstrikes
`round`	rounds a double to the nearest long integer
`sget_b`	get `SINT` array from `BOOL` array
`startup`	establishes communication between the AP and the APL task

The AP Writer's Toolkit
continued…

Function	Description
`termaout`	sets up (nearly) raw terminal handling (applicable under UNIX only)
`termcontext`	restores terminal handling (applicable under UNIX only)
`throwarray`	returns a result from an external function and release the space it occupied in the AP
`unixtoav`	converts an ASCII character into its equivalent value in ⎕AV

The Basic Definitions File, apl.h

The supplied file **`apl.h`** contains the `ARRAY` definition and symbolic constants for auxiliary processors.

These definitions can be used inside a user defined C function by a control line of the form:

```
#include "apl.h"
```

or

```
#include <apl.h>
```

Definition of an Internal Array

`apl.h` contains, in particular, the definition of an `ARRAY` (i.e. the internal representation of Dyalog APL data objects), as well as a set of useful symbolic constants.

An `ARRAY` is defined as follows:

```
…
typedef struct     /* an APL ARRAY                */
{
unsigned length;   /* total length in words
                      including self              */
unsigned type  :4; /* see below                   */
unsigned rank  :4; /* rank of array               */
unsigned eltype:4; /* type of all the elements
                      (see below)                 */
ALIGNWD(12);       /* forces word alignment       */
unsigned shape[1]; /* shape followed by data      */
} ARRAY;
…
```

Listing 7: Extract from include file apl.h

type is either

`SIMPLE (0xf)` signifying a simple APL object
or
`NESTED (0x7)` signifying object is a nested array

eltype is either

`CHAR (0)` signifying elements are characters
`BOOL (1)` signifying elements are boolean
`SINT (2)` signifying elements are integers (small)
`INTG (3)` signifying elements are integers (normal)
`LONG (4)` signifying elements are integers (large)
`DOUB (5)` signifying elements are floating-point
or
`PNTR (6)` signifying elements of a nested array

In addition the name `DINT` is defined to be either 3 (`INTG`) or 4 (`LONG`). `DINT` represents the default integer type and is machine dependent.

The procedure `ALIGNWD(A)` which is called in the `ARRAY` definition is defined as:

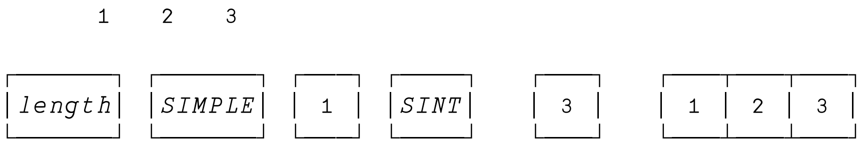
```
unsigned:sizeof(unsigned)*8-A
```

and is necessary to force word boundary alignment.

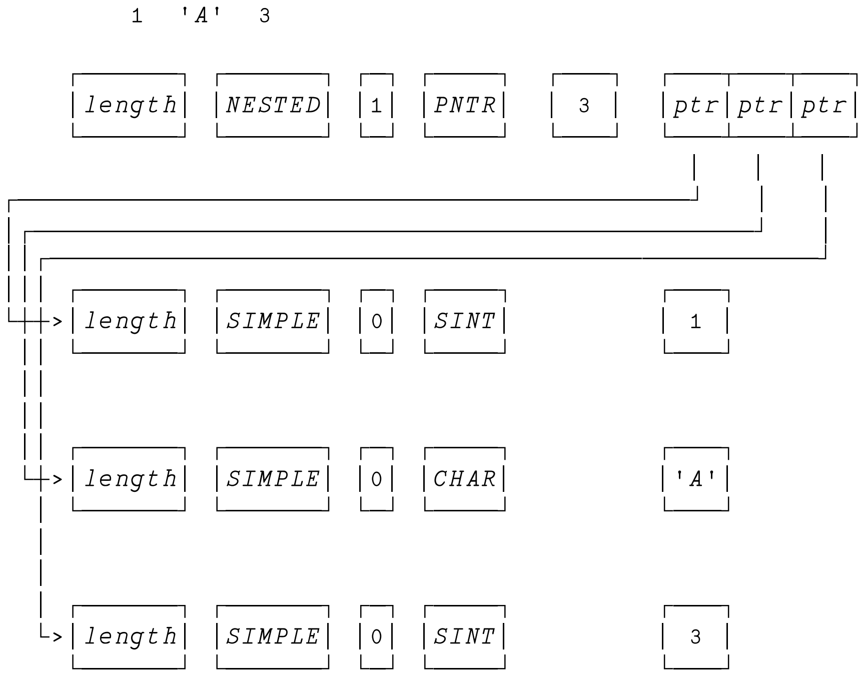
Internal Array Structure

```
|←WORD 1→|<───────WORD 2───────>|
|        |                      |
|        |   aligned on word    | number| aligned to |
|        |    boundary if       |   of  |   whole    |
|        |     necessary        | words | number of  |
|        |                      | = rank|   words    |
|┌──────┐|┌────┐ ┌────┐ ┌──────┐|┌─────┐|┌──────────┐|
||      |||    | |    | |      |||     |||          ||
||length|||type| |rank| |eltype|||shape|||array data||
|└──────┘|└────┘ └────┘ └──────┘|└─────┘|└──────────┘|
            |             |         |         |
            |             |         |         |
            |             |         |         |
            v             v         v         |
                                              |
        ┌──────┐       ┌────┐  ┌─────────┐    |
        |SIMPLE|       |CHAR|  |one word |    |
        |NESTED|       |BOOL|  |per axis |    |
        └──────┘       |SINT|  |starts   |    |
                       |INTG|  |on word  |    |
                       |LONG|  |boundary |    |
                       |DOUB|  └─────────┘    |
                       |    |                 v
                       |PNTR|         ┌──────────────┐
                       └────┘         |              |
                                      |CHAR   1 byte |
                                      |BOOL   1 bit  |
                                      |SINT   1 byte |
                                      |INTG   2 bytes|
                                      |LONG   4 bytes|
                                      |DOUB   8 bytes|
                                      |PNTR          |
                                      └──────────────┘
```

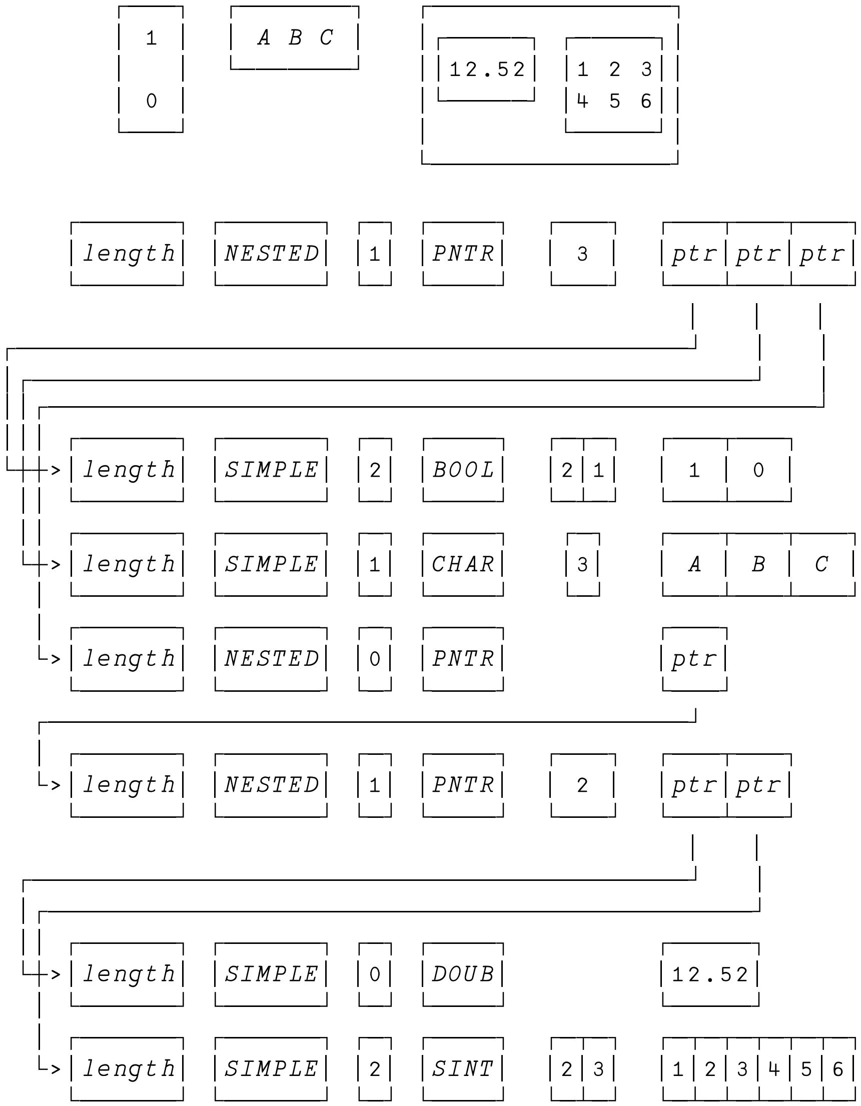
Example 1: Simple Array (Homogeneous)



Example 2: Simple Array (Heterogeneous)



Example 3: Nested Array



Symbolic Constants

Symbolic constants are defined (in apl.h) as follows:

Used for	Name	Value	Description
Pipes	`ARGSPIPE`	3	APL –>AP pipe number
	`RSLTPIPE`	4	AP –>APL pipe number
External function syntax	`RSLT`	0040	result returning
	`SHYR`	0140	shy-result returning
	`NILA`	0000	has no arguments
	`MONA`	0004	takes only a right argument
	`DYAD`	0014	takes both left and right arguments
	`OPTL`	0034	left argument is optional
	`INTR`	0200	handles its own interrupts
APL error codes	`WSFULL`	1	
	`SYNTAX`	2	
	`INDEX`	3	
	`RANK`	4	
	`LENGTH`	5	
	`VALUE`	6	
	`LIMIT`	10	
	`DOMAIN`	11	
	`NONCE`	16	
	`NOPIPES`	72	
	`NOPROCS`	76	
	`INTERRUPT`	1003	
	`EOF_INTERRUPT`	1005	
AP error codes	`PRE_BASE`	200	To avoid clashes in error codes, each supplied AP is allotted 20 unique error codes in the range 200 - 499
	`VIA_ERRBASE`	220	
	`XUT_ERRBASE`	240	
	`OFS_ERRBASE`	260	

The apl.lib Library

Copy Array: `aget_a`

Names

`aget_a, sget_b, iget_b, dget_b, iget_b, iget_s, dget_s, lget_i, dget_i, dget_l` - array conversion routines.

Synopsis

```
ARRAY *aget_a(source)
ARRAY *source;
```

Description

The array conversion routines are a collection of functions to convert APL data arrays between various different internal representations. They are used in writing Auxiliary Processors where the programmer wishes to create arrays of a specific type, whatever the type of the arguments supplied. They comprise the following functions, which are defined in `aconvs.c`:

`aget_a` – direct copy of an APL array

`sget_b` – get `SINT` array from `BOOL` array
`iget_b` – get `INTG` array from `BOOL` array
`lget_b` – get `LONG` array from `BOOL` array
`dget_b` – get `DOUB` array from `BOOL` array
`iget_s` – get `INTG` array from `SINT` array
`lget_s` – get `LONG` array from `SINT` array
`dget_s` – get `DOUB` array from `SINT` array
`lget_i` – get `LONG` array from `INTG` array
`dget_i` – get `DOUB` array from `INTG` array
`dget_l` – get `DOUB` array from `LONG` array

They are used via **`convfns`**, a 2-dimensional table of function pointers. The first dimension indicates the element type to be converted from, the second dimension the element type to be converted to. As there are 7 different element types (`CHAR`, `BOOL`, `SINT`, `INTG`, `LONG`, `DOUB`, `PNTR`), **`convfns`** is a 7 by 7 array. Where conversion from one element type to another is invalid or inappropriate, **`convfns`** contains pointers to **`domain`** or **`nonce`**, which signal *`DOMAIN ERROR`* and *`NONCE ERROR`* when used. The full table is:

to from	CHAR	BOOL	SINT	INTG	LONG	DOUB	PNTR
CHAR	aget_a	domain	domain	domain	domain	domain	nonce
BOOL	domain	aget_a	sget_b	iget_b	lget_b	dget_b	nonce
SINT	domain	nonce	aget_a	iget_s	lget_s	dget_s	nonce
INTG	domain	nonce	nonce	aget_a	lget_l	dget_i	nonce
LONG	domain	nonce	nonce	nonce	aget_a	dget_l	nonce
DOUB	domain	nonce	nonce	nonce	nonce	aget_a	nonce
PNTR	nonce	nonce	nonce	nonce	nonce	nonce	nonce

Using this table, the programmer needs only to specify the element type desired in the new array. For example, the expression:

```
new = (*convfns[old->eltype][DOUB])(old)
```

produces an APL array **`new`** with element type `DOUB` from the APL array **`old`**.

Number of Elements: `arraybound`

Name

`arraybound` - returns the number of elements in an `ARRAY`

Synopsis

```
unsigned arraybound(array)
ARRAY *array;
```

Description

Calculates the number of elements in an APL array. (This function is defined in `support.c`.)

Start of Array: arraystart

Name

`arraystart` - returns a pointer to the beginning of data in an `ARRAY`

Synopsis

```
char *arraystart(array)
ARRAY *array;
```

Description

Returns a pointer to the position in an APL array where its data contents begin. The data is preceded by a header word and shape vector. (This function is defined in `support.c`.)

Output Character: avtoasc

Name

`avtoasc` - displays a ⎕*AV* character on the current output device

Synopsis

```
void avtoasc(c)
unsigned char c;
```

Description

`avtoasc` (defined in `trans.c`.) uses the appropriate Output Translate Table to display a character from ⎕*AV*.

See also...

`init_out`

Convert APL to ANSI: avtounix

Name

`avtounix` - converts a value in ⎕AV into its ANSI or nearest ASCII equivalent

Synopsis

```
unsigned char avtounix(c)
unsigned char c;
```

Description

`avtounix` converts a value in ⎕AV (0-255) into its ANSI equivalent (0-255) using the appropriate translate table. If no translate table has been loaded, the value is mapped to the nearest ASCII equivalent (0-127) using the internal table **`unixtab`**. In this case, APL underscored characters map to ASCII lower case and special APL characters such as **rho** or **transpose** map to ASCII 127.

Conversion Length: convlen

Name

`convlen` - determines the size of the buffer needed to contain the ⎕AV representation of an ASCII string

Synopsis

```
unsigned convlen(from)
unsigned char *from;
```

Description

`convlen` determines the size of the buffer that would be needed to contain the ⎕AV representation of an ASCII string as produced by **`convtoav`**. This may be larger because **`convtoav`** expands TAB characters to blanks.

Convert String to APL: `convtoav`

Name

`convtoav` - converts an ASCII string to ⎕*AV*

Synopsis

```
unsigned convtoav(to,from,max)
register unsigned char *to, *from;
register unsigned max;
```

Description

Converts the ASCII string **`from`** into its ⎕*AV* representation **`to`**. TAB characters are expanded to blanks, and overstrikes are resolved. **`max`** specifies the maximum size of the output buffer **`to`**. If **`max`** is reached before the end of **`from`** is encountered, the result is truncated.

Returns

The number of characters in the output buffer **`to`**

See also...

`resolve, unixtoav`

Start of Data: `data`

Name

`data` - macro to access the data part of an `ARRAY`

Synopsis

```
define data(array type)
((type *)&array->shape[array->rank])
```

Description

This macro is useful shorthand for accessing the start of the data portion of an `ARRAY`.

Define Function: `define`

Name

`define` - define an external function to APL

Synopsis

```
define(apfn, syntax, fncode)
char *apfn;
int syntax, fncode;
```

Description

Informs APL of the name, calling syntax, and code number associated with an external function. (The function is defined in `support.c`.) **`apfn`** is a pointer to a character string containing the name of the external function to be fixed in the active WS. The string containing the name is translated to an APL name according to the same rules as for any other Dyalog APL name (see *Language Reference*) which may contain ∆ and ⍙, upper case, lower case, numeric, underscore and underscored characters. To cater for all possibilities, the name is read as if it had been entered at the keyboard starting in ASCII mode, except that the control codes must be specified by an escape sequence as follows:

<Backspace>	\b
<Ctrl N>	\016
<Ctrl O>	\017

Examples:

The following examples illustrate the principle.

1. `FOO` translates to *FOO*
2. `foo` translates to *foo*
3. `foo_BAR` translates to *foo_BAR*
4. `F\b_O\b_O\b_` translates to ↑⍙⍙
5. `\016H123\017a` translates to ∆123*a*
6. `f\016H\bFabc` translates to *f*⍙*ABC*
7. `Fff\b_` translates to *Ff_* (see below)

In example 4 the ASCII O and _(underscore) are resolved to O̲. In example 5, <\016> selects APL mode so that the subsequent <H> is read as <∆>, and the <F> as the APL underscore, resolving to <⍙>. In example 7, <f> <backspace> <_> is an invalid overstrike and so resolves to _.

syntax defines the valence of the external function by the following bit settings:

```
                 7   6   5   4   3   2   1   0
               ┌───┬───┬───┬───┬───┬───┬───┬───┐
syntax         │   │   │   │   │   │   │   │   │
               └───┴───┴───┴───┴───┴───┴───┴───┘
                 ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
                 │   │   │   │   │   │   │   │
                 │   │   │   │   │   │   │   └─ (not used)
                 │   │   │   │   │   │   └───── (not used)
                 │   │   │   │   │   └───────── takes a
                 │   │   │   │   │              right arg.
                 │   │   │   │   └───────────── takes a
                 │   │   │   │                  left arg.
                 │   │   │   └───────────────── ambivalent
                 │   │   └───────────────────── returns a
                 │   │                          result
                 │   └───────────────────────── result is
                 │                              shy
                 └───────────────────────────── handles its
                                                own interrupts
```

For example, if **syntax** has the value 36 (00100100), the external function is defined as being monadic, result returning. To simplify syntax definition, a series of globals `MONA`, `DYAD`, `OPTL`, `RSLT`, `SHYR`, and `INTR` are defined (in **apl.h**) which may be combined using the bitwise exclusive OR operator. Thus instead of

```
define("foo",36,1)
```

the statement can read

```
define("foo",RSLT|MONA,1).
```

fncode is an arbitrary number associated with the external function. When invoked, this code number will be transmitted by APL to the Auxiliary Processor, thereby identifying the task to be performed.

Signal Domain Error: `domain`

Name

`domain` - signals a *DOMAIN ERROR*

Synopsis

```
domain()
```

Description

Signals a *DOMAIN ERROR* to the APL task

Signal Error: `error`

Name

`error` - signals an error to the APL task.

Synopsis

```
error(errnumber)
int errnumber;
```

Description

Signals an error to APL. **`errnumber`** specifies an APL error code number, which will, unless trapped, invoke an error report in the active WS.

Returns

None; but after the error has been processed, and the pipes cleaned up, control will return to the statement after the call to **`setjmp`** in the Auxiliary Processor.

Release Memory: `freearray`

Name

`freearray` - releases memory occupied by an `ARRAY`

Synopsis

```
ARRAY *freearray(arr)
ARRAY *arr;
```

Description

Releases the space occupied by an `ARRAY` using the library call **`free`**. If the `ARRAY` is `NESTED`, **`freearray`** (defined in `support.c`) calls itself recursively to free the space occupied by each sub-array.

Returns

`0`

See also...

`throwarray`

Read Word: `fromapl`

Name

`fromapl` - reads a word of information from the APL task.

Synopsis

```
unsigned fromapl()
```

Description

Reads one word from `ARGSPIPE` (under UNIX) or the shared memory segment (under Windows).

See also...

`apl.h`

Read Array: `getarray`

Name

`getarray` - reads an `ARRAY` from the APL task

Synopsis

```
ARRAY *getarray()
```

Description

Reads an APL data array from `ARGSPIPE` (under UNIX) or from the global memory segment (under Windows). If the array is of type `NESTED`, **`getarray`** (defined in `support.c`) calls itself recursively to read each sub-array.

Returns

A pointer to the array, or the value 0. A 0 is returned when an ambivalent external function is invoked without its (optional) left argument.

Get Element: `getd`

Name

`getd` - accesses an element of an `ARRAY` with element type `DOUB`.

Synopsis

```
double getd(base,indx);
double *base;
unsigned indx;
```

Description

`getd` accesses an element of an `ARRAY` with element type `DOUB` and returns a value of type double. Double-word alignment is forced if required by the machine architecture (`#ifdef DALIGN`). **`base`** is the address of the start of the data in the object, **`indx`** is the offset of the element to be accessed.

Get as Int: `igetb`

Names

`igetb, igets, igeti, igetl, igetd` - int access routines.

Synopses

```
int igetb(base,indx);
unsigned char *base;
unsigned indx;

int igets(base,indx);
char *base;
unsigned indx;

int igeti(base,indx);
long *base;
unsigned indx;

int igetd(base,indx);
double *base;
unsigned indx;
```

Description

These routines (defined in `access.c`) access an element of an `ARRAY` and return its value as an int. **`base`** is the address of the start of the data in the `ARRAY`, **`indx`** is the offset of the element to be accessed. **`igetb`**, **`igets`**, **`igeti`**, **`igetl`** and **`igetd`** are used to access APL data of type `BOOL`, `SINT`, `INT`, `LONG` and `DOUB` respectively. **`igetd`** uses **`getd`** which forces double-word alignment if required (`#ifdef DALIGN`).

`igetl` truncates; **`igetd`** rounds and truncates.

Convert to Int: `initint`

Names

`initint`, **`initdint`**, **`initintg`**, **`initlong`** - determines appropriate access function.

Synopses

```
int (*initint(array))();
ARRAY *array;

int (*initdint(array))();
ARRAY *array;

short (*initintg(array))();
ARRAY *array;

long (*initlong(array))();
ARRAY *array;
```

Description

These routines determine which of the access routines is needed to convert elements of the `ARRAY` to int and long respectively.

Returns

A function pointer. If the data to be accessed does not have a valid APL element type, these routines will return a pointer to one of two support functions; **`domain`** or **`nonce`** which, when used, will signal *`DOMAIN ERROR`* or *`NONCE ERROR`* respectively.

Initialise Terminal: `inittty`

Name

`inittty` - initialises terminal control parameters

Synopsis

```
inittty()
```

Description

`inittty` initialises terminal control parameters. It must be called before using **`termaout`** or **`termcontext`**. One of the tasks performed by **`inittty`** is to remember the current settings so that they can be restored later by **`termcontext`**.

This function (defined in `tty.c`.) is applicable only under UNIX.

Initialise for Output: `init_out`

Name

`init_out` - prepares a device for output

Synopsis

```
init_out(describe,fail)
unsigned describe;
int (*fail)();
```

Description

`describe` is a file descriptor, a handle, or a stream; for example, 1 or `stdout`. **`fail`** is a pointer to a function to be called should an output error occur. **`init_out`** outputs the codes defined by `INIT` in the Output Table to the device. **`init_out`** must be called before characters are output using **`avtoasc`**.

See also...

`avtoasc`

Read Keystroke: `inkey`

Name

`inkey` - reads a keystroke from an input device

Synopsis

```
int inkey(fd, up)
int fd;
int up;
```

Description

`fd` is a file descriptor or handle. **`up`** is either `USER` or `PROG` (see `keystrokes.h`). The system maintains two instances of the current keyboard mode; one for the APL development environment (`PROG`), and one for user input (⎕*SM*, prefect, etc.).

Returns

The keystroke. Values in the range 1-255 represent ⎕*AV* characters. Other values represent special functions and commands. See `io_maps.h` for details.

Initialise Tables: `i_init`

Name

`i_init` - gets Input and Output Table(s)

Synopsis

```
void i_init()
```

Description

If the Auxiliary Processor requires input or output translation using the Dyalog APL Input and Output Tables, this function must be called using the following mechanism. It reads the appropriate tables into memory. A program that uses **`i_init`** should not also use **`o_init`**, which is automatically called by **`i_init`**.

```
main()
{
i_init();
...
}
```

See also...

`o_init`

Get as Long: `lgetb`

Names

`lgetb, lgets, lgeti, lgetl, lgetd` - long access routines.

Synopses

```
long lgetb(base,indx);
unsigned char *base;
unsigned indx;

long lgets(base,indx);
char *base;
unsigned indx;

long lgeti(base,indx);
short *base;
unsigned indx;

long lgetl(base,indx);
long *base;
unsigned indx;

long lgetd(base,indx);
double *base;
unsigned indx;
```

Description

These routines access an element of an `ARRAY` and return its value as a long. **`base`** is the address of the start of the data in the `ARRAY`. **`indx`** is the offset of the element to be accessed. **`lgetb`**, **`lgeti`**, **`lgeti`**, **`lgetl`** and **`lgetd`** are used to access APL data of type `BOOL`, `SINT`, `INT`, `LONG` and `DOUB` respectively. **`lgetd`** uses **`getd`** which forces double-word alignment (`#ifdef DALIGN`) if required.

`lgetd` rounds and truncates.

Initialise Array: `mkarray`

Name

`mkarray` - initialises an `ARRAY`

Synopsis

```
ARRAY *mkarray(eltype, rank, shape)
unsigned eltype;
unsigned rank;
unsigned *shape;
```

Description

Allocates space for an `ARRAY` and initialises its header. **`eltype`** specifies the type of each element of the array, and may be an integer in the range 0-6 corresponding to `CHAR`, `BOOL`, `SINT`, `INTG`, `LONG`, `DOUB`, or `PNTR` respectively. **`rank`** specifies the rank of the object. **`shape`** is a pointer to an unsigned array of length **`rank`** and specifies its shape. (The function `mkarray` is defined in `support.c`.)

Returns

A pointer to the `ARRAY`.

Initialise Output Table: `o_init`

Name

`o_init` - gets Output Table(s)

Synopsis

```
void o_init()
```

Description

If the Auxiliary Processor requires output translation using the Dyalog APL Output Tables, this function must be called using the following mechanism. It reads the appropriate tables into memory. If a program requires both Input and Output Tables, it should use **`i_init`** instead.

```
main()
{
o_init();
...
}
```

See also...

`i_init, avtounix, unixtoav`

Transmit Array: `putarray`

Name

`putarray` - transmits an `ARRAY` to the APL task

Synopsis

```
ARRAY *putarray(array)
ARRAY *array;
```

Description

Transmits an `ARRAY` from the Auxiliary Processor to the APL task. If the `ARRAY` is `NESTED`, **`putarray`** calls itself recursively to transmit each sub-array. (This function is defined in `support.c`.)

See also...

`throwarray`

Put as Double: `putd`

Name

`putd` - places a double value into an element of an `ARRAY`

Synopsis

```
putd(val, base, indx);
double val, *base;
unsigned indx;
```

Description

`putd` places a value of type double into an element of an APL data object. Double-word alignment is taken into account if required by the machine architecture. **`val`** is the value to be stored. **`base`** is the address of the start of the data in the object, and **`indx`** is the offset of the element in which **`val`** is to be placed.

Resolve Overstrikes: `resolve`

Name

`resolve` - resolves APL overstrikes

Synopsis

```
unsigned resolve(a,b)
register unsigned a,b;
```

Description

Returns the value of the character in ⎕*AV* formed by overstriking (superimposing) two other characters **`a`** and **`b`**. (The function `resolve` is defined in `resolve.c`.)

Round to Long: `round`

Name

`round` - rounds a double to a long integer

Synopsis

```
static long round(fval);
double fval;
```

Description

Rounds a double to the nearest long integer.

Establish AP Link: `startup`

Name

`startup` - establish communication with APL.

Synopsis

`startup(n_fns)`

Description

`startup` establishes communication between the Auxiliary Processor and APL. (This function is defined in `support.c`.) **`startup`** transmits the process number of the Auxiliary Processor. It then receives the corresponding process number of the APL task, which it stores in **`parentid`**. **`startup`** completes this phase of the initialisation by transmitting **`n_fns`**, the number of external functions to be fixed. Under UNIX **`startup`** also sets up a signal handler for `SIGHANDSHAKE`, and resets signal status to ignore `INTERRUPT` and `QUIT` but take default action on `HANGUP`.

See also...

Error handling.

Handle Terminal: `termaout`

Name

`termaout` - sets up (nearly) raw terminal handling

Synopsis

`termaout()`

Description

`termaout` sets up nearly raw terminal handling for character at a time i/o. The following settings are selected:

Input modes:

`-inlcr`	do not map NL to CR on input
`-igncr`	do not ignore CR on input
`-icrnl`	do not map CR to NL on input
`-ixon`	enable START/STOP output control
`-ixoff`	request that the system send START/STOP characters when the input queue is nearly full/empty

Output modes:

`-onlcr`	do not map NL to CR-NL on output
`-ocrnl`	do not map CR to NL on output
`tab0`	do not expand tabs to spaces, 0 delay

Local modes:

`-isig`	disable interrupts, i.e. disable the checking of characters against the special control characters `INTR` and `QUIT`
`-icanon`	disable canonical input
`-echo`	do not echo back every character typed

Input is set for 1 character at a time, and the time-out value is set to `TIMEOV`

This function is only relevant under UNIX.

See also...

`apl.h, inittty, termcontext`

Reset Terminal Handling: termcontext

Name

`termcontext` - restores terminal handling

Synopsis

`termcontext()`

Description

Restores terminal handling parameters to those in use when **`inittty`** was called. It is used to reset terminal handling after calls to **`termaout`**.

This function is only relevant under UNIX.

See also...

`inittty`, `termaout`

Transmit Array: `throwarray`

Name

`throwarray` - transmits an `ARRAY` to the APL task

Synopsis

```
ARRAY *throwarray(arr)
ARRAY *arr;
```

Description

Transmits an `ARRAY` from the Auxiliary Processor to the APL task and releases the memory it occupied in the Auxiliary Processor. (This function is defined in `support.c`.)

Returns

0. This allows one to use the expression

```
larg = throwarray(larg);
```

thus clearing the pointer to the array which was released.

See also...

`putarray`, `freearray`

Convert Character to APL: unixtoav

Name

unixtoav - converts an ASCII character into a value in ⎕AV

Synopsis

```
unsigned char unixtoav(a)
unsigned char a;
```

Description

unixtoav converts an ANSI character to its equivalent value in ⎕AV using the inverse of the translate table attached to file descriptor 1 (normally `WIN.DOT`). If no translate table has been loaded, **unixtoav** uses the internal table **asciimap**

See also...

`o_init`

Index