DYALOG

The tool of thought for expert programming

Dyalog™ for Windows

Guide to the Code Libraries

Version 13.0

Dyalog Limited

Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom

tel: +44 (0)1256 830030
fax: +44 (0)1256 830031
email: support@dyalog.com
http://www.dyalog.com

Version 13.0

First Edition March 2011

Contents

Introduction

This is a guide to the workspaces, scripts and DLLs supplied with Dyalog 13.0.

Since version 12.0, Dyalog has begun a programme of developing new exemplary and tutorial material for programmers, which had remained fairly static for several releases prior to that time. This guide was first provided with v12.0, and is the result of the first step of the programme: to review the existing material, removing or archiving what is no longer relevant, and bringing what remains up to date.

From version 12.1, Dyalog intends to accelerate the production of "library" code, and will publish new components of the Code Libraries on the internet, in order to allow more frequent updates to recently added parts of the library than would be possible if the library can only be updated with each release of Dyalog APL itself. Please visit:

`http://www.dyalog.com/library`

for up-to-date information on additions and updates to the code library.

Since version 12.0, much sample code is shipped in the form of *scripts*, rather than workspaces. The `Classes` folder contains exemplary class and namespace scripts. These are described in the documents *Simple APL Library Toolkit (SALT)* and *Spice command bar*. Version 12.1 added the ability to encapsulate tools as *User Commands*, and a number of such tools are included with v12.1. See the separate *User Commands* manual for more information.

For migration tools, see the *Workspace Transfer* document.

Many workspaces remain in the folders in which they have been shipped previously. Some earlier versions contained location dependencies, which have now been removed. In future releases workspaces may be shipped in different paths, reflecting progressive reorganisation of the code libraries.

The material here comprises workspaces and DLLs, labelled as follows.

Use	This code is offered by Dyalog for common tasks. You should find it runs efficiently and reliably. The code is not offered as examples for study and in some cases, such as the XUtils Auxiliary Processor, the source is not provided.
Study	This code is offered for study as examples of how to write certain kinds of programs, such as a GUI, or as tutorials. Some of it supports descriptions in other manuals, such as the *Interface Guide*. It is not generally optimised for any performance characteristics, but written primarily for clarity.
Archive	These workspaces contain solutions to problems now rarely encountered, or for which different solutions are now recommended. They are provided for continuity.

Some workspaces not listed here remain in the `ws` folder while their eventual fates are decided. Dyalog welcomes any views you might have about them.

Communication

Calls to the Windows API

QuadNA workspace

STUDY

This workspace contains functions that illustrate the use of ⎕NA to invoke the Windows API.

`Beep`	Beep N times on the system speaker
`Blink`	Set the cursor blink time
`cd`	Change directory
`DllVersion`	Major and minor version numbers of a DLL
`DumpWindow`	Copy a form's window to the Clipboard as a bitmap
`Env`	Return the environment variables as strings
`GetLocalTime`	Return local time as a vector
`GetSystemTime`	Return system time as a vector
`GetVersion`	Return operating system version
`HTTPDate`	Current date and time in RFC1123 date format
`MsgBox`	Pop a Windows message box
`Replace`	Replace or insert text into an Edit or RichEdit object
`SetSesh`	Maximise, minimise or restore the session window
`SetTabs`	Set tab stops in a Windows ListBox object

Calls to C functions

XUtils Auxiliary Processor

C utility functions

USE

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. Note that Version 13.0 introduces native support for "PCRE" regular expressions using system operators ⎕R and ⎕S.
`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 zero-origin 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 4⍴'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 3⍴⍳6)
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 0⍴''`. 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
      ⍴⎕←',' 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 0⍴⊂''. 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

NOTE: Version 13.0 introduces native support for "PCRE" regular expressions using system operators ⎕R and ⎕S. Dyalog recommends that these are used in preference to ss.

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 1-element vector it specifies whether case is significant. If it is a 2-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 1s 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
```

`ss` (like many other regular expression engines) understands an empty search string to represent the previous search string; if no previous search has been performed since XUtils was started, a zero-length search string will result in an `ERROR 243.`

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)`	Parentheses 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 2⍴'ABC' 'DEFG' 'HI' 'JKLMNO'
      A
 ABC  DEFG
 HI   JKLMNO

      ⍴⍞←',' vtol A
ABC,DEFG,HI,JKLMNO,
19
```

OLE

Object Linking and Embedding

Loan workspace

STUDY This workspace illustrates an OLE Server using a loan sheet example. Visual Basic and Excel client samples are included.

CFiles workspace

STUDY This workspace illustrates an OLE Server that allows you to read Dyalog component files into Excel.

DCOMReg workspace

USE This workspace contains functions that may be used to register an OLE Server, written in Dyalog APL, for DCOM.

OLEAuto workspace

STUDY This workspace illustrates how you can access OLE Servers such as Microsoft Access and Microsoft Excel.

OLEAsync workspace

STUDY This workspace illustrates how an OLE Server written in Dyalog may be called so that it executes in parallel (asynchronously), possibly on a different computer.

Shortcut workspace

STUDY This workspace illustrates how you may call OLE objects via non-standard interfaces. This example creates a shortcut on your desktop.

Sockets

Communicating through TCP/IP sockets

APLServe folder

A simple webserver

ARCHIVE

The `SERVER` workspace provides the framework for a multi-threaded APL webserver that may be used to deliver Dyalog applications via the Web or an intranet. See `WebServer` in the `CONGA` workspace for this function.

As supplied, the `SERVER` workspace contains a number of example applications designed to illustrate the principles involved. When you load the workspace, its `⎕LX` starts the `SERVER` running on your computer. You may then access the webserver using a browser such as Internet Explorer or Firefox.

The `SERVER` workspace and the files it uses are supplied in the `APLSERVE` folder.

Chat workspace

STUDY

This workspace illustrates how the TCP/IP interface can be used to chat between two or more APL workspaces.

Conga workspace

USE

This workspace contains classes and namespaces for communicating through TCP/IP sockets.

`DRC`	Tools for socket communications
`FTPClient`	Implement a simple, passive-mode FTP client
`HTTPUtils`	Tools for HTTP communications
`Parser`	Parse a string and set switch values accordingly
`RPCServer`	Server for Remote Procedure Calls
`TelnetClient`	Simple Telnet client
`TelnetServer`	Telnet server
`TODServer`	Time-of-day server
`WebServer`	Minimal web server

The `Samples` namespace contains examples of using the Conga classes.

QFiles workspace

STUDY

This workspace illustrates how the TCP/IP interface may be used to implement a client/server component file system.

RExec workspace

STUDY

This workspace illustrates how the TCP/IP interface may be used to implement client/server (remote execution) operations.

WWW workspace

ARCHIVE

This workspace contains basic functions to illustrate the principles of internet access using Dyalog. See `WebServer` in the `CONGA` workspace for this function.

Development tools

BuildSE workspace

USE

This workspace is used to build the default APL session. To configure the session differently, you may edit the functions and rebuild and save the session.

Display workspace

Exhibiting array structure

USE

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:

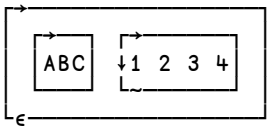
```
)COPY DISPLAY
```

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 borders 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
ϕ	Empty along other than last axis
∈	Nested array
~	Numeric data
-	Character data
+	Mixed character and numeric data
∇	⎕OR object

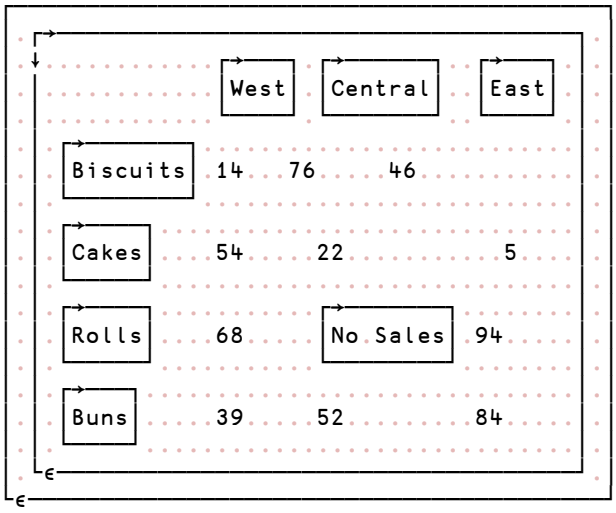
Example:

```
      DISPLAY 'ABC' (1 4⍴1 2 3 4)
```



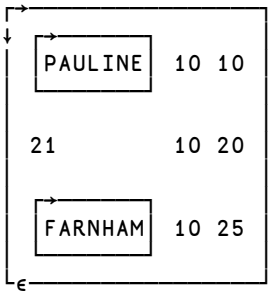
Example:

```
AREAS←'West' 'Central' 'East'
PRODUCTS←'Biscuits' 'Cakes' 'Rolls' 'Buns'
SALES←?4 3⍴100 ⋄ SALES[3;2]←⊂'No Sales'
DISPLAY ' ' PRODUCTS⍪..,AREAS SALES
```



Example:

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



Math workspace

Extended mathematical functions

USE

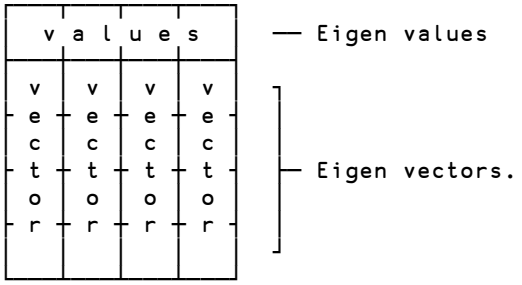
The functions in this workspace perform complex arithmetic. Complex arrays are represented by depth-2 arrays of real-imaginary pairs.

The code consists entirely of dynamic functions, and illustrates encapsulation through dfns.

The workspace requires dynamic link libraries `LAPACK.DLL` and `FFTW.DLL`.

There are six functions:

`Eigen` takes an n×n real or complex matrix and returns an (n+1)×n result of Eigen: `Values⍪⍉Vectors`



`Domino` is a complex generalisation of APL's primitive ⌹ function.

`Fourier` takes a real or complex array right argument and performs a Fourier Transform, or its inverse.

`Hermite` Hermite polynomials

`Laguerre` Laguerre polynomials

`Legendre` Legendre polynomials

Patch workspace

Updates to the interpreter

USE

The `PATCH` workspace allows you to update the interpreter from the Dyalog website. (Patches contain bug fixes and/or enhancements.)

In each edition, there are four files that can be patched. You *need* patch only the files you use; but you might consider it wise to patch all four so no doubt arises.

The Dyalog installation folder has the interpreter as an executable and as a Dynamic Link Library. It also has Runtime versions of the same files. The names vary slightly between the Classic and Unicode editions, and between 32 and 64 bit editions.

Classic Edition 32-bit example:

```
dyalog.exe    dyalog130.dll

dyalogrt.exe  dyalog130rt.dll
```

Unicode Edition 64-bit example:

```
dyalog.exe    dyalog130_64_unicode.dll

dyalogrt.exe  dyalog130_64rt_unicode.dll
```

In previous versions the DLLs resided in a separate subdirectory called `bin`.

The DLLs are used by Dyalog applications, exported as OCX controls, etc. If you run all your applications using the Dyalog executable file, you can neglect patching the DLLs.

You cannot patch a file that is the result of patching. Patches are applied only to the original version as shipped; this is called the *base version*. Base versions are already saved in the `Base` folder of the Dyalog installation directory.

When you load the `PATCH` workspace, its wizard starts. The wizard allows you to patch files straight from the Dyalog webserver, or from patch files you have downloaded.

By default, in patching the executables, the wizard attempts to apply patches to the base version, and overwrite the current executables with your latest patched versions. So, when patching, it is convenient to start by

- closing all Dyalog sessions;
- starting the **base version**;
- loading the `PATCH` workspace.

When the wizard finishes, it will close the session, and you can start a new session with your updated interpreter.

If the wizard is writing the patched file outside the Dyalog installation folder, or if it is unable to overwrite `dyalog.exe` (because you are using it to run the wizard) it will give the patched file a suffix of `.patched.exe` or `.patched.dll`. You can then rename them in Explorer to replace current EXEs or DLLs.

Util workspace

APL utility functions

USE

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
```

`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

Dynamic functions

Dyalog's lambda

Dynamic functions (or *Dfns*) are a simplified form for defining functions and operators. While they sacrifice certain features of traditionally-defined functions (or *tradfns*) such as control structures, they give programmers a compact form for simple functions and clear way to:

- write functions that define and localise their own tools
- use anonymous functions, eg `{⍵/⍳⍴⍵}` to minimise repetition or avoid assigning names to functions or arrays that will have no further use

DDB workspace

USE This workspace contains a lightweight database system that can replace a SQL database for many simple applications. See the section on *Storage* for details.

DFns workspace

STUDY This workspace, kept up to date on the Web, is an encyclopaedia of examples of programming with dynamic functions and operators.

Eval workspace

STUDY This workspace contains tools for studying the evaluation of expressions using syntax rules that include but are not limited to Dyalog's. It is not optimised for performance, but could be used in applications for processing custom domain-specific languages.

Min workspace

STUDY Implements a minimal programming language using only dynamic functions

Max workspace

STUDY Implements an extended version of the MIN language

Tube workspace

STUDY Demonstrates graph searching, applied to the underground rail networks of London, Paris, New York and other cities

Graphical User Interfaces

Dyalog system classes

Arachnid workspace

STUDY This is a card game that demonstrates various Dyalog GUI features, including the use of the BitMap and Image objects.

BMEd workspace

USE This workspace contains functions for editing bitmaps and for creating picture buttons.

CPro workspace

Causeway Pro framework for GUI

USE Causeway Pro is a framework for designing and building GUIs (graphical user interfaces) for applications.

The workspace includes samples from the tutorial in *Getting Started With Causeway*.

Note that the workspace predates user-defined classes in Dyalog and includes its own implementation in the `Class` namespace.

Graphs workspace

STUDY This contains some business graphics utility functions and a graphics demonstration.

PocketWD workspace

ARCHIVE This is a version of the `WDESIGN` workspace for the PocketPC platform. See the notes on `WDESIGN` below.

WDesign workspace

ARCHIVE This workspace contains a graphical tool for designing GUI forms and populating them with controls. It resembles tools widely used for this purposes in other languages. The developer defines a form by gesturing with the mouse; the tool provides immediate visual feedback, and finally writes a function that recreates the form.

`WDESIGN` is invitingly easy to use and automates work that in other languages is laborious. Its use is now deprecated for a combination of reasons.

The ability to lay out forms without mastering the corresponding APL expressions is deceptive. Getting a GUI working requires understanding the code behind it. A beginner is better served by studying and trying examples than using `WDESIGN`.

The Dyalog expressions required to generate a form are very simple. For someone who has learned them, `WDESIGN` does not save much work. And a human can write clearer GUI code than `WDESIGN` does.

The chief value of WDESIGN is in graphically positioning elements on a form. But Dyalog developers rarely invest heavily in interface design; clear and simple is the common standard. For this standard of presentation, it is hardly more difficult to guess the desired control coordinates and then tweak them.

WDESIGN does not allow you to develop GUIs without understanding the code behind them. The extra value of precise visual positioning is outweighed by clearer code and writing without a development tool. A beginner's time is better invested mastering GUI code than learning WDESIGN.

WIntro workspace

STUDY

The WINTRO workspace contains a tutorial introduction to the GUI features in Dyalog. 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

STUDY

The WTUTOR workspace contains a more elaborate set of tutorials to help you explore further aspects of Dyalog's GUI support.

WTutor95 workspace

STUDY

The WTUTOR95 workspace contains an additional set of tutorials.

WinForms

GDIForms workspace

STUDY

This example is converted from the `GDIPlusShape` sample provided on the *Visual Studio.NET Beta 2 Resource* CD. It illustrates non-rectangular forms. See the *DotNet Interface Guide*.

Tetris workspace

STUDY

This example is converted from the `TETRIS` sample that is provided on the *Visual Studio.NET Beta 2 Resource* CD. See the *DotNet Interface Guide*.

It illustrates how to use some of the graphical objects and methods provided by the .NET Framework. The original C# code that handles the GUI and drawing functions has been translated directly into APL. The code that handles the application logic has been completely re-written to take advantage of APL's array-handling capabilities.

WinForms workspace

STUDY

This workspace contains functions that demonstrate how you can use the `System.Windows.Forms` .Net class library to drive the Windows GUI. See the *DotNet Interface Guide*.

Object orientation

User-defined classes

The guides *OO for APLers* and *OO for Impatient APLers* introduce the use of native Dyalog support for user-defined classes.

OO4APL folder

STUDY

This folder contains workspaces supporting the examples in *OO for APLers*.

Presentation

Newleaf workspace

Composing paged output

USE

Newleaf is a toolkit for composing document pages. These can be presented in a choice of formats, including HTML, PDF, PostScript and RTF.

Newleaf supports flowed text columns, tables, frames and graphics and includes a viewer for proofing results.

The workspace includes an extensive set of examples.

`Help`	Displays expressions that compose and display a simple document, then several other examples
`Seatrial 0`	Exercises all the examples
`Describe`	A short tutorial, analysing an example document

RainPro

Scalable vector graphics

USE

RainPro is a toolkit for producing scalable vector graphics to a very high standard. The workspace includes extensive examples, of which the following is a sample.

`Colours`	Shows chart colours and line styles
`Patterns`	Show all fill patterns, markers, etc.
`Sample`	Assorted charts to show some options
`Sambars`	Test barchart options … see Praxis (trend chart)
`Rain 02`	Compute rainfall graph to date (see Rain 90 etc)
`Temp 02`	ditto for temperatures (see Temp 89, Temp 90 etc)
`Scatter`	Sample scatter plot … see also Surface
`Barchart ¯8+?13⍴15`	with hanging bars
`Bench`	APL benchmarks from long ago
`Pie 30+?7⍴120`	Pie chart … see also Twopies
`Step`	Step chart … note irregular X-ticks and labels
`Dupax`	Independent Y axes … see Intercepts for axes
`Frequency`	Frequency distributions, with annotations
`Timeseries`	Simple timeseries, showing use of date labels
`Notes`	Testbed for assorted notes and headings
`View PG`	to show it on screen
`Perspectives`	3D charts now available!
`Experiments`	Other interesting stuff
`Seatrial 1`	Complete test for all the above charts and more

Storage

DDB workspace

A lightweight database system

USE

The functions in the `ddb` namespace are used to maintain simple data arrays in a single mapped file. They provide a robust alternative to an 'inverted' component file, as long as the maximum size of the data in each field may be fixed at creation time.

`create`	Create table
`remove`	Remove table
`append`	Append row/s to table
`retain`	Retain only selected rows
`open`	Open table (read/write)
`defs`	Field definitions
`get`	Get field/s from table
`put`	Replace values in field(s)

Files workspace

Handling files and directories

USE

This workspace provides cover functions for common operations in the file system, encapsulating both native file-system primitives such as ⎕NTIE and Windows API calls.

See the source for function syntax.

`AppendText`	Appends single-byte text to a named file
`Copy`	Copy one file to another; protected mode optional
`Delete`	Delete a named file
`Dir`	Directory information for a filepath
`DirX`	Extended directory information for a filepath
`Exists`	Boolean result indicates whether file exists
`GetCurrentDirectory`	Get current directory
`ReadAllLines`	Read a text file as single-byte text; return lines as nested character vector
`ReadAllText`	Read a text file as single-byte text
`MkDir`	Make a directory
`Move`	Named file to another location
`PutText`	Write single-byte text to a file, accepting scalar, vector, matrix or nested character arrays
`RmDir`	Remove a directory
`SetCurrentDirectory`	Set current directory

SQAPL workspace

USE

The ODBC interface is provided by SQAPL for ODBC which is included with Dyalog APL for Windows and distributed under licence from Insight Systems ApS.

SQAPL for ODBC is an interface between APL and database drivers which conform to the Microsoft ODBC specification.

ODBC drivers exist for a wide variety of databases, from simple drivers which give limited access to 'flat' DOS files, through more sophisticated local database managers such as Access, dBase and Paradox, to multi-user DBMS systems such as Oracle, Ingres, Sybase or DB2 running on remote hosts.

See the separate *ODBC User Guide* for more details.

Threads

Dividing a process between multiple threads

Lift workspace

STUDY

This workspace simulates a lift taking people to the floor of their choice. Two lifts are used, but the example could easily be extended to more.

People arrive at the lift entrance pseudo-randomly. People get into the lift one at a time, in orderly fashion. When the lift is full, if there is nobody waiting, the lift door closes and the lift rises. The lift stops only at floors where people want to get out. People get out of the lift in a disorderly fashion.

Each lift and each person in the simulation is implemented as a separate thread.

Packaging

Using Dyalog programs in other contexts

Programs written in Dyalog may be components of other systems. (See also the section on Communications.)

OCXBrows workspace

USE The `OCXBROWS` workspace lists the ActiveX (OCX) controls installed on the machine. Its Detail button displays the control's properties, events and methods – and its appearance if it has one. The utility of the workspace is limited by the paucity of information embedded in most controls; but where help files are supplied (see if you have `Calendar Control 9.0` installed) it reveals a wealth of useful information.

ActiveX folder

STUDY These three workspaces, `DUAL`, `DUALBASE` and `DUALFNS`, support the ActiveX control example described in the *Interface Guide*.

APLClasses folder

STUDY These workspaces and DLLs, from `APLCLASSES1` to `APLCLASSES5`, contain examples to support the chapter on writing.Net classes in the *DotNet Interface Guide*.

APLScript folder

STUDY This folder contains scripts and executables supporting the examples in the *DotNet Interface Guide* of creating executable applications from script files.

ASP.Net folder

STUDY This folder contains materials to support the chapters in the *DotNet Interface Guide* on working with ASP.Net.

Appendix: Obsolete workspaces

The following workspaces from earlier versions of Dyalog are no longer thought applicable, and have been retired.

```
ATFIN
DOSUTILS
FASTFNS
FONTS
FTP
GROUPS
KIBITZER
NTUTILS
OPS
POSTSCRI
PREDEMO
PREFECT
SMDEMO
SMTUTOR
TUTOR
XLATE
```