DYALOG

The tool of thought for expert programming

Dyalog™ for Windows

User Commands

for Dyalog APL V13.2

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

DYALOG

Table of Contents

Introduction

Version 12.1 introduced "User Commands" to Dyalog APL. Like system commands, *user commands* are tools which are available to developers at any time, in any workspace – as part of the development environment. Unlike system commands, user commands are written in APL. Dyalog APL is shipped with a set of user commands, with APL source code that you can inspect and modify – or use as the basis for writing completely new user commands of your own. User commands are intended to make it easy to write and share development tools. A section of the APL Wiki, `http://aplwiki.com/UserCmdsDyalog`, is devoted to sharing user commands.

If an input line begins with a closing square bracket "`]`", the system will interpret the line as a user command, temporarily load the required code into the session namespace where it cannot conflict with any code in the active workspace, and execute it. For example:

```
      )load util
util saved …
      ]fns S*
SET    SETMON SETWX  SM_TS  SNAP
```

Help is easily accessible for user commands:

```
      ]?fns
Command "fnsLike"
Syntax:  accepts switches -format -regex -date=

Arg: pattern; Produces a list of fns & ops whose names match
the pattern given
-format   Return result as )FNS would (*)
-regex    uses full regular expression
-date     takes a YYMMDD value preceded by > or <
(*) -format is implied if the command was entered on the
command line and the result was not captured

Script location: C:\ProgramFiles\D121U\SALT\Spice\wsutils
```

As we can see above, the full name of the command is **`fnslike`**, but unambiguous abbreviations are allowed. The source code is in a file called **`wsutils.dyalog`** in the folder which is identified in the above output. New user commands can be installed simply by dropping new source code files into the command folder, making them instantly accessible without restarting any part of the system. A full list of installed user commands is available at any time:

```
       ]?
75 commands:

aplmon  calendar cd      commentalign    cfcompare    compare
…
varslike         wscompare       wsloc   wspeek  xref

Type "]?+" for a summary or "]??" for general help or "]?CMD"
for info on command CMD
```

Implementation

When an input line begins with a closing square bracket, the system will look for a function named ⎕SE.UCMD and – if it exists – call this function passing the rest of the input line as the right argument. The default session files (.DSE) all contain a function which passes the command to the *Spice* command processor, which is based on the simple tool for managing APL code in Unicode text files known as *SALT*. *SALT* and *Spice* were introduced with version 11.0. As a result any *Spice* commands that you may have developed are now available as *user commands* since version 12.1.

You *can* write your own user command implementation by redefining ⎕SE.UCMD, but Dyalog recommends that you refrain from doing so, in order to promote a single user command format that allows all user commands to be shared. If you use the *Spice* framework, this will also allow the use of any user commands that you develop with versions 11.0 and 12.0 via the "*Spice* command line" (see *Help* | *Documentation Centre* for more information about *SALT* and *Spice*).

In the longer term, Dyalog aims to add the ability to load, edit and save APL source code held in Unicode files into the interpreter itself. Through *SALT* and *Spice*, user commands are thus built on the framework which is likely to become the recommended mode of development in the future.

Dyalog's user commands are similar in concept to those implemented in other APL systems in the past – but the text based implementation is intended to allow much easier sharing of development tools.

Using User Commands

All *user commands* are entered in the session starting with a right bracket, in the same way that *system commands* start with a right parenthesis.

To execute command **xyz** type `]xyz`

To find all available commands type `]?`

To get a summarized list of all commands type `]?+`

To get more general help type `]??` or `]help`

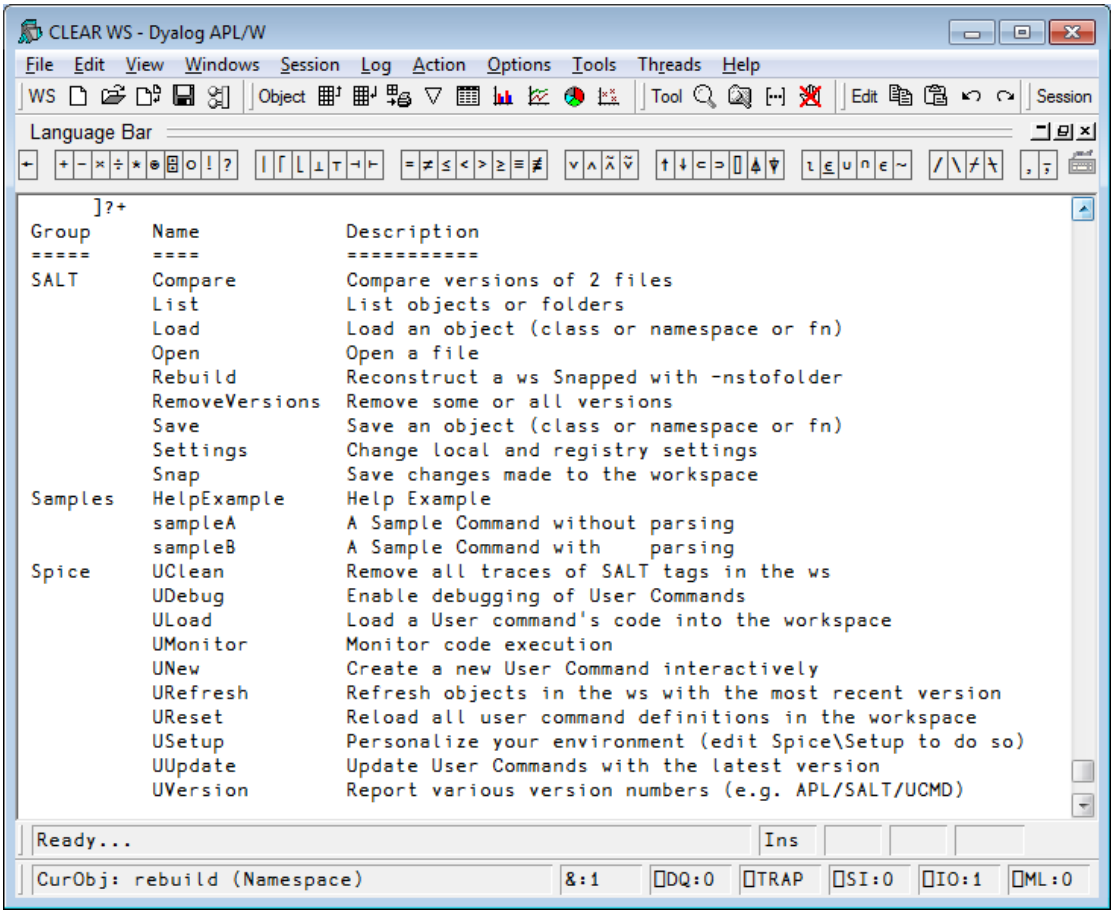
To find all the available commands in a specific folder type `]? \folder\name`

To get info on command XYZ type `]?xyz` or `]help xyz`

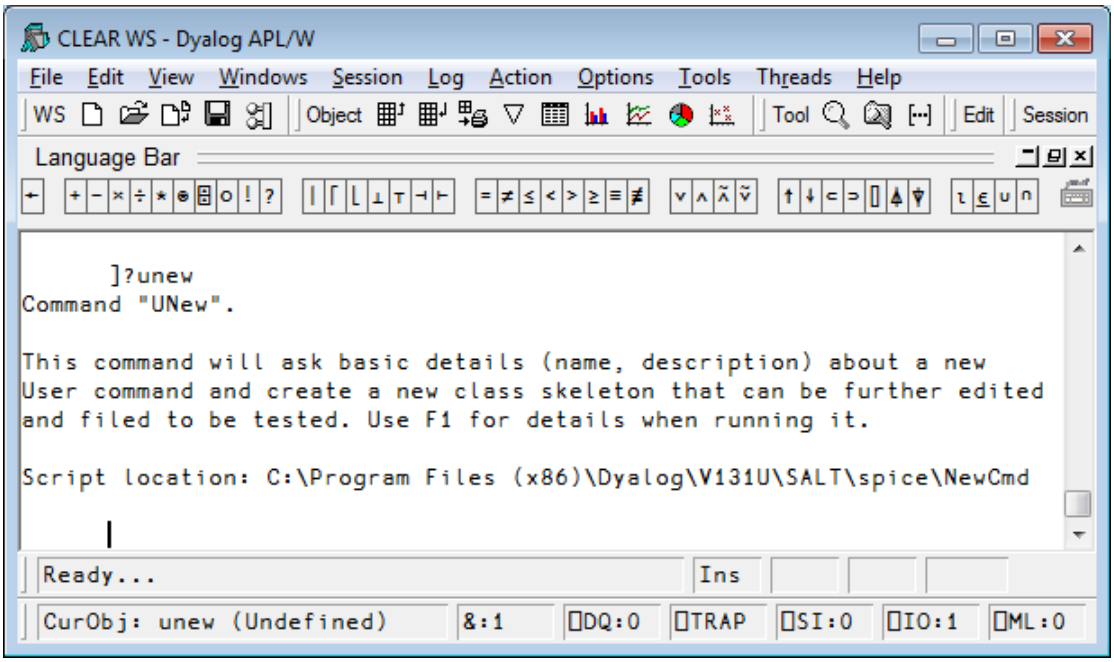
To get detailed help/info on command XYZ type `]??xyz`

To assign the result of a command to a variable type `]nl←cmdx …`

Example:



To view help on a particular command type `]?cmdname` . For example, to find help on command '`UNew`':



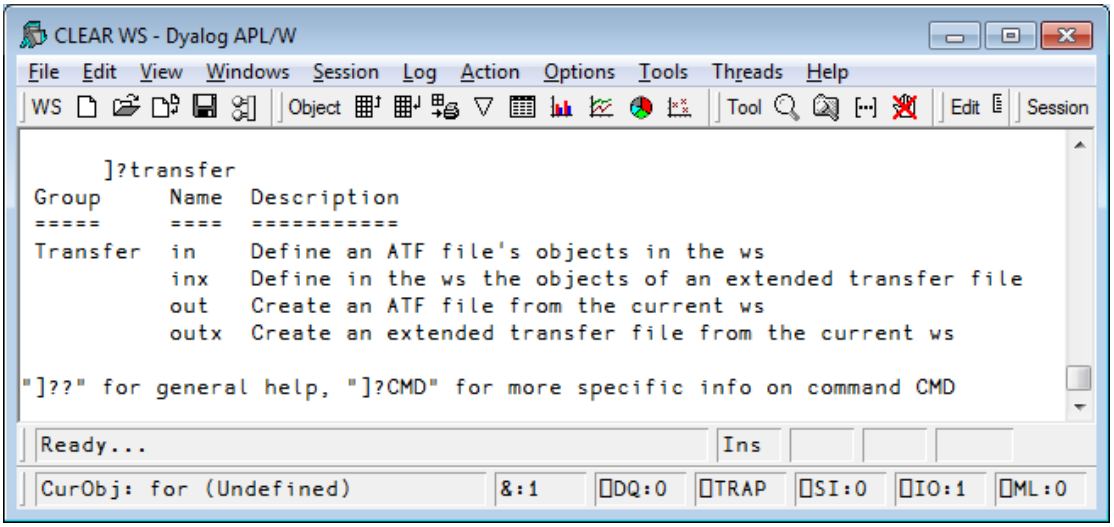
The names of commands are case insensitive, so *UNew* and *unew* are the same command.

Upon hitting Enter, the line is sent to the user command processor which determines which command has been selected, brings in the code, runs it, and then cleans up.

Groups

Commands with common features can be regrouped under a single name. To find all the commands related to a particular group type **`]?grpname`**

For example, to list all the commands in the **`transfer`** group:



Creating Commands

A user command is implemented as a namespace or class saved in a Unicode text file with the extension .dyalog. You just need to write three simple APL functions called `List`, `Help` and `Run` (plus any additional functions that you need for your implementation) - and there are worked examples which show you how to wrap it all up as a class. You do NOT need to use a class but if you decide to do so you have to make sure the above functions are public. A single namespace/class can host as many commands as you like, so you can get away with doing it once only if you prefer. The following examples will act as a tutorial and hopefully get you started as an implementer of user commands.

Examples

Example #1: The TIME command

Here is a very simple example: A user command that will show us the current time.

Here is the code required to implement our first user command:

```
:Namespace timefns
    ⎕ML ⎕IO←1 ⍝ always set to avoid inheriting external values

    ∇ r←List
      r←⎕NS¨1⍴⊂''
      r.(Group Parse Name)←⊂'TimeGrp' '' 'Time'
      r[1].Desc←'Time example Script'
    ∇

    ∇ r←Run(cmd argz)
      r←⎕TS[4 5 6]   ⍝ show time
    ∇

    ∇ r←Help Cmd
      r←'Time (no arguments)'
    ∇
:EndNamespace
```

The `List` function is used to tell the user command framework about the command itself. This allows it to display a little summary when the user types '`]?+`' to list user commands. `List` returns one namespace per command. Each namespace contains 4 variables. The summary is stored in `Desc`. Three more variables must be set: the command `Name`, the `Group` that the command belongs to and the `Parse` information for optional arguments. We'll get to parsing in a bit.

The `Help` function is used to report more detailed information when you type `]?time`. Since the class may harbour more than one command, the functions `Help` and `Run` both take the command name as an argument to decide what to do. Here there is only one command and the argument will always be 'Time' so we ignore it and always return something (e.g. help) for that command.

The `Run` function is the one executing your code for the command. It is always called with 2 arguments: the command name and the rest of the line after it. Here we ignore both as all we do is call `⎕TS`.

We can write this code in a file named `timefns.dyalog` using Notepad and put it in the `SALT\spice` folder or write it in APL and use the `]save` command[1] to put it there.

Once in the *spice* folder (the default location for user commands), it is available for use. All we need to do is type `]time`. Et voilà! The current time appears in the session as 3 numbers[2].

Example #2: Another command in the same class: UTC

We may want to have another command to display the current UTC time instead of the current local time. Since this new command is related to our first 'Time' command, we could – and should – put the new code in the same namespace, adding a new function `Zulu`[3] and modifying `Run`, `List` & `Help` accordingly. Like this:

```
:Namespace timefns
    ⎕ML ⎕IO←1

    ∇ r←List
      r←⎕NS¨2⍴⊂''
      r.(Group Parse)←⊂'TimeGrp' ''
      r.Name←'Time' 'UTC'
      r.Desc←'Shown local time' 'Show UTC time'
    ∇

    ∇ r←Run(Cmd argz);dt
      ⎕USING←'System'
      dt←DateTime.Now
      :If 'UTC'≡Cmd
            dt←Zulu dt
      :EndIf
      r←(⍕dt)↓⍨r←⍳⍨r=' ' ⍝ remove date
    ∇

    ∇ 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
```

[1] `⎕SE.SALT.Save 'timefns spice\timefns'` can also be used.
[2] This requires SALT version 2.22 or later. If you are not using Dyalog v13.1 or later, check which version you are using by typing `⎕SE.SALT.Version` and download a new one if necessary.
[3] UTC is sometimes denoted as **Z** time – Zero offset zone time – or **Zulu** time from the NATO phonetic alphabet

```
      r←TimeZone.CurrentTimeZone.ToUniversalTime date
    ∇
:EndNamespace
```

The `List` function now accounts for the '**UTC**' command and returns a list of 2 namespaces so `]?+` will now return info for both commands. Same for `Help` which makes use of its argument to return the proper help information.

The `Run` function now makes use of the `Cmd` argument and, if it is 'UTC', calls the `Zulu` function. It still does not use the 2nd argument, it ignores it. It then returns the data nicely formatted, an improvement over the previous code.

Example #3: Time in Cities around the world

We could then add a new function to tell the time in Paris, another one for Toronto, etc. Each time we would have to modify the 3 shared functions above, OR, we could have a single function that takes an argument (the location) and computes the time accordingly[4]. Like this:

```
:Namespace timefns

    ⎕ML ⎕IO←1

    ∇ r←List
      r←⎕NS¨2⍴⊂''
      r.(Group Parse)←⊂'TimeGrp' ''
      r.Name←'Time' 'UTC'
      r.Desc←'Show local time in a city' 'Show UTC time'
    ∇

    ∇ r←Run(Cmd Arg);dt;offset;cities;diff;lc
      ⎕USING←'System'
      dt←DateTime.Now ⋄ offset←0
      :If 'UTC'≡Cmd
          cities←'l.a.' 'montreal' 'copenhagen' 'sydney'
          lc←⎕SE.Dyalog.Utils.lcase Arg~' '
          offset←¯8 ¯5 2 10 0[cities⍳⊂lc]
      :OrIf ' '∨.≠Args
          dt←Zulu dt
      :EndIf
      diff←⎕NEW TimeSpan(3↑offset)
      r←(r⍳' ')↓r←⍕dt+diff ⍝ remove date
    ∇

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

[4] The function does not deal with daylight savings time. An exercise for the reader?

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

Here, `List` and `Help` have been updated to provide more accurate information but the main changes are in `Run` which now makes use of the 2nd argument. This one is used to determine if we should use the `Zulu` function and compute the offset from UTC by looking it up in the list of cities we know the time zone (offset) for. We allow uppercase letters by simply lowercasing the whole argument with the utility `⎕SE.Dyalog.Utils.lcase` which is available to everyone.

The first argument to `Run` is always the command name (here called `Cmd`) and the second argument is whatever you entered after the command (here it is called `Arg`). When there are no special rules this argument will always be a string.
For example, if the user entered:

```
]time Sydney
```

`Cmd` will contain 'Time' and `Arg` will contain 'Sydney' which is whatever was entered after the first space after the command name. Notice the first argument will contain the Name of the command as spelled out in `List` no matter how the user cases it and that any extra space the user enters will be included in the argument which is why we take care of removing those spaces on line 5.

Switches

There are times when it makes more sense for a command to accept *switches* instead of writing an entirely new (similar) command. A command switch (also known as *modifier* or *option*) is an instruction that the command should change its default behaviour.

For example, the SALT command `list` is used to list `.dyalog` files in a folder. The command accepts an argument which is used as a filter (e.g. '`a*`' to list only the files starting with 'a') and accepts also some switches (e.g. '`-versions`' to list all the versions). Thus the command '`]list a* -ver`' will only list the files starting with 'a' with all their versions instead of listing everything without version, which is the default.

The Spice framework upon which user commands is built allows you to define switches that your command will accept. If the `Parse` element for your command is empty (as defined in your `List` function), `Run`'s second argument will simply contain everything following the command name, and you can interpret it any way you like just like we did so far. By setting Parse to non-empty values, you can get the framework to handle switches for you.[5]

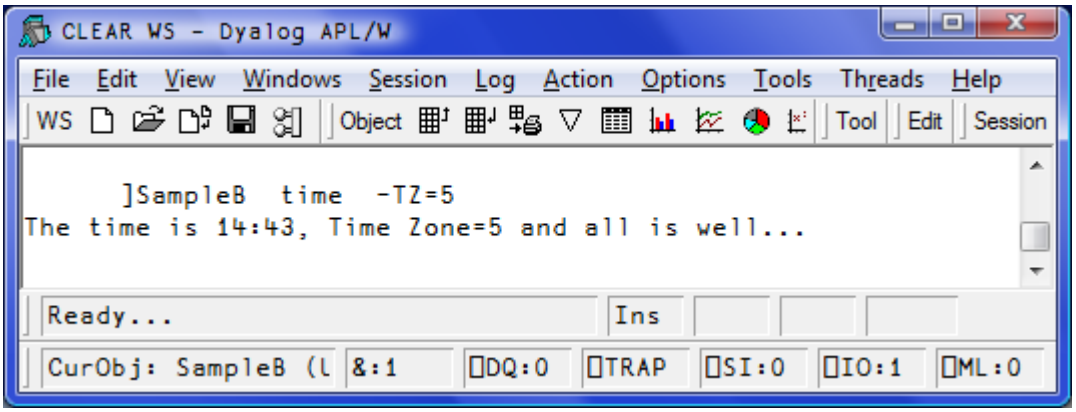
Let's take a look at an example using *parsing*.

[5] The parser upon which this functionality is based is described in an article found in Vector Vol 19, #4: Tools, Part 1: *Basics*.

Example #4: The Supplied sample Command

The default installation includes a sample command to demonstrate the use of arguments and switches.
In file `aSample.dyalog` you will find a class with 2 commands: one named `sampleA` which does not use the parser and one, `sampleB`, that does.

The second command, `sampleB`, is similar to the 'time/utc' command described earlier: it accepts one and only one argument and one switch, called TZ, which MUST be given a value. For example you could write:



The framework is used to validate that there is exactly **one** argument, and that `TZ`, if supplied, has been given a value - and that no unknown switches are used.

This is accomplished by setting the `Parse` variable for that command to '`1 -TZ=`' (in your `List` function).
The `1` means that **one and only one** argument must be present. `-TZ=` declares that '-' is the switch delimiter, that `TZ` is a valid switch for the command, and the trailing '`=`' means that a value must be supplied. The names of switches are case sensitive and follow the same rules as APL identifiers.

If you don't declare the number of arguments to your command, any number of arguments will be accepted (including 0).

When your command is used, your function will only be called if the arguments and switches comply with the rules that you have declared. The framework will package the argument and switch(es) into a namespace and pass this as the second argument to `Run`, which is called `Arg` in our example.

That second argument, a namespace, will contain a vector of strings named `Arguments`, with one string per argument to the command (in our example there will always be a single –enclosed- string because of the '`1`' in '`1 -TZ=`'), and a variable named `TZ` which will either be a numeric scalar 0 if the switch was not specified, or a character vector containing the supplied value.

Let's try to go over that again. The user enters:

```
      ]sampleb  xyz  -TZ=123
```

This is OK since there is one argument (arguments are separated by spaces) which has the value '`xyz`', and the switch `TZ` has been given a value: '`123`'.

`Run` will be called with TWO right arguments where the first one is '`sampleB`' and the second one is a namespace containing `Arguments` (`,⊂'xyz'`) and `TZ` (`'123'`). The `Run` function then runs its course with those values.

Here's another example:

```
      ]sampleB  x  y  z
```

3 arguments have been supplied: x, y and z, so the framework rejects the command without calling your code:

```
Command Execution Failed: too many arguments
```

Another example:

```
      ]SAMPLEB  'x y z'  -TZ
```

Here there is only ONE argument as quotes have been used to delimit the argument of 5 characters: 'x, space, y, space, z' BUT the switch TZ has not been given a value so:

```
Command Execution Failed: value required for switch <TZ>
```

One more:

```
      ]Sampleb  zyx  -TT=321
```

Here one argument is OK but TT is not a recognized switch and:

```
Command Execution Failed: unknown switch: <TT>
```

What if we don't supply ANY argument?

```
      ]Sampleb  -T=xx
```

T is OK as an unambiguous abbreviation for `TZ`, but 0 argument is not enough and:

```
Command Execution Failed: too few arguments
```

The following general rules apply to the parser:

- Commands take 0 or more arguments *followed by* 0 or more switches
- Arguments come first, switches last
- Arguments are separated by spaces
- A special character ('-' is recommended) identifies and precedes a switch
- Switches may be absent or present and may accept a value with the use of '='
- Switches can be entered in any order
- Switches are case sensitive
- Arguments and switch values may be surrounded by single or double quotes[6] in order to embed spaces, quotes or switch delimiters.

[6] If quoted, an argument must begin and end with the same quote symbol (" or '). Whichever is used, the other quote symbol can be embedded within the argument, for example "I'm". The same quote symbol can also be embedded by doubling it, for example 'I''m'.

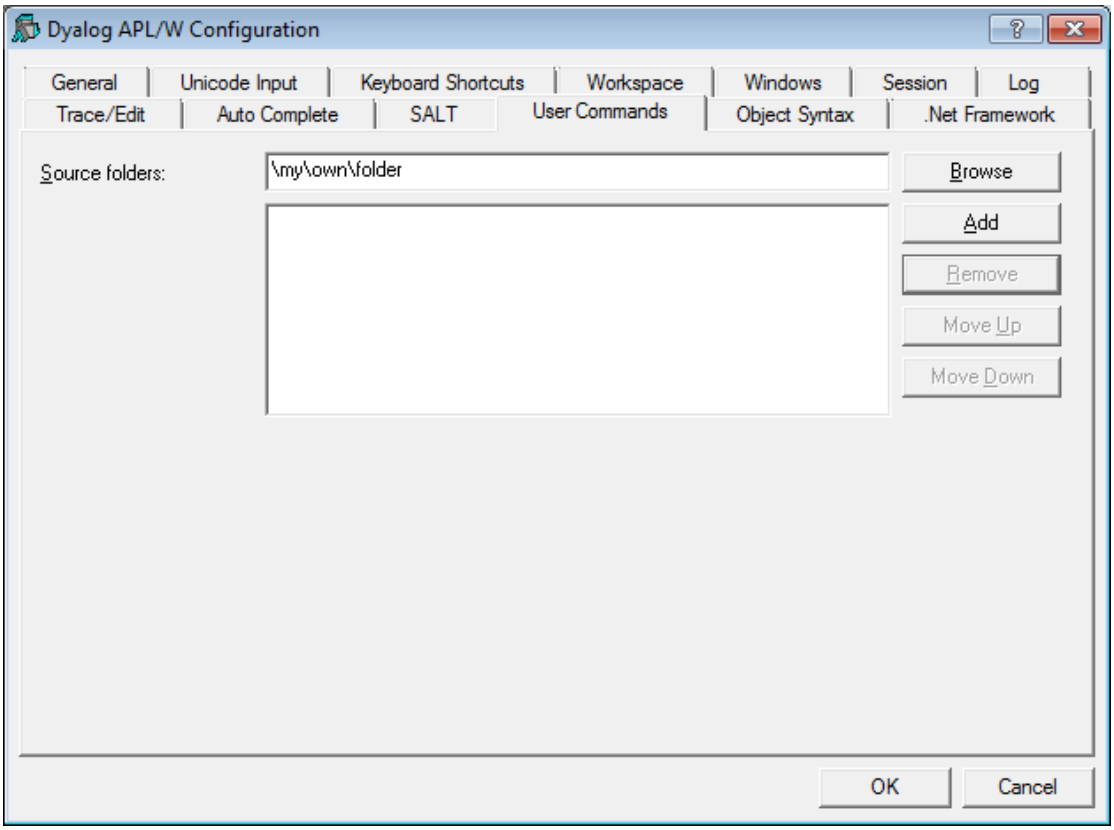
After verifying that the specified rules have been followed, the user command framework will put all the arguments into the variable `Arguments` in a new namespace. It will also insert a variable of the same name as each switch. The namespace is then passed as the second argument to `Run`.

There are a few more things the parser can do but this should cover most cases. See *Advanced Topics* in the following for details.

Location of Commands

By default, the files defining user commands are located in the folder `SALT\spice` below the main Dyalog program folder. You can change that by specifying a new location.

You can change the location using *Options/Configure User Commands Tab*, just remember the change won't become effective until the next APL restart:



You can also change the location of user commands immediately (no need to restart APL) using the command `]settings`.

`]settings` takes 0, 1 or 2 arguments. With 0 argument, it displays the current value of ALL settings. With 1 argument it shows the value of that particular setting. With 2 arguments it resets the value of the setting specified.

The setting to use for the user command folder is '`cmddir`'. Thus

```
      ]settings  cmddir
```

will report the folder(s) currently in use. The installed default is `[SALT]\spice`, where `[SALT]` is shorthand for the SALT program folder. If you wish to use another folder, e.g. `\my\user\cmds` you should type

```
      ]settings   cmddir   \my\user\cmds
```

Note that this will change the setting for the duration of the session only. If you wish to make this permanent you should use the `-permanent` switch:

```
      ]settings   cmddir   \my\user\cmds  -permanent
```

More than one folder can be specified by separating the folders with semi colons (;), e.g.

```
      ]settings  cmddir  \my\user\cmds;\my\other\goodies
```

The folders will be used in the order specified. If a command with the same name appears in more than one folder, only the first occurrence will be used.

Because spaces are important in folder names you must take care NOT to introduce ANY spaces inappropriately.

If you replace the command folder with your own, you effectively disable most installed commands. Only the commands which are part of the *SALT* and *Spice* framework will remain active. See below for details on those.

If you wish to ADD to the existing settings you can either retype the list of folders including the previous ones or precede your new folder with a comma to mean ADD (in front), e.g.

```
      ]settings  cmddir ,\my\spice\cmds;\my\other\goodies
```

will add the 2 folders specified to any existing setting.

If your folder includes spaces or a dash you should use quotes:

```
      ]settings cmd '\tmp\a -b c;\apl\with  2  spaces'
```

When you change the command folder it takes effect immediately. The next time you ask for `]?` or a command it scans the new folder(s) specified to cache the info related to all commands: name, description, parsing rules.

Advanced Topics

By default, all errors in user commands are trapped, possibly making it difficult to debug commands as you are working on them. To prevent this, you can set the `DEBUG` mode ON, as follows:

```
      ]udebug ON
```

Tracing User Commands

You can trace into a user commands just like any other APL expression. Because there is a setup involved in executing a user command it can take quite a few keystrokes to get to the actual code: First the `UCMD` function is called then the Spice processor, and finally your `Run` function. To speed up the process you can ask Spice to stop just prior to calling `Run` by adding a dash at the end of your command expressions, e.g.

```
      ]command  arguments  -
```

The dash will be stripped off and APL will stop on the line calling your `Run` function, allowing you to trace into your code.

This will only work when the `DEBUG` mode, as shown above, is ON.

Default Values for Switches

A switch always has a value, either 0 if not present, 1 if present without a value or a string matching the value of the switch. For example, if you use `-X=123` then **X** will be a 3-element character vector, not an integer.

If you wish to default a switch to a specific value, you can either test its value for 0 and set it to your desired default, e.g.

```
:if  X≡0  ⋄  X←123  ⋄  :endif
```

or you can use the function `Switch` which is found in your namespace (in the 2nd argument).

Monadic `Switch` returns the value of the switch as if it had been requested directly except that it returns 0 for invalid switches (a VALUE error normally).

Dyadic `Switch` returns the value of the left argument if the switch is undefined (0) or the value of the switch if defined but with a twist: if the value of the default is numeric it assumes the value of the switch should be also and will transform it into a number, so if `-X=123` was entered, then

```
99  Args.Switch  'X'  ⍝ default to 99 if undefined
```

will return `(,123)`, not '`123`'[7]

[7] the result is always a vector with Switch, this makes it easy to subsequently tell between 0 (switch not there) and ,0 (value supplied by the user)

Restricted Names

If possible, avoid using switches named `Arguments`, `SwD`, `Switch`, `Propagate` or `Delim`, as these names are used by the parser itself (remember that switch names are case sensitive). You *can* use these names, but they will not be defined as variables in the argument namespace. They will only be available thru function `Switch`, for ex: `Args.Switch 'SwD'` will return the value of switch named `SwD`.

Long arguments

There are times when arguments need to contain spaces. The user can put quotes around related elements. For example, if the user command `newid` accepts 2 arguments, say *full name* and *address* you would set `Parse` to '2' and the user would use, e.g.

```
      ]newid  'joe blough'  '42 Main str'
```

If the command needed arguments *name*, *surname* and *address* (3 arguments), the user would not need the quotes before 'joe' and after 'blough', but would need them for the 3rd argument to keep the three parts of the address together.

If you want the **last** argument to contain "whatever is left", then you can declare the command as '*long*'. If there are too many arguments, the "extra" ones will be merged into the last one (with a single space inserted between them). To do this, append an '`L`' after the number of arguments, for example '`3L`' (plus switches if any).

An example of a command requiring one compulsory *long* argument would be a logging command coded '`1L`':

```
      ]log    all this text is the argument.
```

Note that if there are multiple blanks anywhere in the text, they will be converted into single spaces.

Short arguments

There are times when you only know the MAXIMUM number of arguments. For example there may be 0, 1 or 2 but no more. In that case you would code the parse string as '2S' for 2 Shorted arguments.

Another example is when you have a single argument which can be defaulted if not supplied. You would then use '1S' (plus switches if any) as parse string. If the user enters no argument (0=⍴Args.Arguments) then your program takes the proper action (e.g. default to a specific value).

Forcing a reload of all commands

When you use a command which the framework does not recognize, it can scan the command folder(s) to see whether new commands have been added. This is the default behaviour when the `setting newcmd` is set to 'auto'. However, if you change this setting to 'manual' or make a change to the help or the list of commands, you will need to use the command `]ureset` to force a complete reload of all user commands. This is because some information like the short help is cached in □SE. Using]? Would therefore not reflect the latest changes.

Monitoring your code

It is possible to use ⎕**MONITOR** on your code to find where code is being executed and how long it took to run. To do so use]Umonitor ON|OFF. When ON, each function that is called is recorded along with the monitoring information. See below for details.

SALT Commands

Because SALT is part of the user command framework, the commands which implement SALT itself are always available, even if you remove the default command folder from the *cmddir* setting. The commands in question are `load`, `save`, `compare`, `list`, `open`, `settings`, `rebuild`, `removeversions` and `snap`. If you "shadow" these with your own command with the same names, you will effectively make them invisible, but you will always be able to call them directly by using the functions in `⎕SE.SALT`, for example `⎕SE.SALT.Load`.

Spice Commands

A number of user commands are related to the user command framework (Spice) itself. They are described in details in Appendix A.

Detailed Help

It is possible to provide several levels of help for your commands. When the user enters **]?xyz** the framework calls your <**Help**> function with the name of the command (here 'xyz') as right argument. If your command accepts a left argument it will be given the number 0 for "basic help".

It is possible to use more than one '**?**' to specify the level of help required. Entering **]??xyz** is requesting more help than **]?xyz.]???xyz** even more so. In effect the left argument to your <**Help**> function is the number of extra '?' See command HelpExample for details.

More Implementation Details

User commands are implemented thru a call to `⎕SE.UCMD` which is given the string to the right of the `]` as the right argument and a reference to calling space as the left argument. For example, if you happen to be in namespace `#.ABC` and enter the command

```
]XYZ   -mySwitch=blah
```

APL will make the following call to `⎕SE.UCMD`:

```
#.ABC ⎕SE.UCMD 'XYZ  -mySwitch=blah'
```

preserving the command line exactly**.** The result returned by `UCMD` is displayed in the session.

This means that application code can invoke user commands by calling `⎕SE.UCMD` directly and that if you erase the function, you will disable user commands completely.

By default, ⎕SE.UCMD calls *Spice*, which implements user commands as described in this document. Its right argument is simply passed on to Spice using (here) the call:

```
      ⎕SE.SALTUtils.Spice 'XYZ  -mySwitch=blah'
```

Spice will make UCMD's left argument available to your command via "glocal" ##.THIS so you can reference the calling environment if you need to.

It is not recommended that you call Spice directly. Dyalog reserves the right to change the implementation at this level. However, calling ⎕SE.UCMD should be safe for the foreseeable future. It is possible to modify ⎕SE.UCMD in order to implement your own user command mechanism, but Dyalog recommends that you refrain from this, in order to promote maximum sharing of development tools amongst all users of Dyalog APL.

Assigning the result of a command

It s possible to assign the result of a command by simply inserting an assignment between] and the command name. For example to assign the result of **list** to 'a':

```
      ]a←list -raw
      ⍴⍴a
 <DIR>  lib        2010 7 8 17 38 10 879
 <DIR>  SALT       2010 7 8 17 38 10 889
 <DIR>  spice      2010 7 8 21 0 52 715
 <DIR>  study      2010 7 8 17 38 10 930
 <DIR>  tools      2010 7 8 17 38 10 948
5 5
```

To discard the result, do not specify a variable:

```
      ]←list -raw
```

To display line by line (as opposed to block by block) use quad:

```
      ⎕pw←30
      ]disp 2 32⍴';'
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

      ;;
      ;;
      ]⎕←disp 2 32⍴';'
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;;
```

Detecting if the result will be captured

It may be interesting to know if the result of a command will be used. For example, command **]fnsLike** returns a matrix result but can be formatted to be seen like)FNS.

Since most calls issued in the session would require one to use **–format** to format the result a la)FNS, it is easier to assume this is always the case and make the command show a formatted result whenever called from the session.

For this the framework supplies a Boolean variable, **##.RIU** (Result Is Used), which tells whether the result is captured either because □SE.UCMD was used specifically or]Z←cmd was entered.

In the case above where we would NOT want **]fnsLike** to format the result we can always use]□←fns (quad assign the result).

Source file of the command

Should you need to know which file your command came from, global **##.SourceFile** will provide that information.

Appendix A – Public User Commands

WARNING: Version 13.1 is the third release which includes user commands. The user command mechanism is stable but should still be considered experimental to some extent: while the intention is that user commands built with version 13.1 will continue to work in future releases, the mechanism may be extended and many of the user commands shipped with the product are likely to be renamed or significantly modified in the next couple of releases. V13 and 13.1 already have some changes which are mainly additions to the 12.1 set.

Use `]?` To list the commands currently installed.

Commands are divided into groups. Each group is presented here along with its commands.

To get examples or more information use**]??** ***command*** . For example, to get detailed info on command **wslocate** do

```
]??wslocate
```

Group SALT

This group contains commands that correspond to the SALT functions of the same name found in `⎕SE.SALT`: `Save`, `Load`, `List`, `Compare`, `Settings`, `Open`, `Snap` and `RemoveVersions`. It also includes a separate command, `Rebuild`.

Example:

```
]save  myClass  \tmp\classX  -ver
```

This will do the same as

```
⎕SE.SALT.Save 'myClass  \tmp\classX  -ver'
```

`Rebuild` is used to bring back objects saved with the command `]SNAP … -nstofolder`

For example if you used `]SNAP \tmp\example -nstofolder` to save the contents of your workspace and wanted to retrieve everything that was saved under `\tmp\example` you would do

```
]rebuild \tmp\example
```

Note that this will copy over any existing objects.

Group Sample

There are commands in this group used to demonstrate the use of help and parsing user command lines. You should have a look at the classes and read the comments in them and the description earlier in this document to better understand the examples.

Command HelpExample

This command is an example of a command using different levels of Help with a left argument to the <Help> function.

Command sampleA

This command is an example of a command NOT using parsing, where the argument is the entire string after the command name.

Command sampleB

This command is an example of a command using parsing, where the string after the command name is parsed and turned into a namespace containing the arguments tokenized and each switch identified.

Group Spice

This group contains ten commands: *UClean, UDebug, Uload, Umonitor,UNew, USetup, URefresh, UReset, Uupdate* and *UVersion*. They are used to manage the user command system itself.

Command UClean

This command removes any trace of SALT in the workspace by removing all tags associated with SALT with each object in the workspace. Once you run it the editor will no longer put changes back in the source file(s).

Command UDebug

This command turns debugging ON and OFF in order to stop on errors when they happen. Otherwise the error will be reported in the calling environment. It also enables the 'stop before calling the run function' feature which consists in adding a dash at the end of the command as in

```
      ]mycmd myarg -
```

to make the setting permanent use

```
      ]udebug on -permanent
```

`UDebug` can also turn system debugging on and off[8]. For example, to turn the 'w' debug flag on use

```
      ]udebug on -flag=w
```

to turn it off use

```
      ]udebug off -flag=w
```

Command ULoad

[8] system debug flags are used to debug the interpreter itself. See the User Guide for details on this topic.

This command is used to bring in the workspace the namespace associated with a user command. It is typically used when debugging a user command and you need the code in order to work with it.

Example: load the code for the `CPUTime` command:

```
      ]uload  cpu
Command "CPUTime" is now found in <#.Monitor>
```

The namespace `Monitor` containing the code the for the `CPUTime` user command was brought in from file. We can now edit the namespace and modify the command. When we exit from the editor, the namespace will automatically be saved back to the script file from whence it came. There is no need for a `usave` command since **SALT's** `save` command already saves code and subsequent changes are handled by the editor's callback function. However, there is a command for creating a new command, `UNew`, described below.

Command UMonitor

This command turns monitoring ON or OFF. When ON, all active functions see their ⎕CR and their ⎕MONITOR information paired in global variable #.UCMDMonitor

Results are set in `#.UCMDMonitor` after each invocation of a command.

`-var=` sets the name of the variable to store the result instead of `#.UCMDMonitor`

Command UNew

This command is used to create a namespace containing one or more user commands. It creates a form which is used to input all the basic information about the commands contained in a *Spice* namespace: the command names, their groups, their short and long description and details of switches.

Each command's information is entered one after another.

When finished it creates a namespace which you can edit and finally save as a file.

Command URefresh

This command will reload all SALTed objects if they have changed. This can happen after you)LOAD a workspace containing stale objects. URefresh will let you confirm the]load first unless you use *–noprompt.*

Command UReset

Forces a reload of all user commands – this may be required e.g. after modifying a command's description or parsing rules which are kept in memory.

Command USetup

This command is used by versions prior to V12.1 to automatically initialize Spice's command bar to the user's preferences. It can still be customized and used to modify your session preferences, e.g. to setup your PF keys.

Command UUpdate

This command will update your current version of SALT and user commands to the latest version. This command must be run manually as prompts are issued to do the work although the `-noprompt` switch allows to bypass them.

It takes one argument to specify which tools to update. The default is SALT which includes UCMDs. To update only the user commands use UCMD as argument.

Command UVersion

This command reports various version numbers: for APL, SALT, .NET and UCMD itself. If given the name of a file containing a workspace it will display the minimum version of Dyalog necessary to)LOAD the workspace.

Group svn

This group contains a series of commands used as cover to SubVersion functions of the same name. For example, **svnci** is equivalent to 'svn ci' and commits changes made to the current working copy.

Group SysMon

This group contains three commands for measuring CPU consumption in various ways: `CPUTime` simply measures the total time spent executing a statement, `Monitor` uses MONITOR to break CPU consumption down by line of application code, and `APLMON` breaks consumption down by APL Primitive.

Command APLMON

From version 12.0, Dyalog APL provides a root method which allows profiling of application code execution, breaking CPU usage down by APL primitive rather than by code line. The `APLMON` command gives access to this functionality.

As with `Monitor`, you can either run the command with the switch `-on` to enable monitoring, run your application, and then run the command again with the switch `-report` to produce a report, ***or*** you can pass an expression as an argument, in which case the command will switch monitoring on, run the expression, and produce a report immediately. The only other switch is `-filename=`, which allows specification of the `APLMON` output file to be used. If it is omitted, a filename will be generated in the folder which holds your APL session log file.

Examples:

```
      ]aplmon ⍴{+/1=⍵∨.≠⍳⍵}¨⍳1000 -file=\tmp\data
1000
Written: C:\tmp\data.csv
```

The above command generated a log file name, enabled `APLMON` logging, ran the expression and switched `APLMON` off again. You can report on the contents of this file using the "aplmon" workspace, or send it to Dyalog for analysis.

```
      )load aplmon
      InitMon '\tmp\data.csv'
Total CPU Time    =  0.15 seconds
Total primitives =           5,003
                        count  sum hitcount  sum time    pct
time
-------------------------------------------------------------
 1.      or           │      7         1,000  0.136557  94.03 │
 2.      equal        │      6         1,000  0.00454    3.13 │
 3.      iota         │      1         1,001  0.003087   2.13 │
 4. plus slash        │      6         1,000  0.001038   0.71 │
```

Command CPUTime

This command is used to measure the CPU and Elapsed time required to execute an APL expression. There are three switches, `-repeat=` which allows you to have the expression repeated a number of times and/or some period of time, `-details=` which specifies how much details should be included and `-compare` which allows you to compare the CPU time of all expressions in relation to the first one. By default, the expression is executed once. The report always shows the average time for a single execution.

It can also accept a combination of both iterations and period, for ex the maximum between 10 iterations and 1000 milisecs. If 1 second is not enough to run the expression 10 times it repeats until the expression has been executed 10 times. On the other hand if the expression ran 10 times in less than 1 sec it continues to run until 1 sec has gone by. It would be specified this way: `-repeat=10⌈1s`

With `-details=none` only the numbers are returned as a 2 column matrix (CPU and elapsed), 1 row per expression.

With `-details=ai` only the same numbers plus the 2 ⎕AI numbers are returned (Nx4 matrix).

With `-details` or `-details=all` nothing is returned; instead, a report that includes the number of times repeated and the ⎕AI and ⎕MONITOR numbers is shown.

Examples:

```
      ]cputime {+/1=ω∨ιω}¨ι1000
* Benchmarking "{+/1=ω∨ιω}¨ι1000"
             Exp
 CPU (avg):   63
 Elapsed:     67

      ]cputime {+/1=ω∨ιω}¨ι1000 -repeat=1s
* Benchmarking "{+/1=ω∨ιω}¨ι1000", repeat=1s
              Exp
 CPU (avg):  54.6
 Elapsed:    55.2

      ]cpu  {ω[⍋ω]}b  b[⍋b] -compare

  {ω[⍋ω]}b → 1.8E¯4 |     0% □
  b[⍋b]    → 8.0E¯3 | +4435% □□□□□□□□□□□□□□□□□□□□□□□□□□□□□

      ]cpu  {+/1=ω∨ιω}¨ι100  ι⍨ι1e6  -details=ai -rep=50
 0.64  0.46  0.64  0.46
30.58 30.52 30.58 30.52
```

The last example shows the 2x4 result for the 2 expressions tested.

Command Monitor

This command is used to find out which lines of code in your application are consuming most CPU. You can either run the command with the switch `-on` to enable monitoring, run your application, and then run the command again with the switch `-report` to produce a report, ***or*** you can pass an expression as an argument, in which case the command will switch monitoring on, run the expression, and produce a report immediately. Other switches are:

Switch	Effect
`-top=n`	Limits report to the *n* functions consuming the most CPU
`-min=n`	Only reports lines which consume at least n% of the total, either CPU ***or*** Elapsed time
`-fns=fn1,fn2,…`	Only monitors named functions
`-caption=text`	Caption for the tab created for this report

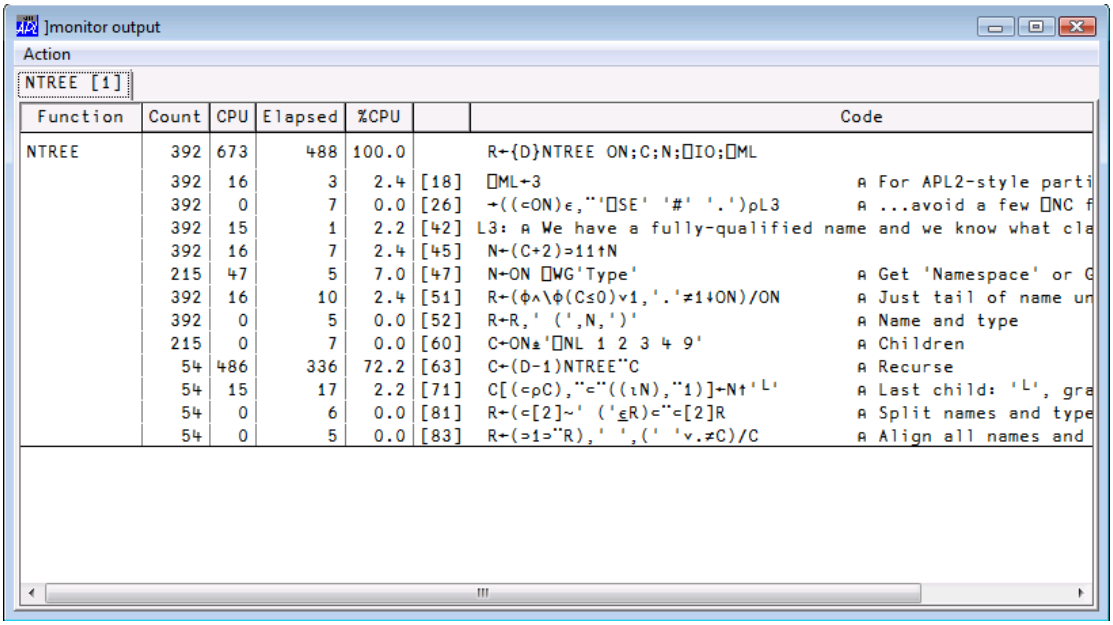
Examples:

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

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

      ]mon -rep -cap=NTREE
```

(Pops up the following dialog)



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)⊃11↑N
	215	47	5	7.0	[47]	N←ON ⎕WG'Type' ⍝ Get 'Namespace' or C
	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),' ',(' '∨.≠C)/C ⍝ Align all names and

Command Profile

This command is used to fine tune your application. See the relevant document for details.

Group System

This group contains operating system related commands.

Command assemblies

This command lists all the assemblies loaded in memory.

Command cd

This command will change directory in Windows only. It reports the previous directory or the current directory if the argument is empty.

Example: switch to directory **\tmp** for the remaining of the session:

```
      ]cd  \tmp
C:\Users\Danb\Desktop
```

Command EFA

This command, Edit File Association, will associate Windows® file extensions **.dws, .dyapp** and **.dyalog** with a specific Dyalog APL version. This is useful only if you have several versions installed and wish to change the current association made with the latest install.

Group Tools

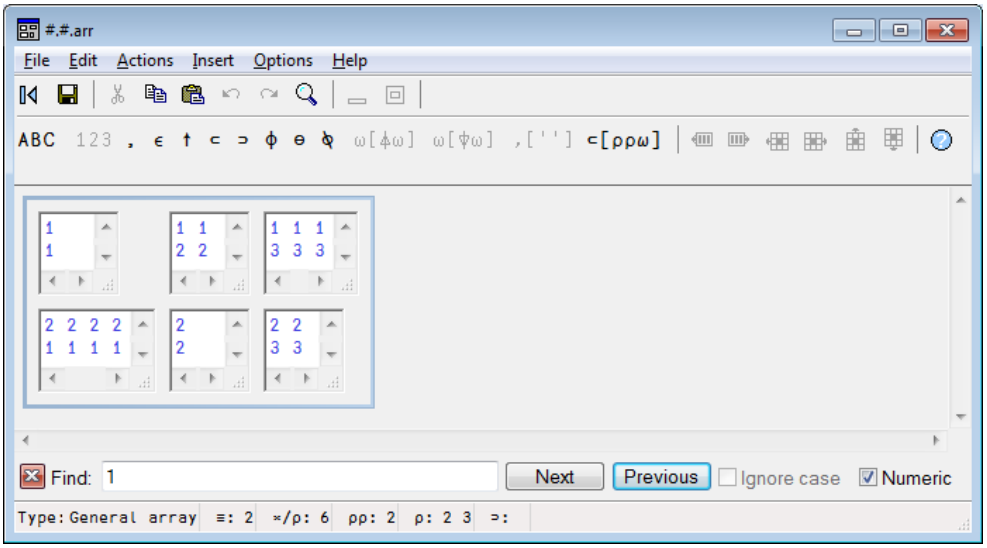
This group contains a series of commands used to perform tasks related to everyday activities.

Command aedit

This command will edit an array using David Liebtag's array editor.

Example:

```
      arr←(2 3⍴1 2 3 4)/⍤¯1⍤⍳2 3
      ]aedit arr
```



Note that this functionnality is also available via the tool bar in the session.

Command calendar

This command is similar to Unix' **cal** program and displays a calendar for the year or the month requested.

Example:

```
      ]cal  2010 3
     March 2010
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 31
```

Command Collect

This command will collect all files with a specific pattern (the argument) into a single file. This is useful when files have been split (see]SPLIT) and need to be reassembled.

Switches

–newname= allows you to select a new name

–erase will erase the file if it exists before attempting to create it.

Example:

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

All files starting with *\tmp\file. zip* and followed by *001*, *002* ,*003*, etc. will be merged into a single file named *\temp\px.zip*

Command Demo

Demo provides a "playback" mechanism for live demonstrations of code written in Dyalog APL

Demo takes a script (a text file) name as argument and executes each APL line in it after displaying it on the screen.

It also sets F12 to display the next line and F11 to display the previous line. This allows you to rehearse a demo comprising a series of lines you call, in sequence, by using F12.

For example, if you wish to demo how to do something special, statement by statement you could put them in file `\tmp\mydemo.txt` and demo it by doing

```
      ]demo \tmp\mydemo
```

The extension TXT will be assumed if no extension is present.

The first line will be shown and executed when you press Enter. F12 will show the next which will be executed when you press Enter, etc.

Command dinput

This command is used to test multi line D-expressions.

Example:

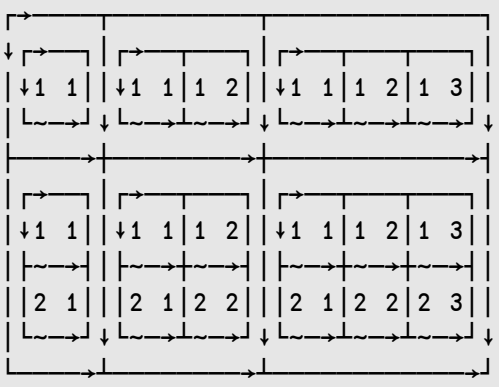
```
      ]Dinput     ⍝ multi-line expression
····{             ⍝ dup:
········⍵ ⍵
····}{            ⍝ twice:
········⍺⍺ ⍺⍺ ⍵
····}7
 7 7  7 7
```

Command disp

This command will display APL expressions using boxes around enclosed elements as per the familiar `disp` function.

Example:

```
      ]disp ⍳¨⍳2 3
```



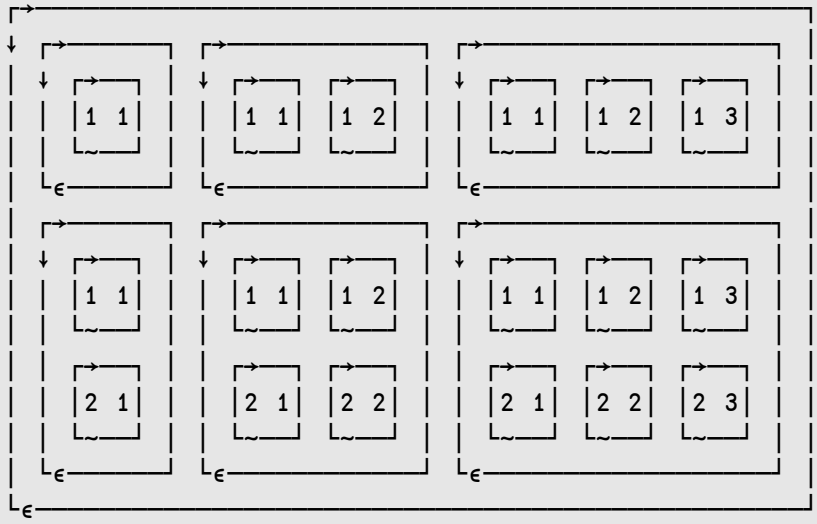
Command display

This command will display APL expressions using boxes around enclosed elements as per the familiar `DISPLAY` function.

Note that this command is different from the `disp` command just like the 2 functions `disp` and `DISPLAY` are different and you must enter at least ‘displ’ to use it.

Example:

```
      ]display ⍳¨⍳2 3
```



Command dmx

This command will provide more detailed information about the last APL error.

Command factorsof

This command will return the factors that constitute a number.

Example:

```
      ]fac 123456789
3 3 3607 3803
```

Command FFind

This command searches the `.dyalog` files in, by default, the current SALT working directory for the string given as argument in SALT script files. It needs ONE long argument which may be a .Net regular expression[9].

It reports all the hits in each script file where found.

To search different directories use the switch `-folder` to specify the new location.

`-options` will accept a value of I (insensitive), S (Singleline mode) or M (Multiline mode – the default) to change search behavior.

`-types` will accept the extensions to use when searching (default .dyalog)

`-regex` will consider the argument to be a regular expression

Example:

```
      ]ffind \b(abc|\w{7})\b -folder=\tmp -typ=txt log -r
```

will find 'abc' or all 7 letter words in all .txt or .log files in `\tmp`, and below. `-r` is short for `-regex`; without it the exact text above would be looked for.

[9] to look for or replace strings in the workspace use command WSLOCATE

Command FnCalls

This command is used to find the calls made by a program in a script file or in the workspace.

It takes 1 or 2 arguments: the function name and the namespace or filename where it resides (default current namespace). With switch `-details` it can provide extra details on all the names involved such as locals, globals, unused, recursively called, etc.

With switch `-treeview` it will show the result in a treeview window instead of the session log.

If the switch `-file` is provided the namespace is assumed to be the name of a file.

Example:

```
     ]fncalls Spice '\Dya -APL\12.1\SALT\SALTUtils'[10] -fil
Level 1:  →Spice
⍝ Handle KeyPress in command window
⍝ The function can also be used directly with a string
F:isChar            F:isHelp            F:isRelPath
F:BootSpice         F:GetSpiceList      F:SpiceHELP

Level 2: Spice→isChar
…
Level 2: Spice→BootSpice
⍝ Set up Spice
F:GetList      R:Spice

Level 3: BootSpice→GetList
⍝ Retrieve the list of all Spice commands
F:getEnvir     F:lCase        F:splitOn     F:ClassFolder

Level 4: GetList→ClassFolder
⍝ Produce full path by merging root and folder name
…
```

At each level the calling function is followed by the called function which is detailed. It lists each function called preceded by either an **F** (for function) or an **R** (for recursive call). We can see at the 1st level that function `Spice` calls 6 other functions and at the 2nd level function `isChar` calls nothing and `BootSpice` calls 2 functions: `GetList` and `Spice`, recursively. At the 3rd level `GetList` calls 4 functions and so on.

With the switch `-full` the output repeats for already shown functions. This may produce output where the same function calls may be different if objects are shadowed up the stack.

With the switch `-details` each object is preceded by either F or R as above or a character meaning:

[10] the 2nd argument is surrounded by quotes because it contains a dash

```
o: local
G: global
!: undefined local
↑: glocal (global localized higher on the stack)
L: label
l: unreferenced label
*: previously described in the output[11]
```

With the switch –`isolate` all the object names required to run the function given as argument are returned in matrix form. Moreover, if –`isolate` takes an unused APL name as value (e.g. –`isolate=newNs`) then all the objects are copied into the new namespace. This allows you to modularize code by isolating individual sections.

Command FReplace

This command searches the .dyalog files in, by default, the SALT current working directory for the string given as first argument in SALT script files and replaces occurrences by the second (long) argument. It needs TWO arguments which may be .Net regular expressions (see http://msdn.microsoft.com/en-us/library/az24scfc(VS.71).aspx for details) if the switch –`regex` is used.

To work on a different directory use the switch `-folder` to specify the new location.

`-options` will accept a value of I (insensitive), S (Singleline mode) or M (Multiline mode – the default) to change search behavior.

`-types` will accept the extensions to use when searching (default .dyalog)

`-regex` will consider the arguments to be regular expressions

Example:

```
    ]frepl Name:\s+(\w+)\s+(\w+) Name: $2, $1 -f=\tmp -r
```

will reverse every occurrence of 2 words (`-r` means "this is a regular expression") when they follow '`Name:`', i.e

```
Name:  Joe   Blough
```

will become

```
Name: Blough, Joe
```

in every file it finds in the directory `\tmp`

Command fromhex

This command will display an hexadecimal value in decimal

Example:

```
      ]fromhex   FFF  A0
4095  160
```

[11] this won't happen when switch **–full** is used

Command Fto64

Since Version 10.1, Dyalog APL has supported large span (64-bit) component files, and since Version 12.0 ⎕FCREATE has created these by default. Small span (or 32-bit) component files are limited in size and do not fully support inter-operability between different platforms. Support for small span files is being gradually withdrawn from Dyalog APL. See the version 13.2 (and 14.0) release notes for more information, and recommendations on how to prepare for the withdrawal of small span support.

The Fto64 user command can be used to locate small span files and convert them to the 64-bit architecture. It takes a folder name as the argument, and accepts a number of switches:

-list produce a list of small span files, but do not convert them
-recursive also check sub-folders for the existence of small span files
-verbose generates detailed output, for every component file found in the folder(s)
-backup= causes backups to be made of all small span files, using the supplied extension

Example:

```
   ]fto64  \big\project -recursive -verbose  -backup=.32
* <C:\big\project\132u64b.DCF> is already 64b
*** <C:\big\project\to\x1.DCF> is tied
…
<C:\big\project\to\x2.DCF> made into 64b format and
backed up to <C:\big\project\to\x2.DCF.32>
27 files modified
```

Note that this command uses ⎕FCOPY to perform the conversion. This means that it could take a considerable amount of time to execute if there are very large files BUT that all the timestamps will be preserved.

Command ftttots

This command will display a number representing a file component time information into a ⎕TS form (7 numbers).

```
      ]ftttots 3⎕⎕frdci 4 1
2011 3 10 23 16 28 0
```

Command GUIProps

This command will report the properties (and their values), *childlist*, *eventlist* and *proplist* of the event given as argument or, if none provided, the object on which the session has focus (the object whose name appears in the bottom left corner of the session log).

Command latest

This command will list the names of the youngest functions changed (most likely today, otherwise of the last changed day), the most recently changed first.

Command SPLIT

This command allows you to split a file into several smaller files. This can be handy when transmitting files over an iffy network. If a file transmission fails the failure is limited to the smaller file which can be retransmitted without having to retransmit the whole original file.

By default the file is split into 10 equal sections. Switch **-n=** allows you to select another number.

If that number is followed by either M or G it is assumed to be the desired size for each section.

Ex:

```
    ]split \projectx\big.zip -n=2M
```

Each section will have 2E6 bytes in size, each file will have the name *\projectx\big.zip001, …002*, etc.

Note that the number of files produced cannot exceed 999.

Command Summary

This command produces a summary of the functions in a class in a script file. It takes a full pathname as single (long) argument. If the switch **–file** is provided the name is assumed to be a file.

```
    ]summary ⎕SE.Parser
name        scope  size  syntax
 fixCase             24   n0f
 if                  24   n0f
 init        PC    4500   n1f
 xCut               532   r2m
 Parse       P     5748   r1f
 Propagate   S     1220   r2f
 Switch            1152   r2f
```

Scope shows S if shared, P if public, C if constructor and D if destructor

Size is in bytes

Syntax is a 3 letter code:

```
[1] n=no result, r=result
[2] # of arguments (valence)
[3] f=function, m=monadic operator, d=dyadic operator
```

Command tohex

This command will display a number in hexadecimal value

Example:

```
      ]tohex   100 256
64  100
```

Command tohtml

This command will produce HTML text that displays a namespace or a class in a browser. It accepts 5 switches:

-title= will take a string to be displayed at the top of the page, e.g. –title= My best Class

-full will include the full HTML code, including the <head> section before the <body>

-filename= write the result to the file specified

-clipboard will put the result on the clipboard, ready to be pasted elsewhere

-xref will produce a Cross-reference of the names used in the class in relation to all the methods

Command WSpeek

Executes an expression in a temp copy a workspace. As its name suggests, WSpeek is used to view, rather than to change, a saved workspace; any changes made in the copy are discarded on termination of the command. A Wsid='.' means the saved copy of the current workspace.

```
      ]WSpeek wsid [expr ...] ⍝ expr defaults to ⎕LX
```

Example: execute the <queens> program from the 'dfns' workspace

```
      ]wsp  dfns  0 disp queens 5
```

Command Xref

This command returns a Cross-reference of the objects in a script file. If the switch `-file` is provided the name is assumed to be a file.

It produces a very crude display of all references on top against all functions to the left. At the intersection of a function and a reference is shown a symbol denoting the nature of the reference in relation to the function: **o** means local, **G** mean global, **F** means function, **L** means label.

Example:

```
      ]xref  \Program Files\Dyalog\SALT\lib\rundemo -file
         ccfkllnpssszzzFFILNPPS
         llieaiaa_cn...iinieaoc
          .lybnms ri.NRllinxtsr
          Tes eet ip.eaeetethni
          e . . e p .sw . . . p
          x . . . t .t. N . . t
          t . . . . . . . . . .
          - - - - : - - - - : -
Edit      . . . . o . . G . : G
Init      . . . .o: . . . .F:GG
Load      .o. .o. : oGGG.F. G G
```

As can be seen in this report, name **script** is a *local* in function **Edit**. The characters dot, dash and semi colon only serve as alignment decorators and have no special meaning.

Command wsdoc

This command will produce a listing of the contents of your workspace possibly onto file. The arguments are the objects to list only (default all objects).

The switches are:

-items= the items to list, can only be one of: wsid fns vars obs fns/ vars/ obs/ salt (a "/" after the item means "do not expand")
-file= output to file mentioned. If ω is given the actual ws name will be used.
-noprompt do not ask me if file exists and overwrite it
-pw= use this width to fold items
-xref will produce a cross reference of objects found in the code

Group Transfer

This group contains four commands: *in*, *out*, *inx* and *outx*. **In** and **Out** read or write APL Transfer Files in the standard ATF format, and should be compatible with similarly named user or system commands in other APL implementations. **Inx** and **Outx** use a format which has been extended to represent elements of a workspace which have been introduced in Dyalog APL since the ATF format was defined.

See the “Dyalog APL Windows Workspace Transfer” for more details.

Group wsutils

This group contains several commands used for workspace management and debugging. Some of the commands take a filter as an argument, to identify a selection of objects. By default, the filters are in the format used for filtering file names under Windows or Unix, using ? as a wildcard for a single character and * for 0 or more characters. For example, to denote all objects starting with the letter **A** you would use the pattern `A*`.

Regular expressions can be used to select objects. You indicate that your filter is a regular expression by providing the switch `-regex`.

The commands which accept filters are *fnslike*, *varslike*, *nameslike*, *reordlocals*, *sizeof* and *commentalign*. They all apply to THE CURRENT NAMESPACE, i.e. you cannot supply a dotted name as argument.

Also, very often the same command will accept a `-date` switch which specifies the date to which the argument applies. This will typically be used when functions are involved, for example when looking for functions older than a date, say 2009-01-01, you would use `-date=<90101`[12]. The century, year and month are assumed to be the current one so if using this expression in 2009 using `-date=<101` would be sufficient. You can use other comparison symbols and `-date=≠80506` would look for dates **different than** 2008-05-06. Ranges are possible too and `-date=81011-90203` would look for dates from 2008-10-11 to 2009-02-03 included.

Command cfcompare

This command compares 2 APL component files. It accepts the same switches as `varcompare` plus switch –`cpts` which list the components to compare.

It accepts 2 arguments: either the path of files or their tie number if already tied. They can be optionally followed by :passnumber it the file(s) require one for reading.

Example:

```
      ]cfcompare  \tmp\abc:123   27  -cpts=5 7, 9-99
```

This will compare file **\tmp\abc** with the file using 27 as tie number. The passnumber 123 will be used for file **\tmp\abc** to read the components. Only components 5, 7 and 9 to 99 will be compared.

Only those components with the same number will be compared; if the 1st file's components range from 1 to 10 and the 2nd's range from 6 to 22 then only components 6 to 10 will be compared. In this case because a set has been specified only components 7, 9 and 10 would be compared.

If either the path of the name of the 2nd file is the same as the 1st's then = can be used to abbreviate the name. For example to compare files ABC and XYZ in folder \tmp\long\path you can enter

```
      ]cfcomp  \tmp\long\path\ABC  =\XYZ
```

See command `varcompare` for the list of other accepted switches.

Command CommentAlign

This command will align all the end of line comments of a series of functions to column 40 or to the column specified with the `-offset` switch.

The arguments are DOS type patterns for names which can be viewed as a regular expression pattern if switch `-regex` is supplied. The `-date` switch can also be applied.

[12] The value of **date** is '<90101', the < is included which is why the syntax includes BOTH = and <

The result is the list of functions that were modified in column format or in `)FNS` format if switch `-format` is supplied.

Example:

```
      ]commentalign  HTML*  -format  -offset=30
```

This will align all comments at column 30 for all functions starting with 'HTML' and display the names of all the functions it modified in `)FNS` format

Command defs

This command will show the definition of single line Dfns and composed functions in order to recall and edit them on the fly.

Example:

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

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

Command findrefs

This command will follow nested references in the workspace until all references are found. Accepts switches:

-root= Namespace (default=#) to start from
-loops Report reference loops
-aliases List all aliases for each ref found
-nolist Do not list namespaces (useful with -loop)

Command fncompare

This command will show the difference between 2 functions, including timestamps. It can handle large functions and has switches to trim the functions first, exclude the timestamps, etc.

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
```

Switches

`-normalize` removes excess space at the ends of each line
`-delins` change the delete/insert characters
`-exts` exclude timestamps in comparison
`-zone` specify how many lines to show before and after each difference
`-nolastline` exclude the last line of each function (ex ignore SALT tag lines)

Command fndiff

This command will show the different lines between 2 functions by showing the differences side by side. It is more suited for small functions. With the same example functions as in fncompare:

Example:

```
      ]fndiff fna fnb
·fna·   ·   ·   ·   ·  │·fnb·   ·   ·   ·   ·   ·
·fna·line 2 ·   ·   ·  │·fnb·line 2 ·   ·   ·   ·
⍝ comment deleted   ·  │⍝ new comment   ·   ·   ·
```

Command fnslike

This command will show all functions names following a pattern in their names. It accepts the `-regex`, `-date` and `-format`[13] switches.

Command nameslike

This command will show all objects following a same pattern in their names. Each name will be followed by the class of the name.

It accepts the `-regex`, `-date`, `-noclass` and `-format`[13] switches.

Example: find all names containing the letter 'a':

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

`-noclass` removes the class number after the name.

Command reordlocals

This command will reorder the local names in the header of the functions given in the argument. The argument is a series of patterns representing the names to be affected. It accepts the `-regex`, `-date` and `-format` switches.

Command sizeof

This command will show you the size of the variables and namespaces given in the argument. The argument is a series of patterns (including none=ALL) representing the names affected. It accepts the **–class** switch to specify the classes involved and the **-top** switch to limit the number of items shown.

Example:

```
      )obs
NStoScript      aplUtils        test
      )vars
CR      DELINS  Describe        FS
      ]size -top=4 -class=2 9
NStoScript 132352
aplUtils    40964
test        31996
Describe    10128
```

Command supdate

This command will update a namespace script with newly added variables and functions.

This can come in handy when you've added code and data inside a scripted namespace.

[13] -format is not needed when the result is not captured see "Detecting if the result will be captured" above

Example:

```
      ]load myns
      )cs myns
      V ←⍳9
      ⎕fx  'myfn'  '2+2'
⍝ Now update the script to include these new objects
      ]supdate
Added 1 variables and 1 functions
```

Command varcompare

This command will compare 2 variables including namespaces which contain functions and other variables and namespaces. For this reason it includes the same switches as command `fncompare` plus the following:

`-exnames=`	exclude names from the comparison
`-nlines=`	show only the 1st n lines of each variable not in the other object
`-show=`	show only specific sections of the comparison report
`-nssrc`	force the use of source for namespaces if they exist instead of comparing object by object

See `]??varcompare` for details

Command varslike

This command will show all variables following a same pattern in their names. It accepts the `-regex` and `-format`[13] switches.

Command wscompare

This command will show the difference between 2 workspaces. It is a combination of the commands `fncompare` and `varcompare` being run on entire workspaces. The workspaces are first copied into temporary namespaces and the comparison then performed. It includes the switches of `fncompare` and `varcompare` plus the following:

`-exstring=`	exclude object containing this string
`-gatheroutput`	gather all the output and return it as a result (can be quite large)

Command wslocate

This command will search strings in the current namespace. It accepts a number of switches that allow it to screen out hits in comments, text, etc. It accepts normal and regular expressions and will perform replacement on most objects. It is a very comprehensive command. For example it allows you to find where 3 names are assigned 3 numbers. See its documentation (`]??wslocate`) for details.

Example: look for the words ending in **AV** (“syntactically to the right”), regardless of case, in text only (exclude Body and Comments):

```
      ]wslocate AV -syntactic=r -insensitive -exclude=bc
Search String (Find and Replace) for Dyalog V6.01

    ∇ #.xfrfrom (3 found)
[57]  ⍎(⎕⎕trans=2)/oNS,'⎕AV←bUf'
                          ^
[72]  ⎕⎕CodT←⎕⎕CodT,(⎕⎕trans=2)/'%⍞av[%⎕AV['
                                   ^    ^
```