SALT II

Dan Baronet, Dyalog Ltd
danb@dyalog.com

Introduction

SALT stands for Simple APL Library Toolkit, a code-management tool for APL. It first appeared in V11 as a prototype. Since then it has undergone many changes and is now fully supported by Dyalog. It allows you to store code in flat Unicode text files, often called scripts, outside the workspace. There are many ways to keep code out of the workspace but text files have advantages over traditional APL ways. they let you:

- Exchange, send or archive the scripts without using the interpreter
- Compare scripts easily
- Divide and work on different sections of code in parallel
- Use an external code-management systems

In environments where the ability to manage code externally is important this is a considerable advantage over storing code in workspaces.

Basics

Besides storing and retrieving, SALT allows you to list and view folders of scripts. It can save multiple versions and manage them locally. It can compare them. And it comes with its own set of utilities.

If you share code it is important to be able to manage it, one way or another. You want to be able to:

- Store multiple versions and retrieve *any* of them
- Compare them
- Do housekeeping

Dyalog offers all this with SALT and more. You don't have to use SALT's ability to store multiple versions but it might well be all you need. If a third-party version-control system is used, SALT's versioning should not be used as it would probably interfere rather than help.

SALT can store functions and sourced namespaces [1] onto file. Non-sourced namespaces can be stored but they need to be converted into sourced form first

[2]. At present, there are restrictions and root variables, for example, cannot be stored.

All the SALT code resides in `⎕SE` and is enabled by default in V12.

Simple examples

1. Function `Foo` is defined and we want to keep a scripted copy of it. To do so we enter

   ```
   ⎕SE.SALT.Save 'Foo  \projectZ\fns\Foo'
   ```

   and file `\projectZ\fns\Foo.dyalog` will now contain a copy of the function in text form.

2. Sourced namespace `Utils` contains functions and variables used everywhere. They constitute a set of utilities that must remain together. We want to store the namespace in file `U1.dyalog`:

   ```
   ⎕SE.SALT.Save 'Utils  \projectZ\utils\U1'
   ```

   Note that in this case we made the filename differ from the namespace's name.

3. Sourced namespace `GUIutils` is another set of utilities relying on the above `Utils` namespace. To save it and start using version numbers we add the `version` switch:

   ```
   ⎕SE.SALT.Save 'GUIutils   \projectZ\utils\U2  -version'
   ```

The next time we want to get those two namespaces (and their contents) all we need to do is

```
⎕SE.SALT.Load  '\projectZ\utils\U*'
```

There may be more sets of utilities starting with the letter `U`. If we don't want them all we must load the ones we want one by one. If there are dependencies we can tell SALT by using the SALT tag `⍝∇:require`. For example, if the above `GUI` namespace requires the `Utils` namespace to be present we should insert the line

```
⍝∇:require  \projectZ\utils\U1
```

preferably at the top of the source. If we know that `Utils` is always in the same folder as `GUI` we can use instead

```
 ⍝∇:require  =\U1
```

with the = meaning *same folder as myself*. We then only need to load U2, even if both files are moved together to a different location.

Automatic update

Because SALT intercepts editor events, it can save changes on file right after editing an object, optionally prompting you for confirmation.

In other words, every time you edit a salted object, SALT will come up and ask you if you wish to save the changes back to file, making a new version if necessary.

Making life easier, the settings

Instead of always specifying the path of the objects to save or load you can tell SALT to look in specific places when a relative path (one that does not start with \) is given. To do so you use

```
 ⎕SE.SALT.Settings 'workdir  location1;loc2;…;locN'
```

SALT will *save* objects in location1 if a relative path is given or *look* in each location until the file is found when a load is requested. Each full path must be separated from the next one by a semi-colon.

If you do not wish SALT to confirm with you when saving the changes every time you modify an object you should do

```
 ⎕SE.SALT.Settings 'edprompt  0'
```

This will skip the prompting and the changes will be made to file automatically.

Everyday examples

Let's have a look at a typical use.

Mike has been gathering functions for years. They're all over the place, in various workspaces, sometimes in duplicates, sometimes in triplicates. He wants to clean this up and start using a system to manage his code. SALT gives him two choices:

1. He can store all his functions, each one in a single script file, grouped by topic in folders of his choice.
2. He can start reorganising each workspace in namespaces and store each namespace in a separate file.

Whichever he chooses, he can snap his workspace straight into files like this:

```
⎕SE.SALT.Snap '\my\APL\folder'
```

Each function and namespace will be saved in a file with the same name followed by the `.dyalog` extension.

If he wants to start keeping track of versions he must add the `-version` switch as well. Let's see both cases:

Method 1: store each function in a separate file

Let's assume the workspace looks like this:

```
      )FNS
main    ublock  ucut    uopen   GUI_close       GUI_open
```

To store each function in separate files in the same folder we simply do

```
⎕SE.SALT.Snap  '\projectX\scripts\APL'
```

We can then use a 'Disk Explorer' type program to organise files in folders.

If we decide to reorganise the functions and, say, put the GUI functions together, the utilities (those functions whose names begin with `u`) together and the rest in the top folder, we can do

```
⎕SE.SALT.Snap  '\projectX\scripts\APL\GUI    -stem=GUI'
⎕SE.SALT.Snap  '\projectX\scripts\APL\utils  -stem=u'
```

We now save the remaining functions in `\projectX\scripts\APL`.

```
⎕SE.SALT.Snap  '\projectX\scripts\APL'
```

SALT keeps track of what has been saved and won't save a new copy again.

Method 2: regroup some functions into namespaces first

Same thing. This time we organise the functions into namespaces first. We must create the namespaces and move functions into them. We can

a. create the namespaces:

   ```
   'GUI'  'utils' ⎕ns¨⊂''
   ```

   and then use Workspace Explorer to reorganise (move) the functions or

b. do it manually, e.g.:

   ```
   'utils' ⎕NS  list ← 'u' ⎕NL 3 4  ⋄  ⎕EX list
   ```

We need to make sure the functions do not appear in two places (hence the `⎕EX`).

Then we do all functions and namespaces at once:

```
⎕SE.SALT.Snap '\projectX\scripts\APL'
```

Note that everything (functions and namespaces) are saved in the same location. If you wish to separate them you could do

```
      ⍝ functions in folder 'base'
⎕SE.SALT.Snap '\projectX\scripts\APL\base   -class=3 4'
      ⍝ namespaces in folder 'groups'
⎕SE.SALT.Snap '\projectX\scripts\APL\groups -class=9'
```

When an object has been salted, modifications can be stored to file automatically after making changes via the editor. If you subsequently erase an object or redefine it the tagged information is lost and no automatic update can occur.

Important note

Only *sourced* functions and namespaces can be tagged. Non-sourced functions (e.g. derived functions) and namespaces cannot be tagged and although some of them can be saved (SALT generates source for them subject to constraints such as no GUI object in them), they cannot be edited and must be resaved manually or skipped. If you wish to convert a non-sourced namespace into a sourced one you should use the SALT utilities [3] provided. Saving it will then allow SALT to pick up subsequent changes automatically.

Saving new code

Once your code is in SALT your changes will be picked up automatically if you wish. If you disable SALT or if you decline to save changes when prompted (perhaps you want to test before filing the changes) you will end up with code that is not in SALT yet. You can `)SAVE` the workspace and resume later as all the tagged information is kept unless you deliberately remove it.

With SALT enabled you can pick up all the new code by using `Snap` again or you can use `Save` for an individual item.

Using your code

We've seen how to move code outside the workspace into text files. Now is the time to use that code. The function to bring the code in is `Load`. You give it a filename and it defines the code in the workspace, ready for use. Let's go back to Mike's code.

To bring everything back in we do

```
⎕SE.SALT.Load '\projectX\scripts\*'
```

If namespaces were saved there they will now also appear as namespaces in the root of the workspace.

If other needed namespaces were saved elsewhere they have to be brought in, separately:

```
⎕SE.SALT.Load '\projectX\scripts\APL\GUI\*'
⎕SE.SALT.Load '\projectX\scripts\APL\utils\*'
```

All these objects are tagged by SALT, and editing any of them will get the modifications saved back to file, if you wish. If you do *not* want the objects to be so tagged use the switch `nolink`, e.g.:

```
⎕SE.SALT.Load '\projectX\scripts\* -nolink'
```

You might typically do that in a production environment.

You might also prefer to keep some functions together, saved in a single file (a namespace), but to have them in the workspace (in the root by default) and not in a namespace. You use the `disperse` switch for that:

```
⎕SE.SALT.Load '\projectX\scripts\* -disperse'
```

You can even choose which objects to disperse:

```
⎕SE.SALT.Load '\projectX\scripts\* -disperse=fn1,fn2,opx,varX'
```

This method has the advantage of being able to define variables. A serious disadvantage, though, is that this version of SALT cannot keep track of where the objects came from: changes are *not* picked up and filed automatically.

Tackling variables

SALT cannot tag variables and save them individually on file.

There are two ways around this:

1. Initialise the variables in a function and call the function before doing anything else:

```
          ⎕SE.SALT.Load 'initfn' ⋄ ⎕VR 'initfn'
         ∇ initfn
    [1]  GlobalSetting←'PROD'
         ∇
```

2. Put the variables in a namespace and disperse its contents. If `globalVars` was saved thus:

```
:Namespace globalVars
   GlobalSetting←'PROD'
:EndNamespace
```

then

```
⎕SE.SALT.Load 'globalVars -disperse'
```

would define `GlobalSetting` in the workspace root.

Version control

One of the main reasons for using version control is to be able to go back in time and retrieve previous versions.

Let's say you had code working fine at version 3 and you made a series of changes that have brought you to version 5 and you now realise there is a problem or a difference in behaviour. If you want to check and run that previous code you can retrieve it simply by doing

```
⎕SE.SALT.Load 'mainCode  -version=3'
```

If you'd rather only see what the difference is you can use `Compare`:

```
⎕SE.SALT.Compare 'mainCode  -version=3 5'
```

SALT will use its own code to do the comparison but you prefer to use your favourite file-comparison program located, say, [4] in `[ProgramFiles]\X` then tell SALT [5]:

```
⎕SE.SALT.Compare'mainCode -v=3 5 -use=[ProgramFiles]\X\cm.exe'
```

SALT's way of tracking versions is very simple and each file can have its own version number. SALT has no locking mechanism and does not allow you to 'lock out' files but it will warn you if it detects that a file has been revised when you try to save back a modified object. If you see such a warning and you don't know why, then you should investigate.

For serious version control in a large system you should use a more robust system like SubVersion or CVS. In that case you should *not* be using SALT's versions at all.

If you decide to use versions you should keep in mind that SALT creates a new file each time a modification is made to an object. After a while you might end up with a large number of files, many of which you are uninterested in keeping. For example, let's say you started working on namespace `NmspX` at version 15. After having modified it sixty times you are now at version 75. If you are happy with the result and confident that versions 16 to 74 are useless you can use Explorer to get rid of the files `NmspX.16.dyalog` to `NmspX.74.dyalog`. There will be a gap between 15 and 75 but that might not matter.

If you would like to collapse those sixty versions into one, bringing the good version 75 to a version just above the current good version 15 (i.e. 16) you can get SALT to do it like this:

```
 ⎕SE.SALT.RemoveVersions 'mainCode  -version=>15 -collapse'
```

SALT will confirm with you the deletion of (here) 60 versions, delete 59 of them and rename the last one to version 16. The collapse switch is used to keep the last version. Without it the last 60 versions would be deleted (which may also be what you want to do).

Epilogue

You might not need SALT or any version control system at all. If all you have is a small system that runs well in 1 or 2 workspaces and needs little or no maintenance then SALT would not be very useful to you.

If you already have your own management system that takes care of all the maintenance problems and you don't need to use text files then, again, SALT would not improve your life much.

But if you don't have such a system, and your code requires a fair amount of maintenance, then SALT can help. It is already there and it is free.

As you have seen, SALT supports many ways to organise your code. It is good at saving and retrieving code, keeping track of versions and managing them. Coupled with a professional external version-control system it could solve many problems.

SALT comes with V12 but will be available online before 2009.

Check it out.

Notes

1. Sourced namespaces are those for which the `⎕SRC` function returns a canonical representation similar to what `⎕NR` returns for functions.

2. There are SALT utilities to do this.
3. You will find those utilities in `lib/NStoScript`. The main function is `Convert`. It takes a namespace as argument and turns it into a sourced namespace.
4. `[ProgramFiles]` is where Windows keeps all programs. SALT will replace it by whatever value is appropriate.
5. SALT uses a consistent syntax for each function: a single string argument that describes the command arguments and the switches that start by a dash. Each switch can be shortened to a non-conflicting length. Here there are no other switches starting with `v` so `-v` is sufficient to denote `-version`.