The tool of thought for software solutions

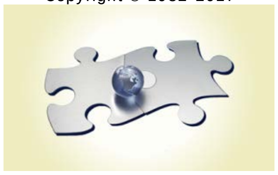
Dyalog Code Libraries Reference Guide

Version 16.0

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



Contents

Introduction

There is a growing collection of tools, samples and other code created for Dyalog APL, that can be used to improve productivity when writing applications, and to facilitate learning about features of Dyalog APL.

Historically, this document has provided an overview of the “workspaces” shipped with Dyalog APL. Version 16.0 is a watershed release, in that nearly all the new APL code that has been produced for this version is in Unicode text files rather than the old binary workspace format.

Some of the most widely used tools will continue to be included with the installation package as described in this document, but many new tools are most easily accessed via the public GitHub repositories where the code is maintained. There are a couple of web pages worth bookmarking:

`http://www.dyalog.com/library`	An overview of tools with links to documentation and downloads.
`https://github.com/Dyalog`	The entry point on GitHub to all the open source projects, including many tools.

Over time, we expect to reduce the number of binary workspaces shipped, and move in the direction of distributing all tools as source files. As the community becomes more comfortable with GitHub, we also expect to decouple the release cycle for tools completely from the interpreter release cycle.

This document is intended to provide an overview of the most important tools that are either shipped as part of the version 16.0 installation, or available on GitHub. The materials are labelled as follows.

USE	This code is offered by Dyalog for common tasks. You should find it runs efficiently and reliably.
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.

Core Library

USE

Version 16.0 marks the beginning of a new set of utility libraries, based on scripts. A new folder called **Library** (below the main **Dyalog** folder) contains the utilities that we believe will be widely used, and will become a new *core library* for Dyalog application development. The folder currently has two sub-folders called **Core** and **Conga**. These two folders are on the default SALT *workdir*, which means that the `]Load` user command will search in them. This means that you can bring these tools into the workspace without providing a path name, making them very easy to load and immediately use:

```
    ]load HttpCommand
#.HttpCommand
    url←'http://hasthelargehadroncollider'
    url,←'destroyedtheworldyet.com/atom.xml'
    answer←HttpCommand.Get url
    xml←⎕XML answer.Data ⍝ Convert XML to array
    (xml[;2]∊⊂'content')⌿xml[;3] ⍝ Tags named “content”
NOPE.
```

Note that the initial load can be done under program control:

```
    ⎕SE.SALT.Load 'HttpCommand'
```

The two folders **Library/Core** and **Library/Conga** contain snapshots of the GitHub repositories Dyalog/library-core and Dyalog/library-conga respectively. A release tag in each repository will identify the exact version shipped with v16.0 and subsequent releases.

The Documentation sub-folder in each of the above folders contain documentation for each tool in the form of MarkDown files (.MD). If you do not have a MarkDown viewer, you can view the files on GitHub: for example, the documentation for everything in Library/Core can be found by selecting the Documentation folder within https://github.com/Dyalog/library-core.

Tool (Library/Core)

Tool is a utility for loading and initialising major tool components. It currently supports ***Conga***, ***RConnect*** (the R interface), ***SharpPlot*** and ***SQAPL*** (the ODBC interface). We intend to add support for other tools in the future.

Tool has two methods, `New` and `Prepare`. `New` returns an initialised instance of the tool in question. `Prepare` materialises the tool in your workspace (so you could save your application and no longer have external dependencies on loading interface code) but does not initialise it (you can use `New` later to do this).

Example 1: Generate 10 numbers with a mean of 100 and a standard deviation of 1 (requires R with rscproxy installed on your machine):

```
     ]load tool
#.Tool
      iR←Tool.New 'RConnect' ⍝ A new instance of RConnect
      1⍕iR.x'rnorm(10,100,1)'
99.6 99.5 101.7 98.7 101.1 100.2 100.1 101.2 99.9 100.8
```

Example 2: Use the ODBC interface to query the zipcodes Access database which is included with MiServer.

```
      Tool.Prepare 'SQAPL'
#.SQA
```

At this point, the SQAPL interface has been copied into your workspace. You could save the workspace and use it on a machine where the SQAPL workspace is not installed (the SQAPL DLLs and the ODBC data source still need to be available). The next steps will use the interface which was materialised by `Tool.Prepare`:

```
      sqa←Tool.New 'SQAPL'
      sqa.Connect 'zip' 'zipcodes'
0
      sqa.Do 'zip' 'select city, zipcode from zipcodes where
city like ''West Hen%'''
0  zip.s1    WEST HENRIETTA  14586        6
```

APLProcess (Library/Core)

APLProcess is a class which provides a mechanism for an APL process to launch and manage other processes. The started process can either be another APL process, or a process based on any other executable. To start a process, you create a new instance of the APLProcess class:

Up to five constructor arguments can be provided:

1	the name of a workspace to load
2	command line arguments
3	1 for runtime (the default), 0 for development system, or a character vector containing the name of an executable (not necessarily an apl interpreter)
4	RIDE_INIT parameters to use for the started process
5	a log-file name to redirect output to

For example: To launch a Dyalog development environment which loads `dfns.dws` with a workspace size of 200M, and then check whether it is up and running:

```
      proc←⎕NEW APLProcess ('dfns.dws' 'MAXWS=200M' 0)
      proc.HasExited
0
```

A typical use of `APLProcess` is to start a runtime process which uses Conga to provide some kind of service. See the Conga Upgrade Guide for examples of this.

FtpClient (Library/Conga)

FTPClient is a simple implementation of a “passive mode” FTP client. It can be used to get directory listings for, or retrieve and update files managed by FTP servers. Examples of use:

```
      creds←'ftp.mirrorservice.org' 'anonymous' 'testing'
      ms←⎕NEW FtpClient creds
      ms.List 'pub/'
0  pub/FreeBSD
pub/GNOME
pub/NetBSD
pub/OpenBSD
...

      30↑2⊃ms.Get 'pub/FreeBSD/README.TXT'
Welcome to the FreeBSD archive!
```

HttpCommand (Library/Conga)

`HttpCommand` is a stand-alone utility to issue HTTP commands and return their results. `HttpCommand` can be used to retrieve the contents of web pages, issue calls to web services, and communicate with any service which uses the HTTP protocol for communications.

`HttpCommand` can be used in two ways:

1) Create an instance of `HttpCommand` using `⎕NEW`
   This gives you very fine control to specify the command's parameters
   You then use the `Run` method to execute the request

```
h←⎕NEW HttpCommand        ⍝ create an instance
h.Command←'get'           ⍝ set the HTTP method
h.URL←'www.dyalog.com'    ⍝ set the URL
r←h.Run                   ⍝ run the request
```

2) Alternatively you can use the `Get` or `Do` methods which make it easier to execute some of the more common use cases.

```
r←HttpCommand.Get 'www.dyalog.com'
r←HttpCommand.Do 'get' 'www.dyalog.com'
```

Up to seven constructor arguments can be provided:

1	The HTTP command (method) for the request
2	The URL of the server
3	Any parameters, e.g. form fields, for the request
4	Any HTTP headers for the request. HttpCommand will set several headers by default, any of which can be overridden
5	In X509 certificate if using SSL (secure sockets)
6	Any SSL flags for the request if using SSL
7	The GNU TLS priority string if using SSL

The methods that execute HTTP requests - `Do`, `Get`, and `Run` - return a namespace containing the variables:

`Data`	the response message payload
`Headers`	the response HTTP headers
`HttpVer`	the server HTTP version
`HttpStatus`	the response HTTP status code (200 means OK)
`HttpStatusMsg`	the response HTTP status message
`PeerCert`	the server (peer) certificate if running secure
`Rc`	the Conga return code (0 means no error)

In addition, `HttpCommand` has several utility functions to compose and parse HTTP messages.

`HttpCommand` uses Conga and will search for the Conga library or copy it in as needed. For efficiency, you may want to copy Conga, for example by using the `Tool` utility before using HttpCommand, to avoid repeated copying of Conga.

Example Use Cases

1) Retrieve the contents of a web page

```
 result←HttpCommand.Get 'www.dyalog.com'
```

2) Update a record in a web service

```
   cmd←⎕NEW HttpCommand  ⍝ create an instance

⍝ set a the HTTP command and URL
   cmd.(Command URL)←'PUT' 'www.somewhere.com'

⍝ set the id and name parameters for the "PUT" command
   (cmd.Params←⎕NS '').(id name)←123 'Fred'

   result←cmd.Run ⍝ and run it
```

HttpUtils (Library/Conga)

`HttpUtils` is a namespace containing classes and utility functions to assist in the processing and creation of HTTP messages. Dyalog v16.0 includes two new features that make use of the HTTP protocol. The first is support for native HTTP mode in Conga v3.0 which greatly simplifies receiving HTTP messages. The second is the HTMLRenderer, an experimental feature, which enables the user to build HTML5-based applications using the Dyalog GUI object framework. HTMLRenderer uses HTTP-like messages to communicate with callback functions in the user's application.

`HttpUtils` provides an easily accessible interface to all the information contained in an HTTP message including headers, cookies, form data, the HTTP command used, and any data contained in the body of the message.

`HttpUtils` contains two classes, `HttpRequest` and `HttpResponse`, which provide a standard means to represent and process HTTP messages.

When using Conga, your application may function as an HTTP client, an HTTP server, or both. When acting an HTTP client, it needs to do two things – 1) format and send HTTP requests, and 2) receive and parse HTTP responses. When acting as an HTTP server, also needs to do two things – 1) receive and parse HTTP requests, and 2) format and send HTTP responses. The `HttpRequest` and `HttpResponse` classes in `HttpUtils` help you do all of these things.

Conga's HTTP mode has 4 new events – HTTPHeader, HTTPBody, HTTPChunk, and HTTPTrailer. The `HttpRequest` and `HttpResponse` classes have methods corresponding to each of these events. Simply calling the appropriate method and passing the data element from the event will update the class thereby giving you easy access to all of the data elements of the request or response.

When using the HTMLRenderer Dyalog GUI object, your callback function for the onHTTPRequest event receives HTTP request information via the data elements of the event. Your application then needs to update those data elements forming the logical equivalent of an HTTP response and pass that back to HTMLRenderer from your callback function The `HttpRequest.AddHtmlRenderer` method is used to parse the event data.

In this example, we simulate acting as an HTTP client and receiving the response from a server.

```
:Repeat
  :If ~done←0≠err←1⊃rc←DRC.Wait clt 5000  ⍝ standard Conga loop
    (err obj evt dat)←4↑rc  ⍝ break out the results

    :Select evt   ⍝  which Conga HTTP mode event?

    :Case 'HTTPHeader'
      resp←⎕NEW #.HttpUtils.HttpResponse dat ⍝ create the response

    :Case 'HTTPBody'
      resp.CongaHttpBody dat  ⍝ process the HTTP message body

    :Case 'HTTPChunk'
      resp.CongaHttpChunk dat  ⍝ process the HTTP message body

    :Case 'HTTPTrailer'
      resp.CongaHttpTrailer dat  ⍝ process the HTTP message body

    :EndSelect
  :EndIf
:Until done∨resp.IsComplete ⍝ do we have the complete resposnse?
```

At this point we have all of the response data in public fields.

```
      resp.(HttpStatus HttpStatusText)
┌───┬──┐
│200│ok│
└───┴──┘
```

```
      resp.Headers   ⍝ HTTP headers
┌────────────────┬─────────────────────────────────────────────────────┐
│cache-control   │private                                              │
├────────────────┼─────────────────────────────────────────────────────┤
│content-type    │text/html                                            │
├────────────────┼─────────────────────────────────────────────────────┤
│content-encoding│gzip                                                 │
├────────────────┼─────────────────────────────────────────────────────┤
│vary            │Accept-Encoding                                      │
├────────────────┼─────────────────────────────────────────────────────┤
│server          │Microsoft-IIS/8.5                                    │
├────────────────┼─────────────────────────────────────────────────────┤
│set-cookie      │ASPSESSIONIDCATBDSCB=ANDNFICBLDIEHFOOINJHFOAL; path=/│
├────────────────┼─────────────────────────────────────────────────────┤
│x-powered-by    │ASP.NET                                              │
├────────────────┼─────────────────────────────────────────────────────┤
│date            │Tue, 27 Jun 2017 12:34:02 GMT                        │
├────────────────┼─────────────────────────────────────────────────────┤
│content-length  │370                                                  │
└────────────────┴─────────────────────────────────────────────────────┘
```

```
      resp.Data  ⍝ response data (edited for presentation here)
<html>
<head>
<meta http-equiv="Content-Language" content="en-us">
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>Some really cool place</title>
</head>
<body bgcolor="#FFFFFF">
<p align="center"><img src="images/BDS.jpg"></p>
<p align="center"> </p>
<p align="center"><b><font face="Arial">Some really cool place...coming
soon! </font></b></p>
</body>
</html>
```

GitHub Libraries Not Installed with v16.0

The following libraries are not currently included with the standard Dyalog APL installation, and need to be accessed directly from GitHub:

Math (https://github.com/Dyalog/Math)

USE

This repository contains a math namespace for Dyalog APL with functions for finding eigenvalues, eigenvectors and discrete Fourier transforms. It is based on calls to the LAPACK library. Source code and build scripts for the libraries are included.

MiServer (https://github.com/Dyalog/MiServer)

USE

MiServer is Dyalog's Web Server framework, written in APL. It includes "widgets" based on SyncFusion libraries, JQueryUI, and other 3rd party components, and contains everything you need to write a stand-alone web server. A running example of a MiServer site which showcases the various widgets and provides information on how to get started is available at http://miserver.dyalog.com.

APLPi (https://github.com/APLPi)

STUDY

These repositories, separate from the main Dyalog repos, contain examples of projects in APL which use the Raspberry Pi to perform experiments and drive robots.

Foreign Function Interface

Calls to the Windows API

QuadNA workspace

STUDY

This workspace contains functions that illustrate the use of `⎕NA` to invoke the Microsoft Windows API. Similar code could be written on any operating system, but no samples are currently provided for Linux or macOS.

`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

OLE

Object Linking and Embedding

OLE is an old technology which was the precursor to Microsoft.NET. It is still widely supported by Microsoft Windows and may still be a suitable mechanism for delivering APL functionality. The workspaces mentioned below have not been modified or tested for some time, but may still be useful:

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.

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, rebuild and subsequently 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
`⌽`	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 3⍴100 ⋄ 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 │
│ └───────┘       │
└∊────────────────┘
```

Math workspace

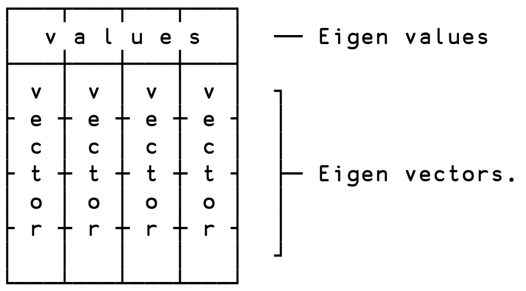
Extended mathematical functions

USE

This tool provides access to LAPACK; it is no longer shipped as part of the standard Dyalog APL installation, but is available on GitHub as the repository https://github.com/Dyalog/math.

The code consists entirely of dynamic functions, and illustrates encapsulation through dfns. There are five functions (the old Domino function is no longer relevant, now that APL supports complex numbers natively).

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



`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

Util workspace

APL utility functions

USE

The UTIL workspace contains a number of APL utility functions. The current list of functions with brief descriptions is contained in the variable `notes.contents`:

```
      )load util
C:\Program Files\Dyalog\Dyalog APL-64 16.0 Unicode\ws\util.dws
saved Thu Jun 22 20:15:28 2017

      notes.contents
-----------------
Utility Functions
-----------------

Edit (double-click or Shift-Enter) <function name> in the
following list for a description. Within the description, edit
<See also:> names for related functions and <#.function> to
view the code.

    ss          ⍝ Approx alternative to xutils' ss.
    ∆VARS       ⍝ Global variable cross-reference.
    ∆MPUT       ⍝ Put array on disk.
    ∆MAPPEND    ⍝ Append array ⍺ to map file ⍵.
    APLVERSION  ⍝ Interpreter version.
    BIG_ENDIAN  ⍝ Machine is "big-endian".
    BMVIEW      ⍝ View BitMap Files.
    CENTRE      ⍝ Centre TEXT in W columns.
    CLIP2PR     ⍝ bmp image of clipboard to default printer
    DETRAIL     ⍝ Remove trailing blank columns
    DIR         ⍝ Directory contents.
    DISP        ⍝ Boxed sketch of nested array.
    DISPLAY     ⍝ Boxed display of array.
    DOIF        ⍝ Return next line number,
    ECHO        ⍝ Value of environment variable ⍵.
    ENLIST      ⍝ Flatten array.
    FIND        ⍝ NAMES in workspaces in DIRS.
    FNGREP      ⍝ Matches of regular expression V in fns F
    FNREPL      ⍝ Replace first string with second in fns/ops
    FTREE       ⍝ (Approximate) function calling tree.
    Globals     ⍝ Name localisation.
    KEYPRESS    ⍝ Trace and decypher KeyPress events
    LJUST       ⍝ Left justify text
    MAKEMAT     ⍝ Return VEC as MATrix
    MATRIX      ⍝ Converts scalars and vectors
    NTREE       ⍝ Tree diagram of named object
    PROP        ⍝ Display Property for each item in obj tree.
    PROPS       ⍝ Display ALL props for NODE.
    RJUST       ⍝ Right justify text
    SET         ⍝ (NAME, VALUE) for environment vars.
    SETWX       ⍝ Set ⎕WX to ⍵ in namespace ⍺ and children.
    SM_TS       ⍝ Convert date format : ⎕SM(IDN) ← 3↑⎕TS.
    SNAP        ⍝ Round values ⍵ to nearest ⍺.
    Time        ⍝ Execution time for expression.
    TRAV        ⍝ Traverse object tree.
    TREE        ⍝ Display tree
    TS_SM       ⍝ Convert date format : 3↑⎕TS ← ⎕SM(IDN)
    WSDIFF      ⍝ Workspace differences.
    WSPACK      ⍝ Shares Identical arrays.
    XVAR        ⍝ all lower case vector
    XWS         ⍝ Evaluate expr in saved wsid.
```

As mentioned at the top of `notes.contents`, more detail is provided for each function in notes. For example:

```
      notes.DISPLAY

cmat ← #.DISPLAY array           ⍝ Pictorial representation of
nested array.

The argument is displayed with a series of boxes bordering each
sub-array. Characters embedded in the borders indicate array
shape and type.

Shape:  ↓ → Non-zero axis.
        ⌽ ⊖ Zero axis.

Type:     ∊ Nested array.
          ~ Numeric.
          - Character.
          # Namespace reference.
          ∇ ⎕OR object.
          + Mixed type.

Prototypical items of empty arrays (⌽ ⊖) are exposed.

    1 DISPLAY 'ABC'(1 4⍴1 2 3 4)(0 1 0⍴0)(⎕SE #) ⍝ Smooth
characters.
┌→────────────────────────────────┐
│ ┌→──┐ ┌→──────┐ ┌┌⊖┐ ┌→───────┐ │
│ │ABC│ ↓1 2 3 4│ ⌽↓0│ │ ⎕SE  # │ │
│ └───┘ └~──────┘ └└~┘ └#───────┘ │
└∊────────────────────────────────┘


      DISPLAY 'ABC'(1 4⍴1 2 3 4)(0 1 0⍴0)(⎕SE #) ⍝ Printable
.→--------------------------------.
| .→--. .→------. ..⊖. .→-------. |
| |ABC| ↓1 2 3 4| ⌽↓0| | ⎕SE  # | |
| '---' '~------' ''~' '#-------' |
'∊--------------------------------'

See also: DISP
```

dfns

Dyalog’s lambda

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 a clear way to:

- write functions that define and localise their own tools
- use anonymous functions, eg `{ω/⍳⍴ω}` 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

Windows GUI

Dyalog APL includes a set of built-in classes which provide easy access to the Win32 API, creating objects such as forms, menus, buttons, edits, grids and so on. These features will remain fully supported by Dyalog for a long time to come. However, they are no longer in active development, simply because Microsoft has more or less ceased enhancing the Win32 API.

Version 16.0 includes an HTMLRenderer object which allows the creation of UI under Microsoft Windows, Apple macOS and Linux – which is still experimental, but expected to replace the Windows GUI as the preferred vehicle for producing user interfaces.

The tools mentioned in this section will remain supported until further notice, although several of them are becoming rather dated.

Arachnid workspace

STUDY

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

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.

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.

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. These examples were created for version 11.0, which was the first version that supported user-defined classes. The examples are a little dated, but still valid.

OO4APL folder

STUDY

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

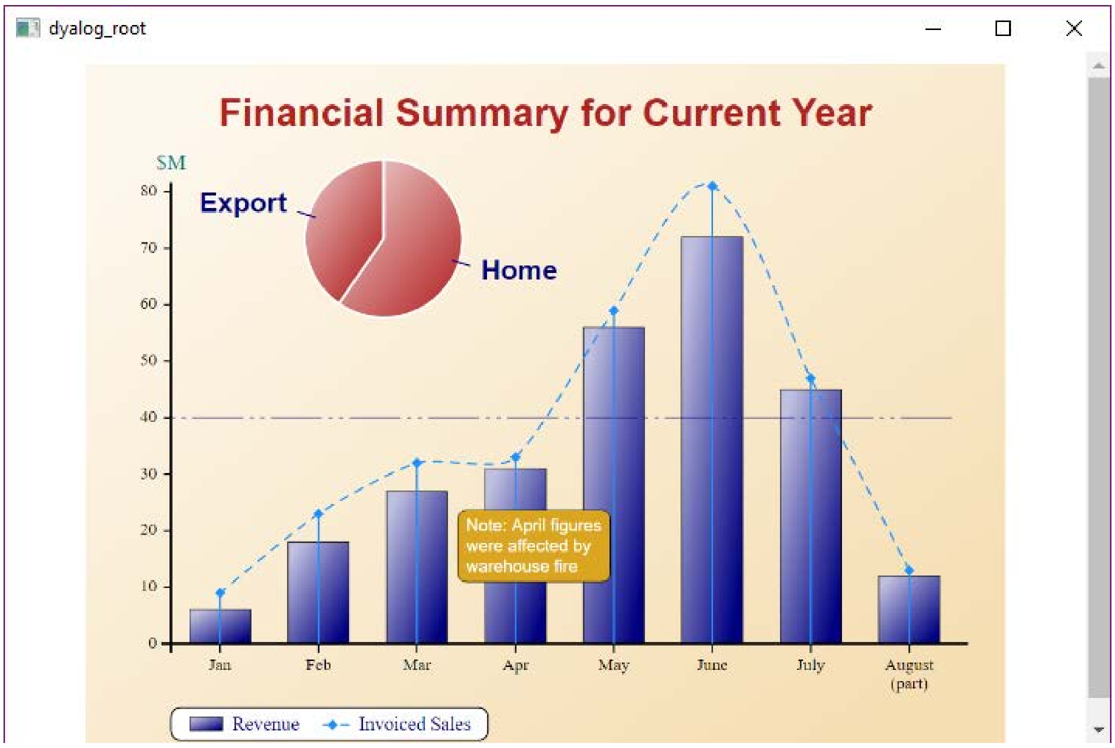
Presentation

SharpPlot

SharpPlot is a toolkit for producing scalable vector graphics to a very high standard, and is a cross-platform replacement for the APL workspace once known as RainPro. SharpPlot also includes the document composition capabilities of the old NewLeaf workspace.

The SharpPlot workspace includes extensive examples in the Samples namespace. The HTMLRenderer object provides a simple mechanism for viewiing the results:

```
      svg←Samples.Sample.RenderSvg #.SvgMode.FixedAspect
      'sp' ⎕WC 'HTMLRenderer' svg
```



You can produce a list of functions in the Samples namespace using:

```
       Samples.⎕nl ¯3
 Animations  Bar  Box  Bubbles  Cloud  Contour Gantt  Histogram
      Line  MinMax  MultDial  MultNested  MyReport  Pie  Polar
      Response  Sample  Scatter  Step  Tower  Trace  TreeMap
      Vectors  Venn  XBar
```

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
`CreateTemp`	Create a temporary file based on a file name template
`Dir`	Directory information for a filepath
`DirX`	Extended directory information for a filepath
`Exists`	Boolean result indicates whether file exists
`GetCurrentDirectory`	Get current directory
`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
`ReadAllLines`	Read a text file as single-byte text; return lines as nested character vector
`ReadAllText`	Read a text file as single-byte text
`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.