

# Release Notes

Version 12.0



*Copyright © 1982-2008 by Dyalog Limited.*

*All rights reserved.*

*Version 12.0.3*

***First Edition July 2008***

*No part of this publication may be reproduced in any form by any means without the prior written permission of Dyalog Limited, South Barn, Minchens Court, Minchens Lane, Bramley, Hampshire, RG26 5BH, United Kingdom.*

*Dyalog 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. Dyalog 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.*

*Microsoft, MS and MS-DOS are registered trademarks of Microsoft Corporation.*

*POSTSCRIPT is a registered 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 a registered 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.*

*All other trademarks and copyrights are acknowledged.*

# Contents

<b>Contents .....</b>	<b>iii</b>	
<b>CHAPTER 1 Introduction.....</b>	<b>1</b>	
Classic and Unicode Editions .....	1	
Component Files .....	1	
Development Environment.....	2	
Additional Tools.....	3	
SALTed Utility Libraries .....	4	
Miscellaneous.....	5	
System Requirements.....	6	
Converting to Unicode .....	7	
Interoperability and Compatibility .....	9	
<b>CHAPTER 2 Unicode Support (Unicode Edition only).....</b>	<b>13</b>	
Introduction .....	13	
Keyboard Input.....	15	
Character Usage and Restrictions.....	17	
Workspace and Performance Considerations .....	18	
Enhancements to □NA .....	19	
Changes to □DR.....	20	
Unicode System Functions and Variables.....	21	
Reading and Writing Unicode Native Files.....	22	
<b>CHAPTER 3 Language Enhancements .....</b>	<b>25</b>	
New and Revised Primitive & System Functions.....	25	
Grade Down (Monadic): .....	R←▽Y .....	26
Grade Up (Monadic): .....	R←▲Y .....	28
Underscored Alphabetic Characters: .....	R←□□ .....	30
Atomic Vector: .....	R←□AV .....	31
Atomic Vector - Unicode: .....	□AVU .....	32
Data Representation (Monadic): .....	R←□DR Y.....	34
Data Representation (Dyadic):.....	R←X □DR Y .....	35
File Copy: .....	R←X □FCOPY Y .....	36
File Create: .....	{R}←X □FCREATE Y.....	37
File Properties: .....	R←X □FPROPS Y.....	38
Map File: .....	R←{X}□MAP Y .....	40
Name Association: .....	{R}←{X}□NA Y .....	42
Native File Append: .....	{R}←X □NAPPEND Y.....	68
Native File Read: .....	R←□NREAD Y.....	69
Native File Replace: .....	{R}←X □NREPLACE Y .....	70

Native File Translate:..... {R}←{X} □NXLATE Y .....	72	
Terminal Control:..... ( □ML ) R←□TC .....	73	
Unicode Convert: .....R←{X} □UCS Y.....	74	
<b>CHAPTER 4 New Session Features .....</b>	<b>77</b>	
APL Keyboard .....	77	
On-Screen Keyboard.....	84	
Language Bar .....	86	
New Help System and Documentation Center.....	88	
New Configuration Dialogs .....	89	
Edit Window Tools.....	93	
SharpPlot Graphics .....	94	
<b>CHAPTER 5 Unicode and the Dyalog GUI.....</b>	<b>99</b>	
KeyPress .....	Event 22 .....	100
<b>Index.....</b>	<b>103</b>	

---

## CHAPTER 1

# Introduction

## Classic and Unicode Editions

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

For this reason, Version 12.0 and a limited number of future Versions of Dyalog will be available in two separate editions; *Unicode* and *Classic*.

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

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

## Component Files

Version 12.0 introduces two new options for Component Files; *journaling* and Unicode support.

### Journaling File System

Version 12.0 allows *journaling* to be enabled for selected component files (see `□FPROPS`). When journaling is enabled, files are updated using a journal which effectively prevents system or network failures from causing file damage. Enabling journaling will cause file write operations to run a little slower (the exact impact on performance will depend on the specific file operations being performed).

Versions of Dyalog prior to Version 12.0 will *not* be able to access files which have journaling enabled, but journaling can be switched off if this is necessary (see □FPROPS).

## Unicode Support

64-bit component files now possess a Unicode Property which determines whether or not character data will be stored as Unicode character data or using the internal representation employed by all previous Versions of Dyalog and by Version 12.0 Classic Edition (data type 82).

32-bit component files do not have this property and may not contain Unicode character data.

## □FCOPY System Function

A new system function, □FCOPY, is provided. This may be used to generate 64-bit component files from 32-bit component files, and as a generally useful tool to copy component files. The key point is that □FCOPY replicates component time-stamps and user information in the files.

# Development Environment

## Integrated APL Keyboard

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

Version 12.0 is supplied with two sets of such keyboards (one using Ctrl and one using AltGr) for a range of different languages. These keyboards were created using the Microsoft Keyboard Layout Creator (MSKLC) and you may use the same tool to customise one of the supplied keyboards or to create a new one.

## On-Screen Keyboard

Included with Dyalog APL Version 12.0 Unicode Edition is the Comfort On-Screen Keyboard 2.1 which has been specially extended for use with Dyalog APL and is distributed under a licence agreement with Comfort Software. The On-Screen keyboard is a really useful tool that works with any Windows application and replaces Kibitzer in the Unicode Edition. Kibitzer remains part of the Classic Edition.

## APL Language Bar

Version 12.0 provides an optional Window which is docked to the Session Window, to make it easy to pick APL symbols without using the keyboard. Furthermore, if you hover the mouse pointer over a symbol in the APL Language Bar, a pop-up tip is displayed to remind you of its usage.

## Built-In SharpPlot Graphics

Initially only in the 32-bit Windows version, there are some new buttons on the Session toolbar which display graphical images of the contents of the current object using SharpPlot

## Additional Tools

### CausewayPro, RainPro, and NewLeaf

In April 2007, Dyalog acquired the Causeway range of productivity tools for APL developers. These tools are now bundled with Dyalog, starting with Version 12.0 for 32-bit Windows.

- RainPro is a graphics package which produces high quality graphs for business and technical applications.
- NewLeaf is a printing utility which can produce output in PDF and a number of other popular formats.
- CausewayPro is a screen designer and application builder for simple applications.

## Conga

*Conga*, also known as the *Dyalog Remote Communicator*, is a tool for communication between applications. Conga can be used to transfer APL Arrays between two Dyalog applications which are both using Conga, but it can also be used to exchange messages with other partners like HTTP Servers (also known as *Web Servers*), Web Browsers, or any other web clients or servers including Telnet, SMTP, POP3 and so forth.

Uses of Conga include, but are not limited to the following:

- Retrieving information from, or posting data to, the internet.
- Accessing internet-based services like FTP, SMTP, or telnet
- Writing an APL application that acts as a Web (HTTP) Server, Mail Server or any other kind of service available over an intranet or the internet.
- *APL Remote Procedure Call* servers which receive APL arrays from client applications, process data, and return APL arrays as the result.

Conga provides a significantly simpler application programming interface compared to the use of TCPSocket objects, and includes support for secure communications using SSL.

## SALTed Utility Libraries

The Simple APL Library Toolkit (SALT) was introduced as an experimental tool with version 11.0. Starting with Version 12.0, Dyalog will gradually move towards using SALT and UTF-8 script files as the distribution mechanism for utilities and code samples. For a time, APL code will be distributed using both workspaces and the UTF-8 script files supported by SALT. In the future, SALT will become the preferred mechanism for this.

Version 12.0 includes tools and documentation on using SALT in combination with *SubVersion*, a popular source code management system. A public SubVersion server hosted by Dyalog, will allow users to collaborate on the development of open source, shared code libraries.

---

## Miscellaneous

### Performance Improvements for Set Functions

Dyadic  $\tau$  and other *set* functions ( $\epsilon \cap \cup \sim$ ) are significantly faster for certain arguments.

### Matrix Iota idiom

The matrix iota idiom  $M\{(\downarrow\alpha)\tau\downarrow\omega\}M$  which used only to operate on *character* matrices, now operates on any matrices.

### New idiom

There is a new idiom *Catenate To* ( $,\leftarrow$ ) which optimises *repeated* catenation of a scalar or vector to a vector.

### Atomic Vector Index idiom

The atomic vector index idiom  $\square AV \tau CA$  is not implemented in Unicode Edition, and should be replaced in both Editions by  $\square UCS$ .

### Default Address Size for $\square FCREATE$

From Version 12.0 onwards, the default *address size* for a component file is 64. In previous Versions it was 32.

The *address size* of a component file is an optional parameter specified in the right argument of  $\square FCREATE$ . A value of 32 causes the internal component addresses to be represented by 32-bit values which allow a maximum file size of 4GB. A value of 64 (which is now the default) causes the internal component addresses to be represented by 64-bit values which allows file sizes up to operating system limits.

Note that a 32-bit component file *may not* contain Unicode character data.

# System Requirements

## Microsoft Windows

Dyalog APL Version 12.0 supports the following Versions of Windows:

- Windows Vista
- Windows XP
- Windows 2000
- Windows Server 2003

Note that Dyalog APL Version 12.0 is not supported under Windows 95, Windows 98, Windows ME or Windows NT4.

## Microsoft .Net Interface

Dyalog APL Version 12.0 .Net Interface requires Version 2.0 or 3.0 of the Microsoft .Net Framework. It does *not* operate with .Net Version 1.0.

---

## Converting to Unicode

The adoption of Unicode is an important milestone in the evolution of Dyalog APL and the change to the way that character data is represented is fundamental. Although we have tried to minimise the impact, the requirement for users to make some changes in application code is inevitable.

If you use any of the features listed below, you will probably have to change your code to convert to Version 12.0 Unicode Edition.

- Monadic Grade Up and Grade Down
- Name Association
- Data Representation
- Native File Operations
- Custom written DLLs and Auxiliary Processors
- Mapped Files
- External Data

### Monadic Grade Up and Grade Down

Monadic Grade Up and Grade Down are based on the internal representation of characters. This means that Unicode arrays will sort in a different order than AVcode arrays. For example  $\uparrow 'aA'$  returns (2 1) in the Unicode Edition and (1 2) in the Classic Edition and all previous versions of Dyalog APL.

It will be necessary to re-evaluate the use of monadic  $\uparrow$  and  $\downarrow$  in Unicode Edition. Backwards compatibility and compatibility with Classic Edition can be achieved (if desired) by supplying  $\square AV$  as a left argument.

### Name Association

Windows DLLs typically contain two versions of functions that process character arguments; one for ASCII and one for Unicode. They are distinguished by appending either "A" (ASCII) or "W" (Wide) to the name of the function in the argument to  $\square NA$ .

For example, the specific name for the `MessageBox()` function is `MessageBoxA` (ASCII) or `MessageBoxW` (Unicode). To convert to Unicode Edition, You will have to change all your  $\square NA$  statements that specify "A" as the function suffix.

## Data Representation

In the Unicode Edition, the data type for character arrays is 80, 160 or 320; and not 82. This means that the results from `⊞DR` will change. In particular, code which determines whether an array is a character array using expressions like `{82=⊞DR ω}` will not work in the Unicode Edition.

## Native File Operations

Although the default data type for file operations involving character arrays will remain unchanged (80), in converting to Unicode Edition you should consider changing your code to handle characters that are not present in `⊞AVU`. You may need to specify different data types for `⊞NAPPEND`, `⊞NREPLACE`, and `⊞NREAD`.

## Custom DLLs and Auxiliary Processors

Custom written DLLs and Auxiliary Processors which receive and return data in internal APL format need to be rewritten to recognise and use the new data types. Data type 82 may be returned and will be translated by Version 12.0. Thus, if an AP or DLL recognizes all the character types, it can work with both the Unicode and Classic Editions.

## Mapped Files

Character data in raw mapped files will be interpreted differently in Unicode Edition.

## External Data

If you start using Unicode characters which are not in `⊞AVU` in your application, and you use external forms of storage like SQL databases or native files, you need to consider which representation, or encoding, that you wish to use. For relational databases, SQAPL supports new Unicode data types, but your database may not have native support for these types. Very often, an encoding called UTF-8 is used to store Unicode characters, both in flat files but also in relational databases which do not have “native” Unicode support.

Version 12.0 provides tools for converting data to and from UTF-8, but you need to decide on a format and possibly convert your existing databases to use new data types and representations. If you use a lot of external data, these conversions may require more planning and implementation work than the changes required to move APL application code to the Unicode edition.

# Interoperability and Compatibility

## Introduction

Workspaces and component files are stored on disk in a binary format (illegible to text editors). This format differs between machine architectures and among versions of Dyalog. For example a file component written by a PC will almost certainly have an internal format that is different from one written by a UNIX machine. Similarly, a workspace saved from Dyalog Version 12 will differ internally from one saved by a previous version of Dyalog APL.

It is convenient for versions of Dyalog APL running on different platforms to be able to *interoperate* by sharing workspaces and component files. However, this is not always possible. For example, if a new internal data structure is introduced in a particular version of Dyalog APL, previous versions could not be expected to make sense of it. In this case the load (or copy) from the older version would fail with the message:

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

Similarly, *large* (64-bit-addressing) component files are inaccessible to versions of the interpreter that pre-dated their introduction.

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

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

If the second item is missing, the file type defaults to 64-bit-addressing.

From Dyalog APL Version 11 onwards, there are two separate versions of programs for 32-bit and 64-bit machine architectures.

Interoperability is summed up in the following tables. Table rows show the version that is attempting to access the file or workspace and columns show the version that saved it:

```
This version can access files created by this version →
↓
```

The row and column titles show the Dyalog version **10.0**, **10.1**, etc; **(32)** and **(64)** indicate a version running on a 32-bit or 64-bit machine architecture, respectively.

## Implementation

The following tables document compatibility between different versions of Dyalog APL. Each row represents a system which is accessing or receiving data, each column represents a system which has saved (or created, or sent) the data.

In each cell, “Yes” means that all data can be transferred successfully. “-“ means that data cannot be accessed. “~” followed by one or more letters means that data can be read, with one or more exceptions:

o	Cannot read <code>⊞ORs</code> . Note that <code>⊞NULL</code> is represented as a namespace.
t	Cannot <b>tie</b> files created on machines with different byte ordering.
r	Cannot <b>read</b> a component with different byte ordering.
w	Can read from but cannot <b>write</b> to files created on machines with different byte ordering (attempting to write generates <code>FILE ACCESS ERROR</code> ).
u	Cannot tie a file with the Unicode property, cannot read components containing Unicode data. For sockets: Cannot read data in encoding Unicode.
j	Cannot tie a file with journaling enabled. Note that no versions prior to Version 12.0 can tie a journaled file.

In general, data is written, saved or transmitted in the format that is native to the writer. Readers do the work of any necessary translation. The exceptions to this rule at that:

- A 64-bit system writing to a 32-bit file will write components in 32-bit format
- Version 12 and above will write character data in either Unicode or non-Unicode format, depending on the Unicode bit of the file. 32-bit files are always non-Unicode.

## Workspaces

Workspaces cannot be loaded if saved by “higher” versions.

	10.0	10.1	11.0(32)	11.0(64)	12.0 (32)	12.0 (64)
10.0	Yes	-	-	-	-	-
10.1	Yes	Yes	-	-	-	-
11.0 (32)	Yes	Yes	Yes	Yes	-	-
11.0 (64)	-	Yes	Yes	Yes	-	-
12.0 (32)	Yes	Yes	Yes	Yes	Yes	Yes
12.0 (64)	-	Yes	Yes	Yes	Yes	Yes

## Small (32-bit) Component files and External Variables

Small component files are limited in size to 4GB and are limited to having the same architecture in all components.

	10.0	10.1	11.0	12.0
10.0	~t	~t	~ot	~otj
10.1	~t	~t	~ot	~otj
11.0	~w	~w	~w	~owj
12.0	~w	~w	~w	~w

## Large (64-bit) Component files

Large component files were introduced in version 10.1, and are the default architecture used by 12.0. In large component files, each component has its own architecture information (byte order, 32/64 data size, unicode).

	10.1	11.0	12.0
10.1	~r	-	-
11.0	Yes	Yes	~ouj
12.0	Yes	Yes	Yes

## Sockets (Type 'APL')

	10.0	10.1	11.0 (32)	11.0 (64)	12.0 (32)	12.0 (64)
10.0	Yes	~o	~o	-	~ou	-
10.1	Yes	Yes	~o	-	~ou	-
11.0 (32)	Yes	Yes	Yes	Yes	~ou	~ou
11.0 (64)	Yes	Yes	Yes	Yes	~ou	~ou
12.0 (32)	Yes	Yes	Yes	Yes	Yes	Yes
12.0 (64)	Yes	Yes	Yes	Yes	Yes	Yes

## Auxiliary Processes

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

## Session Files

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

---

## CHAPTER 2

# Unicode Support (Unicode Edition only)

## Introduction

### Benefits

Unicode is an industry standard allowing computers to consistently represent and manipulate text expressed in any of the world's writing systems. It assigns a number, or code point, to each of approximately 100,000 characters, including the APL character set.

The adoption of Unicode provides users of Dyalog with two important benefits:

- It is now possible to write Dyalog applications that fully and properly support not just American and Western European character sets, but all of the world's languages and writing systems. You can now write applications that input, store, display and print characters in the entire Unicode set. This capability also extends to the Dyalog Development Environment.
- Character data no longer needs to be translated as it enters or leaves Dyalog during inter-operation with other components like database systems or code libraries written in other languages.

### Character Arrays

To support Unicode, the internal form of character arrays has changed

In all Versions of Dyalog APL prior to 12.0 (Unicode Edition), character arrays are stored internally as indices into the *Atomic Vector* ( $\square AV$ ). This contains 256 different characters, so internally, character arrays are represented by sequences of numbers in the range 0-255. Each character consumes 1 byte (8 bits) of storage. The data type of a simple character array, reported by  $\square DR$ , is 82.

In Version 12.0 Unicode Edition, character arrays are stored internally as Unicode *code points*, which are integers that the Unicode consortium (currently) guarantees will never consume more than 21 bits.

For arrays which only contain code points between 0 and 255 (more or less the ANSI set), one byte is sufficient to contain the necessary code point for each character. Characters contained in the first *plane* of Unicode, with code points up to 65,535 (hexadecimal FFFF), can be represented using two bytes per character. Unicode characters beyond the first plane are stored using 4 bytes per character.

In order to conserve space, Version 12.0 Unicode Edition uses three character types, in roughly the same way that Dyalog has three integer types (ignoring Booleans, which *can* be described as 1-bit integers). The three character types have type numbers 80, 160 and 320. Type 82 is no longer used. The system will select the smallest type required to represent the data in an array.

In the same way as for integers, the conversion between the three types is an internal matter and for the most part invisible to the user. The only occasions on which you need to be aware of the internal format is in considering space requirements for arrays, and when using system functions which are sensitive to the internal form, like `⎕DR` and functions for manipulating native files.

Arrays with data type 82 are not supported in a Unicode Edition workspace. Data which has been stored by earlier Versions of Dyalog or by Classic Edition using character type 82 are translated to type 80, 160 or 320 on entry into a Unicode Edition workspace. In most cases, arrays containing APL characters will have type 160, other arrays will have type 80.

---

# Keyboard Input

## Introduction

All Versions of Dyalog APL prior to Version 12 employ a proprietary keyboard input mechanism in which the mapping of keystrokes to characters is defined through the Dyalog APL Input and Output Translate tables.

With the adoption of Unicode, it is no longer necessary for Dyalog to support a proprietary mechanism for entering (and displaying) APL symbols. APL characters are handled in the same way as all other Unicode characters, and like other Unicode characters, may be entered using whatever standard tools are provided to cater for unusual languages and special requirements.

Version 12.0 Unicode Edition no longer uses the Dyalog APL Input Translate Tables but instead works directly off standard Operating System keyboards and (under Windows) IMEs.

Version 12.0 Classic Edition, which does not support Unicode, continues to use the same proprietary keyboard input mechanism as before and the Dyalog Development Environment requires an Input Translate Table to specify the mechanism for entering APL symbols.

## MSKLC Keyboards for APL Characters

Under Windows, Microsoft provides a tool called the Microsoft Keyboard Layout Creator (MSKLC)<sup>1</sup>. This is intended to allow users to create custom keyboards to cater for requirements that are not covered by the standard sets of keyboards shipped with the different international editions of Windows.

Using the MSKLC Version 1.4, Dyalog has created a set of keyboards which are based upon standard National Language keyboards but which have the additional functionality to support APL characters.

Using one of these keyboards, APL characters are entered using a special *meta* key in combination with the normal character keys. Other keystrokes operate as normal. There are two variants supplied; one in which APL symbols are entered using the Ctrl key in combination with the normal character keys, and one in which the AltGr key is used instead. The former is compatible with the keyboard layouts supported by previous Versions of Dyalog APL and will be familiar to existing users. The latter, which uses AltGr in place of Ctrl has the advantage that it does not override the conventional use of keystrokes such as Ctrl-c (Cut).

---

<sup>1</sup> <http://www.microsoft.com/globaldev/tools/msklc.msp>

## The Dyalog APL IME

Since 2001, Dyalog APL for Windows has included an IME for entering APL symbols. This IME first appeared in Dyalog.Net and was intended to allow programmers to enter APL symbols into *non-APL* applications (such as editors and word-processors) to create APLScript files for ASP.NET and other .Net applications.

The Dyalog APL IME may be used with the Unicode Edition although it is effectively replaced by the MSKLC keyboards and will become a deprecated feature.

## Keyboard Shortcuts

The Dyalog Development Environment provides a number of shortcut keys that may be used to perform actions. These are identified by 2-character codes; for example the action to start the Tracer is identified by the code <TC>, and mapped to user-configurable keystrokes.

In all Versions of Dyalog prior to Version 12.0 Unicode Edition and in the Classic Edition, the mapping between keystrokes and actions is defined by the Input Translate Table.

Windows keyboards and IMEs do not support application specific accelerator keys, so for Version 12.0 Unicode Edition, this information is now specified separately using the Keyboard Shortcuts dialog (*Options/Configure/Keyboard Shortcuts*) and stored in the Windows Registry.

## Character Usage and Restrictions

### Atomic Vector $\square AV$

The Unicode Edition continues to support the atomic vector  $\square AV$  as a character array with 256 elements. To the human eye, its appearance is unchanged, and expressions which index  $\square AV$  or search it to find the positions of characters, produce the same results in the Unicode Edition as in the Classic Edition and in previous Versions of Dyalog.

### Underscores

The underscored APL alphabet ABCDEFGHIJKLMN OPQRSTUVWXYZ is not included in the Unicode standard (presumably because *underscore* is seen as a display attribute in the same category as *strikethrough*, *bold* or *italic*), although Unicode does include delta-underscore, Δ. This poses a serious problem because existing Dyalog applications include underscored characters in names (of APL objects) and in character arrays.

To resolve this problem, underscored characters are converted to corresponding characters in the Unicode *circled* alphabet, which contains the letters from  $\square$  to  $\square$ . Note that the APL 385 Unicode font, supplied with Dyalog displays the Unicode symbols  $\square$  to  $\square$  using the glyphs A to Z, thereby (somewhat artificially) allowing application code and data to have the same visual appearance as before.

Dyalog **strongly** recommends that you move away from the use of the underscored alphabet, as these symbols now constitute the sole remaining non-standard use of characters in Dyalog applications.

### Characters Allowed in User-Defined Names

In the Unicode character set, there are thousands of symbols which represent letters and ideograms. In principle, it would be possible to allow the use of all of these letters in user-defined names.

However, there are a number of interesting anomalies. For example, it might be confusing to allow the use of Greek lower-case letters  $\alpha$   $\epsilon$   $\rho$   $\omega$  in user-defined names, because they so closely resemble the APL symbols  $\alpha$   $\epsilon$   $\tau$   $\rho$   $\omega$ .

In order to allow time for an orderly discussion and resolution of naming issues, Version 12.0 Unicode Edition takes a conservative approach and only allows letters which were already allowed in previous Versions of Dyalog APL, namely:

```

0123456789 (but not as the 1st character in a name)
ABCDEFGHIJKLMN OPQRSTUVWXYZ_
abcdefghijklmnopqr stuvwxyz
ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖÙÚÛÜÝß
àáâãäåæçèéêëìíîïðñòóôõöùúûüþ
ΔΔ
ABCDEFGHIJKLMN OPQRSTUVWXYZ

```

Note that using a standard Unicode font (rather than APL385 Unicode used in the table above), the last row above would appear as the circled alphabet, □ to □.

## Workspace and Performance Considerations

In the Unicode Edition, character arrays (including constants embedded in functions), which contain characters that are not in the first 256 Unicode code points, will consume twice as much space as in previous Versions of Dyalog.

When a function is imported into the Unicode Edition from a Classic Edition workspace, component file, or a workspace saved by a previous Version of Dyalog,, comments in functions are converted to Unicode and the function is re-fixed. This has implications for space and speed.

- Following conversion to unicode, the function may consume more space. However, in Version 12.0 comments are stored as references to a table, where identical comments are shared, so this *may* actually reduce the space requirement.
- The operation to re-fix the function takes time, more if the function is very large. It is therefore recommended that you do not dynamically import functions from a Classic Edition workspace into the Unicode Edition (for example, using □CY or by □FX'ing a □OR) in an application.

## Enhancements to `□NA`

The system function `□NA` (Name Association) is used to load dynamically linked libraries, and perform name association between entry points in the libraries and function names in the workspace.

Version 12.0 contains a couple of enhancements aimed at making it easier to write name associations that are portable between the Unicode and Classic Editions.

### Default width of T arguments

In Version 12.0 Unicode Edition, an argument type of T (without a width specification) is interpreted to mean a *wide character* according to the convention of the host operating system. This translates to T2 under Windows and T4 under Unix or Linux.

Note therefore that the use of type T with default width (arguments coded as <0T, >0T and =0T) is **portable between Classic and Unicode Editions**. This is because, in the Classic Edition, T (with no width specifier) implies 1-byte characters, translated from `□AV` to ASCII, while In the Unicode Edition, T (with no width specifier) implies 2-byte characters.

If arguments or results need to be encoded in or decoded from UTF-8 or other formats, it is the programmer's task to do this using dyadic `□UCS`.

### Automatic Selection of A or W Functions

Under Windows, Win32 library calls are frequently available in two variants:

- An ANSI (narrow) version with a name ending in A
- a Unicode (wide) version with a name ending in W

For example, the function to create a Message Box is available as *MessageBoxA* and *MessageBoxW*. In Version 12.0, you may specify the character \* instead of A or W at the end of a function name. This will be replaced by A in the Classic Edition and W in the Unicode Edition. The default name of the associated function (if no left argument is given to `□NA`), will be without the trailing letter (**MessageBox**).

For example, the following function will display a Message Box with OK & Cancel buttons in both Editions (under Windows):

```

▽ ok←title MsgBox msg;MessageBox
[1]   □NA'I user32|MessageBox* I <0T <0T I'
[2]   ok←1=MessageBox 0 msg title 1   A 1=OK, 2=Cancel.
▽

```

## Changes to ⎕DR

In the Unicode Edition, the data type for character arrays is 80, 160 or 320; and not 82. However, ⎕DR continues to recognise a left argument of 82 (when the right argument contains numeric data) and returns *the same* characters as in previous Versions of Dyalog, except that the results are converted (and translated) to one of the new character types 80, 160 or 320. A left argument of 82 is not allowed if the right argument contains Unicode character data.

**Warning:** Conversions of Unicode character data to numeric types will often give different results in the Unicode Edition, as the internal representation of characters is different.

Left arguments of 80, 160 and 320 can be used with ⎕DR in both the Classic and Unicode Editions. In the Classic Edition, a TRANSLATION ERROR (event number 92) will be signalled if the result would contain characters not in ⎕AVU.

⎕AVU is a new system variable which is described overleaf.

## Monadic ⎕DR on Character Arrays

Code which determines whether an array is a character array using expressions like {82=⎕DR ω} will not work in the Unicode Edition.

The following expression can be used in both Editions:

```
ischar←{(10|⎕DR ω)ε0 2}
```

## Dyadic ⎕DR on Character Arrays

Because the internal representation of character arrays has changed, the result of dyadic ⎕DR on character arrays has changed:

Classic Edition	Unicode Edition
⎕DR 'A'	⎕DR 'A'
82	160
83 ⎕DR 'A'	83 ⎕DR 'A'
-109	75 35
163 ⎕DR 'A'	163 ⎕DR 'A'
LENGTH ERROR	9035

## Unicode System Functions and Variables

### Atomic Vector - Unicode $\square$ AVU

The Dyalog APL atomic vector  $\square$ AV has always been, to some extent, user definable, as the mapping between extended ASCII symbols (actually, the indices into an ASCII font) and  $\square$ AV positions was defined by the Output Translate Table. The *appearance* of the characters in  $\square$ AV also depended upon which font was used.

In Version 12.0 Unicode Edition, the contents of  $\square$ AV are defined, not by the Output Translate Table, but by a new system variable  $\square$ AVU (Atomic Vector – Unicode) . Its purpose is twofold:

1. To make it possible for each user to ensure that (each element of)  $\square$ AV contains the same character as before.
2. To guide the translation of character data read from old files and workspaces to Unicode in such a way that the appearance or content of this data is unchanged.

$\square$ AVU is an integer vector with 256 elements, containing the Unicode code points corresponding to the symbols represented by  $\square$ AV.

When the Unicode Edition imports character data from a Classic Edition workspace or from a workspace saved by Version 11 or earlier, the conversion of data from type 82 is controlled by the value of  $\square$ AVU. This also applies to character data stored in non-Unicode component files or received on non-Unicode TCPSocket connections.

When you load or copy an old workspace into the Unicode Edition, you can control the conversion of character data from that workspace by setting  $\square$ AVU first. The default value of  $\square$ AVU corresponds to the use of the **Dyalog Alt** font with the default win.dot Output Translate Table in the Classic Edition or in earlier versions of Dyalog APL.

### Unicode Convert $\square$ UCS

$\square$ UCS converts (Unicode) characters into integers and vice versa.

Used monadically, it simply converts between Unicode characters and Unicode code points. Used dyadically, the optional left argument converts a character string to or from its UTF-8, UTF-16 or UTF-32 encoding schemes.

## Reading and Writing Unicode Native Files

Text files (native files) may be written in a variety of different formats. Notepad, for example, allows you to save files in *ANSI*, *Unicode*, *Unicode big endian*, and *UTF-8*<sup>2</sup> encodings.

In ANSI format, each character is stored as a single byte. This format is only suitable for storing ANSI characters (Unicode code point range 0-255) and corresponds directly to Dyalog data type 80.

In Unicode format (UTF-16) a single 16-bit code unit is used to encode the first 65,536 most commonly used characters, and a pair of 16-bit code units, called surrogates, to encode the remaining less commonly used characters in Unicode.

In UTF-8 format, characters are represented by a mixture of single and multi-byte characters. Some character codes in the range (128-256) are used as lead-bytes to mark the start of multi-byte character codes. Using two or more bytes per character provides plenty of room to represent all the commonly used world characters.

Under Windows, native files are identified as containing Unicode data by the file signature which is a special prefix of 2 or 3 bytes at the beginning of the file. The following Unicode formats are frequently encountered.

Description	Signature
UTF-16 LE (Little Endian)	First 2 bytes are 0xFF, 0xFE (equivalent signed integers $-1$ $-2$ )
UTF-8	First 3 bytes are 0xEF, 0xBB, 0xBF (equivalent signed integers $-17$ $-69$ $-65$ )

The following functions `ReadFile` and `WriteFile` may be used to read and write files that are compatible with Notepad. Note that the functions do not support files in *UTF-32* or *Unicode big endian* formats nor 32-bit Unicode characters (data type 320).

---

<sup>2</sup> UTF stands for Universal Character Set Transformation Format.

```

▽ Chars←ReadFile name;nid;signature;nums
[1]  A Read ANSI or Unicode character file (Windows)
[2]  nid←name □NTIE 0
[3]  signature←3↑□NREAD nid 83 3 0
[4]  :If signature≡~17 ~69 ~65  A UTF-8
[5]      nums←□NREAD nid 83(~2+□NSIZE nid)3
[6]      Chars←'UTF-8'□UCS{ω+256×ω<0}nums A Signed ints
[7]  :ElseIf (2↑signature)≡~1 ~2  A Unicode (UTF-16)
[8]      Chars←□NREAD nid 160(~1+□NSIZE nid)2
[9]  :Else
[10]     Chars←□NREAD nid 80(□NSIZE nid)0
[11]  :EndIf
[12]  □NUNTIE nid

```

▽

```

▽ {format}WriteFile(name chars);nid;signature;nums
[1]  A Write ANSI or Unicode character file (Windows)
[2]  A format = ANSI or UTF-8 or UTF-16
[3]  □SIGNAL(~v/80 160=□DR chars)/11
[4]  :If 0=□NC'format'
[5]      format←(80 160i□DR chars)>'ANSI' 'UTF-16'
[6]  :EndIf
[7]  □SIGNAL(~(cformat)ε'ANSI' 'UTF-8' 'UTF-16')/11
[8]  :Trap 22
[9]      nid←name □NCREATE 0
[10] :Else
[11]     nid←name □NTIE 0
[12]     name □NERASE nid
[13]     nid←name □NCREATE 0
[14] :EndTrap
[15] :Select format
[16] :Case 'ANSI'
[17]     chars □NAPPEND nid 80
[18] :Case 'UTF-8'
[19]     ~17 ~69 ~65 □NAPPEND nid 83
[20]     nums←□UCS'UTF-8'□UCS chars
[21]     nums □NAPPEND nid 80
[22] :Else
[23]     ~1 ~2 □NAPPEND nid 83
[24]     chars □NAPPEND nid 160
[25] :EndSelect
[26] □NUNTIE nid

```

▽



## CHAPTER 3

# Language Enhancements

## New and Revised Primitive & System Functions

### New System Functions & Variables

□AVU	Atomic Vector - Unicode
□FCOPY	Copy File
□FPROPS	File Properties
□UCS	Unicode Convert

### Revised Primitive Functions, System Functions & Variables

Ψ	Grade Down
⤴	Grade Up
□A	Underscored Alphabet
□AV	Atomic Vector
□DR	Data Representation
□FCREATE	Create component file
□MAP	Map File
□NA	Name Association
□NAPPEND	Native File Append
□NREAD	Native File Read
□NREPLACE	Native File Replace
□NXLATE	Native File Translate
□TC	Terminal Control

**Grade Down (Monadic):** **$R \leftarrow \Psi Y$** 

$Y$  must be a simple character or simple numeric array of rank greater than 0.  $R$  is an integer vector being the permutation of  $\iota 1 \uparrow \rho Y$  that places the sub-arrays of  $Y$  along the first axis in descending order. The indices of any set of identical sub-arrays in  $Y$  occur in  $R$  in ascending order.

If  $Y$  is a numeric array of rank greater than 1, the elements in each of the sub-arrays along the first axis are compared in ravel order with greatest weight being given to the first element and least weight being given to the last element.

**Example**

```

      M
2 5 3 2
3 4 1 1
2 5 4 5
2 5 3 2
2 5 3 4

      ΨM
2 3 5 1 4

      M[ΨM; ]
3 4 1 1
2 5 4 5
2 5 3 4
2 5 3 2
2 5 3 2

```

If  $Y$  is a character array, the implied collating sequence is the numerical order of the corresponding Unicode code points (Unicode Edition) or the ordering of characters in  $\square AV$  (Classic Edition).

$\square IO$  is an implicit argument of Grade Down.

Note that character arrays sort differently in the Unicode and Classic Editions.

### Example

```

      M
Goldilocks
porridge
Porridge
3 bears

```

Unicode Edition	Classic Edition
$\Psi M$ 2 3 1 4	$\Psi M$ 3 1 4 2
$M[\Psi M; ]$ porridge Porridge Goldilocks 3 bears	$M[\Psi M; ]$ Porridge Goldilocks 3 bears porridge

**Grade Up (Monadic):** **$R \leftarrow \Uparrow Y$** 

$Y$  must be a simple character or simple numeric array of rank greater than 0.  $R$  is an integer vector being the permutation of  $\iota 1 \uparrow \rho Y$  that places the sub-arrays along the first axis in ascending order.

If  $Y$  is a numeric array of rank greater than 1, the elements in each of the sub-arrays along the first axis are compared in ravel order with greatest weight being given to the first element and least weight being given to the last element.

**Examples**

```

       $\Uparrow 22.5 \ 1 \ 15 \ 3 \ ^{-}4$ 
5 2 4 3 1

```

```

      M
2 3 5
1 4 7

```

```

2 3 5
1 2 6

```

```

2 3 4
5 2 4

```

```

       $\Uparrow M$ 
3 2 1

```

If  $Y$  is a character array, the implied collating sequence is the numerical order of the corresponding Unicode code points (Unicode Edition) or the ordering of characters in  $\square AV$  (Classic Edition).

$\square IO$  is an implicit argument of Grade Up

Note that character arrays sort differently in the Unicode and Classic Editions.

```

      M
Goldilocks
porridge
Porridge
3 bears

```

Unicode Edition	Classic Edition
$\uparrow$ M 4 1 3 2	$\uparrow$ M 2 4 1 3
M[ $\uparrow$ M; ] 3 bears Goldilocks Porridge porridge	M[ $\uparrow$ M; ] porridge 3 bears Goldilocks Porridge

## Underscored Alphabetic Characters:

R←A

A is a deprecated feature. Dyalog **strongly** recommends that you move away from the use of A and of the underscored alphabet itself, as these symbols now constitute the sole remaining non-standard use of characters in Dyalog applications.

In Versions of Dyalog APL prior to Version 11.0, A was a simple character vector, composed of the letters of the alphabet with underscores. If the Dyalog Alt font was in use, these symbols displayed as additional National Language characters.

### Version 10.1 and Earlier

A  
ABCDEFGHIJKLMN  
OPQRSTUVWXYZ

For compatibility with previous versions of Dyalog APL, functions that contain references to A will continue to return characters with the same *index* in AV as before. However, the display of A is now Á, and the old underscored symbols appear as they did when the Dyalog Alt font was in use.

### Current Version

Á  
 ÁÂÃÇÈÉÊËÌÍÎÏÐÒÓÔÕÙÚÝþǎìðð



## Atomic Vector - Unicode:

⎕AVU

⎕AVU specifies the contents of the atomic vector, ⎕AV, and is used to translate data between Unicode and non-Unicode character formats when required, for example when:

- Unicode Edition loads or copies a Classic Edition workspace or a workspace saved by a Version prior to Version 12.0.
- Unicode Edition reads character data from a non-Unicode component file, or receives data type 82 from a TCP socket.
- Unicode Edition writes data to a non-Unicode component file
- Unicode Edition reads or writes data from or to a Native File using conversion code 82.
- Classic Edition loads or copies a Unicode Edition workspace
- Classic Edition reads character data from a Unicode component file, or receives data type 80, 160, or 320 from a TCP socket.
- Classic Edition writes data to a Unicode component file.

⎕AVU is an integer vector with 256 elements, containing the Unicode code points which define the characters in ⎕AV.

### Note:

In Versions of Dyalog prior to Version 12.0 and in the Classic Edition, a character is stored internally as an index into the atomic vector, ⎕AV. When a character is displayed or printed, the index in ⎕AV is translated to a number in the range 0-255 which represents the index of the character in an Extended ASCII font. This mapping is done by the Output Translate Table which is user-configurable. Note that although ASCII fonts typically all contain the same symbols in the range 0-127, there are a number of different Extended ASCII font layouts, including proprietary APL fonts, which provide different symbols in positions 128-255. The actual symbol that appears on the screen or on the printed page is therefore a function of the Output Translate Table and the font in use. Classic Edition provides two different fonts (and thus two different ⎕AV layouts) for use with the Development Environment, named *Dyalog Std* (with APL underscores) and *Dyalog Alt* (without APL underscores)

The default value of ⎕AVU corresponds to the use of the **Dyalog Alt** Output Translate Table and font in the Classic Edition or in earlier versions of Dyalog APL.

```

      2 13ρ⎕AVU[97+ι26]
193 194 195 199 200 202 203 204 205 206 207 208 210
211 212 213 217 218 219 221 254 227 236 240 242 245
      ⎕UCS 2 13ρ⎕AVU[97+ι26]
ÁÂÃÇÈÊËÌÍÎÏÐÒ
ÓÔÕŨÚÛÝþǎïðō
```

␣AVU has namespace scope and can be localised, in order to make it straightforward to write access functions which receive or read data from systems with varying atomic vectors. If you have been using Dyalog Alt for most things but have some older code which uses underscores, you can bring this code together in the same workspace and have it all look “as it should” by using the Alt and Std definitions for ␣AVU as you copy each part of the code into the same Unicode Edition workspace.

```
)COPY avu.dws Std.␣AVU
C:\Program Files\Dyalog\Dyalog APL 12.0 Unicode\ws\avu
saved Thu Dec 06 11:24:32 2007

      2 13ρ␣AVU[97+ι26]
9398 9399 9400 9401 9402 9403 9404 9405 9406 9407 9408
9409 9410
9411 9412 9413 9414 9415 9416 9417 9418 9419 9420 9421
9422 9423
      ␣UCS 2 13ρ␣AVU[97+ι26]
ABCDEFGHIJKLM
NOPQRSTUVWXYZ
```

### Rules for Conversion on Import

When the Unicode Edition imports APL objects from a non-Unicode source, function comments and character data of type 82 are converted to Unicode. When the Classic Edition imports APL objects from a Unicode source, this translation is performed in reverse.

If the objects are imported from a Version 12.0 (or later) workspace (i.e. from a workspace that contains its own value of ␣AVU) the value of #.␣AVU (the value of ␣AVU in the root) in the *source* workspace is used. Otherwise, such as when APL objects are imported from a pre-Version 12 workspace, from a component file, or from a TCP socket, the local value of ␣AVU in the *target* workspace is used.

### Rules for Conversion on Export

When the Unicode Edition exports APL objects to a non-Unicode destination, such as a non-Unicode Component File or non-Unicode TCPSocket Object, function comments (in ␣ORs) and character data of type 82 are converted to ␣AV indices using the local value of ␣AVU.

When the Classic Edition exports APL objects to a Unicode destination, such as a Unicode Component File or Unicode TCPSocket Object, function comments (in ␣ORs) and character data of type 82 are converted to Unicode using the local value of ␣AVU.

In all cases, if a character to be translated is not defined in ␣AVU, a TRANSLATION ERROR (event number 92) will be signalled.

**Data Representation (Monadic):** **$R \leftarrow \square DR \ Y$** 

Monadic  $\square DR$  returns the type of its argument  $Y$ . The result  $R$  is an integer scalar containing one of the following values. Note that the internal representation and data types for character data differs between the Unicode and Classic Editions.

Value	Data Type
11	1 bit Boolean
<b>80</b>	<b>8 bits character</b>
<b>83</b>	<b>8 bits signed integer</b>
<b>160</b>	<b>16 bits character</b>
<b>163</b>	<b>16 bits signed integer</b>
<b>320</b>	<b>32 bits character</b>
323	32 bits signed integer
326	32 bits Pointer
645	64 bits Floating

**Unicode Edition**

Value	Data Type
11	1 bit Boolean
82	8 bits character
<b>83</b>	<b>8 bits signed integer</b>
<b>163</b>	<b>16 bits signed integer</b>
323	32 bits signed integer
326	32 bits Pointer
645	64 bits Floating

**Classic Edition**

Note that types **80**, **160** and **320** and **83** and **163** are exclusive to Dyalog APL.

## Data Representation (Dyadic):

$$R \leftarrow X \ \square\text{DR} \ Y$$

Dyadic  $\square\text{DR}$  converts the data type of its argument  $Y$  according to the type specification  $X$ . See monadic  $\square\text{DR}$  on the previous page for a list of data types.

### Case 1:

$X$  is a single integer value. The bits in the right argument are interpreted as elements of an array of type  $X$ . The shape of the resulting new array will typically be changed along the last axis. For example, a character array seen as Boolean will have 8 times as many elements along the last axis.

### Case 2:

$X$  is a 2-element integer value. The bits in the right argument are interpreted as type  $X[1]$ . The system then attempts to convert the elements of the resulting array to type  $X[2]$  without loss of precision. The result  $R$  is a two element nested array comprised of:

- [1] The converted elements or a fill element (0 or blank) where the conversion failed
- [2] A Boolean array of the same shape indicating which elements were successfully converted.

### Case 3: Classic Edition Only

$X$  is a 3-element integer value and  $X[2\ 3]$  is 163 82. The bits in the right argument are interpreted as elements of an array of type  $X[1]$ . The system then converts them to the character representation of the corresponding 16 bit integers. This case is provided primarily for compatibility with APL\*PLUS. For new applications, the use of the [conv] field with  $\square\text{NAPPEND}$  and  $\square\text{NREPLACE}$  is recommended.

Conversion to and from character (data type 82) uses the translate vector given by  $\square\text{NXLATE} \ 0$ . By default this is the mapping defined by the current output translate table (usually WIN.DOT).

**Note.** The internal representation of data may be modified during workspace compaction. For example, numeric arrays and (in the Unicode Edition) character arrays will if possible, be squeezed to occupy the least possible amount of memory. However, the internal representation of the result  $R$  is guaranteed to remain unmodified until it is re-assigned (or partially re-assigned) with the result of any function.

**File Copy:****R←X □FCOPY Y****Access Code: 4609**

Y must be a simple integer scalar or 1 or 2-element vector containing the file tie number and optional passnumber. The file need not be tied exclusively.

X is a character vector containing the name of a new file to be copied to.

The result R is the size of the new file in bytes.

The new file will be a 64-bit file, but will otherwise be identical to the original file. In particular, the file creation, modification and access times will be identical and all component level information, including the user number and update time, will be the same.

**Example**

```

tn←'oldfile32' □FTIE 0
'S' □FPROPS tn
32
'newfile64' □FCOPY tn
407796
tn←'newfile64' □FTIE 0
'S' □FPROPS tn
64
```

If X specifies the name of an existing file, the operation fails with a **FILE NAME ERROR**.

Note: This operation is atomic. If an error occurs during the copy operation (such as disk full) or if a strong interrupt is issued, the copy will be aborted and the new file X will not be created.

**File Create:****{R}←X □FCREATE Y**

Y must be a simple integer scalar or a 1 or 2 element vector containing the *file tie number* followed by an optional *address size*.

The *file tie number* must not be the tie number associated with another tied file.

The *address size* is an integer and may be either 32 or 64. A value of 32 causes the internal component addresses to be represented by 32-bit values which allow a maximum file size of 4GB. A value of 64 (the default) causes the internal component addresses to be represented by 64-bit values which allows file sizes up to operating system limits.

Note:

- a 32-bit component file *may not* contain Unicode character data.
- a 64-bit component file may not be accessed by versions of Dyalog APL prior to Version 10.1.0

X must be either

- a) a simple character scalar or vector which specifies the name of the file to be created. See *User Guide* for file naming conventions under UNIX and DOS.
- b) a vector of length 1 or 2 whose items are:
  - i. a simple character scalar or vector as above.
  - ii. an integer scalar specifying the file size limit in bytes.

The newly created file is tied for exclusive use.

The shy result of □FCREATE is the tie number of the new file.

**Automatic Tie Number Allocation**

A tie number of 0 as argument to a create or tie operation, allocates, and returns as an explicit result, the first (closest to zero) available tie number. This allows you to simplify code. For example:

from:

```
tie←1+[ /0, □FNUMS      A With next available number,
file □FCREATE tie     A ... create file.
```

to:

```
tie←file □FCREATE 0 A Create with first available..
```

**Examples**

```

'..\BUDGET\SALES'   □FCREATE 2   A Windows
'../budget/SALES.85' □FCREATE 2   A UNIX

'COSTS' 200000 □FCREATE 4           A max size 200000

'LARGE' □FCREATE 5 64              A 64-bit file
'SMALL' □FCREATE 6 32              A 32-bit file

```

**File Properties:****R←X □FPROPS Y****Access Code 1 or 8192**

□FPROPS reports and sets the properties of a component file.

Y must be a simple integer scalar or vector containing the file tie number.

X must be a simple character scalar or vector containing one or more valid Identifiers listed in the table below, or a vector of 2-element vectors, each of which contains an Identifier and a (new) value for that property.

If the left argument is a simple character array, the result R contains the current values for the properties identified by X. If the left argument is nested, the result R contains the previous values for the properties identified by X

Identifier	Property	Description / Legal Values
S	File Size (read only)	32 = Small Component Files (<4Gb) 64 = Large Component Files
E	Endian-ness (read only)	0 = Little-endian 1 = Big-endian
U	Unicode	0 = Characters must be written as type 82 arrays 1 = Characters must be written as Unicode arrays
J	Journaling	0 = Disable Journaling 1 = Enable Journaling

The default properties for a newly created file are as follows:

- S = 64
- U = 1 (Unicode Edition and 64-bit file) or 0 (otherwise)
- J = 0.
- E depends upon the computer architecture.

### Example

```
tn←'myfile64' □FCREATE 0
'SEUI' □FPROPS tn
64 0 1 0
```

```
tn←'myfile32' □FCREATE 0 32
'SEUI' □FPROPS tn
32 0 0 0
```

The following expression disables Unicode and switches Journaling on. The function returns the previous settings:

```
('U' 0)('J' 1) □FPROPS tn
1 0
```

The Unicode property applies only to 64-bit component files. 32-bit component files *may not* contain Unicode character data and the value of the Unicode property is always 0. To convert a 32-bit component file to a 64-bit component file, use □FCOPY.

Properties may be read by a task with □FREAD permission (access code 1), and set by a task with □FSTAC access (8192). To set the value of the Journaling property, the file must be exclusively tied.

If Journaling or Unicode properties are set, the file cannot be tied by Versions prior to Version 12.0.

**Map File:** **$R \leftarrow \{X\} \square \text{MAP } Y$** 

$\square \text{MAP}$  function associates a mapped file with an APL array in the workspace.

Two types of mapped files are supported; *APL* and *raw*. An *APL* mapped file contains the binary representation of a Dyalog APL array, including its header. A file of this type must be created using the supplied utility function  $\Delta \text{MPUT}$ . When you map an APL file, the rank, shape and data type of the array is obtained from the information on the file.

A *raw* mapped file is an arbitrary collection of bytes. When you map a raw file, you must specify the characteristics of the APL array to be associated with this data. In particular, the data type and its shape.

The type of mapping is determined by the presence (*raw*) or absence (*APL*) of the left argument to  $\square \text{MAP}$ .

The right argument *Y* specifies the name of the file to be mapped and, optionally, the access type and a start byte in the file. *Y* may be a simple character vector, or a 2 or 3-element nested vector containing:

1. file name (character scalar/vector)
2. access code (character scalar/vector) : one of : 'R', 'W', 'r' or 'w'
3. start byte offset (integer scalar/vector). Must be a multiple of 4 (default 0)

If *X* is specified, it defines the type and shape to be associated with *raw* data on file. *X* must be an integer scalar or vector. The first item of *X* specifies the data type and must be one of the following values:

Classic Edition	11, 82, 83, 163, 323 or 645
Unicode Edition	11, 80, 83, 160, 163, 320, 323 or 645

Following items determine the shape of the mapped array. A value of  $\bar{1}$  on any (but normally the first) axis in the shape is replaced by the system to mean: read as many complete records from the file as possible. Only one axis may be specified in this way.

If no left argument is given, file is assumed to contain a simple APL array, complete with header information (type, rank, shape, etc).

Mapped files may be updated by changing the associated array using indexed assignment:  $\text{var}[a] \leftarrow b$ .

Note that a *raw* mapped file may be updated *only* if its *file offset* is 0.

## Examples

Map raw file as a read-only *vector* of doubles:

```
vec←645 ^1 □MAP'c:\myfile'
```

Map raw file as a 20-column read-write *matrix* of 1-byte integers:

```
mat←83 ^1 20 □MAP'c:\myfile' 'W'
```

Replace some items in mapped file:

```
mat[2 3;4 5]←2 2ρι4
```

Map bytes 100-180 in raw file as a 5×2 read-only matrix of doubles:

```
dat←645 5 2 □MAP'c:\myfile' 'R' 100
```

Put simple 4-byte integer array on disk ready for mapping:

```
(=83 323 □DR 2 3 4ρι24)ΔMPUT'c:\myvar'
```

Then, map a read-write variable:

```
var←□MAP'c:\myvar' 'w'
```

Note that a mapped array need not be *named*. In the following example, a 'raw' file is mapped, summed and released, all in a single expression:

```
+ /163 ^1 □MAP'c:\shorts.dat'  
42
```

## Compatibility between Editions

In the Unicode Edition □MAP will fail with a TRANSLATION ERROR (event number 92) if you attempt to map an APL file which contains character data type 82.

In order for the Unicode Edition to correctly interpret data in a raw file that was written using data type 82, the file may be mapped with data type 83 and the characters extracted by indexing into □AVU.

## Name Association:

 $\{R\} \leftarrow \{X\} \square NA \ Y$ 

$\square NA$  provides access from APL to compiled functions within a **Dynamic Link Library (DLL)**. A DLL is a collection of functions typically written in C (or C++) each of which may take arguments and return a result.

Instructional examples using  $\square NA$  can be found in supplied workspace: `QUADNA.DWS`.

The DLL may be part of the standard operating system software, purchased from a third party supplier, or one that you have written yourself.

The right argument  $Y$  is a character vector that identifies the name and syntax of the function to be associated. The left argument  $X$  is a character vector that contains the name to be associated with the external function. If the  $\square NA$  is successful, a function (name class 3) is established in the active workspace with name  $X$ . If  $X$  is omitted, the name of the external function itself is used for the association.

The shy result  $R$  is a character vector containing the name of the external function that was fixed.

For example, `math.dll` might be a library of mathematical functions containing a function `divide`. To associate the APL name `div` with this external function:

```
'div' □NA 'F8 math|divide I4 I4'
```

where `F8` and `I4`, specify the types of the result and arguments expected by `divide`. The association has the effect of establishing a new function: `div` in the workspace, which when called, passes its arguments to `divide` and returns the result.

```
      )fns
div   div 10 4
2.5
```

## Type Declaration

In a compiled language such as C, the types of arguments and results of functions must be declared explicitly. Typically, these types will be published with the documentation that accompanies the DLL. For example, function `divide` might be declared:

```
double divide(long int, long int);
```

which means that it expects two long (4-byte) integer arguments and returns a double (8-byte) floating point result. Notice the correspondence between the C declaration and the right argument of `⎕NA`:

```
C:           double           divide (long int, long int);
```

```
APL: 'div' ⎕NA 'F8  math|divide           I4           I4 '
```

It is imperative that care be taken when coding type declarations. A DLL *cannot* check types of data passed from APL. A wrong type declaration will lead to erroneous results or may even cause the workspace to become corrupted and crash.

The full syntax for the right argument of `⎕NA` is:

**[result] library|function [arg1] [arg2] ...**

Note that functions associated with DLLs are never dyadic. All arguments are passed as items of a (possibly nested) vector on the right of the function.

## Locating the DLL

The DLL may be specified using a full pathname, file extension, and function type.

**Pathname:** If the full pathname is omitted, APL looks for the DLL in ‘standard’ Windows directories. Specifically, APL uses the `LoadLibrary` system function to locate the library, the exact workings of which can be found in the appropriate Windows documentation for the Software Development Kit.

Alternatively, a full pathname may be supplied in the usual way:

```
⎕NA'... c:\mydir\mydll|foo ...'
```

**Errors:** If the specified DLL (or a dependent DLL) fails to load it will generate:

```
FILE ERROR 1 No such file or directory
```

If the DLL loads successfully, but the specified library function is not accessible, it will generate:

```
VALUE ERROR
```

**File Extension:** If the file extension is omitted, `.dll` is assumed. Note that some DLLs are in fact `.exe` files, and in this case the extension must be specified explicitly:

```
⊞NA '... mydll.exe|foo ...'
```

**Function Type:** On a Windows computer, two distinct conventions, namely ‘C’ and ‘Pascal’ are in use for passing of arguments and receipt of results. If the type of the function you are calling differs from the default, for your version of Dyalog APL (see below) you must specify the function type explicitly immediately following the DLL name. Combinations are

```
.C32  32 bit, C calling convention          (the default).
.P32  32 bit, Pascal calling convention
```

### Example

```
⊞NA '... mydll.exe.P32|foo ...' ⍝ 32 bit Pascal
```

### Call by Ordinal Number

A DLL may associate an *ordinal number* with any of its functions. This number may then be used to call the function as an alternative to calling it by name. Using `⊞NA` to call by ordinal number uses the same syntax but with the function name replaced with its ordinal number. For example:

```
⊞NA '... mydll|57 ...'
```

### Multi-Threading

Appending the ‘&’ character to the function name causes the external function to be run in its own system thread. For example:

```
⊞NA '... mydll|foo& ...'
```

This means that other APL threads can run concurrently with the one that is calling the `⊞NA` function.

## Data Type Coding Scheme

The type coding scheme introduced above is of the form:

**[direction] [special] type [width] [array]**

The options are summarised in the following table and their functions detailed below.

Description	Symbol	Meaning
Direction	<	Pointer to array <i>input</i> to DLL function.
	>	Pointer to array <i>output</i> from DLL function
	=	Pointer to input/output array.
Special	0	Null-terminated string.
	#	Byte-counted string
Type	I	int
	U	unsigned int
	C	char
	T	Classic Edition char: translated to/from ANSI Unicode Edition char
	F	float
	A	APL array
Width	1	1-byte
	2	2-byte
	4	4-byte
	8	8-byte
Array	[ <i>n</i> ]	Array of length <i>n</i> elements
	[ ]	Array, length determined at call-time
Structure	{ ... }	Structure.

In the Classic Edition, C specifies untranslated character, whereas T specifies that the character data will be translated to/from  $\square AV$ .

In the Unicode Edition, C and T are identical (no translation of character data is performed) except that for C the default width is 1 and for T the default width is "wide" (2 bytes under Windows).

The use of T with default width is recommended to ensure portability between Editions.

## Direction

C functions accept data arguments either by *value* or by *address*. This distinction is indicated by the presence of a '\*' character in the argument declaration:

```
int  num1;           // value of num1 passed.  
int *num2;          // Address of num2 passed.
```

An argument (or result) of an external function of type pointer, must be matched in the `⊞NA` call by a declaration starting with one of the characters: `<`, `>`, or `=`.

In C, when an address is passed, the corresponding value can be used as either an *input* or an *output* variable. An output variable means that the C function overwrites values at the supplied address. Because APL is a call-by-value language, and doesn't have pointer types, we accommodate this mechanism by distinguishing output variables, and having them returned explicitly as part of the result of the call.

This means that where the C function indicates a *pointer type*, we must code this as starting with one of the characters: `<`, `>` or `=`.

- `<` indicates that the address of the argument will be used by C as an input variable and values at the address will *not* be over-written.
- `>` indicates that C will use the address as an output variable. In this case, APL must allocate an output array over which C can write values. After the call, this array will be included in the nested result of the call to the external function.
- `=` indicates that C will use the address for both input and output. In this case, APL duplicates the argument array into an output buffer whose address is passed to the external function. As in the case of an output only array, the newly modified copy will be included in the nested result of the call to the external function.

## Examples

- `<I2` Pointer to 2-byte integer - *input* to external function
- `>C` Pointer to character *output* from external function.
- `=T` Pointer to character *input* to and *output* from function.
- `=A` Pointer to APL array *modified* by function.

## Special

In C it is common to represent character strings as *null-terminated* or *byte counted* arrays. These special data types are indicated by inserting the symbol `0` (null-terminated) or `#` (byte counted) between the direction indicator (`<`, `>`, `=`) and the type (`T` or `C`) specification. For example, a pointer to a null-terminated input character string is coded as `<0T[ ]`, and an output one coded as `>0T[ ]`.

Note that while appending the array specifier `[ ]` is formally correct, because the presence of the special qualifier (`0` or `#`) *implies* an array, the `[ ]` may be omitted: `<0T`, `>0T`, `=#C`, etc.

Note also that the `0` and `#` specifiers may be used with data of all types and widths. For example, in the Classic Edition, `<0U2` may be useful for dealing with Unicode.

## Type

The data type of the argument is represented by one of the symbols **i**, **u**, **c**, **t**, **f**, **a**, which may be specified in lower or upper case:

	Type	Description
I	Integer	The value is interpreted as a 2s complement signed integer.
U	Unsigned integer	The value is interpreted as an unsigned integer.
C	Character	<p>The value is interpreted as a character.</p> <p>In the Unicode Edition, the value maps directly onto a Unicode code point.</p> <p>In the Classic Edition, the value is interpreted as an index into <code>⎕AV</code>. This means that <code>⎕AV positions</code> map onto corresponding ANSI <i>positions</i>.</p> <p>For example, with <code>⎕IO=0</code>:  <code>⎕AV[35] = 's'</code>, maps to <code>ANSI[35] = 's'</code></p>
T	Translated character	<p>The value is interpreted as a character.</p> <p>In the Unicode Edition, the value maps directly onto a Unicode code point.</p> <p>In the Classic Edition, the value is <i>translated</i> using standard Dyalog <code>⎕AV</code> to ANSI translation. This means that <code>⎕AV characters</code> map onto corresponding ANSI <i>characters</i>.</p> <p>For example, with <code>⎕IO=0</code>:  <code>⎕AV[35] = 's'</code>, maps to <code>ANSI[115] = 's'</code>.</p>
F	Float	The value is interpreted as an IEEE floating point number.
A	APL array	A pointer to the whole array (including header information) is passed. This type is used to communicate with DLL functions which have been written specifically to work with Dyalog APL. See the <i>User Guide</i> section on Writing Auxiliary Processors. Note that type A is always passed as a pointer, so is of the form <code>&lt;A</code> , <code>=A</code> or <code>&gt;A</code> .

## Width

The type specifier may be followed by the width of the value in bytes. For example:

**I4** 4-byte signed integer.  
**U2** 2-byte unsigned integer.  
**F8** 8-byte floating point number.  
**F4** 4-byte floating point number.

Type	Possible values for Width	Default value for Width
I	1, 2, 4, 8.	4 for 32-bit DLLs 8 for 64-bit DLLs
U	1, 2, 4, 8.	4 for 32-bit DLLs 8 for 32-bit DLLs
C	1,2,4	1
T	1,2,4	wide character(see below)
F	4, 8.	8
A	Not applicable.	

In the Unicode Edition, the default width is the width of a *wide character* according to the convention of the host operating system. This translates to T2 under Windows and T4 under Unix or Linux.

## Examples

**I2** 16-bit integer  
**<I4** Pointer to input 4-byte integer  
**U** Default width unsigned integer.  
**=F4** Pointer to input/output 4-byte floating point number.

## Arrays

Arrays are specified by following the basic data type with `[n]` or `[]`, where `n` indicates the number of elements in the array. In the C declaration, the number of elements in an array may be specified explicitly at compile time, or determined dynamically at runtime. In the latter case, the size of the array is often passed along with the array, in a separate argument. In this case, `n`, the number of elements is omitted from the specification. Note that C deals only in scalars and rank 1 (vector) arrays.

```
int vec[10];           // explicit vector length.
unsigned size, list[]; // undetermined length.
```

could be coded as:

```
I[10]  vector of 10 ints.
U U[]  unsigned integer followed by an array of unsigned integers.
```

Confusion sometimes arises over a difference in the declaration syntax between C and `⊞NA`. In C, an argument declaration may be given to receive a pointer to either a single scalar item, or to the first element of an array. This is because in C, the address of an array is deemed to be the address of its first element.

```
void foo (char *string);

char ch = 'a', ptr = "abc";

foo(&ch);           // call with address of scalar.
foo(ptr);           // call with address of array.
```

However, from APL's point of view, these two cases are distinct and if the function is to be called with the address of (pointer to) a *scalar*, it must be declared: `'<T'`. Otherwise, to be called with the address of an *array*, it must be declared: `'<T[]'`. Note that it is perfectly acceptable in such circumstances to define more than one name association to the same DLL function specifying different argument types:

```
'FooScalar'⊞NA'mydll|foo <T'   ⋄ FooScalar'a'
'FooVector'⊞NA'mydll|foo <T[]' ⋄ FooVector'abc'
```

## Structures

Arbitrary data structures, which are akin to nested arrays, are specified using the symbols `{}`. For example, the code `{F8 I2}` indicates a structure comprised of an 8-byte *float* followed by a 2-byte *int*. Furthermore, the code `<{F8 I2}[3]` means an input pointer to an array of 3 such structures.

For example, this structure might be defined in C thus:

```
typedef struct
{
    double  f;
    short   i;
} mystruct;
```

A function defined to receive a count followed by an *input* pointer to an array of such structures:

```
void foo(unsigned count, mystruct *str);
```

An appropriate `□NA` declaration would be:

```
□NA 'myd11.foo U <{F8 I2}[]'
```

A call on the function with two arguments - a count followed by a vector of structures:

```
foo 4, c(1.4 3)(5.9 1)(6.5 2)(0 0)
```

Notice that for the above call, APL converts the two Boolean `(0 0)` elements to an 8-byte float and a 2-byte int, respectively.

## Specifying Pointers Explicitly

⊞NA syntax enables APL to pass arguments to DLL functions by *value* or *address* as appropriate. For example if a function requires an integer followed by a *pointer* to an integer:

```
void fun(int valu, int *addr);
```

You might declare and call it:

```
⊞NA 'mydll|fun I <I' ⋄ fun 42 42
```

The interpreter passes the *value* of the first argument and the *address* of the second one.

Two common cases occur where it is necessary to pass a pointer explicitly. The first is if the DLL function requires a *null pointer*, and the second is where you want to pass on a pointer which itself is a result from a DLL function.

In both cases, the pointer argument should be coded as **I4**. This causes APL to pass the pointer unchanged, *by value*, to the DLL function.

In the previous example, to pass a null pointer, (or one returned from another DLL function), you must code a separate ⊞NA definition.

```
'fun_null'⊞NA 'mydll|fun I I4' ⋄ fun_null 42 0
```

Now APL passes the *value* of the second argument (in this case 0 - the null pointer), rather than its address.

## Using a Function

A DLL function may or may not return a result, and may take zero or more arguments. This syntax is reflected in the coding of the right argument of `⊞NA`. Notice that the corresponding associated APL function is niladic or monadic (never dyadic), and that it *always* returns a vector result - a null one if there is no output from the function. See Result Vector section below. Examples of the various combinations are:

DLL function Non-result-returning:

```
⊞NA 'mydll|fn1'           A Niladic
⊞NA 'mydll|fn2 <0T'      A Monadic - 1-element arg
⊞NA 'mydll|fn3 =0T <0T' A Monadic - 2-element arg
```

DLL function Result-returning:

```
⊞NA 'I4 mydll|fn4'       A Niladic
⊞NA 'I4 mydll|fn5 F8'    A Monadic - 1-element arg
⊞NA 'I4 mydll|fn6 >I4[] <0T' A Monadic - 2-element arg
```

When the external function is called, the number of elements in the argument must match the number defined in the `⊞NA` definition. Using the example functions defined above:

```
fn1           A Niladic Function.
fn2 ←'Single String' A 1-element arg
fn3 'This' 'That'  A 2-element arg
```

Note in the second example, that you must enclose the argument string to produce a single item (nested) array in order to match the declaration. Dyalog converts the type of a numeric argument if necessary, so for example in `fn5` defined above, a Boolean value would be converted to double floating point (F8) prior to being passed to the DLL function.

## Pointer Arguments

When passing pointer arguments there are three cases to consider.

- < **Input pointer:** In this case you must supply the data array itself as argument to the function. A pointer to its first element is then passed to the DLL function.

```
fn2 ← 'hello'
```

- > **Output pointer:** Here, you must supply the **number of elements** that the output will need in order for APL to allocate memory to accommodate the resulting array.

```
fn6 10 'world' A 1st arg needs space for 10 ints.
```

Note that if you were to reserve fewer elements than the DLL function actually used, the DLL function would write beyond the end of the reserved array and may cause the interpreter to crash with a System Error.

- = **Input/Output:** As with the input-only case, a pointer to the first element of the argument is passed to the DLL function. The DLL function then overwrites some or all of the elements of the array, and the new value is passed back as part of the result of the call. As with the output pointer case, if the input array were too short, so that the DLL wrote beyond the end of the array, the interpreter would almost certainly crash.

```
fn3 '.....' 'hello'
```

## Result Vector

In APL, a function cannot overwrite its arguments. This means that any output from a DLL function must be returned as part of the explicit result, and this includes output via ‘output’ or ‘input/output’ pointer arguments.

The general form of the result from calling a DLL function is a nested vector. The first item of the result is the defined explicit result of the external function, and subsequent items are implicit results from output, or input/output pointer arguments.

The length of the result vector is therefore: 1 (if the function was declared to return an explicit result) + the number of output or input/output arguments.

ONA Declaration	Result	Output Arguments	Result Length
<code>mydll fn1</code>	0		0
<code>mydll fn2 &lt;OT</code>	0	0	0
<code>mydll fn3 =OT &lt;OT</code>	0	1 0	1
<code>I4 mydll fn4</code>	1		1
<code>I4 mydll fn5 F8</code>	1	0	1
<code>I4 mydll fn6 &gt;I4[] &lt;OT</code>	1	1 0	2

As a convenience, if the result would otherwise be a 1-item vector, it is disclosed. Using the third example above:

```

      ρfn3 '.....' 'abc'
5

```

`fn3` has no explicit result; its first argument is input/output pointer; and its second argument is input pointer. Therefore as the length of the result would be 1, it has been disclosed.

## ANSI /Unicode Versions of Library Calls

Under Windows, most library functions that take character arguments, or return character results have two forms: one Unicode (Wide) and one ANSI. For example, a function such as `MessageBox()`, has two forms `MessageBoxA()` and `MessageBoxW()`. The `A` stands for ANSI (1-byte) characters, and the `W` for wide (2-byte Unicode) characters.

It is essential that you associate the form of the library function that is appropriate for the Dyalog Edition you are using, i.e. `MessageBoxA()` for the Classic Edition, but `MessageBoxW()` for the Unicode Edition.

To simplify writing portable code for both Editions, you may specify the character `*` instead of `A` or `W` at the end of a function name. This will be replaced by `A` in the Classic Edition and `W` in the Unicode Edition.

The default name of the associated function (if no left argument is given to `⎕NA`), will be without the trailing letter (`MessageBox`).

## Type Definitions (typedefs)

The C language encourages the assignment of defined names to primitive and complex data types using its `#define` and `typedef` mechanisms. Using such abstractions enables the C programmer to write code that will be portable across many operating systems and hardware platforms.

Windows software uses many such names and Microsoft documentation will normally refer to the type of function arguments using defined names such as `HANDLE` or `LPSTR` rather than their equivalent C primitive types: `int` or `char*`.

It is beyond the scope of this manual to list *all* the Microsoft definitions and their C primitive equivalents, and indeed, DLLs from sources other than Microsoft may well employ their own distinct naming conventions.

In general, you should consult the documentation that accompanies the DLL in order to convert typedefs to primitive C types and thence to `⎕NA` declarations. The documentation may well refer you to the 'include' files which are part of the Software Development Kit, and in which the types are defined.

The following table of some commonly encountered Windows typedefs and their `⎕NA` equivalents might prove useful.

Windows typedef	□NA equivalent
HWND	I
HANDLE	I
GLOBALHANDLE	I
LOCALHANDLE	I
DWORD	U4
WORD	U2
BYTE	U1
LPSTR	=0T[] (note 1)
LPCSTR	<0T[] (note 2)
WPARAM	U
LPARAM	U4
LRESULT	I4
BOOL	I
UINT	U
ULONG	U4
ATOM	U2
HDC	I
HBITMAP	I
HBRUSH	I
HFONT	I
HICON	I
HMENU	I
HPALETTE	I
HMETAFILE	I
HMODULE	I
HINSTANCE	I
COLORREF	{U1[4]}
POINT	{I I}
POINTS	{I2 I2}
RECT	{I I I I}
CHAR	T or C

## Notes

1. LPSTR is a pointer to a null-terminated string. The definition does not indicate whether this is input or output, so the safest coding would be =OT[ ] (providing the vector you supply for input is long enough to accommodate the result). You may be able to improve simplicity or performance if the documentation indicates that the pointer is 'input only' (<OT[ ]) or 'output only' (>OT[ ]). See **Direction** above.
2. LPCSTR is a pointer to a *constant* null-terminated string and therefore coding <OT[ ] is safe.
3. Note that the use of type T with default width ensures portability of code between Classic and Unicode Editions. In the Classic Edition, T (with no width specifier) implies 1-byte characters which are translated between  $\square AV$  and ASCII, while In the Unicode Edition, T (with no width specifier) implies 2-byte (Unicode) characters.

## Dyalog32.dll

Included with Dyalog APL is a utility DLL which is called dyalog32.dll.

The DLL contains two functions: MEMCPY and STRNCPY.

### MEMCPY

MEMCPY is an extremely versatile function used for moving arbitrary data between memory buffers.

Its C definition is:

```
void MEMCPY(           // copy memory
    void *to,          // target address
    void *fm,          // source address
    unsigned size      // number of bytes to copy
);
```

MEMCPY copies `size` bytes starting from source address `fm`, to destination address `to`. If the source and destination areas overlap, the result is undefined.

MEMCPY's versatility stems from being able to associate to it using many different type declarations.

### Example

Suppose a global buffer (at address: `addr`) contains (`numb`) double floating point numbers. To copy these to an APL array, we could define the association:

```
'doubles' ⍵NA 'dya1og32|MEMCPY >F8[] I4 U4'
doubles numb addr (numb×8)
```

Notice that:

As the first argument to `doubles` is an output argument, we must supply the number of elements to reserve for the output data.

`MEMCPY` is defined to take the number of *bytes* to copy, so we must multiply the number of elements by the element size in bytes.

### Example

Suppose that a database application requires that we construct a record in global memory prior to writing it to file. The record structure might look like this:

```
typedef struct {
    int empno;           // employee number.
    float salary;       // salary.
    char name[20];      // name.
} person;
```

Then, having previously allocated memory (`addr`) to receive the record, we can define:

```
'prec' ⍵NA 'dya1og32|MEMCPY I4 <{I4 F4 T[20]} U4'
prec addr(99 12345.60 'Charlie Brown
')(4+4+20)
```

### STRNCPY

`STRNCPY` is used to copy null-terminated strings between memory buffers. Its C definition is:

```
void STRNCPY(           // copy null-terminated string
    char *to,           // target address
    char *fm,           // source address
    unsigned size       // MAX number of chars to copy
);
```

`STRNCPY` copies a maximum of `size` characters from the null-terminated source string at address `fm`, to the destination address `to`. If the source and destination strings overlap, the result is undefined.

If the source string is shorter than `size`, null characters are appended to the destination string.

If the source string (including its terminating null) is longer than `size`, only `size` characters are copied and the resulting destination string is not null-terminated

### Example

Suppose that a database application returns a pointer (`addr`) to a structure that contains two pointers to (max 20-char) null-terminated strings.

```
typedef struct {      // null-terminated strings:
    char *first;     // first name (max 19 chars + 1 null).
    char *last;      // last name. (max 19 chars + 1 null).
} name;
```

To copy the names *from* the structure:

```
'get'⊂NA'dyalog32|STRNCPY >OT[] I4 U4'
get 20 addr 20
Charlie
get 20 (addr+4) 20
Brown
```

To copy data *from* the workspace *into* an already allocated (`new`) structure:

```
'put'⊂NA'dyalog32|STRNCPY I4 <OT[] U4'
put new 'Bo' 20
put (new+4) 'Peep' 20
```

Notice in this example that you must ensure that names no longer than 19 characters are passed to `put`. More than 19 characters would not leave `STRNCPY` enough space to include the trailing null, which would probably cause the application to fail.

## Examples

The following examples all use functions from the Microsoft Windows USER32.DLL.

This DLL should be located in a standard Windows directory, so you should not normally need to give the full path name of the library. However if trying these examples results in the error message 'FILE ERROR 1 No such file or directory', you must locate the DLL and supply the full path name (and possibly extension).

### Example 1

The Windows function "GetCaretBlinkTime" retrieves the caret blink rate. It takes no arguments and returns an unsigned *int* and is declared as follows:

```
UINT GetCaretBlinkTime(void);
```

The following statements would provide access to this routine through an APL function of the same name.

```

      □NA 'U User32|GetCaretBlinkTime'
      GetCaretBlinkTime
530
```

```
      □NA 'U User32|GetCaretBlinkTime'
```

The following statement would achieve the same thing, but using an APL function called BLINK.

```

      'BLINK' □NA 'U User32|GetCaretBlinkTime'
      BLINK
530
```

### Example 2

The Windows function "SetCaretBlinkTime" sets the caret blink rate. It takes a single unsigned *int* argument, does not return a result and is declared as follows:

```
void SetCaretBlinkTime(UINT);
```

The following statements would provide access to this routine through an APL function of the same name :

```

      □NA 'User32|SetCaretBlinkTime U'
      SetCaretBlinkTime 1000
```

### Example 3

The Windows function "MessageBox" displays a standard dialog box on the screen and awaits a response from the user. It takes 4 arguments. The first is the window handle for the window that owns the message box. This is declared as an unsigned *int*. The second and third arguments are both pointers to null-terminated strings containing the message to be displayed in the Message Box and the caption to be used in the window title bar. The 4th argument is an unsigned *int* that specifies the Message Box type. The result is an *int* which indicates which of the buttons in the message box the user has pressed. The function is declared as follows:

```
int MessageBox(HWND, LPCSTR, LPCSTR, UINT);
```

The following statements provide access to this routine through an APL function of the same name. Note that the 2nd and 3rd arguments are both coded as input pointers to type T null-terminated character arrays which ensures portability between Editions.

```
⊞NA 'I User32|MessageBox* U <0T <0T U'
```

The following statement displays a Message Box with a stop sign icon together with 2 push buttons labelled OK and Cancel (this is specified by the value 19).

```
MessageBox 0 'Message' 'Title' 19
```

The function works equally well in the Unicode Edition because the <0T specification is portable.

```
MessageBox 0 'Το Μήνυμα' 'Ο Τίτλος' 19
```

Note that a simpler, portable (and safer) method for displaying a Message Box is to use Dyalog APL's primitive `MsgBox` object.

### Example 4

The Windows function "FindWindow" obtains the window handle of a window which has a given character string in its title bar. The function takes two arguments. The first is a pointer to a null-terminated character string that specifies the window's class name. However, if you are not interested in the class name, this argument should be a NULL pointer. The second is a pointer to a character string that specifies the title that identifies the window in question. This is an example of a case described above where two instances of the function must be defined to cater for the two different types of argument. However, in practice this function is most often used without specifying the class name. The function is declared as follows:

```
HWND FindWindow(LPCSTR, LPCSTR);
```

The following statement associates the APL function `FW` with the second variant of the `FindWindow` call, where the class name is specified as a NULL pointer. To indicate that APL is to pass the *value* of the NULL pointer, rather than its address, we need to code this argument as `I4`.

```
'FW' ⍋NA 'U User32|FindWindow* I4 <OT'
```

To obtain the handle of the window entitled "CLEAR WS - Dyalog APL/W":

```
⍋←HNDL←FW 0 'CLEAR WS - Dyalog APL/W'
59245156
```

### Example 5

The Windows function "GetWindowText" retrieves the caption displayed in a window's title bar. It takes 3 arguments. The first is an unsigned *int* containing the window handle. The second is a pointer to a buffer to receive the caption as a null-terminated character string. This is an example of an output array. The third argument is an *int* which specifies the maximum number of characters to be copied into the output buffer. The function returns an *int* containing the actual number of characters copied into the buffer and is declared as follows:

```
int GetWindowText(HWND, LPSTR, int);
```

The following associates the "GetWindowText" DLL function with an APL function of the same name. Note that the second argument is coded as "`>OT`" indicating that it is a pointer to a character output array.

```
⍋NA 'I User32|GetWindowText* U >OT I'
```

Now change the Session caption using `)WSID` :

```
)WSID MYWS
was CLEAR WS
```

Then retrieve the new caption (max length 255) using window handle `HNDL` from the previous example:

```
DISPLAY GetWindowText HNDL 255 255
→-----
| 19 |MYWS - Dyalog APL/W|
←-----
```

There are three points to note. Firstly, the number 255 is supplied as the second argument. This instructs APL to allocate a buffer large enough for a 255-element character vector into which the DLL routine will write. Secondly, the result of the APL function is a nested vector of 2 elements. The first element is the result of the DLL function. The second element is the output character array.

Finally, notice that although we reserved space for 255 elements, the result reflects the length of the actual text (19).

An alternative way of coding and using this function is to treat the second argument as an input/output array.

e.g.

```

      □NA 'I User32|GetWindowText* U =0T I'
      DISPLAY GetWindowText HNDL (255p' ') 255
      ┌───────────────────────────────────────────────────────────────────────────────────┐
      │ 19 |MYWS - Dyalog APL/W| ───────────────────────────────────────────────────────────┘
      └───────────────────────────────────────────────────────────────────────────────────┘
  
```

In this case, the second argument is coded as =0T, so when the function is called an array of the appropriate size must be supplied. This method uses more space in the workspace, although for small arrays (as in this case) the real impact of doing so is negligible.

### Example 6

The function "GetCharWidth" returns the width of each character in a given range. Its first argument is a device context (handle). Its second and third arguments specify font positions (start and end). The third argument is the resulting integer vector that contains the character widths (this is an example of an output array). The function returns a Boolean value to indicate success or failure. The function is defined as follows. Note that this function is provided in the library: GDI32.DLL.

```

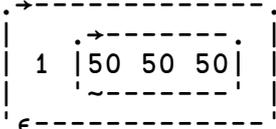
      BOOL GetCharWidth(HDC, UINT, UINT, int FAR*);
  
```

The following statements provide access to this routine through an APL function of the same name :

```

      ⍎NA 'U4 GDI32|GetCharWidth* I U U >I[]'
      'P'⍎WC'Printer'
      DISPLAY GetCharWidth ('P' ⍎WG 'Handle') 65 67 3

```



Note: 'P'⍎WG'Handle' returns a 32-bit handle which, if the *top bit* is set, will appear in APL as a negative integer. Attempting to supply such a negative number as an argument to a DLL function when the argument is declared *unsigned* will result in a **DOMAIN ERROR**. Window handles should therefore be declared as **I** rather than **U**.

### Example 7

The following example from the supplied workspace: **QUADNA.DWS** illustrates several techniques which are important in advanced ⍎NA programming. Function **D11Version** returns the major and minor version number for a given DLL.

In advanced DLL programming, it is often necessary to administer memory outside APL's workspace. In general, the procedure for such use is:

1. Allocate global memory.
2. Lock the memory.
3. Copy any DLL input information from workspace into memory.
4. Call the DLL function.
5. Copy any DLL output information from memory to workspace.
6. Unlock the memory.
7. Free the memory.

Notice that steps 1 and 7, and steps 2 and 6 complement each other. That is, if you allocate global system memory, you must free it after you have finished using it. If you continue to use global memory without freeing it, your system will gradually run out of resources. Similarly, if you lock memory (which you must do before using it), then you should unlock it before freeing it. Although on some versions of Windows, freeing the memory will include unlocking it, in the interests of good style, maintaining the symmetry is probably a good thing.

```

▽ version←DllVersion file;Alloc;Free;Lock;Unlock;Size
;Info;Value;Copy;size;hdl;addr;buff;ok
[1]
[2] 'Alloc'⎕NA'U4 kernel32|GlobalAlloc U4 U4'
[3] 'Free'⎕NA'U4 kernel32|GlobalFree U4'
[4] 'Lock'⎕NA'U4 kernel32|GlobalLock U4'
[5] 'Unlock'⎕NA'U4 kernel32|GlobalUnlock U4'
[6]
[7] 'Size'⎕NA'U4 version|GetFileVersionInfoSize* <OT >U4'
[8] 'Info'⎕NA'U4 version|GetFileVersionInfo*<OT U4 U4 U4'
[9] 'Value'⎕NA'U4 version|VerQueryValue* U4 <OT >U4 >U4'
[10]
[11] 'Copy'⎕NA'dyalog32|MEMCPY >U4[] U4 U4'
[12]
[13] :If xsize↔Size file 0 A Size of info
[14] :AndIf xhdl←Alloc 0 size A Alloc memory
[15] :If xaddr←Lock hdl A Lock memory
[16] :If xInfo file 0 size addr A Version info
[17] ok buff size←Value addr'\ ' 0 0 A Version value
[18] :If ok
[19] buff←Copy(size÷4)buff size A Copy info
[20] version←(2/2*16)τ÷2↓buff A Split version
[21] :EndIf
[22] :EndIf
[23] ok←Unlock hdl A Unlock memory
[24] :EndIf
[25] ok←Free hdl A Free memory
[26] :EndIf
▽

```

Lines [2-11] associate APL function names with the DLL functions that will be used.

Lines [2-5] associate functions to administer global memory.

Lines [7-9] associate functions to extract version information from a DLL.

Line[11] associates **Copy** with MEMCPY function from **dyalog32.dll**.

Lines [13-26] call the DLL functions.

Line [13] requests the size of buffer required to receive version information for the DLL. A size of 0 will be returned if the DLL does not contain version information.

**Notice that** care is taken to balance memory allocation and release:

On line [14], the :If clause is taken only if the global memory allocation is successful, in which case (and only then) a corresponding Free is called on line [25].

Unlock on line[23] is called if and only if the call to Lock on line [15] succeeds.

A result is returned from the function *only* if all the calls are successful Otherwise, the calling environment will sustain a **VALUE ERROR**.

## More Examples

```

□NA'U4 ADVAPI32 |RegCloseKey          U4'
□NA'I  ADVAPI32 |RegCreateKeyEx*       U <OT I <OT I I I >U >U'
□NA'U  ADVAPI32 |RegEnumValue*        U U >OT =U U >U >OT =U'
□NA'I  ADVAPI32 |RegOpenKey*          U <OT >U'
□NA'U4 ADVAPI32 |RegOpenKeyEx*       U4 <OT U4 U4 =U4'
□NA'I  ADVAPI32 |RegQueryValueEx*    U <OT U4 >U4 >OT =I4'
□NA'I4 ADVAPI32 |RegSetValueEx*      U <OT U4 U4 <OT U4'
□NA'I  DOS_U32  |Copy                <OT <OT'
□NA'U  DOS_U32  |Dir                  <OT U >OT'
□NA'U  DOS_U32  |DirMore                U >OT'
□NA'   DOS_U32  |DirClose'
□NA'I  DOS_U32  |Rename                <OT <OT'
□NA'I  DOS_U32  |Erase                <OT'
□NA'   dyaLog32 |STRNCPY                >OT I4 U4'
□NA'   dyaLog32 |MEMCPY                >{U1[4]}[16] I4 U4'
□NA'I  gdi32    |AddFontResource*     <OT'
□NA'   GDI32    |BitBlt                U U U U U U U U'
□NA'U4 gdi32    |GetPixel              U4 U4 U4'
□NA'U4 GDI32    |GetStockObject         U4'
□NA'I  gdi32    |RemoveFontResource*  <OT'
□NA'U4 gdi32    |SetPixel              U4 U4 U4 U4'
□NA'U4 GLU32    |gluPerspective        F8 F8 F8 F8'
□NA'I  kernel32 |_lclose                U'
□NA'I  kernel32 |_lcreat                <OT I'
□NA'U4 kernel32 |_llseek                I U4 I'
□NA'I  kernel32 |_lopen                <OT I'
□NA'U  kernel32 |_lread                 U >U1[] U'
□NA'U  kernel32 |_lwrite                U <U1[] U'
□NA'I  kernel32 |CopyFileA             <OT <OT I'
□NA'I4 kernel32 |GetEnvironmentStrings'
□NA'I4 kernel32 |GetLastError'
□NA'I  kernel32 |GetPrivateProfileInt*  <OT <OT I <OT'
□NA'I  kernel32 |GetProfileString*     <OT <OT <OT >OT I'
□NA'U2 kernel32 |GetSystemDirectory*   >OT U2'
□NA'I2 KERNEL32 |GetTempPathA         U4 >OT'
□NA'U  KERNEL32 |GetWindowsDirectory*  >OT U'
□NA'U  kernel32 |GlobalAlloc           U U4'
□NA'U  kernel32 |GlobalFree            U'
□NA'   Kernel32 |GlobalMemoryStatus   ={U4 U4 U4 U4 U4 U4 U4 U4}'
□NA'U  KERNEL32 |WritePrivateProfileString* <OT <OT <OT <OT'
□NA'U4 OpenGL32 |glClearColor                F4 F4 F4 F4'
□NA'U4 OpenGL32 |glClearDepth                F4'
□NA'U4 OpenGL32 |glEnable                    U4'
□NA'U4 OpenGL32 |glMatrixMode                U4'
□NA'U4 USER32   |ClientToScreen            U =U4[2]'
□NA'I  user32   |FindWindow*                I4 <OT'
□NA'   user32   |ShowWindow                 I I'
□NA'I2 USER32   |GetAsyncKeyState          I'
□NA'U4 user32   |GetDC                      U4'
□NA'I4 User32   |GetDialogBaseUnits'
□NA'I  user32   |GetFocus'
□NA'U4 user32   |GetSysColor                I'
□NA'I  user32   |GetSystemMetrics          I'
□NA'   USER32   |InvalidateRgn             I4 U4 U4'
□NA'I  user32   |MessageBox*               I <OT <OT I'
□NA'U4 user32   |ReleaseDC                  U4 U4'
□NA'U4 USER32   |SendMessage*             I4 U4 U4 <I[]'
□NA'I2 user32   |SetFocus                    I'
□NA'I  user32   |WinHelp*                  I <OT I I4'
□NA'U  WINMM    |sndPlaySound*            <OT U'

```

**Native File Append:****{R}←X □NAPPEND Y**

This function appends the ravel of its left argument  $X$  to the end of the designated native file.  $X$  must be a simple homogeneous APL array.  $Y$  is a 1- or 2-element integer vector.  $Y[1]$  is a negative integer that specifies the file number of a native file. The optional second element  $Y[2]$  specifies the data type to which the array  $X$  is to be converted before it is written to the file.

The file index result returned is the position within the file of the end of the record, which is also the start of the following one.

**Unicode Edition**

Unless you specify the data type in  $Y[2]$ , a character array will by default be written using type 80.

If the data will not fit into the specified character width (bytes)  $\square$ NAPPEND will fail with a **DOMAIN ERROR**.

As a consequence of these two rules, you must specify the data type (either 160 or 320) in order to write Unicode characters whose code-point are in the range 256-65535 and >65535 respectively.

**Example**

```

n←'test' □NCREATE 0
'abc' □nappend n
'ταβέρνα' □nappend n
DOMAIN ERROR
'ταβέρνα' □NAPPEND n
^
'ταβέρνα' □NAPPEND n 160
□NREAD n 80 3 0
abc
□NREAD n 160 7
ταβέρνα

```

For compatibility with old files, you may specify that the data be converted to type 82 on output. The conversion (to  $\square$ AV indices) will be determined by the local value of  $\square$ AVU.

**Native File Read:****R←NREAD Y**

This monadic function reads data from a native file. Y is a 3- or 4-element integer vector whose elements are as follows:

- [1] negative tie number,
- [2] conversion code (see below),
- [3] count,
- [4] start byte, counting from 0.

Y[2] specifies conversion to an APL internal form as follows. Note that the internal formats for character arrays differ between the Unicode and Classic Editions.

Value	Number of bytes read	Result Type	Result shape
11	count	1 bit Boolean	8 × count
<b>80</b>	<b>count</b>	<b>8 bits character</b>	<b>count</b>
82 <sup>3</sup>	count	8 bits character	count
<b>83</b>	<b>count</b>	<b>8 bits integer</b>	<b>count</b>
<b>160</b>	<b>2 × count</b>	<b>16-bits character</b>	<b>count</b>
<b>163</b>	<b>2 × count</b>	<b>16 bits integer</b>	<b>count</b>
<b>320</b>	<b>4 × count</b>	<b>32-bits character</b>	<b>count</b>
323	4 × count	32 bits integer	count
645	8 × count	64bits floating	count

**Unicode Edition : Conversion Codes**

Value	Number of bytes read	Result Type	Result shape
11	count	1 bit Boolean	8 × count
82	count	8 bits character	count
<b>83</b>	<b>count</b>	<b>8 bits integer</b>	<b>count</b>
<b>163</b>	<b>2 × count</b>	<b>16 bits integer</b>	<b>count</b>
323	4 × count	32 bits integer	count
645	8 × count	64bits floating	count

**Classic Edition : Conversion Codes**

Note that types **80**, **160** and **320** and **83** and **163** are exclusive to Dyalog APL.

<sup>3</sup> Conversion code 82 is permitted in the Unicode Edition for compatibility and causes 1-byte data on file to be *translated* (according to NXLATE) from AV indices into normal (Unicode) characters of type 80, 160 or 320.

**Example**

```
DATA←⎕NREAD ⍵ 160 (0.5×⎕NSIZE ⍵) 0 ⎕ Unicode
DATA←⎕NREAD ⍵ 82 (⎕NSIZE ⍵) 0 ⎕ Classic
```

**Native File Replace:****{R}←X ⎕NREPLACE Y**

⎕NREPLACE is used to write data to a native file, replacing data which is already there.

X must be a simple homogeneous APL array containing the data to be written.

Y is a 2- or 3-element integer vector whose elements are as follows:

- [1] negative tie number,
- [2] start byte, counting from 0, at which the data is to be written,
- [3] conversion code (optional).

See ⎕NREAD for a list of valid conversion codes.

The shy result is the position within the file of the end of the record, or, equivalently, the start of the following one. Used, for example, in:

```
⎕ Replace sequentially from indx.
{α ⎕NREPLACE tie ω}/vec,indx
```

**Unicode Edition**

Unless you specify the data type in Y[2], a character array will by default be written using type 80. .

If the data will not fit into the specified character width (bytes) ⎕NREPLACE will fail with a **DOMAIN ERROR**.

As a consequence of these two rules, you must specify the data type (either 160 or 320) in order to write Unicode characters whose code-point are in the range 256-65535 and >65535 respectively.

**Example**

```

n←'test'⊞NTIE 0 ρ See ⊞NAPPEND example
⊞NREAD n 80 3 0
abc
⊞NREAD n 160 7
ταβέρνα
⊞←'εστιατόριο'⊞NREPLACE n 3
DOMAIN ERROR
⊞←'εστιατόριο'⊞NREPLACE n 3
^
⊞←'εστιατόριο'⊞NREPLACE n 3 160
23
⊞NREAD n 80 3 0
abc
⊞NREAD n 160 10
εστιατόριο

```

For compatibility with old files, you may specify that the data be converted to type 82 on output. The conversion (to ⊞AV indices) will be determined by the local value of ⊞AVU.

## Native File Translate:

**{R}←{X}⊖NXLATE Y**

This associates a character translation vector with a native file or, if Y is 0, with the use by ⊖DR.

A translate vector is a 256-element vector of integers from 0-255. Each element maps the corresponding ⊖AV position onto an ANSI character code.

For example, to map ⊖AV[17+⊖IO] onto ANSI 'a' (code 97), element 17 of the translate vector is set to 97.

⊖NXLATE is a non-Unicode (Classic Editon) feature and is retained in the Unicode Edition, only for compatibility.

Y is either a negative integer tie number associated with a tied native file or 0. If Y is negative, monadic ⊖NXLATE returns the current translation vector associated with the corresponding native file. If specified, the left argument X is a 256-element vector of integers that specifies a new translate vector. In this case, the old translate vector is returned as a shy result. If Y is 0, it refers to the translate vector used by ⊖DR to convert to and from character data.

The system treats a translate vector with value  $(\iota 256) - \ominus\text{IO}$  as meaning *no translation* and thus provides raw input/output bypassing the whole translation process.

The default translation vector established at ⊖NTIE or ⊖NCREATE time, maps ⊖AV characters to their corresponding ANSI positions and is derived from the mapping defined in the current output translation table (normally WIN.DOT)

Between them, ANSI and RAW translations should cater for most uses.

### Unicode Edition

⊖NXLATE is relevant in the Unicode Edition only to process Native Files that contain characters expressed as indices into ⊖AV, such as files written by the Classic Edition.

In the Unicode Edition, when reading data from a Native File using conversion code 82, incoming bytes are translated first to ⊖AV indices using the translation table specified by ⊖NXLATE, and then to type 80, 160 or 320 using ⊖AVU. When writing data to a Native File using conversion code 82, characters are converted using these two translation tables in reverse.

<b>Terminal Control:</b>	<b>( <math>\square</math>ML )</b>	<b>R <math>\leftarrow</math> <math>\square</math>TC</b>
--------------------------	-----------------------------------	---

$\square$ TC is a deprecated feature and is replaced by  $\square$ UCS (see note).

$\square$ TC is a simple three element vector. If  $\square$ ML  $<$  3 this is ordered as follows:

$\square$ TC[ 1 ] - Backspace  
 $\square$ TC[ 2 ] - Linefeed  
 $\square$ TC[ 3 ] - Newline

Note that  $\square$ TC  $\equiv$   $\square$ AV[  $\square$ IO+ 1 3 ] for  $\square$ ML  $<$  3 .

If  $\square$ ML  $\geq$  3 the order of the elements of  $\square$ TC is instead compatible with IBM's APL2:

$\square$ TC[ 1 ] - Backspace  
 $\square$ TC[ 2 ] - Newline  
 $\square$ TC[ 3 ] - Linefeed

Elements of  $\square$ TC beyond 3 are not defined but are reserved.

### Note

With the introduction of  $\square$ UCS in Version 12.0, the use of  $\square$ TC is discouraged and it is strongly recommended that you generate control characters using  $\square$ UCS instead. This recommendation holds true even if you continue to use the Classic Edition.

Control Character	Old	New
Backspace	$\square$ TC[ 1 ]	$\square$ UCS 8
Linefeed	$\square$ TC[ 2 ] ( $\square$ ML $<$ 3) $\square$ TC[ 3 ] ( $\square$ ML $\geq$ 3)	$\square$ UCS 10
Newline	$\square$ TC[ 3 ] ( $\square$ ML $<$ 3) $\square$ TC[ 2 ] ( $\square$ ML $\geq$ 3)	$\square$ UCS 13

**Unicode Convert:** **$R \leftarrow \{X\} \square \text{UCS } Y$** 

$\square \text{UCS}$  converts (Unicode) characters into integers and vice versa.

The optional left argument  $X$  is a character vector containing the name of a variable-length Unicode encoding scheme which must be one of:

- 'UTF-8'
- 'UTF-16'
- 'UTF-32'

If not, a `DOMAIN ERROR` is issued.

If  $X$  is omitted,  $Y$  is a simple character or integer array, and the result  $R$  is a simple integer or character array with the same rank and shape as  $Y$ .

If  $X$  is specified,  $Y$  must be a simple character or integer vector, and the result  $R$  is a simple integer or character vector.

**Monadic  $\square \text{UCS}$** 

Used monadically,  $\square \text{UCS}$  simply converts characters to Unicode code points and vice-versa.

With a few exceptions, the first 256 Unicode code points correspond to the ANSI character set.

```
 $\square \text{UCS 'Hello World'}$ 
72 101 108 108 111 32 87 111 114 108 100
```

```
 $\square \text{UCS } 2 \ 11\rho 72 \ 101 \ 108 \ 108 \ 111 \ 32 \ 87 \ 111 \ 114 \ 108 \ 100$ 
Hello World
Hello World
```

The code points for the Greek alphabet are situated in the 900's:

```
 $\square \text{UCS 'καλημέρα Ελλάδα'}$ 
954 945 955 951 956 941 961 945 32 949 955 955 940 948 945
```

Thanks to work done by the APL Standards committee in the previous millennium, Unicode also contains the APL character set. For example:

```
 $\square \text{UCS } 123 \ 40 \ 43 \ 47 \ 9077 \ 41 \ 247 \ 9076 \ 9077 \ 125$ 
{ (+/ω) ÷ ρω }
```

## Dyadic `UCS`

Dyadic `UCS` is used to translate between Unicode characters and one of three standard variable-length Unicode encoding schemes, UTF-8, UTF-16 and UTF-32. These represent a Unicode character string as a vector of 1-byte (UTF-8), 2-byte (UTF-16) and 4-byte (UTF-32) signed integer values respectively.

```
'UTF-8' UCS 'ABC'
65 66 67
'UTF-8' UCS 'ABCÆØÅ'
65 66 67 195 134 195 152 195 133
'UTF-8' UCS 195 134, 195 152, 195 133
ÆØÅ

'UTF-8' UCS 'γεια σου'
206 179 206 181 206 185 206 177 32 207 131 206 191 207 133
'UTF-16' UCS 'γεια σου'
947 949 953 945 32 963 959 965
'UTF-32' UCS 'γεια σου'
947 949 953 945 32 963 959 965
```

Because integers are *signed*, numbers greater than 127 will be represented as 2-byte integers (type 163), and are thus not suitable for writing directly to a native file. To write the above data to file, the easiest solution is to use `UCS` to convert the data to 1-byte characters and append this data to the file:

```
(UCS 'UTF-8' UCS 'ABCÆØÅ') NAPPEND tn
```

**Note regarding UTF-16:** For most characters in the first plane of Unicode (0000-FFFF), UTF-16 and UCS-2 are identical. However, UTF-16 has the potential to encode all Unicode characters, by using more than 2 bytes for characters outside plane 1.

```
'UTF-16' UCS 'ABCÆØÅΨΔ'
65 66 67 198 216 197 9042 9035
←unihan←UCS (2×2×16)+13 A x20001-x200034
专丕
'UTF-16' UCS unihan
55360 56321 55360 56322 55360 56323
```

## Translation Error

`UCS` will generate `TRANSLATION ERROR` (event number 92) if the argument cannot be converted or, in the Classic Edition, if the result is not in `AV`.

<sup>4</sup> See for example <http://unicode.org/cgi-bin/GetUnihanData.pl?codepoint=20001>



## CHAPTER 4

# New Session Features

## APL Keyboard

### Introduction

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

Version 12.0 is supplied with two sets of such keyboards (one using Ctrl and one using AltGr) for a range of different languages as listed below. These keyboards were created using the Microsoft Keyboard Layout Creator (MSKLC) and you may use the same tool to customise one of the supplied keyboards or to create a new one..

## Installation

During the Installation of Dyalog Version 12.0 Unicode Edition, setup installs one or two APL keyboard layouts onto your system. These keyboard layouts are installed as additional services for your default Input Language.

The following table lists the APL keyboards included with Dyalog APL Version 12.0 Unicode Edition at the time of publication. Other keyboards will be included as they are developed.

<b>Ctrl Keyboards</b>	<b>AltGr Keyboards</b>
Danish - Dyalog Ctrl	Danish - Dyalog AltGr
Finnish - Dyalog Ctrl	Finnish - Dyalog AltGr
French - Dyalog Ctrl	French - Dyalog AltGr
German - Dyalog Ctrl	German Dyalog AltGr
Icelandic - Dyalog Ctrl	
Italian - Dyalog Ctrl	Italian - Dyalog AltGr
Norwegian - Dyalog Ctrl	
Russian - Dyalog Ctrl	
Swedish - Dyalog Ctrl	
UK - Dyalog Ctrl	UK - Dyalog AltGr
US - Dyalog Ctrl	US - Dyalog AltGr

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

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

The following picture illustrates the *Text Services and Input Languages* configuration pane after installing Unicode Edition onto a Windows XP system on which the default Input Language is English (United Kingdom). Incidentally, on this particular system, the Danish and Greek languages are also installed.



## Configuring your APL Keyboard for Use

There are 3 different ways to use your APL keyboard:

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

### **Making your APL Keyboard the default Windows keyboard**

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

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

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





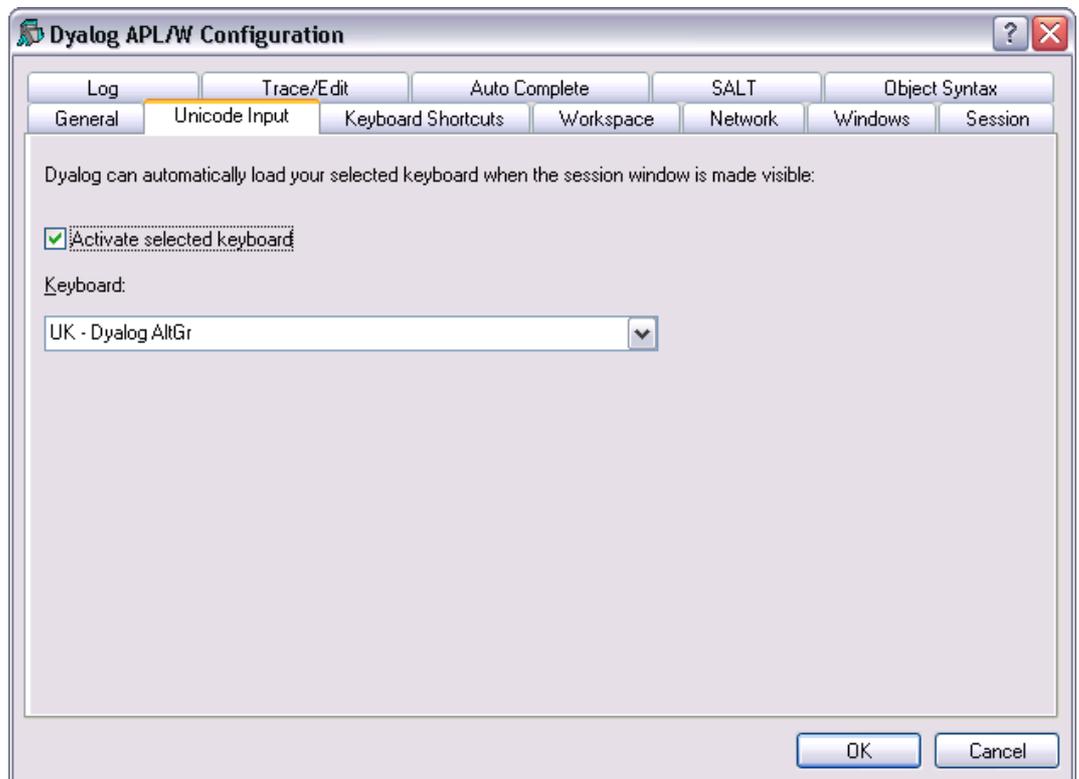
If you wish to, you can select a keystroke to enable you to select it quickly from the keyboard.



## Automatic Keyboard Selection

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

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



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

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

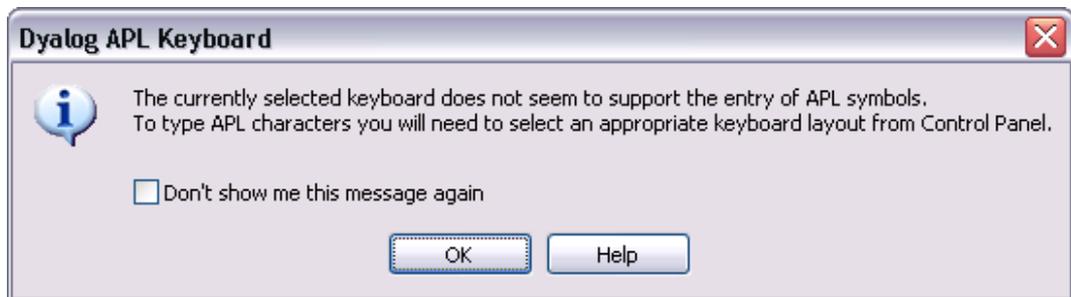
```
InitialKeyboardLayoutInUse = 1
```

```
InitialKeyboardLayout = " UK - Dyalog AltGr"
```

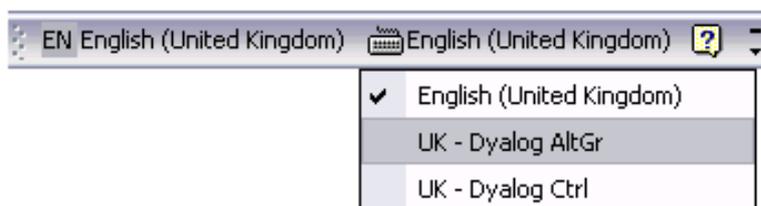
## Manual Keyboard Selection

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

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



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



## On-Screen Keyboard

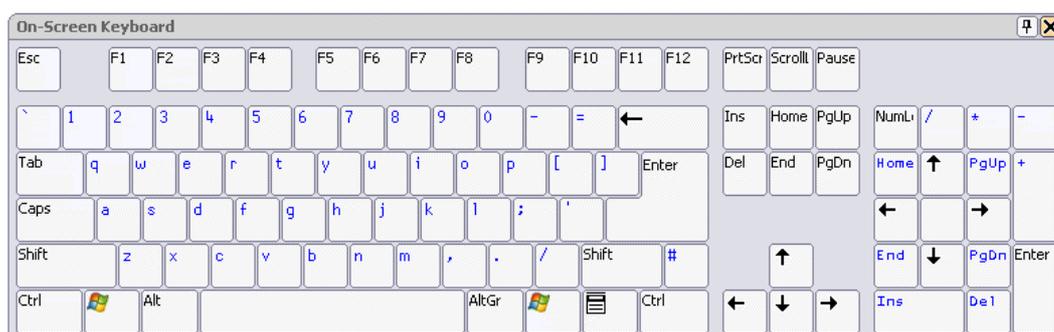
Included with Dyalog APL Version 12.0 is the Comfort On-Screen Keyboard 2.1 which is specially configured for use with Dyalog APL (Unicode Edition) and distributed under a licence agreement with Comfort Software.

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

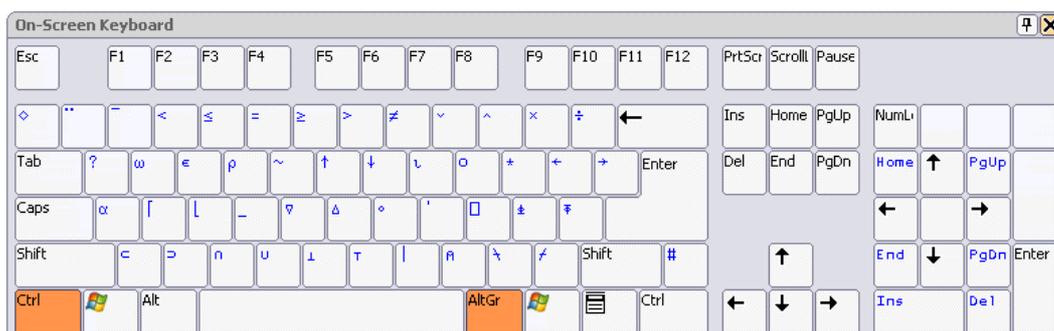
Not only does it support a large number of standard physical keyboards, but it includes a tool to design your own layout corresponding to the actual keyboard attached to your computer.

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

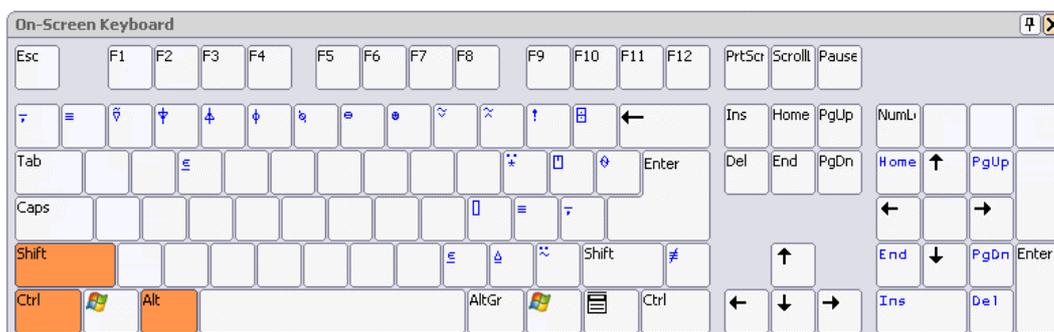
The following pictures illustrate the appearance of a UK - Dyalog AltGr keyboard, in Normal, AltGr and AltGr+Shift modes.



Normal



AltGr Mode

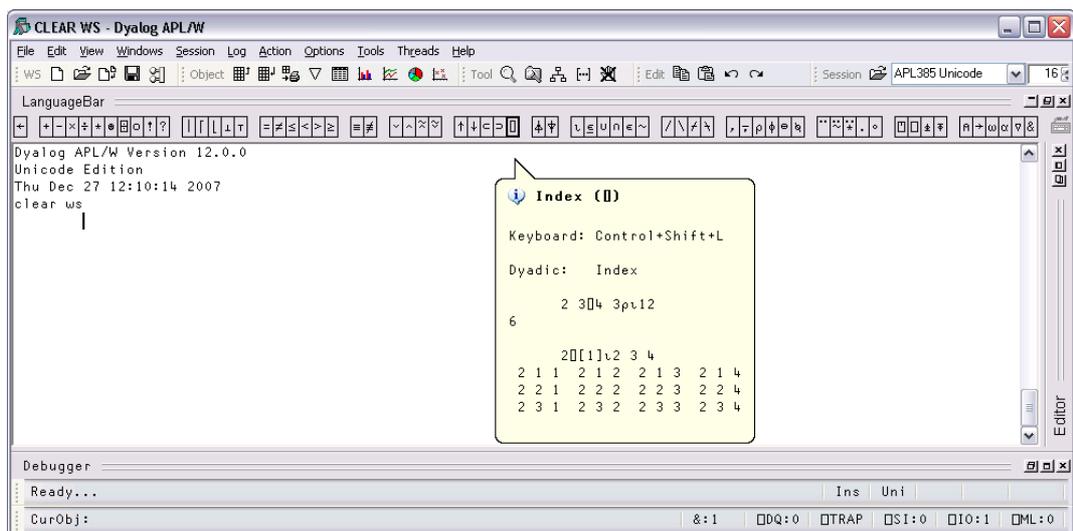


AltGr+Shift

## Language Bar

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

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



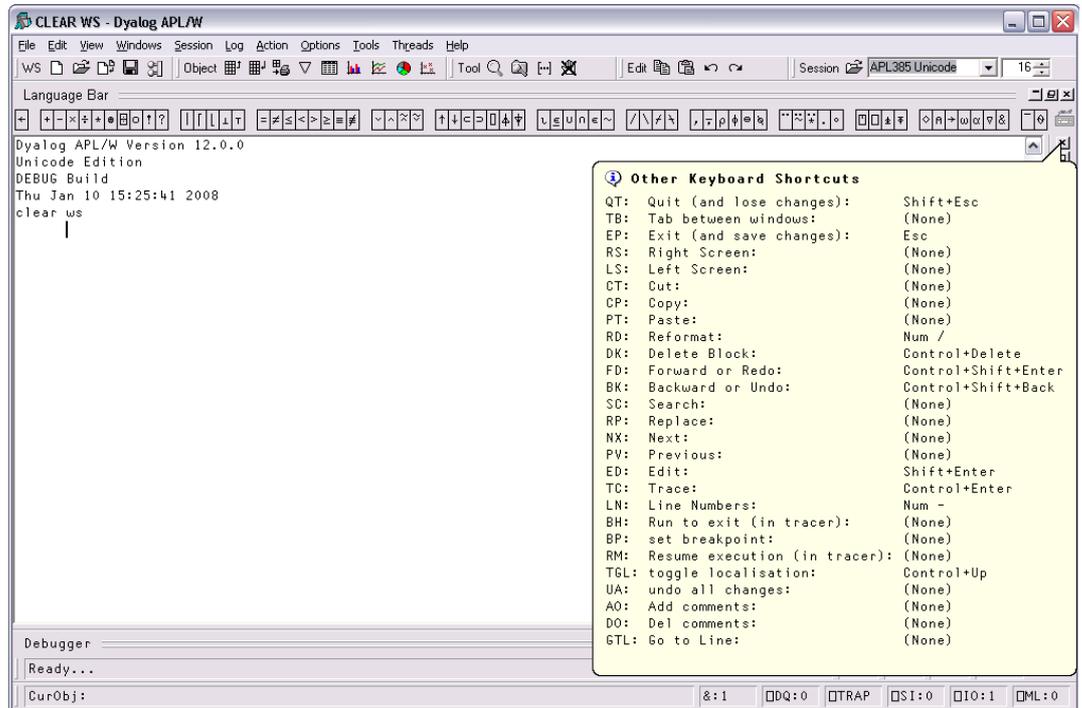
## Keyboard Shortcuts

The Dyalog Development Environment provides a number of shortcut keys that may be used to perform actions. These are identified by 2-character codes; for example the action to start the Tracer is identified by the code <TC>, and mapped to user-configurable keystrokes.

In the Unicode Edition, Keyboard Shortcuts are defined using *Options/Configure/Keyboard Shortcuts* and stored in the Windows Registry.

To the right of the last symbol in the Language Bar is the Keyboard Shortcut icon 

If you hover the mouse over this icon, a pop-up tip is displayed to remind you of your keyboard shortcuts, as illustrated below.



If you click on the Keyboard Shortcut icon , the *Options/Configure/Keyboard Shortcuts* dialog box is displayed. This allows you to change your keyboard shortcut settings.

## New Help System and Documentation Center

The Dyalog APL Version 12.0 help system is packaged as a single *Microsoft HTML Help* compiled help file named help\dyalog.chm.

### Documentation Center

In the same help sub-directory, you will find the complete collection of system documentation in PDF format which may conveniently be accessed via the *Documentation Center* menu item.



### Help Menu

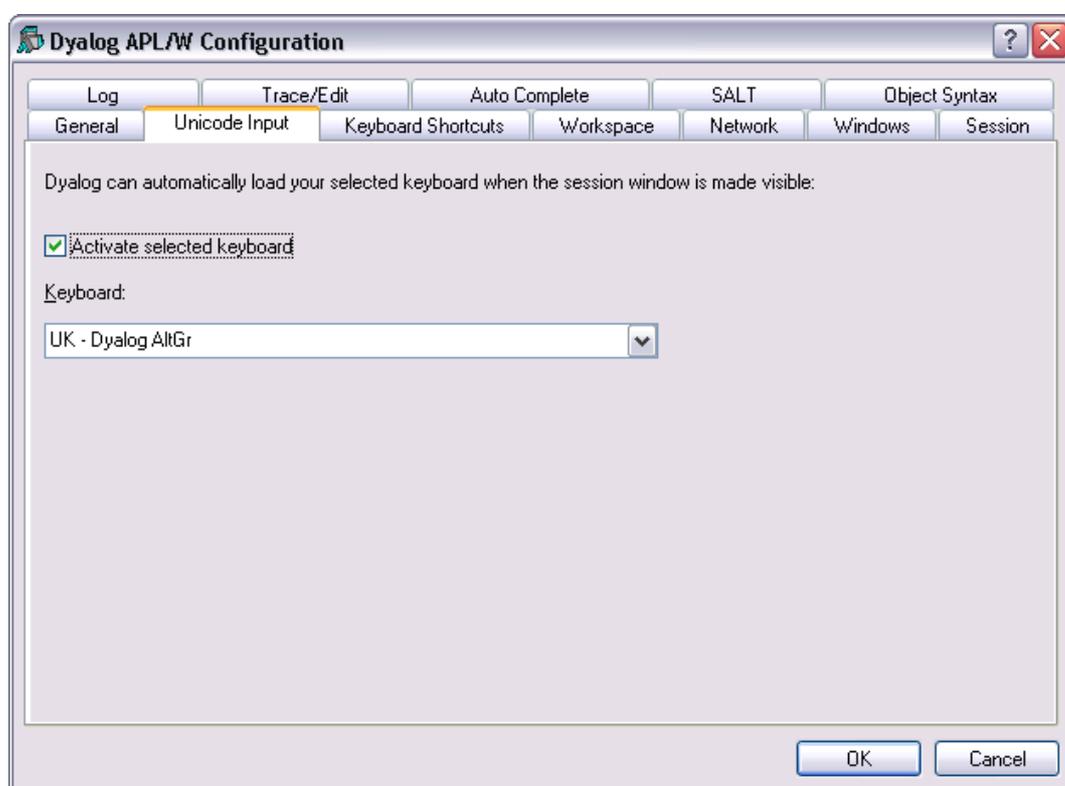
Label	Action	Description
Documentation Center	[DocCenter]	Opens your web browser on help\index.html which displays an index to the on-line PDF documentation and selected internet links.
Latest Enhancements	[Re1Notes]	Opens help\dyalog.chm, starting at the first topic in the Version 12.0 Release Notes section. Note that the Version 11.0 Release Notes are also included.
Language Help	[LangHelp]	Opens help\dyalog.chm, starting at the first topic in the Language Reference.
Gui Help	[GuiHelp]	Opens help\dyalog.chm, starting at the first topic in the Object Reference.
Dyalog Web Site	[DyalogWeb]	Opens your web browser on the Dyalog home page.
Email Dyalog	[DyalogEmail]	Opens your email client and creates a new message to Dyalog Support, with information about the Version of Dyalog APL you are running.
About Dyalog APL	[About]	Displays an <i>About</i> dialog box

## New Configuration Dialogs

### Unicode Input

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

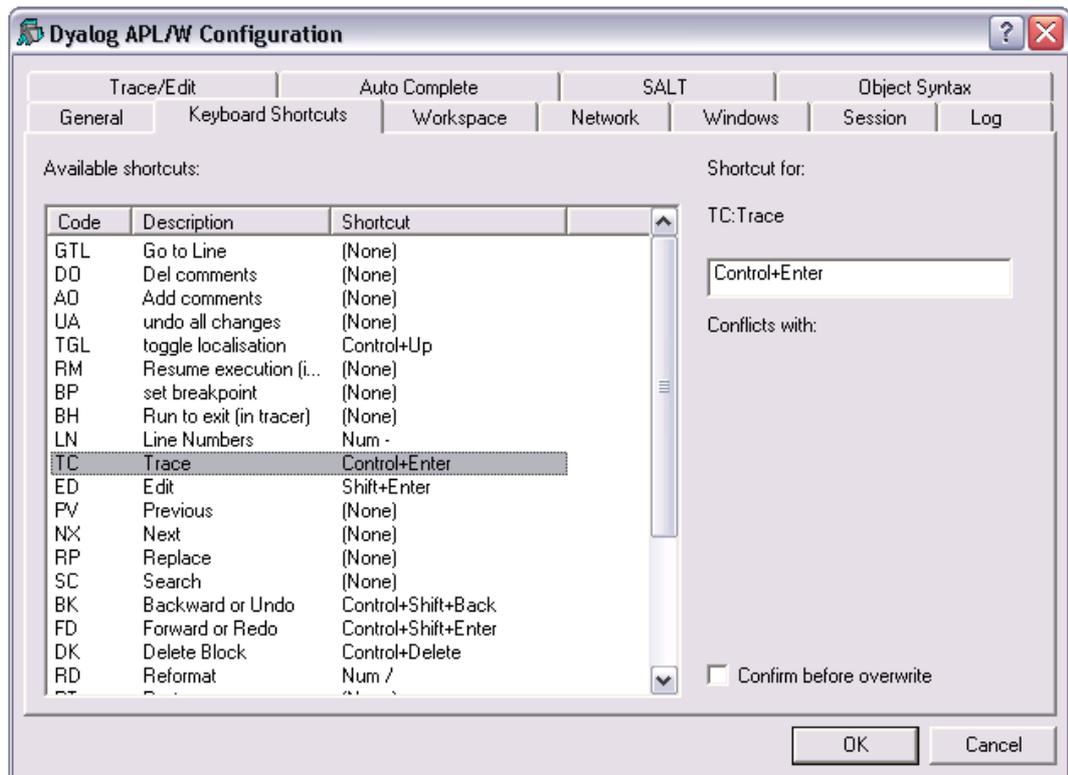
To choose this option, open *Options/Configure/Unicode Input*. In the *Keyboard* dropdown, select one of your installed APL keyboards, enable the *Activate selected keyboard* checkbox, then click *OK*



Label	Parameter	Description
Activate selected keyboard	InitialKeyboardLayoutInUse	1 = automatically select the specified APL keyboard on startup. 0 = no action
Keyboard	InitialKeyboardLayout	the name of the APL keyboard to be selected.

## Keyboard Shortcuts

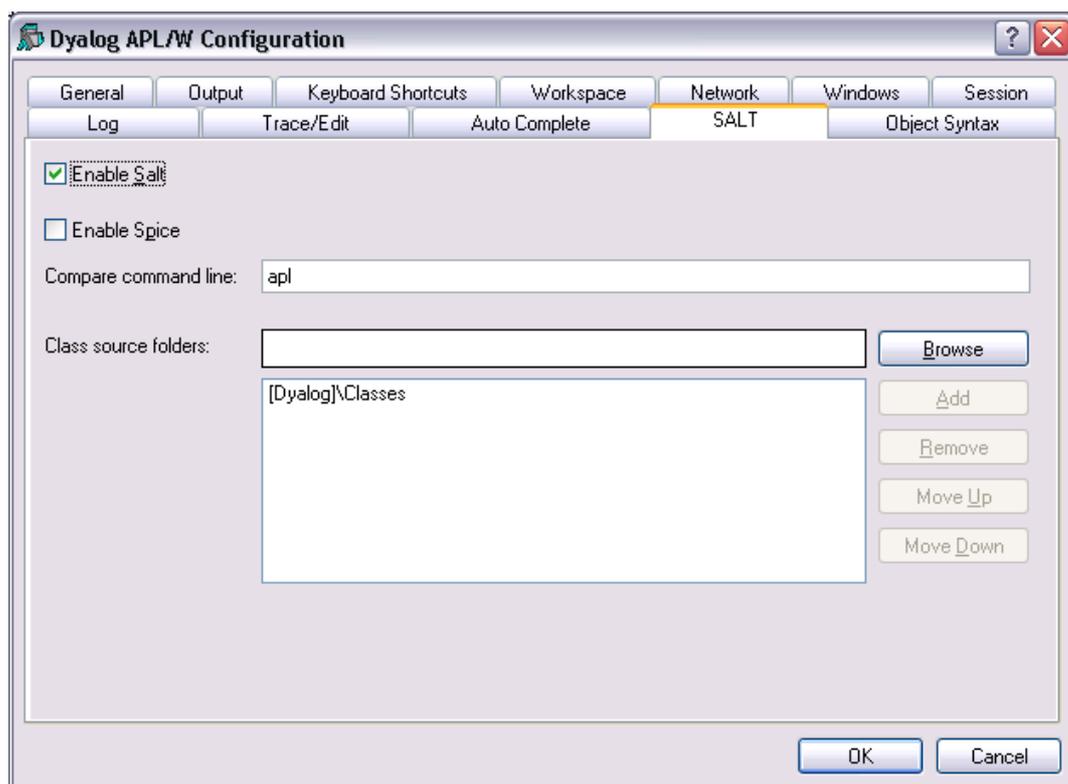
The Dyalog Development Environment provides a number of shortcut keys that may be used to perform actions. These are identified by 2-character codes; for example the action to start the Tracer is identified by the code <TC>, and mapped to user-configurable keystrokes. In the Unicode Edition, Keyboard Shortcuts are defined using *Options/Configure/Keyboard Shortcuts* and stored in the Windows Registry.



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

## SALT

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



Label	Parameter	Description
Enable Salt	AddSALT	Specifies whether or not SALT is enabled
Enable Spice	AddSPICE	Specifies whether or not SPICE is enabled. Note that SPICE cannot be enabled without SALT.
Compare command line	CompareCMD	The command line for a 3 <sup>rd</sup> party file comparison tool to be used to compare two versions of a file. See note.
Editor	Editor	Name of the program to be used to edit script files (default "Notepad").
Class source folders	SourceFolder	Sets the SALT working directory; a list of folders to be searched for source code.

### Configuration dialog: SALT

#### Compare command line

The default value of "apl" for the Compare command line instructs SALT to use built-in APL code when comparing two versions of a file. If you have a 3rd party file comparison tool which can compare UTF-8 files, enter the name of the program here. For example, if you have installed "Compare It!", you would enter: "[ProgramFiles]\Compare It!\wincmp3" (this assumes that the program will take two parameters containing the names of two files to be compared).

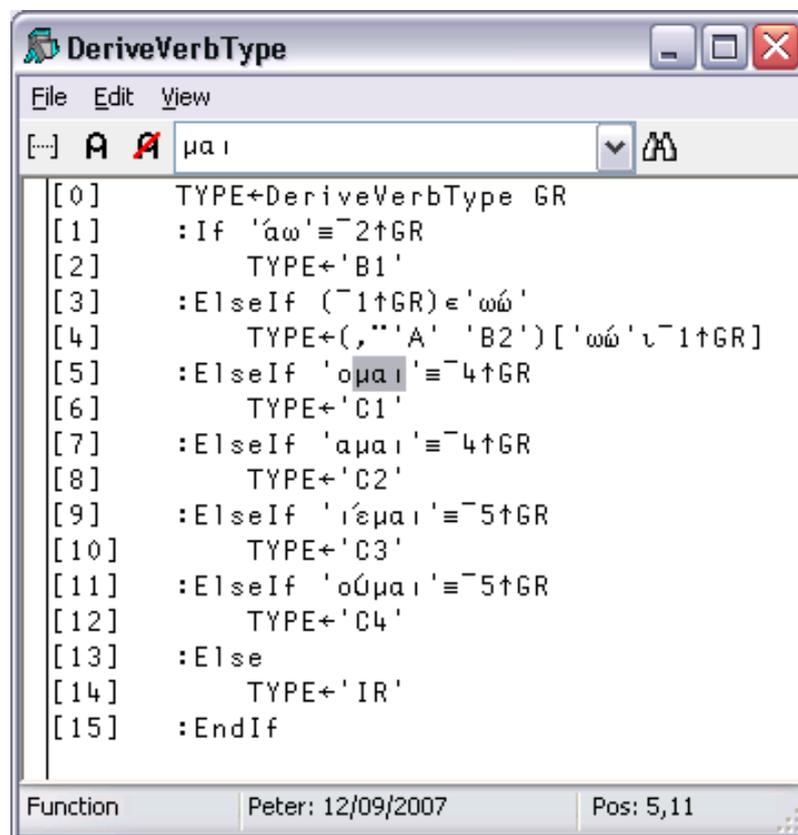
#### More Information

For more information about using SALT and SPICE, see the SALT and SPICE manuals, which can be most easily located via the Documentation Centre.

## Edit Window Tools

The Edit Window now has a toolbar to simplify frequently used operations, namely:

- Toggle line numbers
- Comment selected lines
- Remove comments from selected lines
- Search



The screenshot shows a window titled "DeriveVerbType" with a menu bar (File, Edit, View) and a toolbar containing icons for line numbers, comment, and search. The main area displays code with line numbers [0] through [15]. The code defines a function that sets the TYPE of a verb based on its GR (Greek) form. The code is as follows:

```
[0] TYPE←DeriveVerbType GR
[1] :If 'áω'≡-2↑GR
[2]     TYPE←'B1'
[3] :ElseIf (↑GR)ε'ωó'
[4]     TYPE←(, "'A' 'B2') ['ωó' ↓-1↑GR]
[5] :ElseIf 'ομαι'≡-4↑GR
[6]     TYPE←'C1'
[7] :ElseIf 'αμαι'≡-4↑GR
[8]     TYPE←'C2'
[9] :ElseIf 'ίεμαι'≡-5↑GR
[10]    TYPE←'C3'
[11] :ElseIf 'ούμαι'≡-5↑GR
[12]    TYPE←'C4'
[13] :Else
[14]    TYPE←'IR'
[15] :EndIf
```

At the bottom of the window, the status bar shows "Function", "Peter: 12/09/2007", and "Pos: 5,11".

# SharpPlot Graphics

## Introduction

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

The Version 12.0 Session includes 4 buttons which use SharpPlot to generate simple graphical pictures of the contents of the Current Object (identified by the name under or to the left of the cursor).

For example, if you have a numerical matrix in a variable called **MAT**, you can plot it by first positioning the cursor on the name **MAT** in the Session window, and then clicking one of the 4 graphical buttons in the Session toolbar.

## Data Structures

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

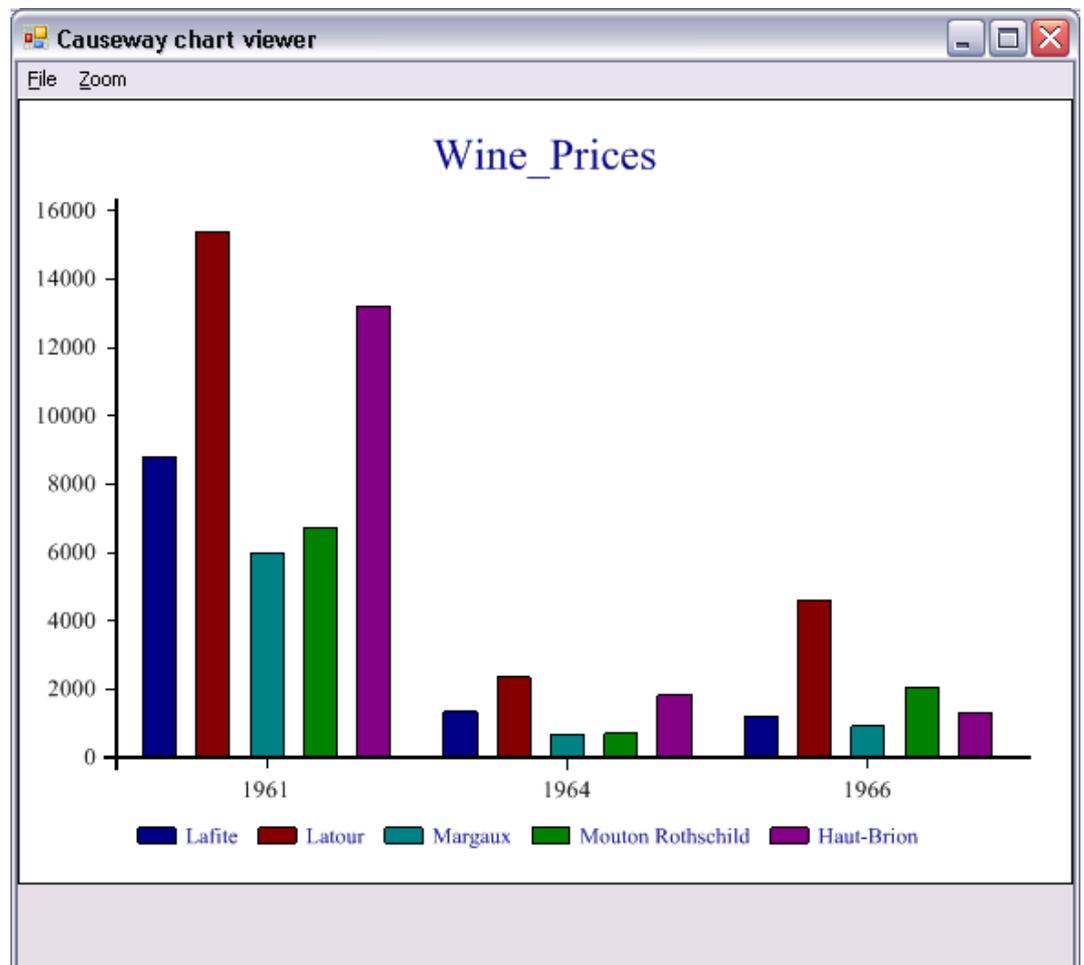
- a simple numeric vector
- a vector of simple numeric vectors
- a simple numeric matrix
- a matrix whose first row contains simple character vectors and whose other elements are simple numerics. In bar and line charts, the column headings in row 1 are used as x-axis labels.
- a matrix whose first column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

## Examples

### Bar Chart

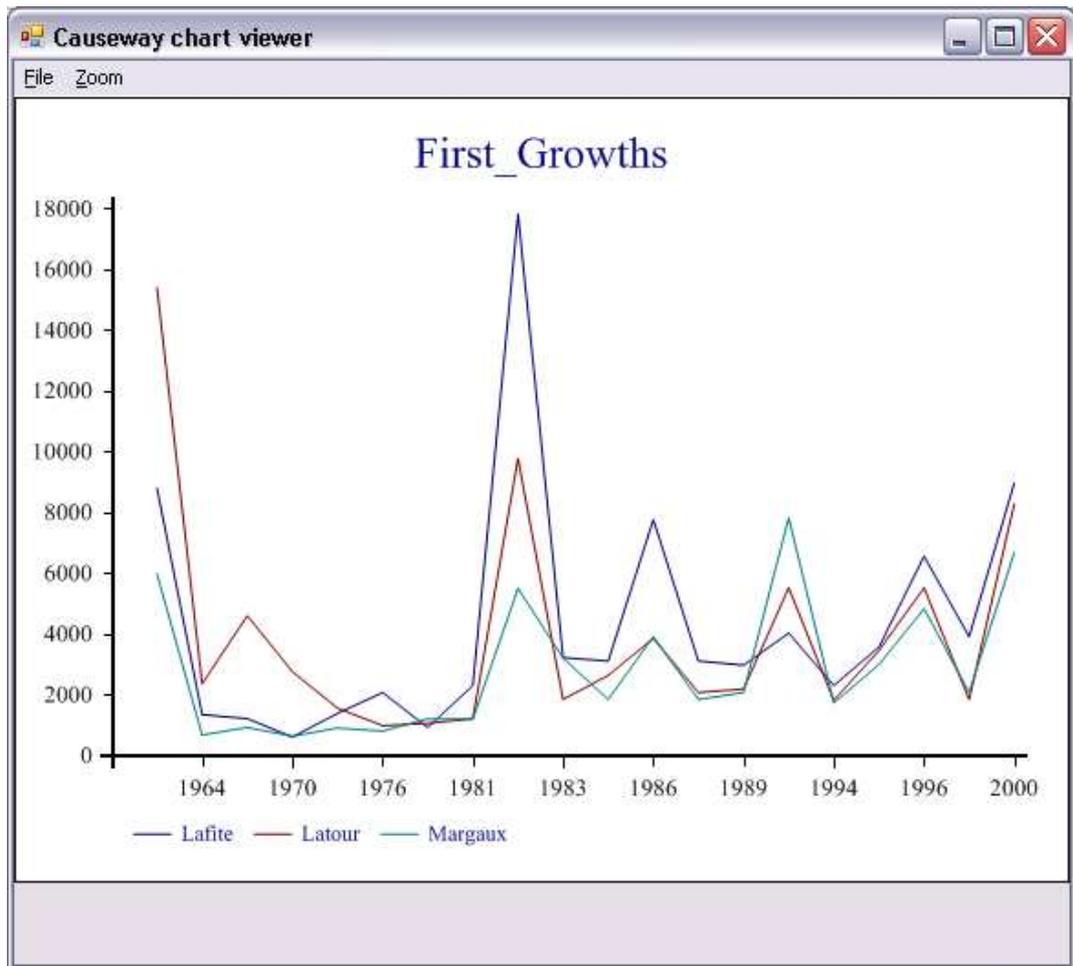
Wine\_Prices

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



Line Chart 

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



## Implementation

The SharpPlot tools are implemented by four buttons in the Session toolbar. Each button has a Select callback which runs the function `SE.Chart.DoChart`. This runs `SE.Chart.Do` which constructs and then runs a function named `SE.Chart.MyChart`.

`SE.Chart.MyChart` uses an instance of the SharpPlot graphics class to produce a chart of your data, which it saves as a temporary file. It then calls the SharpPlot viewer to display the file on your screen.

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

## Notes

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

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

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

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

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



## CHAPTER 5

# Unicode and the Dyalog GUI

The Unicode Edition allows users to enter *any* characters using the Dyalog APL GUI. This capability is reflected by some changes to certain Properties and Events, notably the KeyPress Event. These changes are described in this section.

# KeyPress

## Event 22

**Applies to** ActiveXControl, Animation, Button, Calendar, ColorButton, Combo, ComboEx, DateTimePicker, Edit, Form, Grid, Group, List, ListView, MDIClient, ProgressBar, PropertyPage, RichEdit, Scroll, Spinner, SubForm, TrackBar, TreeView

If enabled, this event is generated when the user presses and releases a key on the keyboard. It is reported for whichever object has the keyboard focus at the time.

The event message reported as the result of `⎕DQ`, or supplied as the right argument to your callback function, is a 6-element vector as follows :

[1] Object:	ref or character vector
[2] Event code:	'KeyPress' or 22
[3] Input Code:	character scalar or vector
[4] Character Code:	integer scalar
[5] Key Number:	integer scalar
[6] Shift State:	integer scalar

If the keystroke resolves to a character, the Input Code is a character scalar. If the keystroke resolves to a command recognised by Dyalog APL, such as UC (Up Cursor) or ER (Enter) the Input Code contains the corresponding 2-element character vector. In the Classic Edition, the resolution of the keystroke to a character (in `⎕AV`) or to a command, is performed using the Input Translate Table. In the Unicode Edition, the resolution is performed by the Operating System. However, if the keystroke resolves to a navigation or control key (such as Cursor Up or Enter), the same 2-character "command" is reported. Note however that commands that are purely internal to Dyalog APL (such as Trace, commonly Ctrl+Enter) are not reported as such and the Input Code will be empty.

In the Unicode Edition, the Character Code is the Unicode code point of the character that the user entered. In the Classic Edition, it is a number in the range 0-255 which specifies the ASCII character that would normally be generated by the keystroke, and is independent of the Input Translate Table. If there is no corresponding ASCII character, the ASCII code reported is 0.

The key number is the physical key number reported by Windows when the key is pressed.

The Shift State indicates which (if any) of the Shift, Ctrl and Alt keys are down at the same time as the key is pressed. It is the sum of the following numbers :

Shift key down	:	1
Ctrl key down	:	2
Alt key down	:	4

Thus a Shift State of 3 indicates that the user has pressed the key in conjunction with both the Shift and Ctrl keys. A Shift State of 0 indicates that the user pressed the key on its own.

### Example

```

    ▾ Key;Form1
[1]   'Form1'␣WC'Form'('Event' 'KeyPress' 'Keycb')
[2]   ␣DQ'Form1'
    ▾
    ▾ Keycb msg
[1]   DISPLAY msg
    ▾

```

On running function `Key`, the following output will be displayed as a result of the user pressing the following 5 keys in succession:

1. "a"
2. Shift+"a"
3. Cursor Up
4. β ("b" using a Greek keyboard)
5. ι (Ctrl+"i" using a UK APL keyboard)

### Unicode Edition

```

.→-----
| .→----- .→----- |
| |Form1| |KeyPress| a 97 65 0 |
|-----|-----|-----|
ε
.→-----
| .→----- .→----- |
| |Form1| |KeyPress| A 65 65 1 |
|-----|-----|-----|
ε
.→-----
| .→----- .→----- .→----- |
| |Form1| |KeyPress| |UC| 0 38 0 |
|-----|-----|-----|
ε

```

```

→-----
|Form1| |KeyPress| β 946 66 0 |
|-----|
ε-----

```

```

→-----
|Form1| |KeyPress| τ 9075 73 2 |
|-----|
ε-----

```

### Classic Edition

```

→-----
|Form1| |KeyPress| a 97 65 0 |
|-----|
ε-----

```

```

→-----
|Form1| |KeyPress| A 65 65 1 |
|-----|
ε-----

```

```

→-----
|Form1| |KeyPress| |UC| 0 38 0 |
|-----|
ε-----

```

```

→-----
|Form1| |KeyPress| | | 223 66 0 |
|-----|
ε-----

```

```

→-----
|Form1| |KeyPress| τ 9 73 2 |
|-----|
ε-----

```

# Index

## □

□AV .....	17, 22
□AVU .....	22, 32, 41, 68, 71
□DR .....	21
□FCOPY .....	2, 36
□FCREATE .....	5, 37
□MAP .....	40
□NA .....	19, 42
□UCS .....	5, 19, 73

## A

address size	
of component file .....	5
APL	
characters .....	31
appending to native file .....	68
atomic vector .....	17, 22, 31
atomic vector - Unicode .....	22, 32
atomic vector index	
idiom .....	5

## C

Causeway .....	3
Classic Edition .....	1, 26, 28, 31, 72, 73
Compatibility .....	9
component files .....	1
compatibility .....	9
Conga .....	4
copying component files .....	2, 36
creating component files .....	37

## D

data representation .....	21
dyadic .....	35
monadic .....	34
Documentation Centre .....	92

## D

dynamic link libraries .....	42
------------------------------	----

## F

file	
copy .....	36
create .....	37
files	
APL component files .....	36, 37
mapped .....	40

## G

grade-down function	
monadic .....	26
grade-up function	
monadic .....	28

## I

Input Method Editor .....	16
Interoperability .....	9

## J

journaling	
component files .....	1

## K

keyboard shortcuts .....	16, 86, 87, 90
keyboards .....	15
KeyPress event .....	100
Kibitzer .....	2

## L

language bar	
Session Window .....	3, 86

## M

mapped files .....	40
matrix iota idiom .....	5
monadic primitive functions	
grade down .....	26
grade up .....	28
MPUT utility .....	40
MSKLC .....	15

**N**

name association .....	19, 42
names .....	17
native file	
append .....	68
read.....	69
replace .....	70
translate .....	72
native files.....	23
NewLeaf.....	3

**O**

On-Screen Keyboard.....	2
-------------------------	---

**P**

performance improvements.....	5
-------------------------------	---

**R**

RainPro .....	3
reading native files .....	69
replacing data in native files .....	70

**S**

SALT .....	4, 91
SharpPlot.....	3, 94
SPICE.....	91

SubVersion .....	4
------------------	---

**T**

terminal control vector .....	73
translating native files .....	72
TRANSLATION ERROR.....	21, 33, 41

**U**

underscored alphabetic characters.....	30
underscores.....	17
Unicode .....	13
Unicode Convert .....	19
Unicode Edition .....	1, 5, 26, 28, 31, 70, 72
Unicode support	
component files .....	2
UTF-16.....	75
UTF-32.....	75
UTF-8.....	75

**V**

viewcmd registry entry.....	97
-----------------------------	----

**W**

wide character .....	49
----------------------	----



---

**DYALOG** <sup>APL</sup>

**Dyalog Ltd**  
South Barn  
Minchens Court  
Minchens Lane  
Bramley  
Hampshire  
RG26 5BH  
United Kingdom  
[www.dyalog.com](http://www.dyalog.com)