The tool of thought for expert programming

Dyalog User Commands Reference Guide

User Commands Version 2.00

Dyalog Limited

Minchens Court, Minchens Lane
Bramley, Hampshire
RG26 5BH
United Kingdom

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

Contents

1 About This Document

This document is intended as an introduction to user commands, a guide to creating and implementing new user commands and a summary of the predefined user commands supplied with Dyalog.

Although the behaviour of user commands is generally independent of the operating system and whether a classic/Unicode installation is used, some of the information in this document is operating system-specific (for example, the ability to auto-complete the names of user commands when running them). The differences between this document and the user commands experience on a UNIX operating system are detailed in the *Dyalog for UNIX User Guide*.

1.1 Audience

It is assumed that the reader has a reasonable understanding of Dyalog.

2 Introduction

User commands are tools that are available at any time, in any workspace, as extensions to the Dyalog development environment. The text-based implementation of user commands allows development tools to be easily shared between users, and the ability to create custom user commands in addition to the predefined user commands that are supplied with Dyalog means that it is simple to write utility tools for your environment that can be easily issued to an entire development team.

Custom user commands produced using this version of the user command framework (version 2.00) are fully compatible with Dyalog version 13.1 and later; the exception to this is user commands that invoke language features or functionality not supported in the release in which they are run. For compatibility with earlier versions of Dyalog, please contact support@dyalog.com.

User commands are entered in an APL Session by starting an input line with a `]` character, for example:

```
      ]ToHex 250+⍳5
 FB  FC  FD  FE  FF
```

A section of the APL Wiki is devoted to sharing custom user commands (see http://aplwiki.com/UserCmdsDyalog).

2.1 Cache File

The first time that you start a Dyalog Session after installing/updating Dyalog, a cache file is created comprising the name of each of the user commands and the file in which it is defined. This can take a few seconds. If any of the files that contain user commands are altered, then the cache file is rebuilt:

- the next time a Dyalog Session is started.
- when the `]Ureset` user command is run (forces an in-Session recache).

By default, the cache file is located in:
Documents\Dyalog APL <version> Files\UserCommand20.cache

By default, the cache file is located in:
~/.dyalog/UserCommand20.cache
(it is likely that this location will change in a future version of Dyalog).

The name and location of the cache file can be changed from its default by setting the UCMDCACHEFILE registry entry/environment variable.

3 Using User Commands

This chapter introduces some of the concepts that underpin user commands in Dyalog.

3.1 Installation

A set of predefined user commands is installed automatically with Dyalog.

For a summary of these user commands, see Chapter *5*.

3.2 File Structure

The **<path to Dyalog>\SALT\spice** directory contains the predefined user commands that are installed with Dyalog.

The **spice** directory can only be moved to a different location by moving its parent **SALT** directory and setting the SALT environment variable accordingly. For information on moving the **SALT** directory and setting the environment variable, see the *Dyalog SALT Reference Guide*.

Although the **spice** directory can be moved, it must always remain directly beneath the **SALT** directory and must not be renamed.

3.3 Implementation

When an input line in a Session starts with a `]` character, Dyalog makes a call to the dyadic function `⎕SE.UCMD` – if this function exists, then it is called with the rest of the input line as the right argument and a reference to calling space as the left argument.

For example:

The following command is entered in the namespace `#.ABC`:

```
      ]<cmd> -myModifier=value
```

Dyalog's interpreter preserves this exactly and makes the following call:

```
#.ABC ⎕SE.UCMD '<cmd> -myModifier=value'
```

`⎕SE.UCMD` converts this into a call to the user command framework:

```
⎕SE.SALTUtils.Spice '<cmd> -myModifier=value'
```

The functions defined for `<cmd>` are actioned with the `-myModifier` modifier applied with a value of `value` and the result is displayed in the Session.

This implementation means that application code can invoke user commands by calling `⎕SE.UCMD` directly; if this function is deleted then user commands are disabled completely.

Dyalog Ltd reserves the right to change the implementation of the user command framework; for this reason, the user command framework should never be called directly instead of through `⎕SE.UCMD`.

3.3.1 Customising the Implementation

Although it is possible to implement a custom user command system by redefining `⎕SE.UCMD`, Dyalog Ltd does not recommend this approach – adhering to the user command framework supplied with Dyalog promotes a single, consistent, format that enables all custom user commands to be shared between Dyalog Sessions.

3.4 File Format

Each user command comprises a script containing a single namespace object (for more information on scripted files, including declaration statements and permitted constructs, see the *Dyalog Programmer's Reference Guide*) and must be stored as files with the **.dyalog** extension.

If an extension is not specified when using user commands to save a script file, then **.dyalog** is automatically appended.

By default, double-clicking on a file with the **.dyalog** extension opens it using a text editor (in Microsoft Windows this is the Microsoft Windows Notepad program).

Files with the **.dyalog** extension are Unicode text files that use UTF-8 character encoding. This means that they can store any text that uses Unicode characters. This format includes most of the world's languages and the Dyalog character set, and is supported by many software applications. By using text files as a storage mechanism, user commands and other tools written using Dyalog can be combined with industry-standard tools for source code management.

3.5 Groups

User commands with common features can be grouped together under a single name. These groups have no effect on the functionality of the individual user commands but enable related user commands to be gathered together for ease of reference and provide a means of sorting and classifying user commands that can be very useful as the number of user commands increases.

User command names must be unique within a group but do not have to be unique across all groups. This means that groups allow a systematic naming convention for user commands that perform similar functions on different types of APL object, for example, `]FILE.Compare` to compare two files, `]ARRAY.Compare` to compare two arrays and `]FN.Compare` to compare two functions.

When running (or asking for help on) a user command, the group name can be prefixed to the user command name, separated by a `.` character; this group name prefix is mandatory if the user command name is not unique across all groups.

Every user command must be in a group, and every group must comprise at least one user command.

3.6 Syntax in Dyalog Sessions

User commands are entered in a Dyalog Session with a preceding right bracket. The basic syntax is as follows:

- to run a user command: `] <cmd>...`
- to list all user commands: `]?`
- to list all user commands (with descriptions) in their groups: `]?+`
- to list all the available commands defined in **.dyalog** files in a directory: `]? <full path to directory>/<directory name>`
- to list all user commands that start with "<string>": `]?<string>*`
- to list all the available commands in a specific group: `]?<groupname>`
- to assign the result of a command to a variable: `]<var>←<cmd>...`

The names of user commands and groups are not case-sensitive although their arguments, modifiers and modifier values might be. The convention used in this document is that group names are shown in upper case and user command names are shown in camel case.

3.6.1 Requesting Additional Information

Help can be requested in an APL Session using the following syntax:

- for general help on user commands: `]??` or `]Help`
- for help on a specific user command: `]?<cmd>` or `]Help <cmd>`
- for more detailed help on a specific user command: `]??<cmd>`

For a specific user command, the information that is returned is dependent on the level of help requested. This is determined by the number of `?` characters entered between the `]` character and the user command name; for example, `]??<cmd>` returns the information defined for level 2 of the `<cmd>` user command. The number of levels of help available depends on a user command's definition (for information on defining multiple levels of help in custom user commands, see Section *4.4.1*).

When requesting help on a user command, the name of that user command does not always need to be entered in full – as long as enough of the name is entered for it to be interpreted unambiguously. For example, if a user command is called `Time` and no other user commands start with the letter `T` then help can be successfully requested by calling `]?T`, `]?Ti`, `]?Tim` or `]?Time`.

3.7 Running User Commands

User commands are run with the following syntax:

```
] <cmd> <-modifiers/arguments>
```

For information on the precise syntax for each user command, the arguments that can be supplied to it and the modifiers that it can take, enter `]Help <cmd>` or `]?<cmd>` in a Dyalog Session.

When running a user command, the name of that command must be entered in full.

Dyalog's auto-complete functionality means that any user commands that match the entered string are presented as selectable options, making it easy to correctly specify the requisite user command.

The names of user commands are not case-sensitive although their arguments, modifiers and modifier values might be.

3.7.1 Arguments

Some user commands can accept (or require) one or more arguments. To see a list of the possible arguments for a user command, enter `]?<cmd>` or `]Help <cmd>` in a Dyalog Session.

For example, the behaviour of the user command `]CD` depends on the argument supplied when calling it. If it is run with no argument, then it returns the current working directory – this is equivalent to entering `cd` on the command line of a Microsoft Windows operating system or `pwd` in UNIX. However, if a single argument specifying the full path to a directory is supplied, then the user command changes the current working directory to be the one specified by the argument.

3.7.2 Modifiers and Modifier Values

The default behaviour of a user command can be altered through the application of *modifiers* (instructions that the command should change its default behaviour). To see a list of the possible modifiers and their modifier values for a user command, enter `]?<cmd>` or `]Help <cmd>` in a Dyalog Session.

Modifiers must be prefixed with the `-` character and are separated from any associated modifier values with the `=` character, for example `-version=3` or `-format=APL`. A modifier that does not accept a modifier value but can only be present or absent is sometimes referred to as a *flag* or a *switch*, for example, `-protect`.

When running a user command with a specified modifier, the name of the modifier does not always need to be entered in full – as long as enough of the modifier's name is entered for it to be interpreted unambiguously. For example, if a user command has a modifier called `-version` and does not have any other modifiers starting with the letter `v` then the function can be successfully called with modifiers `-version`, `-vers`, `-v` and so on.

Multiple modifiers can be included in a user command call – in this situation they must be separated by a space character. The order in which they are specified is irrelevant.

4 Creating User Commands

When an instruction is called repeatedly it can improve efficiency to have that instruction in a script file. The user command framework provides a very efficient mechanism for doing this, allowing a user to create and update instructions without the necessity of maintaining a workspace. Unlike a workspace, user commands do not need to be loaded into each Session that wants to employ them. In addition, their text-based implementation makes them easy to store in a repository and share between users.

This chapter describes the syntax, rules and conventions governing the creation of custom user commands.

4.1 Basic Definition

A new user command can be defined in one of the following ways:

- in a text file (for example, using Microsoft Notepad) and then saved as a **.dyalog** file
- in a Dyalog Session and saved as a **.dyalog** file using the `]Save` user command.

Once in the appropriate directory (see Section *4.7*), the new user command can be run from the Dyalog Session.

User commands are defined by three specific APL functions (along with any additional functions needed for the particular purpose of the user command). The three functions must be called:

- `List` – for information on the `List` function, see Section *4.2*.
- `Run` – for information on the `Run` function, see Section *4.3*.
- `Help` – for information on the `Help` function, see Section *4.4*.

These functions are wrapped together in a namespace (the order in which the functions are specified within the namespace is not important). A single namespace can host multiple user commands.

Examples of user commands wrapped in a namespace are included in Appendix *A* – these show how the `List`, `Help` and `Run` functions are defined.

4.2 The List Function

The `List` function informs the user command framework about the command being defined, enabling it to display a summary of the command when requested to list all available commands (with descriptions) in their groups (`]?+`).

The `List` function returns one namespace for each user command defined within it. Each namespace contains four variables:

- Desc – a summary of the user command's functionality
- Name – the name of the user command (see Section *4.2.1*)
- Group – the name of the group to which the command belongs (see Section *4.2.2*)
- Parse – parsing information for the framework (see Section *4.2.3*)

4.2.1 Name

User commands must have unique names within a group (names can be replicated across different groups if required). They must be valid APL identifier names (for more information on legal names, see the *Dyalog Programmer's Reference Guide*)

Modifiers must have unique names within the user command but do not have to be unique within the superset of user commands. Modifier names are case-sensitive.

The names of user commands and modifiers cannot contain space characters.

When naming a modifier, avoid the names Arguments, Delim, Propagate, SwD and Switch as these names are used by the parser.

4.2.2 Group

Every user command must be a member of a group (but can only be a member of one group). In addition:

- the user commands for a single group do not all need to be defined within a single namespace/**.dyalog** file
- a single namespace/**.dyalog** file can include user commands for several different groups
- user command names must be unique within a group but do not have to be unique across all groups

Although it is possible to add a custom user command to one of the predefined user command groups, Dyalog Ltd recommends that this is avoided as there could be unforeseen consequences (especially with the SALT and UCMD groups).

4.2.3 Parse

If the Parse variable for a user command is empty, then the `Run` function's second argument will comprise everything following the command name. By setting the Parse variable to non-empty values, the user command framework is able to handle arguments and modifiers.

For more information on modifiers and modifier values, see Section *4.5*. For more information on arguments, see Section *4.6*.

The following general rules apply when processing a call to a user command:

- user commands take 0 or more arguments followed by 0 or more modifiers (the arguments must come before the modifiers)
- individual arguments and modifiers are separated by space characters

- modifiers can be optional or mandatory
- modifiers are identified by a preceding `-` character
- modifier values are identified by a preceding `=` character
- modifiers can be specified in any order
- modifier names are case-sensitive
- individual arguments and modifiers can be delimited by single or double quotes to allow space characters within them.

The user command framework verifies that these rules have been adhered to before creating a new namespace. It then populates this namespace with a variable called Arguments (containing all the arguments) and a variable for each of the modifiers with names matching those of the modifiers. Other tools for manipulating the user command are also added to the namespace, for example, the `Switch` function – see Section *4.5.1*. This namespace is passed to the `Run` function (see Section *4.3*) as its second argument.

If the `Parse` variable defined in a user command's `List` function is empty, then the user command will accept anything; the entire string is the argument.

If the `Parse` variable defined in a user command's `List` function is not empty, then it must describe the number of arguments and the modifiers used. The number of arguments is a simple number and the modifier list must include the delimiter, the modifier name and whether it accepts a value.

4.3 The Run Function

The `Run` function executes the code for the command. It is always called with two arguments; the user command's name and the supplied arguments/modifiers. As a single namespace can host multiple user commands, the `Run` function uses the command name to determine the appropriate actions to perform.

4.4 The Help Function

The `Help` function reports detailed information on the user command when this is requested (by entering `]?<cmd>` or `]Help <cmd>` in a Dyalog Session). As a single namespace can host multiple user commands, the `Help` function uses the command name to determine the appropriate information to return.

When a user requests help for a particular user command, the `Help` function returns a specific set of information by default:

`Command “<commandname>”`
`Syntax: accepts switches <modifiers>` only if modifiers are defined
`<commandname> (no arguments)` only if no arguments are defined
`<commandname> <arguments>` only if arguments are defined
`<specific defined help information>`
`Script location: <location>`

The only part of this that is not auto-populated is the specific defined help information (see Section *4.4.1*).

4.4.1 Defining Multiple Levels of Help

The specific defined help information that is presented to a user when requesting help in an APL Session is dependent on the level of help requested. The level is determined by the number of `?` characters that a user enters between the `]` character and the command name; for example, `]??<cmd>` returns the information defined for level 2 of the `<cmd>` user command.

As with the predefined user commands, increasingly detailed levels of information can be provided for custom user commands. If multiple levels of help are defined, then Dyalog Ltd recommends including information to that effect in each level, for example, the information that is displayed in response to a `]??<cmd>` request should state that more detailed information is available if `]???<cmd>` is entered.

Any valid Dyalog algorithmic syntax can be used in the `Help` function to define different levels of help, for example, control structures or branching. Optionally, the different levels of help can be cumulative so that, for example, `]???<cmd>` returns the help information for levels 1 and 2 as well as the help for level 3.

The following code fragments are examples showing how separate (non-cumulative) levels of help can be defined within the Help function:

```
∇ r←level Help Cmd; CR
    CR←⎕UCS 10
    r←'This is basic help.'
    :If level=1
        r,←CR,'This is level 1 help.'
    :ElseIf level=2
        r,←CR,'This is level 2 help.'
    :ElseIf level>2
        r,←CR,'This is level 3 help.'
    :EndIf
∇
```

Alternatively, the same can be achieved with:

```
∇ r←level Help Cmd; CR
    CR←⎕UCS 10
    r←'This is basic help.'
    r,←⊂'This is level 1 help.'
    r,←⊂'This is level 2 help.'
    r,←⊂'This is level 3 help.'
    r←⊃((¯1+⍴r)⌊level)↓r
∇
```

In these cases:

- `]? <cmd>` gives `This is basic help.`
- `]?? <cmd>` gives `This is level 1 help.`
- `]??? <cmd>` gives `This is level 2 help.`
- `]???? <cmd>` gives `This is level 3 help.`
- `]????? <cmd>` gives `This is level 3 help.`

The following code fragments are examples showing how cumulative levels of help can be defined within the `Help` function:

```
∇ r←level Help Cmd; CR
    CR←⎕UCS 10
    r←'This is basic help.'
    :If level>0
        r,←CR,'This is level 1 help.'
    :AndIf level>1
        r,←CR,'This is level 2 help.'
    :AndIf level>2
        r,←CR,'This is level 3 help.'
    :EndIf
∇
```

Alternatively, the same can be achieved with:

```
∇ r←level Help Cmd; CR
    CR←⎕UCS 10
    r←'This is basic help.'
    r,←⊂CR,'This is level 1 help.'
    r,←⊂CR,'This is level 2 help.'
    r,←⊂CR,'This is level 3 help.'
    r←(⍴,CR)↓ ⊃,/(1+level⌊⍴r)↑r
∇
```

In these cases:

- `]? <cmd>` gives `This is basic help.`
- `]?? <cmd>` gives `This is basic help. This is level 1 help.`
- `]??? <cmd>` gives `This is basic help. This is level 1 help. This is level 2 help.`
- `]???? <cmd>` gives `This is basic help. This is level 1 help. This is level 2 help. This is level 3 help.`

If only a single level of help is required, then the `Help` function should be defined without a left argument, that is, `r←Help Cmd`.

Entering `]Help <cmd>` in an APL Session always presents the user with the same level of help as `]? <cmd>` even if there are multiple levels of help defined.

4.5 Modifiers

Modifiers enable a user command to apply filters and rules so that an entirely new (similar) user command does not need to be written. The user command framework allows you to define the modifiers that your user command will accept. The rules when defining each modifier in the `Parse` variable are:

- If a modifier accepts characters in a set, then the `Parse` variable includes the modifier and possible values with the `∊` character as a separator. For example:
  `-<modifier name>∊<set of characters>`
  so `-XYZ∊abc012` means that the modifier `-XYZ` can accept any number and combination of characters in the set `abc012`, such as `ab2a0b`.

- If a modifier accepts specific strings, then the `Parse` variable includes the modifier and possible values with the `=` character as a separator and the strings separated by space characters. For example:
  `-<modifier name>=<string1> <string2> <string3>`
  so `-XYZ=abc 012` means that the modifier `-XYZ` can accept either `abc` or `012` as a modifier value.
- If a modifier accepts any string, then the `Parse` variable includes the modifier a `=` character with nothing after it. For example:
  `-<modifier name>=`
  so `-XYZ=` means that the modifier `-XYZ` can accept any value.

4.5.1 Default Modifier Values

A modifier always has an internal value. This is one of the following:

- 0 if the modifier is not included when running the user command
- 1 if the modifier is included when running the user command but no modifier value is included
- a string matching the specified modifier value

A modifier can be configured to default to a specific value in one of three ways; these approaches are shown in this section with the modifier `-X` defaulting to a modifier value of `123` (a three-element character vector).

Approach 1: Assign a default value to the modifier using the ":" character as the separator:

```
List[i].Parse←'-X:123'
```

Approach 2: Test whether the modifier value is 0 and, if it is, then set it to the required default value.

For example:

```
:if  X≡0  ⋄  X←123  ⋄  :endif
```

Approach 3: Define the default value using the dyadic form of the function `Switch` function (automatically defined in the namespace that is passed to the `Run` function (see Section *4.3*) as its second argument).

The default modifier value must be numeric when using this approach.

Given the name of a modifier as an argument:

- monadic `Switch` returns:
  - 0 if an invalid modifier name is specified
  - 0 if the modifier is not specified and no default value has been set for that modifier
  - 1 if the modifier is specified without a modifier value
  - a string matching the specified modifier value
  - a string matching the default modifier value if a modifier is not specified. and a default value has been set for that modifier

- dyadic `Switch` returns:
    - the value of the left argument (default value) if an invalid modifier value is specified
    - the value of the left argument (default value) if a modifier is not specified (irrespective of whether that modifier is mandatory) and no default value has been set for that modifier
    - the specified modifier value if defined – however, if the value of the default is numeric then it assumes that the specified modifier value should also be numeric and transforms it into a number. This means that, if the modifier and modifier value `-X=123` is entered, the expression `99 Args.Switch 'X'` will return `(,123)` not `'123'`; the `Switch` function always returns a vector, making it very easy to differentiate between `0` (the modifier is not included when running the user command) and `,0` (a modifier value of 0 was specified when running the user command).

4.6 Arguments

Unlike modifiers, arguments do not have names. However, as arguments must be specified in a particular order and each have a specific purpose, they should be given an appropriate name in the `Help` function to make their purpose clear.

The number of arguments that a user command can take is specified in the `Parse` variable (see Section *4.2.3*); this section explains the rules for determining the value to specify there.

4.6.1 Default Argument Values

A default value can be defined for an argument – this value is automatically used if the argument is not specified when running the user command. Default values are defined within the `Run` function. For example:

```
args←a.Arguments,(⍴a.Arguments)↓0 0 0 ‘defaultfor4th’
A4←'defaultvalue' a.Switch '_4'
```

4.6.2 Arguments Including Space Characters

Arguments that contain space characters must be delimited with `'` characters. For example, if the user command `]NewID` must have 2 arguments supplied, *full name* and *address*, then `Parse` should be set to `'2'` and the user command must be run as follows:

```
      ]NewID 'Morten Kromberg' 'Dyalog Ltd'
```

If the user command `]NewID` accepts 3 arguments, *name*, *surname* and *address*, then `Parse` should be set to `'3'` and the user command must be run as follows:

```
      ]NewID Morten Kromberg 'Dyalog Ltd'
```

4.6.3 Minimum Number of Arguments

If a user command must have a minimum number of arguments, then `Parse` can be coded to that effect by assigning it a range of numbers of arguments, that is: `Parse←'<min number of args>-<max number of args>'`.

A minimum number of arguments cannot be specified without also specifying a maximum number of arguments. However, if there is no maximum number of arguments then an arbitrary high number can be used. For example, if at least three arguments must be supplied when calling a user command but there is no limit to the number of arguments that the user command can process, then `Parse` could be assigned as: `Parse←'3-9999'`.

4.6.4 Maximum Number of Arguments

If a user command can only process a limited number of arguments, then `Parse` can be coded to that effect by appending `S` to the maximum number of arguments. For example, if the user command can accept 0, 1 or 2 arguments but no more, then `Parse` should be set to `'2S'`.

4.6.5 Long Arguments

The last argument can be defined to comprise anything that remains after removing the other arguments. `Parse` can be coded to that effect by appending `L` to the maximum number of arguments. For example, if the user command can accept 1 argument consisting of everything that is included when running the command, then `Parse` should be set to `'1L'`. Any additional arguments are merged into the last argument (separated by a space character). If there are multiple space characters anywhere in the text, they are converted into single spaces.

The long argument `L` can be appended to the maximum number of arguments `S` to specify that any additional arguments after the maximum number has been supplied should be merged into the last one supplied. For example, if `'3SL'` is specified, then 0, 1, 2 or 3 arguments can be supplied when calling the user command but any more than this will be merged with the third argument. This means that:

```
]xyz a1 a2 a3 a4 a5 a6
```

runs the user command `xyz` with three arguments: `a1`, `a2` and `'a3 a4 a5 a6'`.

4.6.6 Summary of Argument Specification in the Parser

`Parse←'n'` where `n` can be:

- n_1 : exactly n_1 arguments must be supplied
- n_2-n_3 : a minimum of n_2 arguments and a maximum of n_3 arguments can be supplied
- n_4S : a maximum of n_4 arguments can be supplied (equivalent to $0-n_2$)
- n_5L : n_5 arguments must be supplied; if more than this are supplied then the first n_5-1 arguments are taken and the rest are merged together into the final n_5 argument
- n_6-n_7L : a minimum of n_6 arguments and a maximum of n_7 arguments can be supplied; if more than this are supplied then the first n_7-1 arguments are taken and the rest are merged together into the final n_7 argument
- n_8SL : a maximum of n_8 arguments can be supplied; if more than this are supplied then the first n_8-1 arguments are taken and the rest are merged together into the final n_8 argument (equivalent to $0-n_8$L)

4.7 Saving Custom User Commands

Custom user commands must be saved in a **.dyalog** file (if a custom user command has been created in a scripted namespace in an APL Session, then it can be saved as a **.dyalog** file using the `]Save` user command).

The predefined user commands are located in the **<path to Dyalog>\SALT\spice** directory. Dyalog Ltd recommends that you save custom user commands in a different directory that is not located beneath the **SALT** directory; this is because there might be permissions issues with accessing custom commands beneath this directory and there is always the possibility that Dyalog Ltd might issue a user command with the same filename as your custom user command at a future date.

The custom user command directory must be added to the user command search path to enable the user commands within it to be detected. To do this, run the `]Settings` user command (see Section *5.8.8*) with the `cmddir` global parameter set to the full path and name of the directory.

When adding a new directory to the list of directories searched by the user command framework, you must precede its path with a `,` character.

4.8 Detecting New Custom User Commands

If the `newcmd` global parameter is set to *auto* and a user command is entered in a Dyalog Session that the user command framework does not recognise, then the Dyalog interpreter scans the user command folder(s) to see whether user new commands have been added.

However, if the `newcmd` global parameter is set to *manual* or a change is made to the `Help` function or `List` function of an existing user command, then the user command `]UReset` must be run to force a complete reload of all user commands.

5 Predefined User Commands

Related user commands with common features can be grouped under a single name (see Section *3.5*). This chapter introduces the predefined groups (as summarised in Table *1*) and their constituent user commands.

Table 1. User Command Groups

Group	Description
ARRAY	User commands that relate to arrays or variables.
FILE	User commands that relate to files.
FN	User commands that relate to functions and operators.
MISC	User commands that do not obviously fit into any other category.
NS	User commands that relate to namespaces.
OUTPUT	User commands that change the way in which arrays are displayed in a Session.
PERFORMANCE	User commands that collect and analyse CPU consumption data.
SALT	User commands that perform the same actions as the SALT functions of the same name found in `⎕SE.SALT`.
SAMPLES	User commands that demonstrate the use of multiple levels of help and parsing user command lines.
SVN	User commands that cover svn's (the official command-line client of Subversion) task-specific subcommands of the same name.
TOOLS	User commands that can assist developers by retrieving and presenting information without changing the underlying code.
TRANSFER	User commands that convert workspaces between files written using other dialects of APL or older versions of Dyalog and the current Dyalog version.
UCMD	User commands that manage the user command framework.
WS	User commands that relate to workspaces.

This chapter summarises the user commands in each of these groups.

For information on the precise syntax for each user command, the arguments that can be supplied to it and the modifiers that it can take, enter `]Help <cmd>` or `]? <cmd>` in a Dyalog Session.

5.1 ARRAY Group

The ARRAY group contains user commands that relate to arrays or variables.

5.1.1]Compare

This user command compares any two APL objects for which ⎕NC is 2 (variables) or 9 (namespaces) and returns the differences between them.

For example:

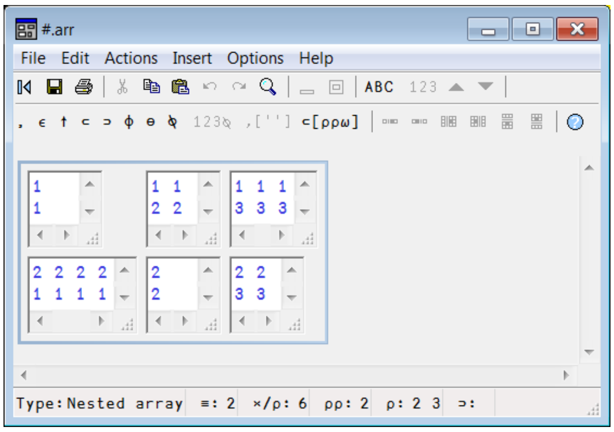
```
      varA←8,1↓varB←⍳9
      ]ARRAY.Compare varA varB

objects are num vectors
elements: 1 different (⎕IO=1)
Var1 8 2 3 4 5 6 7 8 9 Var2 1 2 3 4 5 6 7 8 9
```

5.1.2]Edit

This user command opens the specified array in the Array Editor.

For example:

```
      arr←(2 3⍴1 2 3 4)/¨⍪¨⍳2 3
      ]ARRAY.Edit arr
```



```
Type:Nested array  ≡: 2  ×/⍴: 6  ⍴⍴: 2  ⍴: 2 3  ⊃:
```

5.1.3]ToHTML

This user command outputs the specified namespace/class with the HTML tagging necessary for it to be formatted and displayed in a web browser.

For example:

```
      ]ToHTML ⎕SE.Parser -file=\tmp\x.html
Text in file <\tmp\x.html>
```

5.2 FILE Group

The FILE group contains user commands that relate to files.

5.2.1]CD

This user command reports the current directory if no argument is supplied.

If an argument is specified then this user command changes the current directory to the one specified.

For example:

```
      ]CD
C:\Windows\system32

      ]CD \tmp
C:\Windows\system32

      ]CD
C:\tmp
```

5.2.2]Collect

This user command merges all the files that have a path/name starting with the specified pattern into a single file.

This is particularly useful when `]Split` has been used on a file (see Section *5.2.7*) and the resultant files subsequently need to be reassembled.

For example:

To merge all files starting with **\tmp\file.zip** and followed by **001**, **002** ,**003** and so on into a single file called **\temp\px.zip**:

```
      ]Collect \tmp\file.zip -newname=\temp\px.zip
```

5.2.3]Compare

This user command compares each component within a component file with the component that has the same number in a second component file.

For example:

```
      ]FILE.Compare fileA fileB
Comparing file <fileA>
          with <fileB>
fileA has 2 components starting at 1
fileB has 2 components starting at 1

Comparing 2 components

⍟⍟⍟ Component 2
objects are num vectors
Var1 1 2 3 4 5 Var2 1 2 3 4 5 6 7 8 9 10

⍟⍟⍟ Comparing access matrices (no difference)
```

If fileA comprises components 1 to 10 and fileB comprises components 6 to 22 then only components 6 to 10 will be compared.

5.2.4]Edit

This user command opens the specified native file as an editable text file in the standard in-Session Editor.

For example:

```
]file.edit C:\Users\fiona\Samples\UTF8.txt

]file.edit C:\Users\fiona\Samples\UTF16-BOM.txt
```

5.2.5]Find

This user command only works on the Microsoft Windows operating system.

This user command searches for the specified search string, which can be a .NET regular expression in, by default, **.dyalog** files in the current SALT working directory (as returned by `]Settings workdir`) and its sub-directories.

It returns a list of the files (with full paths) containing the specified string and the line numbers within each file on which the specified string occurs.

For example:

To identify all occurrences of the string "ABC" in all **.dyalog** files in the **\temp** directory and its sub-directories:

```
]Find ABC -folder=\temp
```

To identify all occurrences of the string "ABC" and all seven-letter words in all **.txt** or **.log** files in the current SALT working directory and its sub-directories:

```
]Find \b(ABC|\w{7})\b -typ=txt log -regex
```

5.2.6]Replace

This user command only works on the Microsoft Windows operating system.

This user command searches for the specified search string, which can be a .NET regular expression in, by default, **.dyalog** files in the current SALT working directory (as returned by `]Settings workdir`) and its sub-directories and replaces it with the specified replacement string, which can also be a .NET regular expression.

It returns the number of changes made.

For example:

To replace "ABC" with "XYZ" in all **.dyalog** files in the **\tmp** directory:

```
      ]Replace ABC XYZ -folder=\tmp
23 file(s) changed
```

To reverse every occurrence of two words that follows "Name:" in all **.dyalog** files in the current SALT working directory (for example, "Name: Ken Iverson" becomes "Name: Iverson, Ken"):

```
      ]Replace "Name:\s+(\w+)\s+(\w+)" "Name: $2, $1"
                                                 -regex
31 file(s) changed
```

5.2.7]Split

This user command splits the specified file into the stated number of smaller files (maximum 999) of equal size or multiple individual files of the stated size.

For example:

To split FileA into five individual files (called FileA-01, FileA-02, and so on):

```
      ]Split FileA -n=5
```

To split FileA into individual files (called FileA-01, FileA-02, and so on) of 5 MB each:

```
      ]Split FileA -n=5M
```

5.2.8]ToLarge

This user command converts all small span component files in the specified directory into large span component files.

For example:

```
      ]ToLarge \project -recursive -verbose -backup=.32
* <C:\project\132u64b.DCF> is already 64b
*** <C:\project\to\x1.DCF> is tied
...
<C:\project\to\x2.DCF> made into 64b format and backed up
                             to <C:\project\to\x2.DCF.32>
27 files modified
```

This user command uses `⎕FCOPY` to perform the conversion. This means that it can take a considerable amount of time to execute if there are very large files, but all the timestamps are preserved.

5.2.9]ToQuadTS

This user command takes a timestamp (for example, the last time a component within a component file was updated) and converts it into its `⎕TS` equivalent (a vector of 7 numbers).

For example:

```
      ]ToQuadTS 3⊃⎕frdci 4 1
2013 9 9 23 16 28 0
```

5.2.10]Touch

This user command checks whether the specified file exists in the current/specified location and creates it if it cannot be found.

For example:

```
      ]Touch abc.xyz
```

5.3 FN Group

The FN group contains user commands that relate to functions and operators.

5.3.1]Align

This user command searches for comments at the end of a line of code within the specified function and aligns them to the stated column (defaults to column 40).

For example:

To align all comments at column 30 in functions that start "HTML" and display the names of all the functions that have been modified in `)FNS` format:

```
      ]Align HTML* -offset=30
```

5.3.2]Calls

This user command produces the calling tree of the specified function in the specified class/namespace (defaults to the current namespace).

For example:

```
      ]calls ClassFolder

Level 1:  →ClassFolder
⍝ Produce full path by merging root and folder name
  F:specialName

Level 2: ClassFolder→specialName
⍝ Change any [name] into path
  F:getEnvir     F:lCase        F:uCase

Level 3: specialName→getEnvir
  F:rlb          F:splitOn      F:splitOn1st
F:SALTsetFile

Level 4: getEnvir→SALTsetFile

Level 4: getEnvir→splitOn1st
⍝ Split on 1st occurrence of any chars in str

Level 4: getEnvir→splitOn

Level 4: getEnvir→rlb

Level 3: specialName→uCase
  F:LU

Level 4: uCase→LU
Level 3: specialName→lCase
  *:LU
```

5.3.3]Compare

This user command compares two any APL objects for which ⎕NC is 3 (functions) or 4 (operators) and returns the differences between them (including timestamps).

For example:

```
given:     ∇fna                        ∇fnb
       [1] same line            [1] same line
       [2] fna line 2           [2] fnb line 2
       [3] same line 3          [3] same line 3
       [4] ⍝ comment deleted    [4] new common line
       [5] new common line      [5] ⍝ new comment
           ∇                        ∇

      ]fncomp fna fnb
←[0]   fna
→      fnb
 [1]   same line
←[2]   fna line 2
→      fnb line 2
 [3]   same line 3
←[4]  ⍝ comment deleted
 [5]   new common line
→     ⍝ new comment
```

5.3.4]Defs

This user command lists the names and definitions of all single-line dfns, dops, derived functions and trains, optionally filtered to include only those that contain a specified string or limited to those with the specified names.

For example:

```
      ]Defs
 at←{⍵+(⍴⍵)↑(-⍺⍺)↑⍺}
 derv←{(⍳⍵),¨box⊃⍵*÷2}{⍵+(⍴⍵)↑(-⍺⍺)↑⍺}
 pars←⊃∘(+.×/)
 rcb←{(⍳⍵),¨box⊃⍵*÷2}

      ]Defs ⍺⍺
 at←{⍵+(⍴⍵)↑(-⍺⍺)↑⍺}
 derv←{(⍳⍵),¨box⊃⍵*÷2}{⍵+(⍴⍵)↑(-⍺⍺)↑⍺}

      ]Defs at
 at←{⍵+(⍴⍵)↑(-⍺⍺)↑⍺}
```

5.3.5]DInput

This user command is used to test multi line D-expressions (dfns and dops)

For example:

```
      ]Dinput
····{
········⍵ ⍵
```

```
····}{
········⍺⍺ ⍺⍺ ⍵
····}7
 7 7  7 7
```

5.3.6]Latest

This user command lists the names of any functions changed since the specified date (default is the current system date), with the most recently changed function listed first. Dates are specified as YYYYMMDD but can be shortened to MMDD if the year of interest is the current year; a leading 0 can also then be dropped. For example, `213` is February 13th of the current year

For example:

```
      ]Latest 20140101
#.HelpExample.Help  #.HelpExample.List  #.HelpExample.Run
```

5.3.7]ReorderLocals

This user command changes the order in which the local names in the header of a tradfn are listed.

For example:

To change the order in which the local names in all tradfns that start "F" are listed:

```
      ⎕vr 'Fnml'
    ∇ Fnml;⎕PP;X;⍙;a;_;aa;Aa;aaAA;aA;⎕IO ⍝ locals anyone?
[1]    ...
    ∇

      ]reorderlocals F*
3 fns processed, 1 changed

      ⎕vr 'Fnml'
    ∇ Fnml;a;aa;aA;Aa;aaAA;X;⍙;_;⎕IO;⎕PP ⍝ locals anyone?
[1]    ...
    ∇
```

5.4 MISC Group

The MISC group contains user commands that do not obviously fit into any other category.

5.4.1]Calendar

This user command displays a calendar for the specified month and year (omitting both arguments returns the current month in the current year, omitting the year returns the specified month in the current year, omitting the month returns every month in the specified year).

For example:

```
      ]Calendar 6 1974
     June 1974
Su Mo Tu We Th Fr Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30
```

5.4.2]Factors

This user command returns the factors of the specified integer.

For example:

```
      ]Factors 123456789
3 3 3607 3803
```

5.4.3]PivotTable

This user command requires Dyalog version 14.0 or later.

This user command provides pivot table functionality; the array that is to have pivot table functionality applied to it must have no more than three columns.

For example:

```
      M←(20 2⍴'C3C3C4B4C2B2D1C4A4C1B3B1C1B2A0A1D1B0C1C4'),
4 3 4 8 3 3 9 6 5 9 2 7 7 1 6 5 4 7 6 9

      ]PivotTable M    ⍝ default: count of unique M[;1 2]
        3 4 2 1 0  Total
 C      2 3 1 3 0      9
 B      1 1 2 1 1      6
 D      0 0 0 2 0      2
 A      0 1 0 1 1      3
 Total  3 5 3 7 2     20

      ]PivotTable M  -f=+/  ⍝ sum M[;3] by unique M[;1 2]
        3  4 2  1  0  Total
 C      7 19 3 22  0     51
 B      2  8 4  7  7     28
 D      0  0 0 13  0     13
 A      0  5 0  5  6     16
 Total  9 32 7 47 13    108

      ]PivotTable "(5 2⍴'GrpA' 'case1' 'GrpB' 'case1'
'GrpB' 'case2' 'GrpA' 'case1' 'GrpB' 'case2'),⍳5"  -f=+/
        case1  case2  Total
 GrpA       5      0      5
 GrpB       2      8     10
 Total      7      8     15
```

5.5 NS Group

The NS group contains user commands that relate to namespaces.

5.5.1]ScriptUpdate

This user command updates scripted namespaces/classes to take account of newly added or deleted variables, functions and operators.

In Dyalog the only way to update the source of a scripted object is to edit the source; defining a function using ⎕FX or creating a variable using assignment does not update the source. This user command identifies variables, functions and operators that exist in the specified scripted object but are not part of the source and adds them to the source using ⎕FIX. It also identifies variables, functions and operators that do not exist in the specified scripted object but are part of the source and deletes them from the source using ⎕FIX.

For example:

```
      ]Load myns
      )cs myns
      V←⍳9
      ⎕FX 'myfn' '2+2'

      ]ScriptUpdate
Added 1 variables and 1 functions
```

5.5.2]Summary

This user command returns summary information (scope, size and syntax) of each of the functions in the specified scripted namespace/class.

For example:

```
      ]summary ⎕SE.Parser
 name        scope  size  syntax
 Parse       P      8812   r1f
 Propagate          1584   r2f
 Quotes             1200   r1f
 Switch             1524   r2f
 deQuote             816   r1f
 fixCase              68   n0f
 if                   24   n0f
 init        PC     7976   n1f
 splitParms         2008   r1f
 sqz                 636   r2f
 upperCase           716   r2f
 xCut                524   r2f
```

5.5.3]Xref

This user command generates a cross-reference of the objects in a scripted object.

It produces a table showing all objects referred to (columns) against the function or operator that refers to them (rows). The symbols in the table described the nature of the reference: **o** means local, **G** mean global, **F** means function, **L** means label and **!** identifies an unused localised name.

For example:

```
      src←':Class cl' ':Field myfield←1'
      src,←'∇foo a;var' 'a←1' 'goo' '∇'
      src,←'∇goo;var' 'var←myfield' '∇'
      src,←⊂':EndClass'
      ⎕fix src
      ]Xref cl
       var
  myfield.
     goo..
      a...
      ↓↓↓↓
[FNS]  - -
foo   oF !
goo    .Go
```

This shows that `var` appears in both `foo` and `goo`, but in `foo` it only appears in the function header. `myfield` is referenced in `goo` but is external to it, so appears as a Global to `goo`.

The dot, dash and semi-colon characters only serve as alignment decorators and have no special meaning.

5.6 OUTPUT Group

The OUTPUT group contains user commands that change the way in which arrays are displayed in a Session.

5.6.1]Box

Identical to `]Boxing` (see Section *5.6.2*) – included for convenience when calling on a UNIX installation that does not have the auto-complete feature.

5.6.2]Boxing

This user command requires Dyalog version 14.0 or later.

This user command changes the default display of arrays, functions and operators in the Session. For example, nested arrays can, by default, be displayed as if the `]Disp` or `]Display` user commands had been used (see Sections *5.6.3* and *5.6.4* respectively).

For example:

```
      ]Box on
Was OFF

      ι¨ι2 3
┌─────┬─────────┬─────────────┐
│┌───┐│┌───┬───┐│┌───┬───┬───┐│
││1 1│││1 1│1 2│││1 1│1 2│1 3││
│└───┘│└───┴───┘│└───┴───┴───┘│
├─────┼─────────┼─────────────┤
│┌───┐│┌───┬───┐│┌───┬───┬───┐│
││1 1│││1 1│1 2│││1 1│1 2│1 3││
│├───┤│├───┼───┤│├───┼───┼───┤│
││2 1│││2 1│2 2│││2 1│2 2│2 3││
│└───┘│└───┴───┘│└───┴───┴───┘│
└─────┴─────────┴─────────────┘
```

5.6.3]Disp

This user command displays the specified array with vertical and horizontal lines separating each sub-array. Characters embedded in these borders indicate sub-array shape and type.

Equivalent to the `disp` function from supplied workspace **dfns.dws**.

For example:

```
      ]Disp ι¨ι2 3
┌→────┬─────────┬─────────────┐
↓┌→──┐│┌→──┬───┐│┌→──┬───┬───┐│
│↓1 1││↓1 1│1 2││↓1 1│1 2│1 3││
│└~─→┘↓└~─→┴~─→┘↓└~─→┴~─→┴~─→┘↓
├────→┼────────→┼────────────→┤
│┌→──┐│┌→──┬───┐│┌→──┬───┬───┐│
│↓1 1││↓1 1│1 2││↓1 1│1 2│1 3││
│├~─→┤│├~─→┼~─→┤│├~─→┼~─→┼~─→┤│
││2 1│││2 1│2 2│││2 1│2 2│2 3││
│└~─→┘↓└~─→┴~─→┘↓└~─→┴~─→┴~─→┘↓
└────→┴────────→┴────────────→┘
```

5.6.4]Display

This user command displays the specified array with boxes bordering each sub-array. Characters embedded in the borders indicate sub-array shape and type.

Equivalent to the `display` function from supplied workspace **dfns.dws**.

For example:

```
      ]Display ι¨ι2 3
┌→────────────────────────────────────────────────┐
↓ ┌→──────┐ ┌→────────────┐ ┌→──────────────────┐ │
│ ↓ ┌→──┐ │ ↓ ┌→──┐ ┌→──┐ │ ↓ ┌→──┐ ┌→──┐ ┌→──┐ │ │
│ │ │1 1│ │ │ │1 1│ │1 2│ │ │ │1 1│ │1 2│ │1 3│ │ │
│ │ └~──┘ │ │ └~──┘ └~──┘ │ │ └~──┘ └~──┘ └~──┘ │ │
│ └∊──────┘ └∊────────────┘ └∊──────────────────┘ │
│ ┌→──────┐ ┌→────────────┐ ┌→──────────────────┐ │
│ ↓ ┌→──┐ │ ↓ ┌→──┐ ┌→──┐ │ ↓ ┌→──┐ ┌→──┐ ┌→──┐ │ │
│ │ │1 1│ │ │ │1 1│ │1 2│ │ │ │1 1│ │1 2│ │1 3│ │ │
│ │ └~──┘ │ │ └~──┘ └~──┘ │ │ └~──┘ └~──┘ └~──┘ │ │
│ │ ┌→──┐ │ │ ┌→──┐ ┌→──┐ │ │ ┌→──┐ ┌→──┐ ┌→──┐ │ │
│ │ │2 1│ │ │ │2 1│ │2 2│ │ │ │2 1│ │2 2│ │2 3│ │ │
│ │ └~──┘ │ │ └~──┘ └~──┘ │ │ └~──┘ └~──┘ └~──┘ │ │
│ └∊──────┘ └∊────────────┘ └∊──────────────────┘ │
└∊────────────────────────────────────────────────┘
```

5.6.5]Rows

This user command requires Dyalog version 14.0 or later.

This user command impacts the display of any array that is subsequently entered into the Session by limiting the number of rows that are output.

For example:

```
      ]rows -fold=3
Was off

      ⍳10 4
┌→──────────────────┐
↓1 1 │1 2 │1 3 │1 4 │
├~──→┼~──→┼~──→┼~──→┤
│2 1 │2 2 │2 3 │2 4 │
├~──→┼~──→┼~──→┼~──→┤
│3 1 │3 2 │3 3 │3 4 │
├~──→┼~──→┼~──→┼~──→┤
· · · · · · · · · · ·
├~──→┼~──→┼~──→┼~──→┤
│10 1│10 2│10 3│10 4│
└~──→┴~──→┴~──→┴~──→┘
```

5.7 PERFORMANCE Group

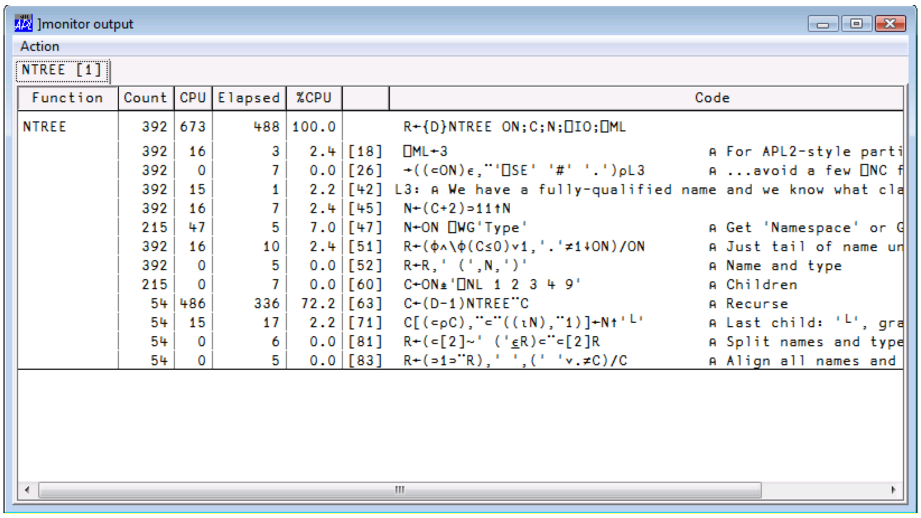
The PERFORMANCE group contains user commands that measure CPU consumption in various ways.

5.7.1]Monitor

This user command reports which lines of code in traditional functions and operators (does not work for dfns and dops) consume the most CPU.

For example:

```
      ]Monitor -on
Monitoring switched on for 44 functions

      5↑[1]NTREE '⎕SE'
⎕SE                     (Session)
├─Chart                 (Namespace)
│ ├─CheckData           (Function)
│ ├─Do                  (Function)
│ ├─DoChart             (Function)

      ]Monitor -report -caption=NTREE
```


]monitor output
Action
NTREE [1]

Function	Count	CPU	Elapsed	%CPU		Code
NTREE	392	673	488	100.0		R←{D}NTREE ON;C;N;⎕IO;⎕ML
	392	16	3	2.4	[18]	⎕ML←3 ⍝ For APL2-style parti
	392	0	7	0.0	[26]	→((⊂ON)∊,¨'⎕SE' '#' '.')⍴L3 ⍝ ...avoid a few ⎕NC f
	392	15	1	2.2	[42]	L3: ⍝ We have a fully-qualified name and we know what cla
	392	16	7	2.4	[45]	N←(C+2)⊃1 1↑N
	215	47	5	7.0	[47]	N←ON ⎕WG'Type' ⍝ Get 'Namespace' or G
	392	16	10	2.4	[51]	R←(⌽∧\⌽(C≤0)∨1,'.'≠1↓ON)/ON ⍝ Just tail of name un
	392	0	5	0.0	[52]	R←R,' (',N,')' ⍝ Name and type
	215	0	7	0.0	[60]	C←ON⍎'⎕NL 1 2 3 4 9' ⍝ Children
	54	486	336	72.2	[63]	C←(D-1)NTREE¨C ⍝ Recurse
	54	15	17	2.2	[71]	C[(⊂⍴C),¨⊂¨((⍳N),¨1)]←N↑'└' ⍝ Last child: '└', gra
	54	0	6	0.0	[81]	R←(⊂[2]~' ('⍷R)⊂¨⊂[2]R ⍝ Split names and type
	54	0	5	0.0	[83]	R←(⊃1⊃¨R),' ',(' 'v.≠C)/C ⍝ Align all names and

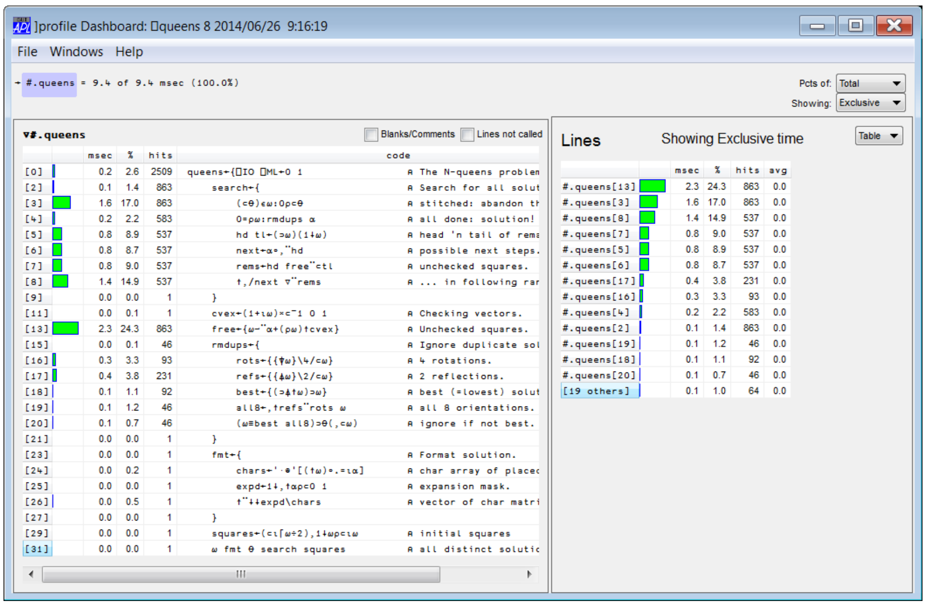
5.7.2]Profile

This user command makes it easy to locate the points in your application at which significant quantities of CPU/elapsed time is spent, facilitating the tuning process.

For more information, see the *Dyalog Application Tuning Guide*.

For example:

```
      )load dfns
      ]Profile -expr="ρqueens 8"
12
```



]profile Dashboard: ⎕queens 8 2014/06/26 9:16:19

→ #.queens = 9.4 of 9.4 msec (100.0%)

Pcts of: Total
Showing: Exclusive

∇#.queens

		msec	%	hits	code	
[0]		0.2	2.6	2509	queens←{⎕IO ⎕ML←0 1	⍝ The N-queens problem
[2]		0.1	1.4	863	search←{	⍝ Search for all solut
[3]	🟩	1.6	17.0	863	(⊂⍬)∊⍵:0⍴⊂⍬	⍝ stitched: abandon th
[4]		0.2	2.2	583	0=⍴⍵:rmdups ⍺	⍝ all done: solution!
[5]	🟩	0.8	8.9	537	hd tl←(⊃⍵)(1↓⍵)	⍝ head 'n tail of rema
[6]	🟩	0.8	8.7	537	next←⍺∘,¨hd	⍝ possible next steps.
[7]	🟩	0.8	9.0	537	rems←hd free¨⊂tl	⍝ unchecked squares.
[8]	🟩	1.4	14.9	537	↑,/next ∇¨rems	⍝ ... in following rar
[9]		0.0	0.0	1	}	
[11]		0.0	0.1	1	cvex←(1+⍳⍵)×⊂¯1 0 1	⍝ Checking vectors.
[13]	🟩	2.3	24.3	863	free←{⍵~¨⍺+(⍴⍵)↑cvex}	⍝ Unchecked squares.
[15]		0.0	0.1	46	rmdups←{	⍝ Ignore duplicate sol
[16]		0.3	3.3	93	rots←{{⍋⍵}\4/⊂⍵}	⍝ 4 rotations.
[17]		0.4	3.8	231	refs←{{⍒⍵}\2/⊂⍵}	⍝ 2 reflections.
[18]		0.1	1.1	92	best←{(⊃⍋↑⍵)⊃⍵}	⍝ best (=lowest) solut
[19]		0.1	1.2	46	all8←,↑refs¨rots ⍵	⍝ all 8 orientations.
[20]		0.1	0.7	46	(⍵≡best all8)⊃⍬(,⊂⍵)	⍝ ignore if not best.
[21]		0.0	0.0	1	}	
[23]		0.0	0.0	1	fmt←{	⍝ Format solution.
[24]		0.0	0.2	1	chars←'·⍟'[(↑⍵)∘.=⍳⍺]	⍝ char array of placed
[25]		0.0	0.0	1	expd←1↓,⍺⍴⊂0 1	⍝ expansion mask.
[26]		0.0	0.5	1	↑¨↓↓expd\chars	⍝ vector of char matri
[27]		0.0	0.0	1	}	
[29]		0.0	0.0	1	squares←(⊂⍳⌈⍵÷2),1↓⍵⍴⊂⍳⍵	⍝ initial squares
[31]		0.0	0.0	1	⍵ fmt ⍬ search squares	⍝ all distinct solutio

Lines   Showing Exclusive time

		msec	%	hits	avg
#.queens[13]	🟩	2.3	24.3	863	0.0
#.queens[3]	🟩	1.6	17.0	863	0.0
#.queens[8]	🟩	1.4	14.9	537	0.0
#.queens[7]	🟩	0.8	9.0	537	0.0
#.queens[5]	🟩	0.8	8.9	537	0.0
#.queens[6]	🟩	0.8	8.7	537	0.0
#.queens[17]		0.4	3.8	231	0.0
#.queens[16]		0.3	3.3	93	0.0
#.queens[4]		0.2	2.2	583	0.0
#.queens[2]		0.1	1.4	863	0.0
#.queens[19]		0.1	1.2	46	0.0
#.queens[18]		0.1	1.1	92	0.0
#.queens[20]		0.1	0.7	46	0.0
[19 others]		0.1	1.0	64	0.0

5.7.3]RunTime

This user command measures and reports the average CPU time and elapsed time required to execute each of the specified APL expressions once.

For example:

To benchmark a single expression by executing that expression once:

```
      ]RunTime {+/1=⍵∨⍳⍵}¨⍳1000
* Benchmarking "{+/1=⍵∨⍳⍵}¨⍳1000"
             Exp
 CPU (avg):   31
 Elapsed:     26
```

To benchmark a single expression by executing that expression repeatedly for 1 second and then averaging the results:

```
      ]RunTime {+/1=⍵∨⍳⍵}¨⍳1000 -repeat=1s
* Benchmarking "{+/1=⍵∨⍳⍵}¨⍳1000", repeat=1s
                     Exp
 CPU (avg):  19.78571429
 Elapsed:    19.94642857
```

To benchmark two expressions by executing them 50 times and then averaging the results, returning the results as a matrix of two rows (for the two expressions) and four columns (⎕MONITOR CPU and elapsed times and ⎕AI CPU and elapsed times):

```
      ]RunTime {+/1=ω∨⍳ω}¨⍳100 ⍳⍨⍳1e6 -details=ai -rep=50
 0.32  0.32  0.32  0.32
22.78 22.88 22.78 22.9
```

To compare the benchmarking statistics of two expressions:

```
      ]runtime ρρ⍳9 ρ∘ρ⍳9 -comp
ρρ⍳9  → 1.6E¯7 |   0% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
ρ∘ρ⍳9 → 3.2E¯7 | +93% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
```

5.7.4]SpaceNeeded

This user command returns the space (in bytes) required to execute the specified expressions.

For example:

```
      ]spaceneeded  ⍳1e6  ρ⍳8
 ⍳1e6  4000102
 ρ⍳8       818
```

5.8 SALT Group

The SALT group contains user commands that perform the same actions as the SALT functions of the same name found in ⎕SE.SALT. For more information on SALT, see the *Dyalog SALT Reference Guide*.

APL objects that have been saved using SALT/user commands (that is, by calling either the Save or the Snap SALT function) or by running the]Save or]Snap user commands are referred to as SALTed.

5.8.1]Clean

This user command removes all the tags associated with SALT from each object in the workspace. Running this user command means that SALT no longer saves changes that are made in the workspace to the objects that were untagged.

This is the only user command in the SALT group that is not analogous to a SALT function.

For example:

To remove the SALT tags from all APL objects in the active workspace:

```
      ]Clean
```

To remove the SALT tags from APL objects objA and objB in the active workspace:

```
      ]Clean objA objB
```

5.8.2]Compare

Analogous to `⎕SE.SALT.Compare`.

This user command identifies the differences between two different versions of the same file or between two similar but distinct files.

For example:

```
      ]SALT.Compare C:\Users\andy\Desktop\abc.dyalog
C:\Users\andy\Desktop\abc2.dyalog

Comparing C:\Users\andy\Desktop\abc.dyalog
     with C:\Users\andy\Desktop\abc2.dyalog

 [0]     cmpx←{                ⍝ Approx expression timings.
-[1]         ⍺←⍬               ⍝ options: raw cpu cols.
+            ⍺←⍬               ⍝ options: raw cp.
 [2]         1=≡,⍵:⍺ ∇⊂,⍵      ⍝ single expression: enclose.
 [3]         ⍺{                ⍝ options.
-[4]             (⍎⍵)-⎕AI      ⍝ time of ⍺ expr-iterations.
 [5]         }{⎕IO ⎕ML←0 1     ⍝ local sysvars (see Notes).
 [6]         dflt←{⍵+⍺×⍵=0}    ⍝ ⍺ default if ⍵=0.
```

5.8.3]List

Analogous to `⎕SE.SALT.List`.

This user command lists the files/directories in a specified location (by default, this is the **[SALT]** directory.

For example:

```
      ]List
 Type    Name    Versions   Size   Last Update
 <DIR>   core                      2014/06/10 10:41:19
 <DIR>   lib                       2014/06/10 10:41:19
 <DIR>   spice                     2014/06/20 10:44:56
 <DIR>   study                     2014/06/10 10:41:19
 <DIR>   tools                     2014/06/10 10:41:19
```

5.8.4]Load

Analogous to `⎕SE.SALT.Load`.

This user command loads the latest (highest numbered) version of an APL object into the namespace that the user command is run in. By default, the link between the loaded APL object and its source is maintained and the loaded APL object is assigned a global name. Depending on the nameclass of the APL object loaded, this user command returns a shy result of:

- a reference to the loaded namespace(s)
- the name of the function/variable/operator loaded

For example:

```
      ]Load C:\Users\jason\Desktop\DIR\abc
ABC
```

5.8.5]Open

Analogous to `⎕SE.SALT.Open`.

This user command opens directories and files, including files that are external to Dyalog, using the appropriate program.

For example:

```
      ]Load C:\Users\jason\Desktop\DIR\abc
```

5.8.6]RemoveVersions

Analogous to `⎕SE.SALT.RemoveVersions`.

This user command deletes a version (or range of versions) of a versioned file and returns the number of versions that have been deleted.

For example:

```
      b←1
      'bb'⎕ns'b'
      ]snap
 #.b  #.bb.b

      ]RemoveVersions b -all
1 version deleted.
```

5.8.7]Save

Analogous to `⎕SE.SALT.Save`.

This user command saves an APL object in a native text file and returns the full path and name of the file that it saves. APL objects that are already SALTed are saved in the original location by default.

For example:

To save APL object ABC as a file called **abc.dyalog** in directory **DIR** (creating directory **DIR** if it does not already exist):

```
      ]Save ABC C:\Users\jason\Desktop\DIR\abc -makedir
C:\Users\jason\Desktop\DIR\abc.dyalog
```

5.8.8]Settings

Analogous to `⎕SE.SALT.Settings`.

The values of certain global parameters are retrieved from the Microsoft Windows Registry at the start of a Dyalog Session. These Session parameters remain active for the Session unless they are modified – one way in which they can be modified is by running the `]Settings` user command.

For example:

```
      ]Settings
 compare        apl
 cmddir         C:\Program Files (x86)\Dyalog\Dyalog APL
14.0 Unicode\SALT\Spice
```

```
 debug          0
 editor         notepad
 edprompt       1
 mapprimitives  1
 newcmd         auto
 track
 varfmt         xml
 workdir        C:\Program Files (x86)\Dyalog\Dyalog APL
14.0 Unicode\SALT
```

> The global parameters that can be changed by running the]Settings user command can also impact SALT functionality – for more information on SALT see the *Dyalog SALT Reference Guide*.

The global parameters that impact user commands are:

- cmddir – the full path to the directory (or list of directories) from which to retrieve user commands
- debug – specifies the level of debugging to use. Possible values are:
  - 0 : no debugging and report errors in the environment
  - >0 : stop if an error is encountered
- edprompt – specifies the frequency at which a user is prompted for confirmation to overwrite the file when modifying a script. Possible values are:
  - 0 or n : the user is never prompted for confirmation
  - 1 or y: the user is prompted for confirmation each time a script is modified
- newcmd – specifies when new user commands become effective in the user interface. Possible values are:
  - auto : new commands are detected automatically
  - manual : new commands do not become effective until the user command]UReset is run.

5.8.9]Snap

Analogous to ⎕SE.SALT.Snap.

Although the]Save user command enables individual APL objects to be saved, saving all the APL objects in a workspace using the]Save user command would be a repetitive process. Instead, the]Snap user command performs a bulk save of every APL object in the workspace in individual native text files – all new APL objects are saved to the specified directory and all modified APL objects are saved to the appropriate location. A list of the names of the APL objects that have been successfully saved is returned. If the]Snap user command stops for any reason, then everything that has already been saved remains saved and a list of the names of the APL objects that have been successfully saved is returned.

For example:

```
      a←1
      'myns'⎕ns'a'
      ]snap
 #.a  #.myns.a
```

5.9 SAMPLES Group

The SAMPLES group contains user commands that demonstrate the use of multiple levels of help and parsing user command lines.

The user commands in this group are not like those in other groups; they do not provide any useful functionality but their code can be examined to assist with understanding when creating custom user commands. This can be achieved by opening them in any text editor, for example, Microsoft Notepad.

This group is only available if `]Settings cmddir ,[SALT]/study` is issued.

5.9.1]UCMDHelp

An example of a custom user command that defines multiple levels of help information in the Help function, selectable by the left argument supplied by the user.

To open the code for this user command in the Editor:

```
      ]ULoad UCMDHelp
The source code for command "ucmdhelp" has been loaded in
namespace "#.HelpExample"

      )ED HelpExample
```

5.9.2]UCMDNoParsing

An example of a custom user command that does not use parsing; the argument is the entire string after the command name.

To open the code for this user command in the Editor:

```
      ]ULoad UCMDNoParsing
The source code for command "ucmdnoparsing" has been
loaded in namespace "#.anyname"

      )ED anyname
```

5.9.3]UCMDParsing

An example of a custom user command that uses parsing; the string after the command name is parsed and turned into a namespace containing the arguments (tokenised) and each of the identified switches.

To open the code for this user command in the Editor:

```
      ]ULoad UCMDParsing
The source code for command "ucmdparsing" has been loaded
in namespace "#.anyname"

      )ED anyname
```

5.10 SVN Group

svn is the official command-line client of Subversion; the SVN group contains user commands that cover svn's task-specific subcommands of the same name.

The user commands in the SVN group only work on the Microsoft Windows operating system.

To be used successfully, the user commands in this group require the command line version of Subversion to be installed. For detailed information and to download this, see http://subversion.apache.org/download/#recommended-release.

5.10.1]Add

This user command saves the specified APL object and adds it to the user's working copy of the repository ready to be committed to the master svn repository.

Similar to the `svn add` subcommand.

For example:

To save the APL object `ns_nick` to a file called **nick** in the user's working copy of the repository:

```
      ]Add ns_nick nick
```

5.10.2]Checkout

This user command checks out a working copy of the specified APL object from the master svn repository.

Similar to the `svn checkout` (or `svn co`) subcommand.

For example:

To check out the contents of the master svn repository **http://svn.local/myproject** to the user's working copy of the repository:

```
      ]Checkout http://svn.local/myproject
```

To check out the contents of the master svn repository **http://svn.local/myproject** to the directory **c:\My Projects\myproject** and set the destination directory to be the current working directory:

```
      ]Checkout http://svn.local/myproject "c:\My
Projects\myproject"
```

5.10.3]Commit

This user command sends changes made in the user's working copy of the repository to the master svn repository.

Similar to the `svn commit` (or `svn ci`) subcommand.

For example:

To commit the changes made in the file **nick** in the user's working copy of the repository with the message "new feature added":

```
]Commit nick -m="New feature added"
```

5.10.4]Delete

This user command deletes the specified file from the user's working copy of the repository and schedules it for deletion from the master svn repository (the deletion will occur the next time the `]commit` user command is run). If the URL of an svn repository is specified instead of a file name, then the specified repository is deleted.

Similar to the `svn delete` subcommand.

For example:

To delete the file **nick** from the user's working copy of the repository and schedule it for deletion from the master svn repository:

```
]Delete nick
```

5.10.5]Diff

This user command returns the differences between the version of the specified file in the user's working copy of the repository and the version of the file in the master svn repository.

Similar to the `svn diff` (or `svn di`) subcommand.

For example:

```
]Diff nick
```

5.10.6]Export

This user command copies the most recent version of the contents of the specified URL of an svn repository and saves those contents to the specified location. No svn metadata is saved with the files.

Similar to the `svn export` subcommand.

For example:

To save the contents of an svn repository with the URL http://svn.mysite.com/Nick to a directory called **nick** in the C directory:

```
]Export "http://svn.mysite.com/Nick" "c:\nick"
```

5.10.7]Import

This user command copies the most recent version of the contents of the specified location and saves those contents to the svn repository that has the specified URL.

Similar to the `svn import` subcommand.

For example:

To save the contents of a directory called **nick** in the C directory to an svn repository with the URL http://svn.mysite.com/Nick:

```
]Import "c:\nick"  "http://svn.mysite.com/Nick"
```

5.10.8]Resolve

This user command informs svn that conflicts between different versions of the specified file have been resolved.

Similar to the `svn resolve` subcommand.

For example:

```
]Resolve nick
```

5.10.9]Status

This user command shows the status of the contents of the svn repository that has the specified URL (for example, unknown or modified).

Similar to the `svn status` subcommand.

For example:

To check the status of every file in the user's working copy of the repository:

```
]Status
```

To check the status of every file in the svn repository that has the URL http://svn.mysite.com/Nick:

```
]Status " http://svn.mysite.com/Nick"
```

5.10.10]Update

This user command updates the user's working copy of the repository with any changes that have been committed to the master svn repository since the original checkout or last time that the `]update` user command was run.

Similar to the `svn update` (or `svn up`) subcommand.

For example:

To update the user's working copy of the repository:

```
]Update
```

To update only the file **nick** in the user's working copy of the repository:

```
]Update nick
```

5.11 TOOLS Group

The TOOLS group contains user commands that can assist developers by retrieving and presenting information without changing the underlying code.

5.11.1]Assemblies

This user command only works on the Microsoft Windows operating system.

This user command lists all the .NET assemblies loaded in the current application domain.

For example:

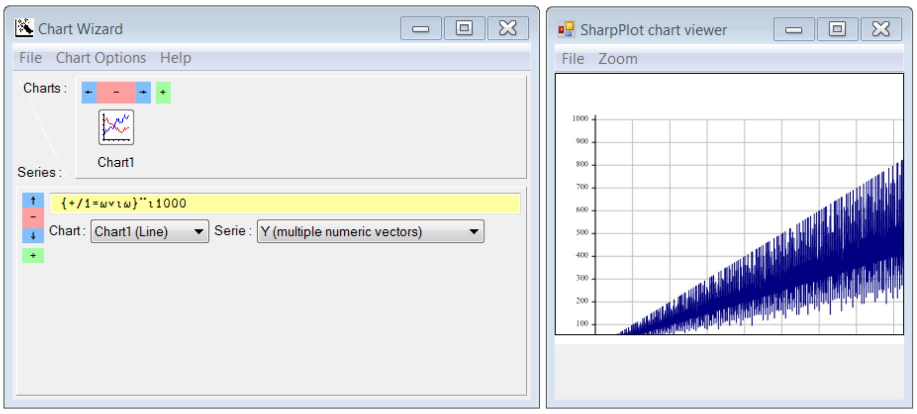
```
      ]Assemblies
mscorlib, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
bridge140_unicode, Version=14.0.20631.0, Culture=neutral,
PublicKeyToken=eb5ebc232de94dcf
msvcm80, Version=8.0.50727.6195, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
dyalognet, Version=14.0.20631.0, Culture=neutral,
PublicKeyToken=eb5ebc232de94dcf
System.Configuration, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
System, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
System.Xml, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
```

5.11.2]Chart

This user command opens the Chart Wizard and SharpPlot Chart Viewer to display the specified expression.

For example:

```
      ]Chart {+/1=ω∨ιω}¨ι1000
```



{+/1=ω∨ιω}¨ι1000
Chart : Chart1 (Line)
Serie : Y (multiple numeric vectors)

5.11.3]Demo

This user command provides a playback mechanism for live demonstrations of code written in Dyalog. It takes a script (the specified text file) name as an argument and executes each APL line in it after displaying it on the screen.

For example:

```
      ]Demo \tmp\mydemo
```

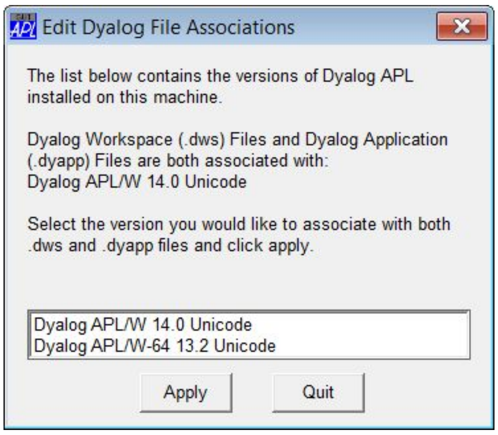
5.11.4]FileAssociations

This user command only works on the Microsoft Windows operating system.

This user command associates files that have the extension **.dws** or **.dyapp** with a specific Dyalog version. This is only relevant if you have multiple versions of Dyalog installed and want to change the version in which **.dws** and **.dyapp** files open when double-clicked on.

For example:

```
      ]FileAssociations
```



Edit Dyalog File Associations

The list below contains the versions of Dyalog APL installed on this machine.

Dyalog Workspace (.dws) Files and Dyalog Application (.dyapp) Files are both associated with:
Dyalog APL/W 14.0 Unicode

Select the version you would like to associate with both .dws and .dyapp files and click apply.

Dyalog APL/W 14.0 Unicode
Dyalog APL/W-64 13.2 Unicode

5.11.5]FromHex

This user command converts a hexadecimal number to its decimal equivalent.

For example:

```
      ]FromHex 64 100
100 256
```

5.11.6]GUIProps

This user command reports the properties (and their values) of the specified GUI object or, if none is provided, the object on which the Session has focus (the object whose name appears in the bottom left corner of the Session log). This only works for GUI objects that have been created using the ⎕WC syntax, not for GUI objects that have been created using other techniques.

For example:

```
      't' ⎕WC 'timer' ('active'0)

      ]GuiProps t
Properties of #.t Interval:1000  Event:     KeepOnClose:0
Type:Timer        Active:0       Data:
Properties of #.t
MethodList: Detach  Wait
ChildList: Timer
EventList: Close  Create  Timer
```

5.11.7]ToHex

This user command converts a decimal number to its hexadecimal equivalent.

For example:

```
      ]ToHex 100 256
 64  100
```

5.12 TRANSFER Group

The TRANSFER group contains user commands that convert workspaces between files written using other dialects of APL or older versions of Dyalog and the current Dyalog version. For more information, see the *Dyalog Workspace Transfer Guide*.

5.12.1]In

This user command imports workspaces between files written using other dialects of APL or older versions of Dyalog and the current Dyalog version.

5.12.2]Out

This user command exports workspaces written using the current Dyalog version into files that are valid for other dialects of APL or older versions of Dyalog.

5.13 UCMD Group

The UCMD group contains user commands that manage the user command framework.

5.13.1]UDebug

This user command facilitates the debugging of custom user commands. When a `-` character is added as the last item of the user command and the user command is executed, the `-` character is removed, a stop is set on line 1 of the `Run` function to suspend its execution and the Debugger is opened when execution reaches that line.

If the namespace containing the user command is within the current namespace, then that version of the namespace is used rather than the script on file.

NOTE: Use of the `-flags` modifier with this user command should only be used as directed by Dyalog Ltd.

5.13.2]ULoad

This user command loads the namespace associated with the specified user command into the active workspace.

For example:

```
      ]ULoad UCMDHelp
The source code for command "ucmdhelp" has been loaded in
namespace "#.HelpExample"
```

5.13.3]UMonitor

This user command turns monitoring on or off. When on, invoking a user command causes its ⎕CR and ⎕MONITOR information to be paired in the global variable #.UCMDMonitor; this information can be further processed to report code coverage.

For example:

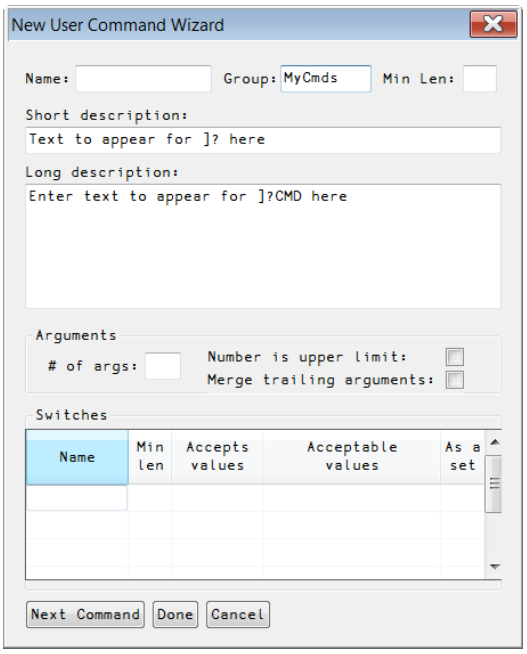
```
      ]UMonitor on
Was OFF

      ]calendar
     June 2014
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

      ]UMonitor -report
 x line never executed
 → branch always taken
 ↓ branch never taken
 : label  never used
 ? questionnable line

∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇∇

       1 [0]   r←Run(Cmd Args)
       1 [1]   :Access Shared Public
     ? 1 [2]   :select AllCmdsι⊂Cmd
 x  :  0 [3]   :case 1⊣'calendar' ⋄ r←calendar Args
  →    1 [4]   :case 2⊣'peek' ⋄ r←wspeek Args
 x  :  0 [5]   :case 3⊣'dinput' ⋄ r←##.THIS dinput Args
 x  :  0 [6]   :case 4⊣'map' ⋄ r←##.THIS map Args
 x     0 [7]   :endselect
```

5.13.4]UNew

This user command opens the New User Command Wizard, a form that can be used to input the basic information pertaining to a new user command (press **F1** for details when running it). A new class skeleton is created from this information that can be further edited.

For example:

```
          ]UNew
```



5.13.5]URefresh

This user command reloads the most recent version of all SALTed objects that have been changed. This situation can occur if you)LOAD a workspace that contains stale objects (for example).

For example:

```
      ]URefresh
8 objects refreshed
```

5.13.6]UReset

This user command forces a rebuild of the user command cache file. This is necessary to pick up changes made to files containing user commands (unless the Session is restarted, in which situation the cache is automatically rebuilt, or the global `newcmd` parameter has been set to *auto* – see Section *5.8.8*).

For example:

```
      ]UReset
88 commands reloaded
```

5.13.7]USetup

This user command is used to initialise files in which Session preferences are customised and modified, for example, configuration of program function (PF) keys. It is analogous to ⎕LX in the Dyalog interpreter.

For example:

```
      ]USetup
```

5.13.8]UUpdate

This user command updates SALT and/or user commands to the latest version. If SALT is updated, then the user command framework is automatically updated too.

NOTE: If you need to update SALT/user commands to a later major version than the one that came with your version of Dyalog, then you will need to use the -version flag.

For example:

```
      ]UUpdate
```

5.13.9]UVersion

This user command reports the version numbers of Dyalog, SALT, UCMD and .NET for the current Session. If the name of a file containing a workspace is specified as an argument, then the minimum version of Dyalog necessary to)LOAD that workspace is returned.

For example:

```
      ]UVersion
  APL  Windows 14.0.21658.0 W Development Unicode 9e252458
  SALT 2.4
  UCMD 2
  .NET 2.0

      ]UVersion 'C:\Users\fiona\Samples\wsA.dws'
14
```

5.14 WS Group

The WS group contains user commands that relate to workspaces.

5.14.1]Compare

This user command compares any two workspaces and returns the size difference and the APL object differences between them; it can be thought of as a combination of the]ARRAY.Compare and]FN.Compare. user commands running at a workspace level.

For example:

```
      ]WS.Compare 'C:\Users\fiona\Samples\wsA.dws'
'C:\Users\fiona\Samples\wsB.dws'
* comparing 'C:\Users\fiona\Samples\wsA.dws'
       with 'C:\Users\fiona\Samples\wsB.dws'

NOTE: total sizes differ by 248 bytes.

<…etc…>
```

5.14.2]Document

This user command lists and details the contents (namespaces, functions, operators and variables) of your workspace.

For example:

To display the contents of the workspace on the screen (this workspace only contains a single variable, name←3):

```
      ]Document
      )wsid
CLEAR WS

      )fns


      )vars
name

      name  (type=I ⍴⍴=0 ⍴=⍬)
3

      )obs
```

To output the contents of the workspace to a file:

```
      ]Document -file=C:\Users\karen\Samples\tmp.txt
Output file = C:\Users\karen\Samples\tmp.txt
```

5.14.3]FindRefs

This user command attempts to find all references in a workspace and identify where they are referenced from.

For example:

```
      a←⎕ns''
      b←1 a 2
      'andy'⎕ns''

      ]FindRefs
 #: Followed 5 pointers to reach a total of 3 "refs" in
0ms.
  Shortest Name  Alias 1
  #
  #.a            #.b[2]
  #.andy
```

5.14.4]FnsLike

This user command returns a list of APL objects for which ⎕NC is 3 (functions) or 4 (operators) that exist in the current namespace and match the specified pattern.

For example:

To find all APL objects for which ⎕NC is 3 or 4:

```
      ]FnsLike
big       det        else      getfile  life
```

To find all APL objects for which ⎕NC is 9 that contain the letter "e" in their name:

```
      ]FnsLike *e*
det        else       getfile  life
```

5.14.5]Locate

This user command searches for the specified string in the current namespace.

For example:

To search for the string "queens":

```
      ]locate queens

    ∇ #.queens (3 found)
[0]   queens←{⎕IO ⎕ML←0 1          ⍝ The N-queens problem.
      ^                                    ^

[24] chars←'·⍟'[(↑⍵)∘.=⍳⍺] ⍝ char array of placed queens.
                                                ^
```

To search for the string "queens" irrespective of case and ignoring comments:

```
      ]locate queens -insensitive -exclude=C

    ∇ #.queens (1 found)
[0]   queens←{⎕IO ⎕ML←0 1         ⍝ The N-queens problem.
      ^
```

5.14.6]Map

This user command displays the structure of the specified namespace (or the current namespace if none is specified) in terms of its constituent variables, functions and operators (identified with ~, ∇ and ∘ respectively). Sub-namespaces are displayed recursively.

This user command uses the `tree` function from supplied workspace **dfns.dws**.

For example:

```
      ]map ⎕SE.Dyalog
⎕SE.Dyalog
·   Callbacks
·   ·   ∇ WSLoaded
·   SEEd → ⎕SE.[SessionEditor]
·   Utils
·   ·   ~ Version lc uc
·   ·   ∇ cut disp display dmb drvSrc dtb fromXML fromto
lcase psmum repObj showCol showRow toMatrix toVector
toXML trimEnds txtreplace ucase where
·   ·   SALT_Data → ⎕SE.[Namespace]
```

5.14.7]NamesLike

This user command returns a list of all APL objects (irrespective of ⎕NC) that exist in the current namespace and match the specified pattern.

For example:

To find all APL objects that contain the letter "a" in their name:

```
      ]NamesLike *a*
aplUtils.9         disableSALT.3      enableSALT.3
commandLineArgs.2 disableSPICE.3     enableSPICE.3
```

To find all APL objects that contain the letter "a" in their name without showing their nameclass:

```
      ]NamesLike *a* -noclass
aplUtils        disableSALT      enableSALT
commandLineArgs disableSPICE     enableSPICE
```

5.14.8]Nms

This user command returns a list of all APL objects (irrespective of ⎕NC) that exist in the current namespace and match the specified pattern.

Almost identical to `]NamesLike` but does not have a modifier for removing the nameclass when displaying results. This slight restriction means that it matches IBM's APL2 system command `)NMS`.

For example:

```
      ]Nms *a*
aplUtils.9         disableSALT.3      enableSALT.3
commandLineArgs.2 disableSPICE.3     enableSPICE.3
```

5.14.9]ObsLike

This user command produces a list of APL objects for which ⎕NC is 9 (namespaces) that exist in the current namespace and match the specified pattern.

For example:

To find all APL objects for which ⎕NC is 9:

```
      ]ObsLike
NStoScript      aplUtils        test
```

To find all APL objects for which ⎕NC is 9 that contain the letter "a" in their name:

```
      ]ObsLike *s*
aplUtils        test
```

5.14.10]Peek

This user command executes the specified expression in a temporary copy of the workspace; any changes made are discarded on termination of the user command, meaning that the current workspace is unchanged.

This user command copies the specified workspace into a temporary namespace in the current process and executes the specified expression in that namespace. It is used to view, rather than to change, a saved workspace; any changes made in the copy are discarded on termination of the command.

For example:

Execute the `queens` program from supplied workspace **dfns.dws**:

```
      ]Peek dfns 0 disp queens 5
```

5.14.11]SizeOf

This user command produces a list of all APL objects that exist in the current namespace and match the specified pattern along with their size (in bytes) in decreasing order.

For example:

```
      )obs
NStoScript      aplUtils        test

      )vars
CR      DELINS  Describe        FS

      ]SizeOf -top=2 -class=2 9
NStoScript 132352    aplUtils    40964
```

5.14.12]VarsLike

This user command returns a list of APL objects for which ⎕NC is 2 (variables) that exist in the current namespace and match the specified pattern.

For example:

To find all APL objects for which ⎕NC is 2:

```
      ]VarsLike
CR      DELINS  Describe        FS
```

To find all APL objects for which ⎕NC is 2 that contain the letter "a" in their name:

```
      ]VarsLike *S*
DELINS  FS
```

Appendix A Example User Commands

This appendix includes examples illustrating the construction of user commands.

The examples in this appendix have been created to illustrate different aspects of user commands. This means that they do not necessarily follow an efficient workflow process or best coding practice.

A.1 Example: Basic User Command Definition

This example illustrates the definition of a basic user command.

A new user command called `Time` is required to display the local time. The necessary functions are defined in a namespace called timefns:

```
:Namespace timefns

    ⎕ML ⎕IO←1   ⍝ set to avoid inheriting external values

    ∇ r←List
      r←⎕NS¨1⍴⊂''    ⍝ r is a vector of length 1 with the
                     ⍝ item set to be a ref to a namespace
      r.(Group Parse Name)←⊂'TimeGrp' '' 'Time'
      r[1].Desc←'Time example Script'
    ∇

    ∇ r←Run(Cmd Args)
      r←1↓,'⊂:⊃,ZI2'⎕FMT ⎕TS[4 5 6]   ⍝ show time
    ∇

    ∇ r←Help Cmd
      r←'Time (no arguments)'
    ∇

:EndNamespace
```

In this example:

- The `List` function sets the four variables Desc, Name, Group and Parse to 'Time example Script', Time, TimeGrp and <null> respectively.
- The `Run` function only needs to call `⎕TS` so the command name and any supplied arguments are ignored. This function also formats the time into a user-friendly format.
- The `Help` function identifies that there is only one user command in the namespace (there is only one user command name, `Time`, defined) and returns the appropriate information for the Time user command.

Running this user command in a Dyalog Session returns three numbers; these three numbers are the current time – respectively they indicate the hour (according to the

24 hour clock), the number of minutes past the hour and the number of seconds elapsed. For example:

```
      ]?Time
Command "Time".

Time (no arguments)

Script location: c:\program files\dyalog\dyalog apl 14.0
unicode\salt\spice\timefns
```

(the same result is returned if `]Help Time` or `]??Time` is entered)

```
      ]Time
13:05:09
```

(indicating that the current system time is 13:05 and 9 seconds)

A.2 Example: Cross-Operating System User Command Definition

This example illustrates the inclusion of two different user commands within a single namespace, different techniques for achieving the same result depending on the operating system being used and using breakout without user commands.

Although the current system time returned by the `Time` user command (see Section *A.1*) is useful, it might be more relevant to have a choice of displaying local time or UTC (Co-ordinated Universal Time). To do this, a new user command called `UTC` is required. As this is closely related to the `Time` user command, it should be created in the same namespace; this involves adding a new function called `Zulu` and modifying the `Run`, `List` and `Help` functions.

To illustrate the ability of a user command to obtain information through a breakout call to .NET, this example also includes options in the `Run` function that are dependent on the operating system that the Dyalog Session is being run on (.NET is only valid when running on the Microsoft Windows operating system). These options ensure that the same user command is cross-system compatible for Microsoft Windows and Linux.

```
:Namespace timefns

    ⎕ML ⎕IO←1    ⍝ set to avoid inheriting external values

    ∇ r←List
      r←⎕NS¨2⍴⊂''    ⍝ r is a vector of length 2 with the
                     ⍝ items set to be refs to namespaces
      r.(Group Parse)←⊂'TimeGrp' ''
      r.Name←'Time' 'UTC'
      r.Desc←'Show local time' 'Show UTC time'
    ∇

    ∇ r←Run(Cmd Args);dt
      :If 'Windows' ≡ 7↑⊃'.'⎕WG 'APLVERSION'    ⍝ Windows
         ⎕USING←'System'                        ⍝ Windows
         dt←DateTime.Now                        ⍝ Windows
            :If 'UTC'≡Cmd                       ⍝ Windows
               dt←Zulu dt                       ⍝ Windows
            :EndIf                              ⍝ Windows
         r←(rι' ')↓r←⍕dt                            ⍝ Windows
      :ElseIf 'Linux' ≡ 5↑⊃'.'⎕WG 'APLVERSION' ⍝ Linux
         dt←('UTC'≡Cmd)/'TZ=UTC'                    ⍝ Linux
         r←⊃⎕SH dt,' date +"%H:%M:%S"'              ⍝ Linux
      :Else
         r←'Unrecognised operating system'       ⍝ neither!
      :EndIf
    ∇

    ∇ r←Help Cmd;which
      which←'Time' 'UTC'ι⊂Cmd
      r←which⊃'Time (no arguments)' 'UTC (no arguments)'
    ∇

    ∇ r←Zulu date
     ⍝ Use .Net to retrieve UTC info
      r←TimeZone.CurrentTimeZone.ToUniversalTime date
    ∇

:EndNamespace
```

In this example:

- The `List` function is amended to allow for two function definitions in the four variable definitions:
  - o Desc is set to to 'Show local time'/'Show UTC time' (two values, therefore the first applies to the first user command and the second applies to the second user command)
  - o Name is set to Time/UTC (two values, therefore the first applies to the first user command and the second applies to the second user command)
  - o Group is set to TimeGrp (only one value so applied to both user commands)
  - o Parse is set to <null> (only one value so applied to both user commands)
- The `Run` function is amended to use the Cmd argument to determine which user command is being run (any further supplied arguments are still ignored). The operating system on which the Dyalog Session is being run is then identified; different actions are taken depending on whether the operating system is Microsoft Windows or Linux (if neither, then a message is returned). The operating system is then used to determine the current system time rather than the APL system function `⎕TS`, for example, if the UTC user command is being run on a Microsoft Windows operating system, then the `Run` function calls the `Zulu` function. The `Run` function also formats the resulting time into a more user-friendly format irrespective of the operating system and user command.
- The `Help` function is amended to enable it to identify that there are two user commands in the namespace (there are two user command names, `Time` and `UTC`, defined) and return the appropriate information according to which name is specified.
- The `Zulu` function is added to retrieve the UTC time through a .NET call – this function is only called if the `Run` function identifies that the Dyalog Session is running on a Microsoft Windows operating system and the UTC user command is specified.

After changing the code but before running these user commands, the `]UReset` user command should be run to force a cache file update (otherwise the code changes will not be detected).

The `Time` and `UTC` user commands can now be run from a Dyalog Session:

```
      ]?TimeGrp
 Group     Name  Description
 =====     ====  ===========
 TimeGrp   Time  Show local time
           UTC   Show UTC time

      ]?Time
Command "Time".

Time (no arguments)

Script location: c:\program files\dyalog\dyalog apl 14.0
unicode\salt\spice\timefns
```

(the same result is returned if `]Help Time` or `]??Time` is entered)

```
      ]Time
13:17:34
```

(indicating that the current system time is 13:17 and 34 seconds)

```
      ]?UTC
Command "UTC".

UTC (no arguments)

Script location: c:\program files\dyalog\dyalog apl 14.0
unicode\salt\spice\timefns
```

(the same result is returned if `]Help UTC` or `]??UTC` is entered)

```
      ]UTC
12:18:15
```

(indicating that the co-ordinated universal time is 12:18 and 15 seconds)

A.3 Example: Optional Arguments

This example illustrates the creation of a user command with an optional argument.

Although the `Time` and `UTC` user commands return the local time and UTC respectively (see Section *A.2*), they only work for the location in which the system is located. To return the time in different locations, new functions could be defined for each location and the `Run`, `List` and `Help` functions modified accordingly. Alternatively, the `Run` function can be modified to use the location as an argument to compute the time (this does not take account of daylight saving time). Using this second approach the **timefns.dyalog** file can be modified as follows (Microsoft Windows only):

```
:Namespace timefns

    ⎕ML ⎕IO←1    ⍝ set to avoid inheriting external values
    ∇ r←List
      r←⎕NS¨2⍴⊂''    ⍝ r is a vector of length 2 with the
                     ⍝ items set to be refs to namespaces
      r.(Group Parse)←⊂'TimeGrp' ''
      r.Name←'Time' 'UTC'
      r.Desc←'Show local time in a city' 'Show UTC time'
    ∇

    ∇ r←Run(Cmd Args);dt;offset;cities;diff;city;lcity;ix
      ⎕USING←'System'
      dt←DateTime.Now
      :Select Cmd
      :Case 'UTC'
          dt←Zulu dt
      :Case 'Time'
          :If 0≠⍴city←Args~' '
              offset←CityTimeOffset city
              'Unknown city'⎕SIGNAL 11⍴⍨0≡offset
              diff←⎕NEW TimeSpan(3↑offset)
              dt←(Zulu dt)+diff
          :EndIf
      :EndSelect
      r←(r⍳' ')↓r←⍕dt
    ∇

    ∇ r←Help Cmd;which
      which←'Time' 'UTC'⍳⊂Cmd
      r←which⊃'Time [city]' 'UTC (no arguments)'
    ∇

    ∇ r←Zulu date
     ⍝ Use .Net to retrieve UTC info
      r←TimeZone.CurrentTimeZone.ToUniversalTime date
    ∇

    ∇ r←CityTimeOffset city;lcity;cities;ix;offsets
      cities←'l.a.' 'montreal' 'copenhagen' 'sydney'
      offsets←¯8 ¯5 2 10
      r←⍬                           ⍝ Assume no match
      lcity←('.'⎕R'\l&')city        ⍝ Name to lowercase
      ix←cities⍳⊂lcity              ⍝ Find city in cities
      :If ix≤⍴cities                ⍝ If present,
          r←ix⌷offsets              ⍝ return the offset
      :EndIf                        ⍝ [else return ⍬]
    ∇

:EndNamespace
```

In this example:

- The `List` function has one small amendment to the description of the Desc variable for the first user command.
- The `Run` function still uses the Cmd argument to determine which user command is being run; different actions are taken according to which is specified. If the Cmd argument is UTC then the function proceeds as before. However, if the Cmd argument is `Time` then the function now takes the second argument into account and passes it to the `CityTimeOffset` function (the `Args~' '` expression removes any extraneous spaces in the

name of the city, so that a user can enter (for example) `'l.a.'` or `'l. a.'` and get a valid result) If the `CityTimeOffset` function returns an offset value then the `Run` function uses this to calculate the time in the specified city, otherwise it generates an "Unknown city" error message.
- The `Help` function has one small amendment to state that an optional argument specifying the location can be included when running the `]Time` user command.
- The `Zulu` function remains unchanged.
- The `CityTimeOffset` function is added to determine whether the second argument matches the name of one of the cities that have had time offsets defined and return the appropriate offset if a match is found. The name of the city entered when running the user command is made case-insensitive by converting them to lower case with the `('.'⎕R'\l&')city` expression.

After changing the code but before running these user commands, the `]UReset` user command should be run to force a cache file update (otherwise the code changes will not be detected).

The `Time` and `UTC` user commands can now be run from a Dyalog Session:

```
      ]?Time
Command "Time".

Time [city]

Script location: c:\program files\dyalog\dyalog apl 14.0
unicode\salt\spice\timefns
```

(the same result is returned if `]Help Time` or `]??Time` is entered)

```
      ]Time
13:17:34
```
(indicating that the current system time is 13:17 and 34 seconds)

```
      ]Time l.a.
04:17:51
```
(indicating that the current time in Los Angeles, ignoring daylight saving time, is 04:17 and 51 seconds)

```
      ]Time l.x.
12:17:59
```
(an invalid city is specified, so the local co-ordinated universal time – based on the current system time – is returned...12:17 and 59 seconds)

```
      ]?UTC
Command "UTC".

UTC (no arguments)

Script location: c:\program files\dyalog\dyalog apl 14.0
unicode\salt\spice\timefns
```

(the same result is returned if `]Help UTC` or `]??UTC` is entered)

```
      ]UTC
06:08:30
```

(indicating that the local co-ordinated universal time is 6:08 and 30 seconds)

```
      ]?TimeGrp
 Group     Name  Description
 =====     ====  ===========
 TimeGrp   Time  Shown local time in a city
           UTC   Show UTC time
```

A.4 Example: The Parse Variable

This example illustrates use of the Parse variable; by setting this to non-empty values, the user command framework is able to handle arguments and modifiers.

For more information on the Parse variable, see Section *4.2.3*. For more information on modifiers and modifier values, see Section *4.5*. For more information on arguments, see Section *4.6*.

A new user command called `Number` is required to display either the age of the specified person or to convert a decimal number into its Hexadecimal equivalent. The necessary functions are defined in a namespace called number:

```
:Namespace number

    ⎕ML ⎕IO←1    ⍝ set to avoid inheriting external values

    ∇ r←List
      r←⎕NS¨1⍴⊂''
      r.(Group Parse Name Desc)←⊂'AgeHex' '' 'Number'
                          'Gives age or Hexadecimal format'
    ∇

    ∇ r←Run(Cmd Args);N;H;alph;Name;Names
      r←⍬
      Names←Args.Arguments
      :For Name :In Names
          :Select Name
          :Case 'Fiona'
              r,←40
          :Case 'Andy'
              r,←51
          :Else
              :Trap 6 ⍝ VALUE ERROR
                  :If ∧/Name∊⎕D   ⍝ If all digits...
                      N←⌈16⍟(⍎Name)
                      H←(N⍴16)⊤(⍎Name)
                      alph←'0123456789ABCDEF'
                      r,⊂←alph[⎕IO+H]
                  :Else
                      r,←⊂'Unrecognised Name'
                  :EndIf
              :Else
                  r,←⊂'Unrecognised Name'
              :EndTrap
          :EndSelect
      :EndFor
    ∇

    ∇ r←Help Cmd
    r←'Enter either a person''s name to return their age
        or a number to return the Hexadecimal equivalent'
    ∇

:EndNamespace
```

In this example, the Parse variable is empty – this means that the Run function's takes everything following the command name as a simple character vector. However, if a valid name is entered with the expectation of having that person's age returned, then an error message is generated:

```
      ]Number Fiona
* Command Execution Failed: SYNTAX ERROR
```

The same error message is generated if a decimal number is entered with the expectation of its Hexadecimal equivalent being returned:

```
      ]Number 42
* Command Execution Failed: SYNTAX ERROR
```

This error arises because the user command is expecting a namespace as its input and instead it is receiving a simple character vector.

These errors arise because the Args parameter in the Run function is a simple character vector rather than a namespace; this is due to the empty Parse variable. Populating the Parse variable means that the Args parameter becomes a namespace.

For this example, the only changes that will be made to the user command's code are to its Parse variable definition.

To enable the user command to perform the necessary namespace conversion, the Parse variable is changed from `''` to `'2S'` – this means that the user command can accept 0, 1 or 2 arguments but no more (for more information on this, see Section *4.6.4*).

```
      ]Number 42
 2A

      ]Number 42 42
 2A  2A

      ]Number 42 42 42
* Command Execution Failed: too many arguments

      ]Number 42 Fiona
 2A  40
```

Changing the Parse variable again, this time from `'2S'` to `'2L'`, means that 2 arguments must be supplied; if more than this are supplied then the first argument is taken as specified and the rest are merged together to become the second argument (for more information on this, see Section 4.6.5).

```
      ]Number 42
```

```
* Command Execution Failed: too few arguments

      ]Number 42 42
 2A   2A

      ]Number 42 42 42
 2A   Unrecognised Name

      ]Number 42 Fiona
 2A   40
```

A.5 Example: Debugging a User Command

This example illustrates using the]UDebug user command to debug a namespace containing a user command group definition.

A user command can be debugged by tracing through the entire namespace. However, a more convenient method is to instruct code to suspend on the first line of the Run function – tracing/debugging can then proceed from there. To do this, debugging mode must be switched on:

```
      ]UDebug on
Was OFF
```

Having debugging enabled does not impact the execution of a user command unless you specify the "-" flag at the end of the command. For example, using the number namespace defined in see Section *A.4* to hold the AgeHex group of user commands:

```
      ]Number 42 Andy
 2A   51

      ]Number 42 Andy -
Run[1]
```

The Debugger should open with the code suspended on Run[1].

To progress through the Run function, press the **<TC>** key combination.

Relevant key combinations on the Microsoft Windows operating system
<TC> is usually **Ctrl** + **Enter**
<ED> is usually **Shift** + **Enter**
<EP> is usually **Escape**

Relevant key combinations on a Linux operating system:
<TC> is usually **APLkey** + **Shift** + **Enter**
<ED> is usually **APLkey** + **Enter**
<EP> is usually **Escape**

You can now trace and debug the code in the namespace.

The debugging window shows that, in the number namespace, the Parse variable is set to `2S`. This means that the Args variable is a namespace. The namespace contains a number of variables, one of which is Arguments:

```
      ]disp Args
⎕SE.[Namespace]
      Args.⎕NL 2
Arguments
SwD
_1
_2
```

```
            ]disp Args.Arguments
┌→─┬────┐
│42│Andy│
└─→┴───→┘
```

This shows that the Arguments variable is a vector comprising two character vectors.

Press the **<ED>** key combination to open the namespace definition in the Editor and change the Parse variable from `'2S'` to `'2L'`. Save the changes and repeatedly hit **<EP>** until you are no longer tracing through code. Then enter:

```
      ]Number 42 Andy 8 9 10 -
```

With the `Run` function suspended, enter:

```
      ]Disp Args.Arguments
┌→─┬─────────┐
│42│Andy 8 9 10│
└─→┴────────→┘
```

This shows that the Arguments variable is still a vector comprising two character vectors. However, the second of the two character vectors now includes everything after the first argument in the call to the user command.

Press the **<ED>** key combination to open the namespace definition in the Editor and change the Parse variable from `'2L'` to `'2S -true'`. The `'-true'` means that the parser now accepts a modifier called `-true` that does not accept a modifier value but can only be present or absent (see Section *3.7.2*). Save the changes and repeatedly hit **<EP>** until you are no longer tracing through code. Then enter:

```
      ]Number 42 Andy -
      Args.⎕NL 2
Arguments
SwD
_1
_2
true
```

This shows an additional variable, `true`, created with the same name as the modifier that was included in the Parse variable. However, when calling the Number user command, this on/off modifier was not specified. Therefore:

```
      Args.true
0
```

To see the effect of calling the Number user command with this modifier specified:

```
      )reset
      ]number 42 Andy -true -
      Args.true
1
```

Press the **<ED>** key combination to open the namespace definition in the Editor and change the Parse variable from `'2S -true'` to `''`. Save the changes and repeatedly hit **<EP>** until you are no longer tracing through code. Then enter:

```
      ]number 42 Andy -
```

With the `Run` function suspended, enter:

```
      ]disp Args
42 Andy

      ⍴Args
8
```

With an empty Parse variable, `Args` is a simple character vector of length 8 (because we have used the "`-`" argument, there is a trailing space after the "`y`" of "`Andy`").

Debugging mode is switched off using:

```
      ]UDebug off
Was ON
```