User Commands
User Guide

User Commands version 2.8



DYALOG

The tool of thought for software solutions

Dyalog is a trademark of Dyalog Limited
Copyright © 1982-2026 by Dyalog Limited
All rights reserved.

User Commands User Guide

User Commands version 2.8
Document Revision: 20260907_280

Unless stated otherwise, all examples in this document assume that `⎕IO ⎕ML ← 1`

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

email: support@dyalog.com
https://www.dyalog.com

TRADEMARKS:

Raspberry Pi is a trademark of the Raspberry Pi Foundation.
Oracle® , MySQL, and Java™ are registered trademarks of Oracle and/or its affiliates.
JavaScript™ is a trademark of Oracle Corporation.
Unicode is a registered trademarks of Unicode, Inc. in the United States and other countries.
UNIX® is a registered trademark in the U.S. and other countries, licensed exclusively through X/Open Company Limited.
Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.
Windows® is a registered trademark of Microsoft Corporation in the U.S. and other countries.
macOS® and OS X® (operating system software) are registered trademarks of Apple Inc. in the U.S. and other countries.

All other trademarks and copyrights are acknowledged.

Contents

1 About This Document **1**
1.1 Audience 1
1.2 Conventions 1
2 Introduction **3**
2.1 Cache File 3
2.1.1 Defining the UCMDCACHEFILE Environment Variable 4
3 Using User Commands **6**
3.1 Installation 6
3.2 Directory Structure 6
3.3 Implementation 6
3.3.1 Customising the Implementation 7
3.4 File Format 8
3.5 Groups 8
3.6 Syntax in Dyalog Sessions 9
3.7 Running User Commands 10
3.7.1 Arguments 10
3.7.2 Modifiers and Modifier Values 11
3.7.3 Errors when Running a User Command 11
4 Creating User Commands **12**
4.1 Basic Definition 12
4.2 The List Function 13
4.2.1 Name 14
4.2.2 Group 14
4.2.3 Parse 15
4.3 The Run Function 16
4.3.1 Defining Multiple Levels of Help 16
4.4 The Help Function 18
4.5 Modifiers 19
4.5.1 Default Modifier Values 20
4.6 Arguments 21
4.6.1 Default Argument Values 21
4.6.2 Arguments Including Space Characters 22
4.6.3 Minimum Number of Arguments 22
4.6.4 Maximum Number of Arguments 22
4.6.5 Long Arguments 22
4.6.6 Summary of Argument Specification in the Parser 23
4.7 Saving Custom User Commands 23
4.8 Detecting New Custom User Commands 24
A SAMPLES Group **26**

A.1]UCMDHelp 26
A.2]UCMDNoParsing 26
A.3]UCMDParsing 27
B Example User Commands 28
B.1 Example: Basic User Command Definition 28
B.2 Example: Cross-Operating System Definition 29
B.3 Example: Optional Arguments 32
B.4 Example: The Parse Variable 35
B.5 Example: Debugging a User Command 37
Index 41

1 About This Document

This document introduces user commands and describes how to create/implement new user commands.

1.1 Audience

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

For information on the resources available to help develop your Dyalog knowledge, see https://www.dyalog.com/introduction.htm.

1.2 Conventions

Unless explicitly stated otherwise, all examples in Dyalog documentation assume that ⎕IO and ⎕ML are both 1.

Various icons are used in this document to emphasise specific material.

General note icons, and the type of material that they are used to emphasise, include:


Hints, tips, best practice and recommendations from Dyalog Ltd.


Material of particular significance or relevance.


Legacy information pertaining to behaviour in earlier releases of Dyalog or to functionality that still exists but has been superseded and is no longer recommended.


Warnings about actions that can impact the behaviour of Dyalog or have unforeseen consequences.

A full list of the platforms on which Dyalog version 21.0 is supported is available at https://www.dyalog.com/dyalog/current-platforms.htm. Although the Dyalog programming language is identical on all platforms, differences do exist in the way some functionality is implemented and in the tools and interfaces that are available. Differences in behaviour between operating systems are identified with the following icons (representing macOS, Linux, Microsoft Windows and UNIX respectively):



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.

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
```

2.1 Cache File

The first time that you start a Dyalog Session after installing 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 or a new file containing user commands is created, 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).
- if a user attempts to run (or get help about) a user command that is not in the cache.

The cache file is also rebuilt if a user command is called after updating the *cmddir* global parameter (using the `]Settings` user command – for more information enter `]Settings cmddir -?` in a Session).

The default name for the cache file uses the following syntax: **UserCommand <UcmdMajor><UcmdMinor>.<DyalogMajor><DyalogMinor><U|C><bits>.cache**. For example, the cache file for the user command framework v2.8, which accompanies Dyalog v21.0, on a 64-bit Unicode system, would be **UserCommand28.210U64.cache**.



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



By default, the cache file is located in **$HOME/.dyalog/**



By default, the cache file is located in **Users/<name>/.dyalog/**

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

2.1.1 Defining the UCMDCACHEFILE Environment Variable

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

Defining an environment variable is operating system-specific.

To define the UCMDCACHEFILE environment variable on Microsoft Windows (permanent method)

1. Open the **Control Panel** and click on the **System** icon.

   The **System** window is displayed.
2. In the **Control Panel Home** pane, click **Advanced system settings**.

   The **System Properties** window is displayed.
3. Navigate to the **Advanced** tab of the **System Properties** window.
4. Click **Environment Variables...**.

   The **Environment Variables** dialog box is displayed.
5. In the **User variables for <user>** pane, click **New...**.

   The **New User Variable** dialog box is displayed.
6. In the **Variable name** field, enter UCMDCACHEFILE.
7. In the **Variable value** field, enter **<full path>\<cache file name>** of the user commands cache file.

8. Click **OK** to create the new environment variable and exit the **New User Variable** dialog box.
9. Click **OK** to exit the **Environment Variables** dialog box.
10. Click **OK** to exit the **System Properties** window.
11. Close the **System** window.

To define the UCMDCACHEFILE environment variable on Microsoft Windows (temporary method – for session duration only)

1. Open the **cmd.exe** application.
2. At the command prompt, enter:
   ```
   dyalog.exe UCMDCACHEFILE=[UCMDCACHEFILE]
   ```
   where `[UCMDCACHEFILE]` is the new **<full path>\<cache file name>** of the user commands cache file.

To define the UCMDCACHEFILE environment variable on Linux and macOS (permanent method)

1. Open the **$HOME/.dyalog/dyalog.config** file in your preferred text editor.
2. Add the following:
   ```
   export UCMDCACHEFILE=[UCMDCACHEFILE]
   ```
   where `[UCMDCACHEFILE]` is the new **<full path>/<cache file name>** of the user commands cache file.

To define the UCMDCACHEFILE environment variable on Linux (temporary method – for session duration only)

1. Open a shell.
2. At the command prompt, enter:
   ```
   UCMDCACHEFILE=[UCMDCACHEFILE] dyalog
   ```
   where `[UCMDCACHEFILE]` is the new **<full path>/<cache file name>** of the user commands cache file.

---

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.



Updates to the set of predefined user commands can be downloaded from https://www.dyalog.com/tools/user-commands.htm.

3.2 Directory Structure

The **[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 *SALT User 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 looks for the 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.
- if this function does not exist, then user commands are disabled.

This implementation means that application code can invoke user commands by calling ⎕SE.UCMD directly.



Dyalog Ltd reserves the right to change the implementation of the user command framework.

EXAMPLE

The following command is entered while in a namespace:

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

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

```
⎕THIS ⎕SE.UCMD '<ucmd> <-myModifier>=<value>'
```

⎕SE.UCMD converts this into a call to the user command framework; the functions defined for `<ucmd>` are actioned with the `<-myModifier>` modifier applied with a value of `<value>` and the result is displayed in the Session.

EXAMPLE

The result of `<ucmd>` is assigned to a variable called `<variable>`:

```
]<variable>←<ucmd> <-myModifier>=<value>
```

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

```
⎕THIS ⎕SE.UCMD '<variable>←<ucmd> <-myModifier>=<value>'
```

⎕SE.UCMD converts this into a call to the user command framework; the functions defined for `<ucmd>` are actioned with the `<-myModifier>` modifier applied with a value of `<value>` and the result is assigned to `<variable>`.

If `<variable>` was not included then the result of `<ucmd>` would be discarded and not shown in the Session, although any non-result output generated would be displayed.

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 Programming Reference Guide*) and must be stored as files with the **.dyalog** extension.



If an extension is not specified when using the `]Snap` or `]Save` user commands to save a script file, then **.dyalog** is automatically appended.



By default, double-clicking on a **.dyalog** file opens that file using the standalone editor.

Files with the **.dyalog** extension are Unicode text files. 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, the predefined user command `]FILE.Compare` compares two files, `]ARRAY.Compare` compares two arrays and `]FN.Compare` compares two functions.



Although a user command can have the same name as its group (or another group), Dyalog Ltd does not recommend this as it can introduce ambiguity to a user reading the code.

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: `]<ucmd>...`
- to display general help information: `]`
- to list all user commands in their groups (without descriptions): `]-?`
- to list all user commands in their groups (with descriptions): `]-??`
- to list all the available commands in a specific group: `]<groupname> -?`
- to display information for a specific user command: `]<ucmd> -?`
- to list all the available user commands defined in **.dyalog** files in a specific directory:
  `]<full path to directory>/<directory name> -?`
- to list all user commands or groups that match pattern `X*YZ*`(`*` is a wildcard):
  `]X*YZ* -?`
- to assign the result of a user command to a variable: `]<var>←<ucmd>...`
- to discard the result of a user command: `]←<ucmd>...`

Multiple levels of help can be defined for each user command; the information that is returned is dependent on the level of help requested. The level is defined to be 1 less than the number of `?` characters entered after the `-` character. For example:

- level 0: `]<ucmd> -?` or `]Help ]<ucmd>`
- level 1: `]<ucmd> -??` or `]Help ]<ucmd> -page=2`
- level 2: `]<ucmd> -???` or `]Help ]<ucmd> -page=3`

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 4.3.1*).



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 UPPERCASE and user command names are shown in Upper CamelCase.

3.7 Running User Commands

User commands are run with the following syntax:

```
]<ucmd> <-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 `]<ucmd> -?` or `]Help ]<ucmd>` 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 text are presented as selectable options, making it easy to correctly specify the requisite user command. (Auto-completion is not available in the TTY version of Dyalog.)



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 `]<ucmd> -?` or `]Help ]<ucmd>` 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).

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.

3.7.3 Errors when Running a User Command

The `]UDebug` user command facilitates the debugging of user commands – switching this on (`]UDebug on`) enables suspension inside a user command's execution.

If an error is generated when running a user command, then `⎕DMX` is cloned to `⎕SE.SALTUtils.dmx`; this means that information pertaining to the error is retained even after the user command framework clears `⎕DMX`.

EXAMPLE

This example shows that information pertaining to the length error is not available from `⎕DMX` but can be retrieved from `⎕SE.SALTUtils.dmx`:

```
      ]Disp 1 2+3 4 5
* Command Execution Failed: LENGTH ERROR

      ⎕DMX

      ⎕SE.SALTUtils.dmx
EM        LENGTH ERROR
Message   Mismatched left and right argument shapes
```

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. 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 several ways, for example:

- 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.
- in a Dyalog Session using the `]UNew` user command – for more information enter `]UNew -?` in a Session.

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



The script for Dyalog's predefined user commands can be a useful starting point when creating a new user command. The location of an existing user command's script can be found in the following ways:

- `]UVersion <ucmd>` returns the script location for the specified user command
- `]ULoad <ucmd>` loads the script for the specified user command into the active workspace and returns the script location.
- `]<ucmd> -?` returns the script location for the specified user command if `]UDebug` is `on`.

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, but must only have one instance of each of the three functions irrespective of how many user commands it contains. (Although a class can be used instead of a namespace, a namespace is the recommended approach.)



See *Appendix A* for some sample user commands that demonstrate the use of multiple levels of help and parsing user command lines. See *Appendix B* for some examples of user commands wrapped in a namespace – 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 (`] -?`), optionally with descriptions (`] -??`) .

The `List` function is niladic and 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 Programming 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; Dyalog recommends using lowercase characters only.

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 (however, custom user commands should not be given the same name as any of the predefined user commands within the SALT group).



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 LINK, 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 and 0 or more modifiers
- individual arguments and modifiers are separated by space characters
- arguments and modifiers can be specified in any order
- arguments can be optional or mandatory
- modifiers are identified by a preceding `-` character
- modifier values are identified by a preceding `=` character
- modifier names are case-sensitive
- individual arguments and modifier values can be delimited by single or double quotes to allow leading/trailing/internal space characters or to allow arguments that have a leading - character.

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 manipulation tools 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 character vector 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 list of modifiers must include, for each modifier, its name, whether it accepts a value and, optionally, any restrictions for that value.

4.3 The Run Function

The `Run` function executes the code for the command. It is always called monadically with a two-element vector argument; 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.

4.3.1 Defining Multiple Levels of Help



See *Appendix A* for some sample user commands that demonstrate the use of multiple levels of help.

The specific defined help information that is presented to a user when requesting help in a Dyalog Session is dependent on the level of help requested. This level is defined to be 1 less than the number of `?` characters entered after the `-` character; for example, `]<ucmd> -??` returns the information defined for level 1 of the `<ucmd>` 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 `]<ucmd> -??` request should state that more detailed information is available if `]<ucmd> -???` 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, `]<ucmd> -???` returns the help information for levels 0 and 1 as well as the help for level 2.

The following code fragment is an example showing how separate (non-cumulative) levels of help can be defined within the `Help` function:

```
∇ r←level Help Cmd
    :Select level
    :Case 0
        r←⊂'This is basic help.'
    :Case 1
        r←⊂'This is level 1 help.'
    :Case 2
        r←⊂'This is level 2 help.'
    :Else
```

```
        r←⊂'This is level 3 help.'
    :EndSelect
∇
```

In this case:

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



The `:Else` control structure in the code fragment ensures that requests for higher levels of help than are defined return the highest-defined level rather than generating an error message.

The following code fragment is an example showing how cumulative levels of help can be defined within the `Help` function:

```
∇ r←level Help Cmd
  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+level)⌊≢r)↑r
∇
```

In these cases:

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



Entering `]Help ]<ucmd>` in a Dyalog Session always presents the user with the same level of help as `]<ucmd> -?` even if there are multiple levels of help defined.

4.4 The Help Function

The `Help` function reports detailed information on the user command when this is requested (by entering `]<ucmd> -?` or `]Help ]<ucmd>` in a Dyalog Session). It is called dyadically; the left argument is the level and the right argument is the name of the user command.

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:

```
]<GROUPNAME>.<commandname>

<specific defined help information>
```

If `]UDebug` is `on`, then the `Help` function returns an enhanced set of information by default:

```
]<GROUPNAME>.<commandname>

Source: <location of the user command's script file>
Version: <version number of the user command>
Syntax: <number of arguments> only if arguments can be specified
Accepts modifiers <list of all modifiers> only if modifiers can be
specified
   <modifier restrictions> only if modifiers exist that have restrictions

<specific defined help information>
```

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 `ϵ` character as a separator. For example:

  `-<modifier name>ϵ<set of characters>`

  so `-XYZϵabc012` 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 character vectors, then the `Parse` variable includes the modifier and possible values with the `=` character as a separator and the character vectors separated by space characters. For example:

  `-<modifier name>=<charvec1> <charvec2> <charvec3>`

  so `-XYZ=abc 012` means that the modifier `-XYZ` can accept either `abc` or `012` as a modifier value.

- If a modifier accepts any character vector, then the `Parse` variable includes the modifier and a `=` character with nothing after it. For example:

  `-<modifier name>=`

  so `-XYZ=` means that the modifier `-XYZ` can accept any value.

For each of these three rules, enclosing the separator character within square brackets means that specification of modifier values is optional. For example,`-XYZ[=]` means that the modifier `-XYZ` can be specified without a value but will 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 character vector 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'
```

With this approach, the default value is reported only if the modifier is not used; a value of 1 is reported if the modifier is used but no value is specified.

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 `Switch` function (automatically defined in the namespace that is passed to the `Run` function (see *Section 4.3*) as its second argument).

Given the name of a modifier as a right 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 character vector matching the specified modifier value
  - a character vector matching the default modifier value if a modifier is not specified but 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 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 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.

EXAMPLE

To set a default value of '*defaultfor4th*' for the 4th argument:

```
args←a.Arguments,(⍴a.Arguments)↓0 0 0 'defaultfor4th'
```

where `a` is the second argument supplied to the `Run` function, that is the arguments/modifiers supplied to the user command (see *Section 4.3*). In this example, the first three arguments have their default values set to 0 if they are optional arguments; if they are mandatory then any value specified here is ignored.

4.6.2 Arguments Including Space Characters

Arguments that contain space characters must be delimited with ' or " 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 is run as follows:

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

If the user command]NewID accepts 3 arguments, *firstname*, *surname* and *address*, then Parse should be set to '3' and the user command is 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 – any additional arguments after the maximum number is reached are merged into the last argument. 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'.

The long argument `L` can be appended to the maximum number of arguments `S` (see *Section 4.6.4*) 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:

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

runs the user command `cmd` 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_4`S` : a maximum of n_4 arguments can be supplied (equivalent to `0-`n_4)
- n_5`L` : 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_7`L` : 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_8`SL` : 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`)
- n_9`-`n_{10}`L` : a minimum of n_9 arguments and a maximum of n_{10} arguments can be supplied; if more than this are supplied then the first n_{10}`-1` arguments are taken and the rest are merged together into the final n_{10} argument (equivalent to `0-`n_{10}`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 namespace in a Dyalog Session, then it can be saved as a **.dyalog** file using the `]Save` user command).

The predefined user commands are located in the **[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, use the `]Settings` user command to set the *cmddir* global parameter to the full path and name of the directory (for more information enter `]Settings -?` in a Session). The new directory is added to the start of the list of directories, making it the first one searched.



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



If the *cmddir* global parameter includes multiple directories, then the user command framework searches the directories in the order listed (starting from the left) and retrieves the first user command it finds with the specified name. To see the list of directories (and the order in which they are searched), enter `]Settings cmddir`.

If the `]UNew` user command is used to create and save a new user command, then its location is automatically added to the list of directories searched.

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 user command directory(s) is scanned to locate new user commands that have been manually 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.

In addition, new user commands that are placed in the **MyUCMDs** directory are automatically active without needing to specify `]Settings cmddir -permanent`.



On the Linux operating system, the **MyUCMDs** directory is located directly under the **$HOME** directory.



On the macOS operating system, the **MyUCMDs** directory is located directly under the **$HOME** directory.



On the Microsoft Windows operating system, the **MyUCMDs** directory is located directly under the **%USERPROFILE%\Documents** directory.

A 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.

A.1]UCMDHelp

An example of a custom user command that defines multiple levels of help information in the Help function, selectable by the number of question marks supplied by the user, for example, `]<ucmd> -???`.

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

```
      ]ULoad UCMDHelp
Namespace #.HelpExample now contains source for
]SAMPLES.UCMDHelp from <full path>\SALT\study\aSampleHelp.dyalog

      )ED HelpExample
```

A.2]UCMDNoParsing

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

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

```
      ]ULoad UCMDNoParsing
Namespace #.anyname now contains source for
]SAMPLES.UCMDNoParsing from <full
path>\SALT\study\aSample.dyalog

      )ED anyname
```

A.3]UCMDParsing

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

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

```
      ]ULoad UCMDParsing
Namespace #.anyname now contains source for ]SAMPLES.UCMDParsing
from <full path>\SALT\study\aSample.dyalog

      )ED anyname
```

B 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.

B.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 `''` 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 that user command.

Running this user command in a Dyalog Session returns three numbers; these three numbers are the current time, indicating the hour (according to the 24 hour clock), the number of minutes past the hour and the number of seconds elapsed respectively.

```
      ]Time -?
```

```
]TIMEGRP.Time

]Time (no arguments)
```

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

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

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

B.2 Example: Cross-Operating System 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 B.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 UNIX.

```
: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'  ⍝ If Windows
        ⎕USING←'System'
        dt←DateTime.Now
          :If 'UTC'≡Cmd
            dt←Zulu dt
          :EndIf
        r←(r⍳' ')↓r←⍕dt
      :Else                                   ⍝ If not Windows
        dt←('UTC'≡Cmd)/'TZ=UTC'               ⍝ set timezone
        r←⊃⎕SH dt,' date +"%H:%M:%S"'         ⍝ and get the time

      :EndIf
    ∇

    ∇ r←Help Cmd;which
      which←'Time' 'UTC'⍳⊂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:
  - `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)
  - `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)
  - `Group` is set to `⊂TimeGrp` (only one value so applied to both user commands)
  - `Parse` is set to `''` (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; this determines whether to use the current system time or 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 -?

 TIMEGRP:
  Time   Show local time in a city
  UTC    Show UTC time

      ]Time -?
─────────────────────────────────────────────────────────────────

]TIMEGRP.Time
```

```
]Time (no arguments)
```

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

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

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

```
      ]UTC -?
```

```
]TIMEGRP.UTC

]UTC (no arguments)
```

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

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

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

B.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 B.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 (example assumes the Microsoft Windows operating system 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⍴⍨⍬≡offset
          diff←⎕NEW TimeSpan(3↑offset)
          dt←(Zulu dt)+diff
        :EndIf
      :EndSelect
      r←(rι' ')↓r←⍕dt
    ∇

    ∇ r←Help Cmd;which
      which←'Time' 'UTC'ι⊂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 1 10
      r←⍬                          ⍝ Assume no match
      lcity←⎕C city                ⍝ Case-fold name
      ix←citiesι⊂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 case-folding it with the ⎕C 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 -?
```

```
]TIMEGRP.Time

]Time [city]
```

(the same result is returned if `]Time -??` or `]Help ]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)_819

```
      ]Time l.x.
* Command Execution Failed: Unknown city
```

(an invalid city was specified)

```
      ]UTC -?
```

```
]TIMEGRP.UTC

]UTC (no arguments)
```

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

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

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

```
      ]TimeGrp -?

 TIMEGRP:
  Time   Show local time in a city
  UTC    Show UTC time
```

B.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←0
      Names←Args.Arguments
      :For Name :In Names
        :Select Name
```

```
        :Case 'Fiona'
          r,←40
        :Case 'Andy'
          r,←51
        :Else
          :If ∧/Name∈⎕D   ⍝ If all digits...
            N←⌈16⍟(⍎Name)
            H←(N⍴16)⊤(⍎Name)
            alph←'0123456789ABCDEF'
            r,←⊂alph[⎕IO+H]
          :Else
            r,←⊂'Unrecognised Name'
          :EndIf
        :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 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
```

B.5 Example: Debugging a User Command

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

Three keyboard shortcuts for command codes are referred to in this example – **<TC>** (*Trace*), **<ED>** (*Edit*) and **<EP>** (*Escape*). The usual key combinations for these are operating-system-dependent.

OS WIN Relevant key combinations on the Microsoft Windows operating system:

- **<TC>** is usually **Ctrl** + **Enter**
- **<ED>** is usually **Shift** + **Enter**
- **<EP>** is usually **Escape**

OS UNIX Relevant key combinations on the UNIX operating system:

- **<TC>** is usually **APLKey** +**Shift** + **Enter**
- **<ED>** is usually **APLKey** + **Enter**
- **<EP>** is usually **Escape**

OS MAC Relevant key combinations on the macOS operating system:

- **<TC>** is usually **Ctrl** + **Enter**
- **<ED>** is usually **Shift** + **Enter**
- **<EP>** is usually **Escape**

A user command can be debugged by tracing through ⎕SE.UCMD (see *Section 3.3*). However, a more convenient method is to instruct the framework to suspend on the first line of the Run or Help function – tracing/debugging can then proceed from there. To do this, debugging mode must be switched on:

```
      ]UDebug on
Was OFF
```

If an error is encountered in debugging mode, execution of the user command is suspended rather than returning to the calling function.

When debugging is enabled, specifying a space followed by the - character at the end of the command opens the **Trace** window with the code suspended on Run[1]. For example, using the number namespace defined in *Section B.4* to hold the AgeHex group of user commands:

```
      ]Number 42 Andy
2A  51

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

To progress through the Run function, enter the *Trace* command (**<TC>**).

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

The **Trace** 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.

Enter the *Edit* command (**<ED>**) to open the namespace definition in the **Edit** window and change the `Parse` variable from `'2S'` to `'2L'`. Save the changes and repeatedly enter the *Escape* command (**<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 **Edit** window 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
```

Debugging mode is switched off using:

```
      ]UDebug off
Was ON
```

Index

A

Arguments 10, 21
  Default values 21
  Including space characters 22
  Long 22
  Maximum number of 22
  Minimum number of 22
  Specification in the parser 23

B

Basic Definition 12

C

Cache file 3
Creating custom user commands 12
  Argument definition 21
  Basic Definition 12
  Detecting new user commands 24
  Help function 18
  List function 13
  Modifier definition
    Default values 20
  Run function 16
  Saving custom user commands 23

D

Detecting new user commands .. 24
Directory structure 6

E

Environment Variables
  UCMDCACHEFILE 4

F

File format 8
Flags See *Modifiers*

G

Groups 8

H

Help function 18
  Defining multiple levels of help 16

I

Implementation 6
Installation 6

L

List function 13
  Group variable 14
  Name variable 14
  Parse variable 15

M

Modifiers 19
  Default values 20

Syntax 11

P

Parse variable 15

R

Run function 16
Running user commands 10

S

Saving custom user commands .. 23
Switch function 20
Switches See *Modifiers*
Syntax 9

U

UCMDCACHEFILE Environment Variable 4
User command groups (predefined)
SAMPLES 26
User commands (predefined)
]UCMDHelp 26
]UCMDNoParsing 26
]UCMDParsing 27