The tool of thought for expert programming

Dyalog™ for Windows

User Commands

Version 1.21

Dyalog Limited

South Barn
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

Introduction

Version 12.1 introduces “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* -format
SET    SETMON SETWX  SM_TS  SNAP
```

Help is easily accessible for user commands:

```
      ]?fns
Command "fnsLike". Syntax:  accepts switches -format -
regex -date=
Script location: C:\Program Files\Dyalog\Dyalog APL 12.1
Unicode\SALT\Spice\wsutils

Arg: pattern; Produces a list of fns & ops whose names
match the filter pattern
-format   Return result as )FNS would
-regex    Filter is a regular expression rather than a
simple name filter in which . denotes any letter and .*
any sequence of letters
-date   can take a value, a pair of [YY]MMDD values (for
from-to) possibly preceded by > or <
```

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:

```
      ]?
"??" for general help, "?CMD" for specific info on CMD

 Group  Name    Description
 =====  ====    ===========
 Demo   Demo    Run a Demo Script
 SALT   Compare Compare versions
        Explore Open OS editor or Disk Explorer
…etc…
```

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 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 have developed are now available as *user commands* in 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.

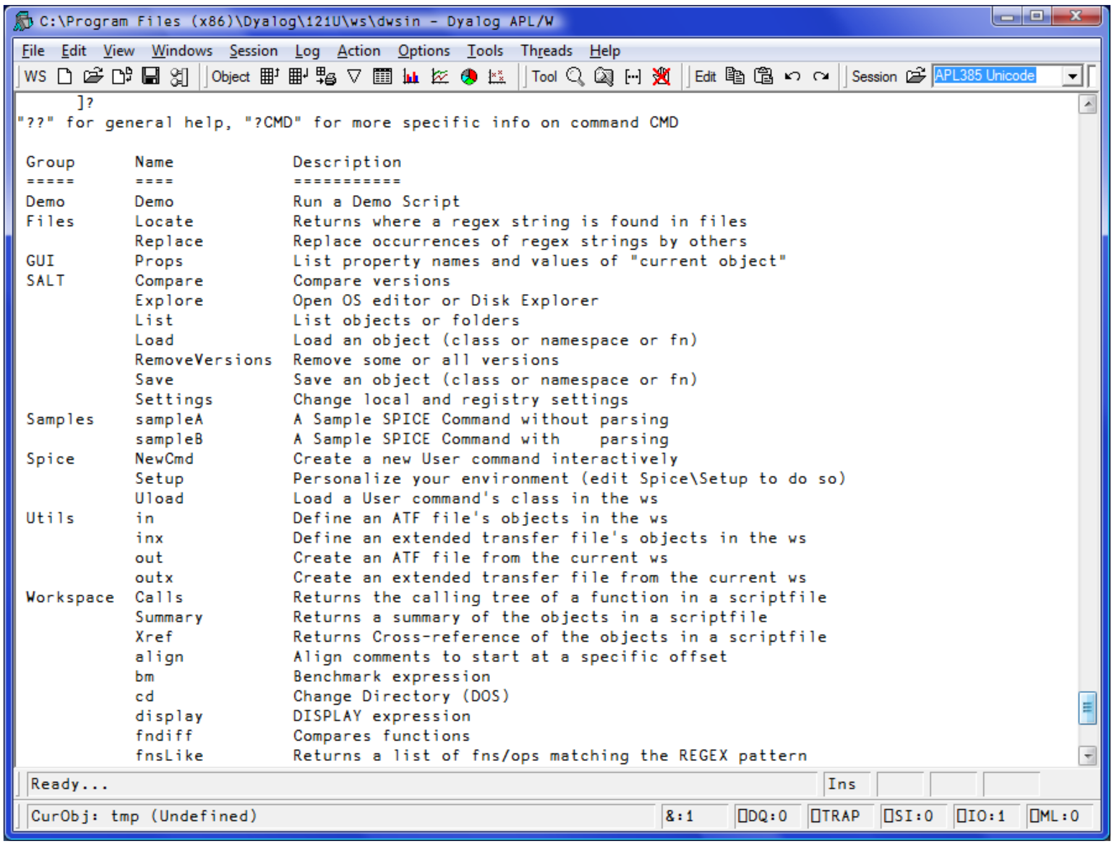
To execute command **xyz** type `]xyz`

To get some general help type `]??`

To find all available commands type `]?`

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

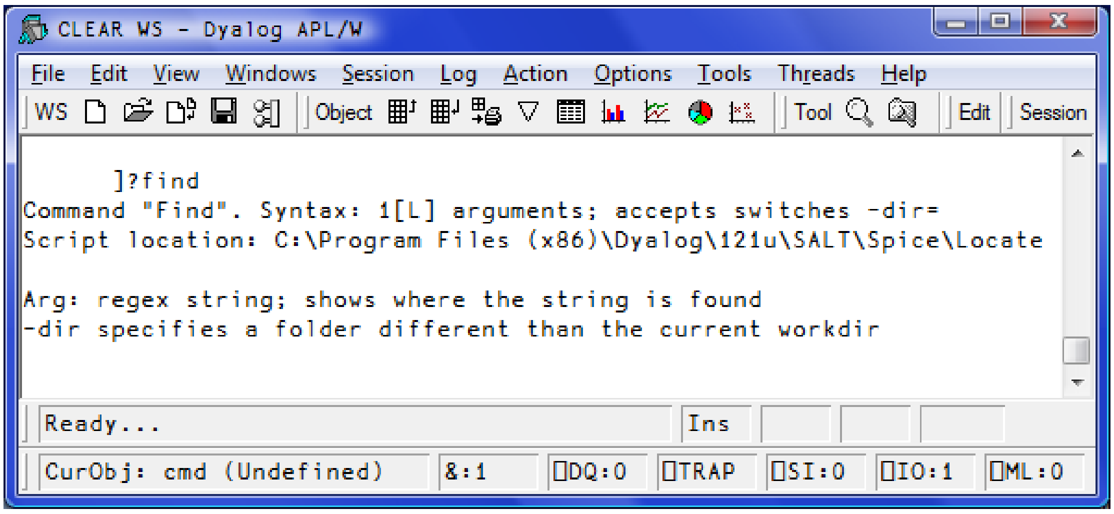
Example:


C:\Program Files (x86)\Dyalog\121U\ws\dwsin - Dyalog APL/W
File Edit View Windows Session Log Action Options Tools Threads Help
WS Object Tool Edit Session APL385 Unicode

```
      ]?
"??" for general help, "?CMD" for more specific info on command CMD

 Group       Name            Description
 =====       ====            ===========
 Demo        Demo            Run a Demo Script
 Files       Locate          Returns where a regex string is found in files
             Replace         Replace occurrences of regex strings by others
 GUI         Props           List property names and values of "current object"
 SALT        Compare         Compare versions
             Explore         Open OS editor or Disk Explorer
             List            List objects or folders
             Load            Load an object (class or namespace or fn)
             RemoveVersions  Remove some or all versions
             Save            Save an object (class or namespace or fn)
             Settings        Change local and registry settings
 Samples     sampleA         A Sample SPICE Command without parsing
             sampleB         A Sample SPICE Command with    parsing
 Spice       NewCmd          Create a new User command interactively
             Setup           Personalize your environment (edit Spice\Setup to do so)
             Uload           Load a User command's class in the ws
 Utils       in              Define an ATF file's objects in the ws
             inx             Define an extended transfer file's objects in the ws
             out             Create an ATF file from the current ws
             outx            Create an extended transfer file from the current ws
 Workspace   Calls           Returns the calling tree of a function in a scriptfile
             Summary         Returns a summary of the objects in a scriptfile
             Xref            Returns Cross-reference of the objects in a scriptfile
             align           Align comments to start at a specific offset
             bm              Benchmark expression
             cd              Change Directory (DOS)
             display         DISPLAY expression
             fndiff          Compares functions
             fnsLike         Returns a list of fns/ops matching the REGEX pattern
```

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


CLEAR WS - Dyalog APL/W
File Edit View Windows Session Log Action Options Tools Threads Help
WS Object Tool Edit Session

```
      ]?find
Command "Find". Syntax: 1[L] arguments; accepts switches -dir=
Script location: C:\Program Files (x86)\Dyalog\121u\SALT\Spice\Locate

Arg: regex string; shows where the string is found
-dir specifies a folder different than the current workdir
```

Ready...
Ins
CurObj: cmd (Undefined)
&:1
⎕DQ:0
⎕TRAP
⎕SI:0
⎕IO:1
⎕ML:0

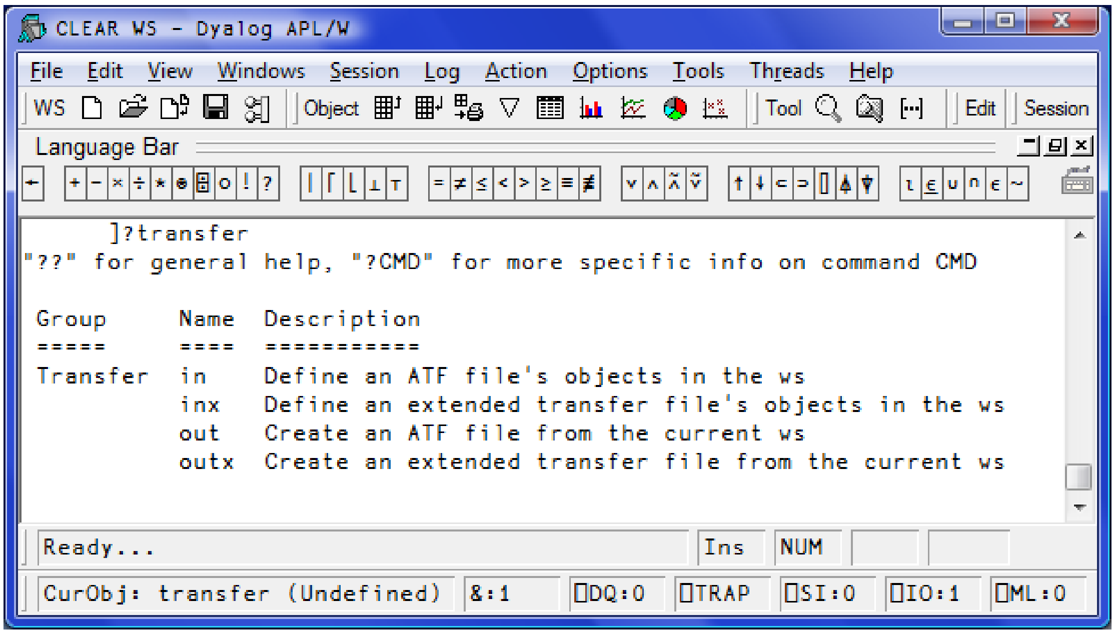
The names of commands are case insensitive, so *FIND* and *find* are the same command.

Upon hitting Enter, the line is sent to the command processor which determines which command has been selected, brings in the code to run it, runs it, and 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:


CLEAR WS - Dyalog APL/W
File Edit View Windows Session Log Action Options Tools Threads Help
WS Object Tool Edit Session
Language Bar

```
      ]?transfer
"??" for general help, "?CMD" for more specific info on command CMD

 Group     Name  Description
 =====     ====  ===========
 Transfer  in    Define an ATF file's objects in the ws
           inx   Define an extended transfer file's objects in the ws
           out   Create an ATF file from the current ws
           outx  Create an extended transfer file from the current ws
```

```
Ready...                                    Ins  NUM
CurObj: transfer (Undefined)  &:1  ⎕DQ:0  ⎕TRAP  ⎕SI:0  ⎕IO:1  ⎕ML:0
```

Creating Commands

A user command is implemented as a class saved in a Unicode text file with the extension .dyalog. If you are unfamiliar with object-oriented programming, do not despair – 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. A single class can host as many commands as you like, so you can get away with doing it once 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:

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

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

    ∇ r←Run(Cmd Args)
      :Access Shared Public
      r←⎕TS[4 5 6]   ⍝ show time
    ∇

    ∇ r←Help Cmd
      :Access Shared Public
      r←'Time (no arguments)'
    ∇
:EndClass
```

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. This information 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. Here there is only one command, so the argument will always be 'time' so we ignore it and always return some help for that command.

The `Run` function is the one executing your code for the command. It is always called with 2 arguments. Here we ignore them as all we do is call `⎕TS`.

We can write this code in a file named `timefns.dyalog` file 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 voila! 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 class, adding a new function `Zulu`[3] and modifying `Run`, `List` & `Help` accordingly. Like this:

```
:Class timefns
    ⎕ML ⎕IO←1

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

    ∇ r←Run(Cmd Args);dt
      :Access Shared Public
      ⎕USING←'System'
      dt←DateTime.Now
      :If 'utc'≡⎕SE.Dyalog.Utils.lcase Cmd
            dt←Zulu dt
      :EndIf
      r←(r⍳' ')↓r←⍕dt ⍝ remove date
    ∇

    ∇ r←Help Cmd;which
      :Access Shared Public
      which←'time' 'utc'⍳⊂⎕SE.U.lcase Cmd
      r←which⊃'Time (no arguments)' 'UTC (no arguments)'
    ∇
```

[1] `⎕SE.SALT.Save 'timefns Spice\timefns'` can also be used.

[2] This requires SALT/Spice version 1.3 or later. If you are not using Dyalog v12.1 or later, check which version you are using type `⎕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←Zulu date
     ⍝ Use .Net to retrieve UTC info
      r←TimeZone.CurrentTimeZone.ToUniversalTime date
    ∇
:EndClass
```

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 the `lcase` utility in **`⎕SE.Dyalog.Utils`**, a namespace of utilities which is used by by SALT, and Spice, but is also available for your commands.

The `Run` function now makes use of the `Cmd` argument and, if it is ‘utc’, calls the `Zulu` function. 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:

```
:Class timefns
    ⎕ML ⎕IO←1

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

    ∇ r←Help Cmd;which
```

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

```
      :Access Shared Public
      which←'time' 'utc'⍳⊂⎕SE.U.lcase Cmd
      r←which⊃'Time [city]' 'UTC (no arguments)'
    ∇

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

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 `Args` 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.

The first argument to `Run` is always the command name (here it is 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’.

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 *flag* 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), `Arg` will simply contain everything following the command name, and you can interpret it any way you like; 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”.

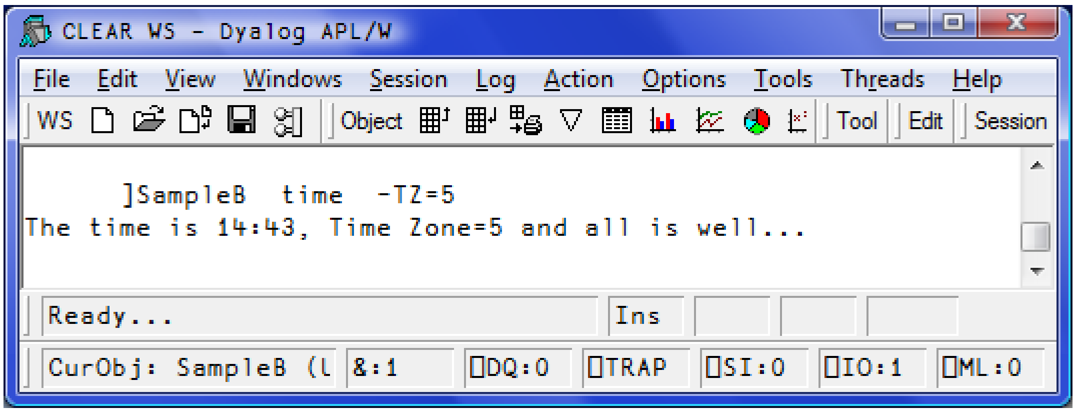
Example #4: The Supplied sample Command

The default installation includes a sample command to demonstrate the use of arguments and switches.

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

In file `aSample.dyalog` you will find a class with 2 commands: one named `sampleA` which does not use the parser and one that does.

The second command, named `sampleB`, uses the parser. It 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:


CLEAR WS - Dyalog APL/W
File Edit View Windows Session Log Action Options Tools Threads Help
WS Object Tool Edit Session
Ready...
Ins
CurObj: SampleB (l
&:1
⎕DQ:0
⎕TRAP
⎕SI:0
⎕IO:1
⎕ML:0

```
      ]SampleB  time  -TZ=5
The time is 14:43, Time Zone=5 and all is well...
```

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 '-' will be 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, 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 element of the argument to `Run Arg` in our example.

`Arg` will contain a vector of text vectors named `Arguments`, with one element per argument (in our example there will always be a single element), 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 the value '`123`'.

`Run` will be called with an argument where the first element is '`sampleB`' and the second element is a namespace containing `Arguments (,⊂'xyz')` and `TZ` (`'123'`). The rest is up to your function.

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
```

Command Execution Failed: too few arguments
T is an unambiguous abbreviation for `TZ`, but 0 arguments was not enough and therefore an error was signaled.

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 or switch delimiters.

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 2nd argument to `Run`.

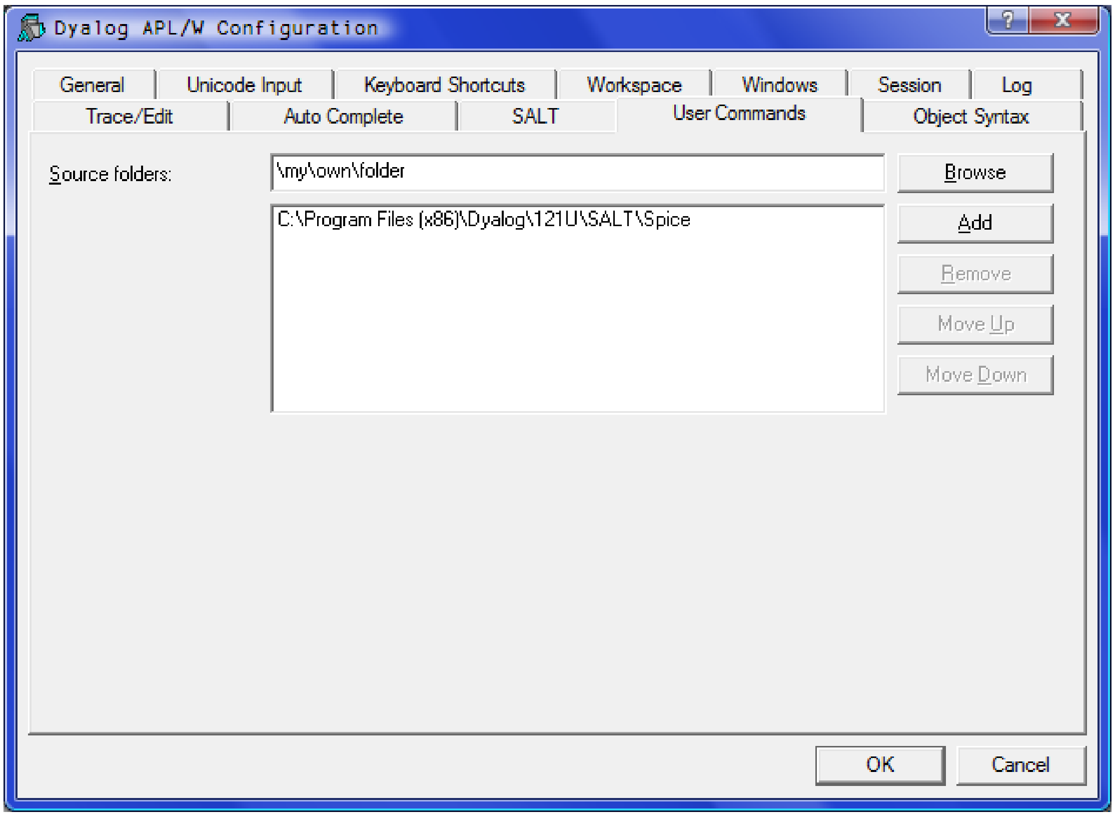
There are a few more things the parser can do but this should cover most cases. See *Advanced Topics* in the following for details.

[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'.

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 effective until the next restart:



Source folders: \my\own\folder

C:\Program Files (x86)\Dyalog\121U\SALT\Spice

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

`]settings` takes 0, 1 or 2 arguments. With 0 arguments, it will display 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 `[Dyalog]\SALT\Spice`, where `[Dyalog]` is shorthand for the mail 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, This will take 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, 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 given as 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 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 e.g. 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`,: `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 had 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 argument, 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= ρ Args.Arguments) then your program takes the proper action.

Activating New Parsing Rules

When you use a command which the framework does not recognize, it will scan the command folder(s) to see whether new commands have been added. However, if you make a change to the parsing rules (or the help) for an existing command, this will *not* be detected automatically. Use the command `]ureset` to force a complete reload of all user commands.

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`, `explore`, `list`, `settings` and `removeversions`. 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

`uload`: allows you to load in the workspace the script related to the command given as argument. This is typically used when you are developing a command in order to modify (debug) it.

There is no need for a `usave` command since the save is handled by the editor's callback function whenever the code is changed. However, there is a command for creating a new command:

`unew`: This command puts up a GUI which allows you to specify command arguments and switches for one or more commands, and puts you into the editor with a script created using the information that you have provided.

`ureset`: Forces a reload of all user commands (required when you have changed the parsing rules for an existing command).

`udebug`: Switches debugging on or off.

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. The example call is simply passed on to Spice using the call:

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

Spice will make UCMD's left argument available to your command via ##.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.

Appendix A – "Public" User Commands

WARNING: Version 12.1 is the first release which includes user commands. The user command mechanism should be considered experimental: While the intention is that user commands built with version 12.1 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 extended and changed in the first couple of releases.

Use `]?` To list commands currently installed.

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

Group Demo

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

Command Demo

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.

Group Monitor

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 CPUTime

This command is used to measure the CPU and Elapsed time required to execute an APL expression. There are two switches, `-repeat=` which allows you to have the expression repeated a number of times, and `-runfor=` which specifies that the expression should be repeated until a number of milliseconds has passed. By default, the expression is executed once. The report always shows the average time for a single execution.

Examples:

```
       ]cputime {+/1=ω∨ιω}¨ι1000
Running "{+/1=ω∨ιω}¨ι1000" 1 times.
 CPU (avg):   94
 Elapsed:    131

       ]cputime {+/1=ω∨ιω}¨ι1000 -runfor=1000
Running "{+/1=ω∨ιω}¨ι1000" for 1000 msec.
 CPU (avg):   98.3
 Elapsed:    100.6
```

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

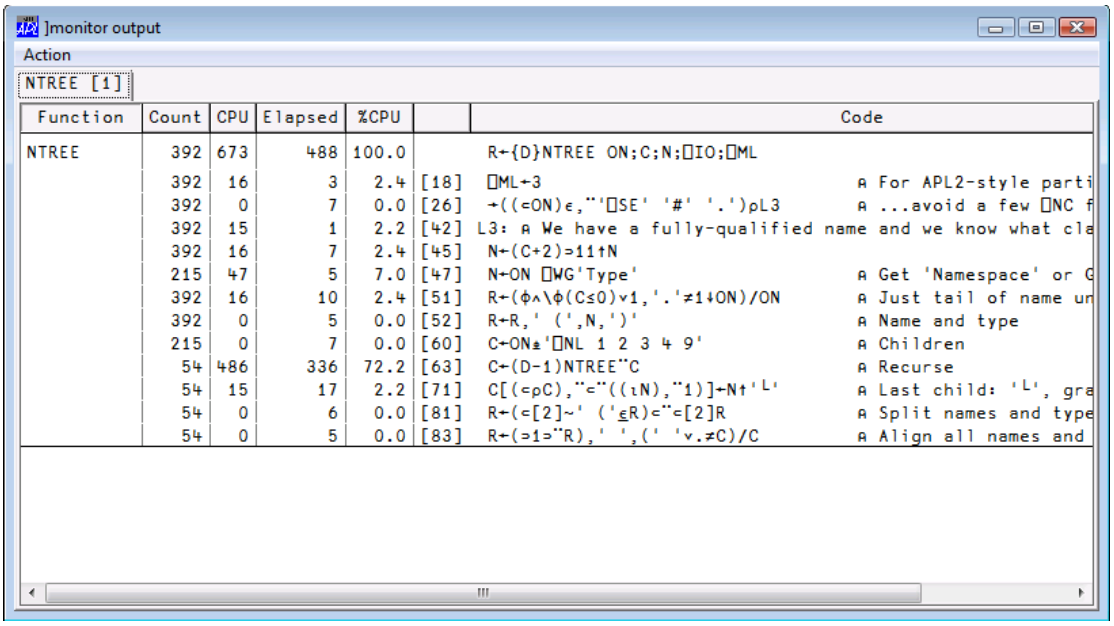
Examples:

```
      ]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 G
	392	16	10	2.4	[51]	R←(⌽∧\⌽(C≤0)∨1,'.'≠1↓ON)/ON ⍝ Just tail of name un
	392	0	5	0.0	[52]	R←R,' (',N,')' ⍝ Name and type
	215	0	7	0.0	[60]	C←ON⍎'⎕NL 1 2 3 4 9' ⍝ Children
	54	486	336	72.2	[63]	C←(D-1)NTREE¨C ⍝ Recurse
	54	15	17	2.2	[71]	C[(⊂⍴C),¨⊂¨((⍳N),¨1)]←N↑'└' ⍝ Last child: '└', gra
	54	0	6	0.0	[81]	R←(⊂[2]~' ('⍷R)⊂¨⊂[2]R ⍝ Split names and type
	54	0	5	0.0	[83]	R←(⊃1⊃¨R),' ',(' '∨.≠C)/C ⍝ Align all names and

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=ω∨ιω}¨ι1000
1000
Written: C:\Users\mkrom.INSIGHT\Documents\Dyalog APL 12.1
Unicode Files\aplmon_20091008004428.csv
```

The above command generated a log file name, enabled APLMON logging, ran the expression (`Foo 1 2 3`), 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 'C:\Users\mkrom.INSIGHT\Documents\Dyalog APL 12.1 Unicode
Files\aplmon_20091008004428.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 |
```

Group SALT

This group contains commands that correspond to the SALT functions of the same name: `Save`, `Load`, `List`, `Compare`, `Explore`, `Settings` and `Rename-Versions`. (found in `⎕SE.SALT`)

Example:

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

This will do the same as

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

Group Sample

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

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 four commands: *UNew*, *USetup*, *UReset* and *Uload*.

Command UNew

This command is used to create a class 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* class: the command names, their groups, their short and long description, details of switches.

Each command's info is entered one after another.

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

Command USetup

This command is used by versions prior to V12.1 to automatically initialize Spice's command bar to the user's preferences.

Command ULoad

This command is used to bring in the workspace the class associated with a user command. It is used when debugging a user command in a *Spice* class.

Example:

```
      ]uload  unew
#.Setup
```

The class `Setup` containing the code the for the `UNew` user command was brought in from file. We can now edit the class and modify the command – when we exit from the editor, the class will automatically be saved back to the script file from whence it came.

Command UReset

Forces a reload of all user commands – this is required when you have changed the parsing rules for an existing command.

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.v12.1” for more details.

Group Tools

Command cd

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

Example:

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

will now switch to \tmp for the remaining of the session.

Command display

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

Example:

```
      ]display ⍳¨⍳2 3
┌→────────────────────────────────────────────────┐
↓ ┌→──────┐ ┌→────────────┐ ┌→──────────────────┐ │
│ ↓ ┌→──┐ │ ↓ ┌→──┐ ┌→──┐ │ ↓ ┌→──┐ ┌→──┐ ┌→──┐ │ │
│ │ │1 1│ │ │ │1 1│ │1 2│ │ │ │1 1│ │1 2│ │1 3│ │ │
│ │ └~──┘ │ │ └~──┘ └~──┘ │ │ └~──┘ └~──┘ └~──┘ │ │
│ └∊──────┘ └∊────────────┘ └∊──────────────────┘ │
│ ┌→──────┐ ┌→────────────┐ ┌→──────────────────┐ │
│ ↓ ┌→──┐ │ ↓ ┌→──┐ ┌→──┐ │ ↓ ┌→──┐ ┌→──┐ ┌→──┐ │ │
│ │ │1 1│ │ │ │1 1│ │1 2│ │ │ │1 1│ │1 2│ │1 3│ │ │
│ │ └~──┘ │ │ └~──┘ └~──┘ │ │ └~──┘ └~──┘ └~──┘ │ │
│ │ ┌→──┐ │ │ ┌→──┐ ┌→──┐ │ │ ┌→──┐ ┌→──┐ ┌→──┐ │ │
│ │ │2 1│ │ │ │2 1│ │2 2│ │ │ │2 1│ │2 2│ │2 3│ │ │
│ │ └~──┘ │ │ └~──┘ └~──┘ │ │ └~──┘ └~──┘ └~──┘ │ │
│ └∊──────┘ └∊────────────┘ └∊──────────────────┘ │
└∊────────────────────────────────────────────────┘
```

Command Find

This command searches the `.dyalog` files in the current working directory for the string given as argument in SALT script files. It needs ONE long argument which is a .Net regular expression.

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

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

Example:

```
      ]find  \b\w{7}\b   -folder=\tmp
```

will find all 7 letter words in .dyalog files in `\tmp`

Command FnCalls

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

It takes 2 arguments: the filename where the class resides and the function to work on. With switch `-details` it can provide details on all the names involved such as if the name is local, global, unused, recursively called, etc.

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

If the switch `-ws` is provided the filename is assumed to be a class in the workspace.

Example:

```
     ]fncalls '\Dyalog APL\12.1\SALT\SALTUtils' Spice[8]
Level 1:  →Spice
⍝ Handle KeyPress in SPICE command window
```

[8] the 1st argument is surrounded by quotes because it contains a space

```
⍝ The function can also be used directly with a string
F:isChar            F:isHelp            F:isRelPath
F:lCase             F:rlb               F:splitOn1st
F:BootSpice         F:GetSpiceList      F:SpiceHELP

Level 2: Spice→isChar
…
Level 2: Spice→BootSpice
⍝ Set up Spice
⍝ In GUI environments we rig Spice fn to be called back
F:GetSpiceList      R:Spice

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

Level 4: GetSpiceList→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 list 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 9 other functions and at the 2nd level function `isChar` calls nothing but `BootSpice` calls 2 functions, `GetSpiceList` and `Spice`, recursively. At the 3rd level `GetSpiceList` calls `ClassFolder` and so on.

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

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

Command fromhex

This command will display an hexadecimal value in decimal

Example:

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

Command Replace

This command searches the `.dyalog` files 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 are .Net regular expressions (see http://msdn.microsoft.com/en-us/library/az24scfc(VS.71).aspx for details).

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

Example:

```
    ]replace Name:\s+(\w+)\s+(\w+) Name: $2, $1 -fol=\tmp
```

will reverse every occurrence of 2 words when they follow 'Name:', i.e

```
Name: Joe Blough
```

will become

```
Name: Blough, Joe
```

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

Command Summary

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

```
    ]summary \Program Files\Dyalog12.1\SALT\Parser.dyalog
name        scope  size  syntax
 fixCase              24   n0f
 if                   24   n0f
 init       PC      4500   n1f
 xCut                532   r2f
 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 Xref

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

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 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
         ccfkllnpsszzzFFILNPPS
         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.

Group WS

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*`.

If the .NET framework is available, 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`[9]. 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 other 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.

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

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.

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=60
```

This will align all comments at column 60 for all functions starting with 'HTML' and display the names of all the functions it modified in `)FNS` format

Command fndiff

This command will show the different lines between 2 functions.

Example:

```
      ⎕fx'f1' 'Af1' '123' 'Ax' 'ef1'
      ⎕fx'f2' 'Af2' '123' 'Ax' 'ef2'
      ]fndiff f1 f2
·f1 ·   ·   ·   ·   ·   ·   │·f2 ·   ·   ·   ·   ·   ·
Af1 ·   ·   ·   ·   ·   ·   │Af2 ·   ·   ·   ·   ·   ·
·ef1·   ·   ·   ·   ·   ·   │·ef2·   ·   ·   ·   ·   ·
```

Command fnslike

This command will show all functions following a same pattern in their names. It accepts the –**regex, -date** and **–format** switches.

Example: display all functions containing the letter 'a'

```
      ]fns a  -format -regex
commandLineArgs disableSPICE   enableSPICE   regGetHandle
disableSALT     enableSALT     qaEmptyCat    BootLocation
```

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 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** and **–format** switches.

Example:

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

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 **-top** switch to limit the number of items shown.

Example:

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

Command varslike

This command will show all variables following a same pattern in their names. It accepts the –**regex** and **–format** switches.

Command wsloc

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 regular expressions and will perform replacement on most objects and APL files. See its documentation (]?wsloc) for details.

Group svn

This group contains a series of commands used as cover to SubVersions functions of the same name. For example, svnci commits changes made to the current working copy.