SALT
User Guide

SALT version 2.9



DYALOG

The tool of thought for software solutions

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

SALT User Guide

SALT version 2.9
Document Revision: 20260907_290

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

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

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

TRADEMARKS:

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

All other trademarks and copyrights are acknowledged.

Contents

1 About This Document **1**
1.1 Audience 1
1.2 Conventions 1
2 Introduction **3**
2.1 History 3
2.2 The Benefits of SALT 4
2.3 SALT as a Source Code Management System 4
3 Using SALT **5**
3.1 Installation 5
3.2 Configuration 5
3.3 Structure within Dyalog 6
3.3.1 Defining the SALT Environment Variable 6
3.4 File Format 8
3.4.1 .dyapp Files 9
3.4.2 .dyalog Files 9
3.5 Nameclasses 10
3.6 Tag Information 10
3.7 SALT Applications 11
3.7.1 Autostarting SALT Applications 11
3.8 Class Dependencies 13
3.9 File Comparison 15
3.10 Version Management 15
4 SALT Functions **16**
4.1 Calling SALT Functions 17
4.1.1 Paths and Filenames 18
4.2 Boot Function 19
4.2.1 Syntax 19
4.2.2 Use 20
4.3 Clean Function 20
4.3.1 Syntax 20
4.3.2 Use 21
4.4 Compare Function 21
4.4.1 Syntax 22
4.4.2 Use 23
4.5 List Function 24
4.5.1 Syntax 24
4.5.2 Use 26
4.6 Load Function 27
4.6.1 Syntax 28

4.6.2 Use 29
4.7 New Function 30
4.7.1 Syntax 31
4.7.2 Use 31
4.8 RemoveVersions Function 31
4.8.1 Syntax 32
4.8.2 Use 33
4.9 Save Function 33
4.9.1 Syntax 34
4.9.2 Use 35
4.10 Settings Function 36
4.10.1 Syntax 36
4.10.2 Use 37
4.10.2.1 Parameters 38
4.11 Snap Function 41
4.11.1 Syntax 41
4.11.2 Use 44
A Configuration Options 47
A.1 Configuration Dialog Box 48
B SALT Function Syntax Summary 49
C Example: SALT in Use 51
Index 56

1 About This Document

This document is intended as an introduction to SALT and a reference guide for its functions, their syntax, modifiers and modifier values.

Although the behaviour of SALT is independent of the operating system and whether a classic/Unicode installation is used, some of the information in this document is operating system-specific (for example, the location of global parameters). The differences between this document and the SALT experience on a UNIX operating system are detailed in the *Dyalog for UNIX Installation and Configuration Guide* and the *Dyalog for UNIX UI Guide*.



Although SALT is still fully supported, Dyalog Ltd expects that *Link* will replace SALT as the mechanism for using text files as APL source code and recommends migrating from SALT to Link as soon as is convenient to do so. For more information, see the *Link User Guide*.

1.1 Audience

It is assumed that the reader has a reasonable understanding of Dyalog and possesses basic computer skills.

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

1.2 Conventions

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

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

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


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


Material of particular significance or relevance.


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


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

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



2 Introduction



Although SALT is still fully supported, Dyalog Ltd expects that *Link* will replace SALT as the mechanism for using text files as APL source code and recommends migrating from SALT to Link as soon as is convenient to do so. For more information, see the *Link User Guide*.

SALT – the Simple APL Library Toolkit – is a technology for storing variables, functions, operators, namespaces and classes in a human-readable form in standard operating-system text files. These files can subsequently be manipulated using a programming interface (API) or by a set of user commands.

User commands are separate from SALT but a group of them perform the same actions as the SALT functions. For more information on user commands, see the *User Commands User Guide*.

2.1 History

The first version of SALT was introduced with Dyalog version 11.0; this introduced scripts representing entire namespaces and classes. Each script was saved as an individual file. However, for many APL users the individual function is a more natural unit and SALT now has the capacity to store scripts representing functions and variables. One of SALT's functions, `Snap`, also enables the construction of a directory structure corresponding to the namespace structure of a workspace, where each file in the structure contains the script of an APL object in the workspace.

Dyalog version 18.0 introduced new mechanisms for launching the APL interpreter from APL source files – the LOAD and LX configuration parameters (for more information on these parameters, see the *Dyalog for Microsoft Windows Installation and Configuration Guide*). This superseded the use of **.dyapp** files described in *Section 3.4.1*, which remain supported for backwards compatibility.

From Dyalog version 18.2, the recommended mechanism for using and managing text files as APL source code changed from SALT to Link. For more information on Link, see the *Link User Guide* (this includes the advantages of using Link over SALT, and guidelines for migrating from SALT to Link).

2.2 The Benefits of SALT

With SALT, the source code (script) of each APL object is stored in a single Unicode (UTF-8) text file – these files can subsequently be loaded into an APL session to recreate the code. Multiple versions of each file can be created and managed locally, and third-party distributed version control and source code management systems can act as repositories for them.

The common file format means that APL users can develop and share code in open source libraries and the files (and their constituent APL source code) can be manipulated by a wide variety of industry standard tools. Each file can be transferred to any version of Dyalog, easily imported into other APL systems, emailed to another user, viewed and edited in a variety of editors or compared with other files (or versions of the same file) using standard comparison tools.

SALT makes it straightforward to use code management systems like Microsoft Visual Studio, Apache Subversion or Git to manage APL source code. SALT is designed to allow the use of these tools without changing the way in which many APL developers often trace and edit code into existence. Whenever a SALTed function, class or namespace is edited using the built-in Dyalog code editor, the changes can automatically be written back to the external source file and then committed to the external repository at some later stage, as appropriate; it is not necessary to bring the system back to a rest state to save code changes.

2.3 SALT as a Source Code Management System

SALT's mechanism for storing and comparing multiple versions of the same source file uses a simple file naming technique that inserts version numbers into the filenames. Although this is sufficient for small projects, for larger projects Dyalog Ltd recommends the use of external source code management systems, for example, Git, Apache Subversion, Concurrent Versions System (CVS) or Microsoft Visual Studio; these include much more sophisticated mechanisms for managing branches, releases and conflict resolution, essential when multiple people are working on the same project.

3 Using SALT

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

3.1 Installation

SALT is installed automatically with Dyalog.

3.2 Configuration

By default, opening a Dyalog session window activates SALT (after start-up, having SALT active has no performance impact on Dyalog). However, if SALT needs to be disabled for any reason then it can be. Disabling SALT has no impact on Dyalog other than the inability to automatically save edited code, for example, user commands can still be run.

SALT can be enabled/disabled by enabling functions in the SALT workspace (a specific workspace that should only be used for enabling/disabling SALT), specifically:

```
)LOAD SALT
enableSALT
```

or:

```
)LOAD SALT
disableSALT
```

respectively.



On the Microsoft Windows operating system, SALT can also be enabled/disabled through the **Configuration** dialog box – this allows additional configuration options to be set at the same time (see *Section A.1*).

This document assumes that SALT is enabled.

3.3 Structure within Dyalog

Within **[SALT]** (by default, this is the **[DYALOG]\SALT** directory) are five sub-directories:

- the **core** directory contains SALT's source code
- the **lib** directory contains SALT utilities
- the **spice** directory contains basic user commands (for more information on user commands, see the *User Commands User Guide*)
- the **study** directory contains code that is referenced in the Dyalog documentation set
- the **tools** directory contains developer tools

The **SALT** directory can be moved to a different location. However, in this situation an environment variable called SALT must be created to inform Dyalog of the **SALT** directory's new location (see *Section 3.3.1*).



The structure under the **SALT** directory must not be modified and the five sub-directories must not be renamed.

SALT comprises a series of programs stored in one class and three namespaces, all within the system namespace ⎕SE. When SALT is enabled, the latest versions of the **SALTUtils.dyalog**, **SALT.dyalog**, **Parser.dyalog** and **Utils.dyalog** files are loaded from the **[SALT]\core** directory into ⎕SE – these files must not be removed if SALT is going to be used.

3.3.1 Defining the SALT Environment Variable

If the **SALT** directory is moved to a different location (see *Section 3.3*) then an environment variable called SALT must be created to inform Dyalog of the **SALT** directory's new location.

Defining an environment variable is operating system-specific.

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

1. Do one of the following (can be version-dependent):
   i. In the Start menu, right-click on **Computer** and select **Properties** from the drop-down menu.
   ii. Press the key and **Pause** key together.

The **System** window is displayed.

2. In the **Control Panel Home** pane, click **Advanced system settings**.

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

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

   The **New User Variable** dialog box is displayed.
6. In the **Variable name** field, enter SALT.
7. In the **Variable value** field, enter **<full path>\<directory name>** where the **SALT** directory is now located.
8. Click **OK** to create the new environment variable and exit the **New User Variable** dialog box.
9. Click **OK** to exit the **Environment Variables** dialog box.
10. Click **OK** to exit the **System Properties** window.
11. Close the **System** window.

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

1. Open the **cmd.exe** application.
2. At the command prompt, enter:

   ```
   dyalog.exe SALT=[SALT]
   ```

   where `[SALT]` is the new **<full path>\<directory name>** of the **SALT** directory.

To define the SALT environment variable on UNIX (temporary method – for session duration only)

1. Open a shell.
2. At the command prompt, enter:

   ```
   SALT=[SALT] dyalog
   ```

   where `[SALT]` is the new **<full path>\<directory name>** of the **SALT** directory.

To define the SALT environment variable on macOS (permanent method)

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

3.4 File Format



For ease of use, Dyalog recommends that the name of a file is the same as the name of the SALTed item it contains.

SALT works with any files, but files with the following extensions are of particular interest:

- **.dyapp** – see *Section 3.4.1*.
- **.dyalog** – see *Section 3.4.2*.



Although **.dyapp** files are supported for backwards compatibility, Dyalog Ltd recommends launching the interpreter directly from any APL source or configuration file (functionality introduced with Dyalog version 18.0) rather than through the now-superseded **.dyapp** file mechanism.

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



By default, double-clicking on a **.dyapp** file opens it using **dyalog.exe** and double-clicking on a **.dyalog** files opens it using **dyalogrt.exe** (that is, the standalone editor).

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



APL objects that have been saved using SALT (that is, by calling either the `Save` or the `Snap` function – see *Section 4.9* and *Section 4.11* respectively) are referred to as *SALTed*.

3.4.1 .dyapp Files



Although **.dyapp** files are supported for backwards compatibility, Dyalog Ltd recommends launching the interpreter directly from any APL source or configuration file (functionality introduced with Dyalog version 18.0) rather than through the now-superseded **.dyapp** file mechanism.

Files with the **.dyapp** extension comprise a **.dyapp** script, each line of which is either a `Load` instruction, a `Target` instruction or a `Run` instruction:

- `Load` instructions specify the full path and filename of the file to be loaded
- `Target` instructions change the target environment
- `Run` instructions specify the name of the method to run

The **.dyapp** script must include at least one `Run` command.

For example, a **.dyapp** file could consist of the following lines:

```
Target #
Load study\files\ComponentFile
Load study\files\KeyedFile
Load MyApp
Run MyApp.Main
```

Files with the **.dyapp** extension can also contain a niladic or monadic tradfn; double-clicking on these files allows *bootstrap loading* of a Dyalog application.

Starting a **.dyapp** file that has been created by the user runs that file in a clear workspace. If the **.dyapp** file has been created by the `Snap` SALT function then it runs in a workspace with the same name as the workspace from which it was created. For more information on the `Snap` function, see *Section 4.11*.

3.4.2 .dyalog Files

Files with the **.dyalog** extension contain the source for a single APL object (that is, variable, function, operator, interface, namespace or class) – SALT identifies the content from the initial characters of the file (for more information on source files, including declaration statements and permitted constructs, see the *Dyalog Programming Reference Guide*).

3.5 Nameclasses

The specific subclasses of nameclasses that can be manipulated using SALT functions comprise:

- nameclass 2 (arrays) – 2.1 (variables)
- nameclass 3 (functions) – 3.1 (tradfns), 3.2 (dfns)
- nameclass 4 (operators) – 4.1 (tradops), 4.2 (dops)
- nameclass 9 (namespaces) – 9.1 (namespaces), 9.4 (classes), 9.5 (interfaces)



Nameclasses 3.3 (primitive or derived function) and 4.3 (primitive or derived operator) cannot be manipulated using SALT – attempting to do so can result in a loss of data.

The source code for each APL object is stored in a single Unicode text file with a default file extension of **.dyalog**. SALT also supports the loading and starting of applications from an application file with an extension of **.dyapp**.

3.6 Tag Information

When SALT first saves an APL object, it applies a *tag* to that object; subsequent saves of the SALTed object update the information contained in the tag. Tag information includes the source filename, the version number (if applicable) and the last write time of the file when it was loaded (which is used to prevent accidental updates of the same version by two different users or from two different sessions). This tag information is recorded in different locations depending on the nameclass:

- for nameclass 2 (variables) the tag information is recorded in a special namespace under `#` called `SALT_Var_Data`. This comprises a table with one row pertaining to each variable maintained in SALT.
- for nameclass 3 (functions) and nameclass 4 (operators) the tag information is recorded in a special comment that is appended to the code.
- for nameclass 9 (namespaces) the tag information is recorded in variables within a special namespace named `SALT_Data`. No tag information is recorded for non-scripted namespaces.



The namespace names `SALT_Data` and `SALT_Var_Data` are reserved for this purpose – no user-defined namespace should use these names.

SALT uses the information stored in the tag to determine where to save any changes to the object, whether a new version is required and whether the original file has been modified externally since it was loaded.

Tags should never be changed manually – removing or altering a tag can cause SALT to behave unpredictably or fail.

3.7 SALT Applications

In addition to managing individual source code files, SALT can load and run applications that are defined by files with an extension of **.dyapp** (for information on the format of **.dyapp** files, see *Section 3.4.1*). SALT starts these applications in Dyalog.

3.7.1 Autostarting SALT Applications

Although **.dyapp** files are supported for backwards compatibility, Dyalog Ltd recommends launching the interpreter directly from any APL source or configuration file (functionality introduced with Dyalog version 18.0) rather than through the now-superseded **.dyapp** file mechanism.

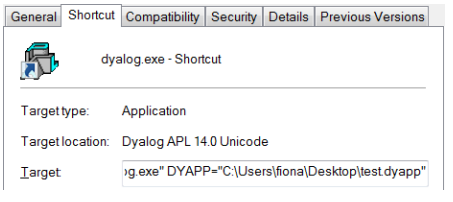
Defining the DYAPP environment variable is operating system-specific.

By default, every Dyalog session opens with a clear workspace – this default can be changed by adding `DYAPP="<path and name of a .dyapp file>"` to the command line that starts Dyalog. In this situation, SALT calls the `Boot` function (see *Section 4.2*) on the specified **.dyapp** file.

To define the DYAPP environment variable on Microsoft Windows

1. Right-click on the Dyalog icon and select **Properties** from the pop-up menu that is displayed.

   The **Properties** dialog box is displayed.
2. In the **Shortcut** tab of the **Properties** dialog box, add `DYAPP="<path and name of a .dyapp file>"` to the end of the path specified in the **Target** field.



3. Click **OK** to close the **Properties** dialog box.

   Opening Dyalog from the icon now automatically loads and runs the specified **.dyapp** file.

or:

1. Do one of the following (can be version-dependent):
   i. In the Start menu, right-click on **Computer** and select **Properties** from the drop-down menu.
   ii. Press the key and **Pause** key together.

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

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

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

   The **New User Variable** dialog box is displayed.
6. In the **Variable name** field, enter SALT.
7. In the **Variable value** field, enter **<full path>\<directory name>** where the **SALT** directory is now located.
8. Click **OK** to create the new environment variable and exit the **New User Variable** dialog box.
9. Click **OK** to exit the **Environment Variables** dialog box.
10. Click **OK** to exit the **System Properties** window.
11. Close the **System** window.

To define the DYAPP environment variable on Linux

1. Open a shell.
2. At the command prompt, enter:

   `DYAPP=<path>\<filename>.dyapp [DYALOG]\mapl`

   where `<path>\<filename>` is that of the **.dyapp** file.

To define the DYAPP environment variable on macOS

1. Open the **$HOME/.dyalog/dyalog.config** file in your preferred text editor.
2. Add the following:
   ```
   export DYAPP=<path>\<filename>.dyapp [DYALOG]\mapl
   ```
   where `<path>\<filename>` is that of the **.dyapp** file.

This means that a **.dyapp** file can be used to auto-start (load and run) Dyalog applications that are based on SALT.



Once an application has been started in this way, additional source code can be added using the ⎕CY system function or other mechanisms; it is not necessary for SALT to be used to include additional source code.

3.8 Class Dependencies

Classes can be defined in a hierarchical structure. A single script file does not have to contain a complete class hierarchy, but can be limited to a single class with zero or more dependencies. This means that a single script file can include a class that has dependencies on another class without the class on which it is dependent being present in the file.

However, SALT cannot successfully load a file that includes dependencies on another class/namespace unless the depended-on class/namespace is already present in the namespace that the file is being loaded to.

SALT does not perform any dependency analysis, therefore, to ensure that the necessary base class/namespace is loaded before a dependent class/namespace, SALT must be instructed to load the pertinent script file to fulfil the class's/namespace's dependency criteria. This is done by adding a statement in the dependent class's/namespace's script file that takes the following format:

```
:Require file://<path/filename>.dyalog
```

where `path/filename.dyalog` is the path (relative to the location of the dependent class's/namespace's script file) and filename of the script file containing the necessary base class/namespace.



Prior to Dyalog version 15.0, the now-deprecated syntax was `]:require path/filename.dyalog` and the `path` could be set to `=` if the file was in the same directory as the script calling it.

SALT follows the path and loads the specified file, thereby satisfying the dependency. This instruction should be included whenever a dependent class is present in a script file – SALT can progress through multiple files and instructions.

EXAMPLE

Class D is derived from base class B. In the **.dyalog** script file that defines class D, this relationship is specified in the initial statement as:

```
:Class D : B
...
:EndClass
```

Classes B and D both exist in the current workspace; this means that, when class D is edited, the reference to class B is found immediately.

SALT is used to store classes B and D as text files.

If an attempt is made to load class D in a clear workspace, then the attempt will be unsuccessful – class D cannot be created because base class B is not present in the clear workspace (class B must be loaded before class D can be loaded).

To instruct SALT that class B is required and must be loaded whenever class D is needed, the following line should be added near the top of class D's declaration (must not be within a function):

```
:Require file://<path to B>/<B>.dyalog
```

If B is located in the same directory as class D, then the path to class B can be omitted, that is:

```
:Require file://<B>.dyalog
```

The **.dyalog** script file that defines class D is, therefore, specified as:

```
:Class D : B
:Require file://<B>.dyalog
...
:EndClass
```

In this situation, class D and class B can both be moved to a different directory without having to change the **.dyalog** script file that defines class D.

3.9 File Comparison

SALT has an integral comparison tool that can identify the differences between two different versions of the same script file (or two different script files) and display the results in the active workspace. However, any Unicode-capable file comparison tool that can be launched using a command which takes as its arguments the name of the two files to be compared can be used instead.

To change the file comparison tool used by SALT, call the `Settings` SALT function (see *Section 4.10*). For example:

```
]SE.SALT.Settings 'compare path/filename of tool'
```

To perform a comparison, SALT appends the names of the files to be compared and calls the specified comparison tool. If this tool is not available, then the task will fail.

3.10 Version Management

By default, SALT maps an APL object to a single file – any change made to the APL object is saved by overwriting that file. However, SALT allows versioning to be applied to files. Versioning is switched on for a file by including the `-version` modifier, optionally with a numerical modifier value, when saving that file (see *Section 4.9*). In this situation, SALT saves the file with the specified name and adds a version number immediately before the **.dyalog** extension, for example, **MyClass.3.dyalog**. The `List` SALT function shows this number in `[]`, for example, `[3]` (see *Section 4.5*).

Each time that an APL object within a versioned file is changed, SALT creates a new file with an incremented number. Over time, this can result in a large number of superfluous files – the `RemoveVersions` SALT function can be used to delete a specified range of these (see *Section 4.8*).



If a SALTed function is updated or created in any way other than through the editor (for example, using]FX or creating a single-line dfn or dop by direct assignment), then SALT does not create a new version of the file.

Once versioning has been switched on for a file, it remains switched on until specifically switched off. To switch off versioning and return to a single instance of the file, the `RemoveVersions` SALT function must be called with the `-all` modifier and without the `-collapse` modifier (see *Section 4.8*); this removes the version number from the latest (highest numbered) file and deletes all other versions of that file.

4 SALT Functions

SALT's functionality is accessed through the functions summarised in *Table 4-1*.

An example including calls to all SALT's functions is described in *Appendix C*.

Table 4-1: *SALT Functions*

Function	Description
`Boot`	Executes a script file or loads and initialises an application using a script instead of a saved workspace.
`Clean`	Removes the tag information from an object.
`Compare`	Compares two versions of an APL object or two different APL objects.
`List`	Lists the files and/or directories in a specified location.
`Load`	Loads an APL object from a file.
`New`	Instantiates an object from a class without naming the class in the workspace.
`RemoveVersions`	Deletes a version (or range of versions) of a versioned file.
`Save`	Saves an APL object to a file.
`Settings`	Reports/Changes session/external repository settings.
`Snap`	Saves all the new and modified APL objects in a workspace to files.

This chapter details these functions, their syntax, modifiers and modifier values.



The `Compare` and `RemoveVersions` functions have been deprecated in favour of third-party version control software.

4.1 Calling SALT Functions

SALT functions are called with the following syntax:

```
]SE.SALT.<function> <arguments> <-modifiers>
```

Within this syntax, `SALT` and `<function>` are case sensitive but `]SE`, `<arguments>`and `<-modifiers>` are not.

Modifiers and their associated modifier values must be separated by the `=` character, for example `-version=3` or `-format=APL`. A modifier that cannot have a modifier value but can only be present or absent is sometimes referred to as a *flag*.

When multiple modifiers are included in a SALT function call, the order in which they are specified is irrelevant.

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



Although functions can be successfully called with abbreviated modifiers, good practice dictates that function calls within programs should always use the full name of the modifier – this future-proofs the calling code against enhancements that might otherwise result in ambiguity.

The notation used when describing the syntax for a SALT function in this document is as follows:

- square brackets `[ ]` indicate an optional modifier
- curly braces `{ }` indicate a mandatory modifier
- a vertical line `|` separates mutually exclusive modifiers
- italic text indicates an element that must be populated by the user

Calling any SALT function with an argument of `'?'` returns a list of all available modifiers for that function. The `Load` and `RemoveVersions` functions return shy results, so a `⊢` should also be included before `]SE` to view the list of all available modifiers, for example, `⊢]SE.SALT.Load '?'`.

4.1.1 Paths and Filenames

Most SALT functions require the file on which they are to act to be specified by providing a path and filename. The path can either be an absolute path or a relative path following a specific convention:

- `./<relative path starting from the current directory>`

  To identify the current directory, enter the `]CD` user command – the value returned is the absolute path to the current directory and can be replaced in your absolute path by `.`.

  For example, if `]CD` returns a value of `c:/Users/Andy`, then `.` is `c:/Users/Andy`.

- `../<relative path starting from the directory that is the parent of the current directory>`

  To identify the directory that is the parent of the current directory, enter the `]CD` user command – the value returned is the absolute path to the current directory. This, when truncated by one level, can be replaced in your absolute path by `..`.

  For example, if `]CD` returns a value of `c:/Users/Andy`, then `..` is `c:/Users`.

- `[ws]/<relative path starting from the directory containing the active workspace>`

  A previous convention that used `⍵/` instead of `[ws]/` has been deprecated; although still supported in this version of SALT, support will be removed in a later version and Dyalog Ltd does not encourage its use.

  To identify the directory containing the active workspace, enter the `)WSID` system command – the value returned is the absolute path and name of the active workspace, the path component of which can be replaced in your absolute path by `[ws]/`.

  For example, if `)WSID` returns a value of `c:/Users/Vince/myworkspace`, then `[ws]` is `c:/Users/Vince`.

  If `)WSID` returns a value that does not have a path (that is, only the name of the workspace is returned), then `[ws]/` acts in the same way as `./`.

- `<relative path starting from the first directory named in the workdir session parameter>` (for details of this session parameter, see *Section 4.10.2.1*)

  To identify the first directory named in the *workdir* session parameter, enter the `⎕SE.SALT.Settings 'workdir'` function call.

When specifying a path as an argument:

- SALT accepts either \ or / as the separator character.

  Dyalog Ltd recommends using / as the separator character for cross-platform compatibility

- if the path (or filename) contains space characters, then the entire path and filename should be enclosed within single or double quotation marks.

If no extension is specified for a filename, then the file is assumed to be a **.dyalog** file.

When saving a file using the `Save` or `Snap` functions, omitting the filename and specifying the path with a trailing \ or / separator character sets the filename(s) to be the same as the name of the SALTed item it contains.

4.2 Boot Function

The `Boot` function either executes a **.dyalog** script file containing a function or uses a **.dyapp** file to describe the loading and initialisation of an application instead of a saved workspace.



Although **.dyapp** files are supported for backwards compatibility, Dyalog Ltd recommends launching the interpreter directly from any APL source or configuration file (functionality introduced with Dyalog version 18.0) rather than through the now-superseded **.dyapp** file mechanism.



If a **.dyalog** script file is used then it can only comprise a single niladic or monadic traditional function.

The `Boot` function does not return any results although the executed function might; in this situation the result returned by the executed function is ignored.

4.2.1 Syntax

for a **.dyapp** file:

```
]SE.SALT.Boot '{path/filename}[.dyapp]'
```

for a **.dyalog** file:

```
]SE.SALT.Boot '{path/filename}{.dyalog} [-xload]' ['argument']
```

where:

- `path/filename` is the full path and filename (without an extension) of the script file to load and initialise.

- `-xload` prevents the information recorded by ⎕LX from being executed when recreating a workspace.
- `argument` is the right hand argument to supply to the monadic function in the **.dyalog** script file.

4.2.2 Use

When the `Boot` function is called to execute a **.dyalog** script file containing a function, the function could be a monadic traditional function. In this situation the function requires a right argument before it can be executed. For example:

```
⎕SE.SALT.Boot 'c:\longpath\myFn.dyalog' 'ABC'
```

The `Boot` function passes the value `'ABC'` as a right argument to the function resulting from the load of the **myFn.dyalog** file. No result is required, so any returned value is discarded. If the function within the **myFn.dyalog** file does not take an argument then the specified argument is ignored.

In practice, the `Boot` function is often used in conjunction with the `Snap` function (see *Section 4.11*). In this situation the code includes a statement to execute ⎕LX. To prevent ⎕LX from executing, the modifier `-xload` must be specified.

4.3 Clean Function

SALT maintains the links between SALTed objects and their external source files using tags (see *Section 3.6*). The `Clean` function can be used to safely remove the tag information from an object (something that should not be done manually), breaking its link with all external source files.

4.3.1 Syntax

```
⎕SE.SALT.Clean '[objects] [-deletefiles]'
```

where:

- `objects` is a space-separated list of the specific objects in the current namespace from which tag information is to be removed. Objects can be namespaces, scripts, or variables. By default, tag information is removed from all objects in the current namespace.
- `-deletefiles` deletes the files that were associated with the specified objects (in addition to removing the tag information from the objects).

Unlike most modifiers, the name of the `-deletefiles` modifier must be entered in full. This is to reduce the risk of deleting files unintentionally.

4.3.2 Use

When in the root namespace, all tags on all objects in the workspace can be removed with:

```
SE.SALT.Clean ''
```

The `Clean` function works recursively, so removing tags from objects at the root namespace level also removes tags from objects in all namespaces beneath the root level.

The removal of tags can be limited to only those on specific objects in the current namespace. For example, to remove the tags from objects `A` and `B`:

```
SE.SALT.Clean 'A B'
```

In addition to removing the tag information from objects in the namespace, the files that were associated with those objects can be deleted completely. For example, to remove the tag information from objects `A` and `B` and delete the files that were previously associated with them:

```
SE.SALT.Clean 'A B -deletefiles'
```

4.4 Compare Function

The `Compare` function has been deprecated in favour of third-party version control software.

Knowledge of the differences between two different versions of the same file or between two similar but distinct files can be a useful analytical tool. The `Compare` function can be called to perform either of these comparisons as long as the specified files are scripted.

SALT's integral comparison tool can be used to perform the analysis or a comparison tool of the user's choice can be specified instead. If SALT's integral comparison tool is used, then the output produced states the APL objects compared and emphasises the lines of text that differ between the two files. For example, output generated using SALT's integral comparison tool could resemble the following:

```
      ⎕SE.SALT.Compare'\tmp\MyProd'
Comparing \tmp\myprod.3.dyalog
     with \tmp\myprod.4.dyalog

 [0]    :Namespace MyProd
-[1]        rlb←{(+/∧\' '=⍵)↓⍵}
+           rlb←{(+/∧\' '=⍵)↓⍵}  ⍝ rem lead ' '
-[2]        rtb←{⍵↓⍨⌽∧\' '=⍵}
+           rtb←{⍵↓⍨⌽∧\' '=⍵}     ⍝ rem last ' '
 [3]    :EndNamespace
```

4.4.1 Syntax

```
⎕SE.SALT.Compare '{path/filename} [-version{=vers}] [-using
{=program}] [-permanent] [-window{=lines}] [-trim] [-symbols
{=symbols}]'
```

where:

- `path/filename` specifies the full path and filename of the versioned APL object whose versions are to be compared. If two different APL objects are to be compared, then the full path and filename of each APL object should be specified separated by a space character.
- `-version` must have a modifier value (*`vers`*) that specifies the versions of the file that are to be compared:
  - a modifier value of *`n`* compares the previous version (that is *`n-1`*) with version *`n`*
  - a modifier value of *`n1,n2`* compares version *`n1`* with version *`n2`*
  - a modifier value of *`ws`* compares the version currently in the active workspace with the latest saved version
  - a modifier value of *`ws n`* compares the version currently in the active workspace with version *`n`*

  If this modifier is not included then the two most recent (highest numbered) versions of the file are compared.
- `-using` must have a modifier value (*`program`*) that specifies the full path of the program to use to perform the comparison. If this modifier is not specified then SALT performs the comparison using the comparison tool named in the

compare session parameter (for details of this session parameter, see *Section 4.10.2.1*).

- `-permanent` changes the program named in the compare session parameter to be the program specified by the `-using` modifier.
- `-window` must have a modifier value (*`lines`*) that specifies the number of lines of code from the script to display in the results of the comparison before and after each line of the script that has been changed. If this modifier is not specified then the default value of 2 is used. Only relevant if SALT's integral comparison tool is being used.
- `-trim` removes leading and trailing spaces from each line of the script prior to performing the comparison. Only relevant if SALT's integral comparison tool is being used.
- `-symbols` must have a modifier value (*`symbols`*) that specifies the two symbols to use in the results of the comparison to indicate whether a line has been deleted or inserted (by default these are - and + respectively). Must be used with a modifier value comprising the deletion indicator followed by the addition indicator without a separating space, for example, *`-+`*. Only relevant if SALT's integral comparison tool is being used.

4.4.2 Use

When specifying the `-version` modifier, a modifier value of `n1,n2` compares version `n1` with version `n2`. If `n` is a negative number then it is subtracted from the highest version number. For example, if there are 5 versions of the specified file, then `-version=1,¯3` compares version `1` with version `2`.

The `-version` modifier can also be used when two different files are compared. In this situation, a modifier value that specifies one version number results in that version of each of the files being compared. For example:

```
      ]SE.SALT.Compare '\firstpath\firstfile.dyalog
\secondpath\secondfile.dyalog -version=3'
```

This compares **firstfile.3.dyalog** with **secondfile.3.dyalog**. However, if the modifier value specifies two version numbers, then the first version number is applied to the first specified APL object and the second version number is applied to the second specified APL object – these two files are then compared. For example:

```
      ]SE.SALT.Compare '\firstpath\firstfile.dyalog
\secondpath\secondfile.dyalog -version=3,7'
```

To perform a comparison using (for example) *Beyond Compare* (a comparison tool available from http://www.scootersoftware.com/download.php) rather than SALT's integral comparison tool, specify the location and executable name for your Beyond Compare installation; make this the permanent comparison tool by including the `-permanent` modifier in the call. For example:

```
SE.SALT.Compare '[ws]\classes\firstclass.dyalog
-using="c:\Program Files\BC\BC2.exe" -permanent'
```

If the first element of `-version` is `ws` then the contents of the specified object in the active workspace are compared with the latest saved version of the file containing that object. For example:

```
SE.SALT.Compare 'NS -version=ws'
```

This identifies the changes made to namespace `NS` since it was last saved. It is not necessary to specify a path to the object being compared as SALT uses the tag information on the object to locate the file (see *Section 3.6*).

4.5 List Function

The directories and **.dyalog** files under a specified directory can be listed using the `List` function. By default, a single path leading to a directory name returns the following information for the directories and **.dyalog** files in the specified location:

- type (`<DIR>` for directories, blank for **.dyalog** files)
- name
- version (the number of versions of the file) – files only
- size (in bytes) – files only
- date of last update

The same information is returned if the path leads to a **.dyalog** file, but relates to that file only.

This information can be filtered or amended using modifiers.

4.5.1 Syntax

```
SE.SALT.List '[directory|.dyalog file] [-folders] [-versions]
[-extension[=ext]] [-full[=value]] [-recursive] [-raw] [-type]'
```

where:

- `directory|.dyalog file` specifies either the full path to the directory whose contents are to be listed or the **.dyalog** file whose versions are to be listed. If no path is specified then the first directory named in the *workdir* session parameter is used (for details of this session parameter, see *Section 4.10.2.1*). If the path specifies a **.dyalog** file then the extension does not have to be included.
- `-folders` restricts the list to directories.
- `-versions` displays each item's version number in the list. If this modifier is not specified, then versioned files are indicated by having the total number of versions displayed in the version column.
- `-extension` can have a modifier value (*`ext`*) that restricts the files included in the list to files with the extension specified by the modifier value. If no modifier value is specified or the modifier value is `*` then all the files are listed with their extension displayed. Unless this modifier is specified, no extensions are displayed in the list. Only one extension can be specified. Wildcards cannot be used.
- `-full` can have a modifier value (*`value`*) that specifies the pathname origin for each item's `Name` information in the list:
  - a modifier value of *`1`* (or no modifier value) displays the full pathname from the specified directory.
  - a modifier value of *`2`* displays the full pathname from root.
- `-recursive` expands the list to include all directories and files within the specified directory recursively.
- `-raw` removes the titles and automatic formatting from all items in the list, thereby making it easier for APL functions to process the returned data.
- `-type` displays the type of each **.dyalog** file. SALT examines a file's script to identify its content from the start and end statements, determining whether it comprises a variable, function, operator, interface, namespace or class – if SALT cannot identify the type, then a value of `Fn` is reported. Although this information can be useful, the `-type` modifier adversely impacts performance.



For more information on scripted files, including declaration statements and permitted constructs, see the *Dyalog Programming Reference Guide*.

4.5.2 Use

Calling the `List` function without an argument returns a list of all the top-level directories and **.dyalog** files within the first directory named in the *workdir* session parameter (for details of this session parameter, see *Section 4.10.2.1*).

For example:

```
      ⎕SE.SALT.List ''
Type    Name    Versions   Size   Last Update
```

If a directory is specified as the argument then a list of all the top-level directories and **.dyalog** files within this directory is returned.

For example:

```
      ⎕SE.SALT.List '[SALT]'
Type    Name    Versions   Size   Last Update
<DIR>   core                      2013/04/22 16:02:34
<DIR>   lib                       2013/04/22 16:02:34
<DIR>   spice                     2013/04/22 16:02:34
<DIR>   study                     2013/04/22 16:02:34
<DIR>   tools                     2013/04/22 16:02:34
```

This is the content of the SALT directory itself. For more information on this content, modifiers must be specified. The `-recursive` modifier can be included in the call to provide details of the content of each directory and the `-type` modifier can be included to identify the type of APL object in each **.dyalog** file, for example:

```
      ⎕SE.SALT.List '[SALT] -recursive -type'
Type    Name                  Versions    Size   Last Update
<DIR>   core                                     2013/04/22 16:02:34
Cl      core\Parser                     11442    2013/01/30 17:15:20
Cl      core\SALT                       61386    2013/01/30 17:15:20
Ns      core\SALTUtils                  64605    2013/01/30 17:15:20
...
...
Cl      tools\special\asymmetric         8234    2013/01/30 17:15:18
Ns      tools\special\crTools            1163    2013/01/30 17:15:18
Cl      tools\special\symmetric          7446    2013/01/30 17:15:18
```

Other modifiers, such as `-folders` and `-raw`, can change the filters applied to the list and how it is presented. Two of the modifiers that can be specified with the `Load` function can take modifier values. The `-full` modifier specifies the pathname origin

for each item's `Name` information in the list – setting this to 2 (when no value is supplied it is assumed to be 1) means that the full pathname from root is displayed instead of the full pathname from the specified directory. For example:

```
      ]SE.SALT.List '[SALT] -full=2'
```

This changes the `Name` information in the list from `core`, `lib`, `spice`, `study` and `tools` (see first example output) to:

```
C:\Program Files\Dyalog\Dyalog APL 13.2 Unicode\SALT\core
C:\Program Files\Dyalog\Dyalog APL 13.2 Unicode\SALT\lib
C:\Program Files\Dyalog\Dyalog APL 13.2 Unicode\SALT\spice
C:\Program Files\Dyalog\Dyalog APL 13.2 Unicode\SALT\study
C:\Program Files\Dyalog\Dyalog APL 13.2 Unicode\SALT\tools
```

The `-extension` modifier can be specified without a modifier value to include all files in the list with their extensions displayed (effectively, a directory listing). Alternatively, a modifier value of a specific extension can be included to restrict the files included in the list to those that match the specified extension. For example:

```
      ]SE.SALT.List '\project\test -extension'
Type    Name              Versions    Size  Last Update
        first.dyalog                 19130  2013/01/30 17:16:31
        process.docx                 14632  2013/01/30 17:16:31
        review.docx                  75776  2013/01/30 17:16:31
        Dyalog.flprj                   359  2013/01/30 17:16:31
<DIR>   images                       11731  2013/01/30 17:16:31

      ]SE.SALT.List '\project\test -extension=docx'
Type    Name              Versions    Size  Last Update
        process                      14632  2013/01/30 17:16:31
        review                       75776  2013/01/30 17:16:31
<DIR>   images                       11731  2013/01/30 17:16:31
```

4.6 Load Function

The `Load` function can be called to load the latest (highest numbered) version of an APL object into the namespace that the `Load` function is called from, irrespective of whether the APL object is SALTed. By default, the `Load` function maintains the link between the loaded APL object and its source and assigns the loaded APL object a tag. Various modifiers can be specified to qualify this functionality.

Depending on the nameclass of the APL object loaded, the `Load` function returns a shy result of:

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

4.6.1 Syntax

```
⎕SE.SALT.Load '{path/name} [-target{=namespace}] [-noname]
[-disperse[=objects]|-nolink] [-protect] [-version{=vers}]
[-source[=no]]'
```

where:

- `path/name` specifies either the full path and name of the file to load or the full path and single pattern that identifies the APL objects to load (a single pattern can result in multiple APL objects being loaded).
- `-target` must have a modifier value (*`namespace`*) that specifies the full path and name of the appropriate namespace into which the APL object should be loaded. If this modifier is not specified then the APL object is loaded into the namespace that the `Load` function is called from. If the specified namespace does not exist (or is not a namespace), then the function call fails.
- `-noname` causes the `Load` function to return the value of the specified object instead of creating it in the workspace. For a class or namespace a reference is returned; for a function or operator, its `⎕OR` is returned with the function/operator inside. As no name is defined, no tag can be associated with the object.
- `-disperse` imports the APL objects within the specified file directly into the target namespace rather than importing the namespace contained by the specified file. When used without a modifier value, all objects in the specified namespace are imported into the target namespace along with the values of the system variables `⎕CT`, `⎕FR`, `⎕IO`, `⎕ML`, `PP` and `⎕WX`. If only a subset of the APL objects in the specified file are required, then the modifier value (*`objects`*) can be included to state which APL object or APL objects (separated with the `,` character) are required. If this modifier is specified then a shy message is returned by the `Load` function indicating the number of APL objects successfully loaded. Only relevant if the file loaded contains a namespace.
- `-nolink` removes the link between a loaded APL object and its source file. Using this modifier prevents SALT from managing the source for the APL object after loading it into the workspace – changes to the APL object will not be automatically saved until either the `Save` or `Snap` function has been called to save the APL object again.

- `-protect` prevents the specified APL object from being loaded if an APL object of that name is already defined in the namespace that the APL object is being loaded into. This modifier protects existing APL objects from being redefined.
- `-version` must have a modifier value (*`vers`*) that specifies the version to load. Only relevant if a version other than the latest version is required.
- `-source` returns the specified namespace as a nested vector instead of defining it in the workspace. If a modifier value of *`no`* is included, then a non-scripted version of the scripted namespace is loaded. Only relevant if the file loaded contains a namespace.

4.6.2 Use

The `Load` function takes either a filename or a filename pattern as its argument and retrieves the APL object defined in the specified path/file or all APL objects defined in files that match the specified filename pattern in the specified path. For example, the function call:

```
]SE.SALT.Load 'study\files\ComponentFile'
```

loads the APL object defined in the **ComponentFile** file (containing a class) from the **study\files** directory into the namespace, and the function call:

```
]SE.SALT.Load '\myutils\gui*'
```

loads all the APL objects that are defined in files with names starting with **GUI** in the **\myutils** directory into the current namespace. This works recursively – if **\myutils** contains other directories that include files with names starting with **GUI** then the APL objects in those files will also be loaded.

If the APL object should be loaded into a namespace other than the namespace that the `Load` function is called from, then the modifier `-target` must be used with a modifier value that defines the destination namespace. For example:

```
]SE.SALT.Load 'study\files\ComponentFile -target=MyFiles'
```

loads the APL object defined in the **ComponentFile** file from the **study\files** directory into the **MyFiles** namespace within the current namespace (a relative path was specified).

By default, the loaded APL object is assigned a tag pertaining to its original APL object. To instantiate a class in the **ComponentFile** file in the **study\files** directory using the argument **c:\temp\cfile** without naming the ComponentFile class in the namespace, either the `Load` function or the `New` function can be called. The following statement performs this action by calling the `Load` function:

```
      ⎕NEW (⎕SE.SALT.Load 'study\files\ComponentFile -NoName')
'c:\temp\cfile'
```

Alternatively, the following statement performs this action in one step rather than two by calling the `New` function (see *Section 4.7*):

```
      ⎕SE.SALT.New 'study\files\ComponentFile' 'c:\temp\cfile'
```

An APL object can be a namespace containing other APL objects, only a subset of which should be loaded. In this situation, the `-disperse` modifier can specify exactly which APL objects should be extracted from the specified file and loaded into the target namespace. For example, if a namespace in file **NS1** contains APL objects called **Obj1**, **Obj2**, **Obj3**, **Obj4**, **Obj5** and **Obj6**, then the following command would bring the APL objects with even numbers in their names into the current namespace:

```
      ⎕SE.SALT.Load 'study\files\NS1 -disperse=Obj2,Obj4,Obj6'
```

If the `-disperse` modifier is not used, then the `-nolink` modifier can be specified (these modifiers are mutually exclusive). This removes the link between a loaded APL object and its source file (the tag), thereby preventing SALT from managing the source for the APL object after loading it into the workspace. It has the effect that editing the APL object does not result in automatic saves; either the `Save` or `Snap` function has to be called to save the APL object again.

4.7 New Function

When instantiating an object from a class (object oriented programming), it can be beneficial to avoid naming the class in the namespace; this avoids potential name clashes. Although this can be achieved by calling the `Load` function within the `⎕NEW` system function (see *Section 4.6*), it is more computationally efficient to call the `New` function.

The `New` function returns an instance of the class, for example, `#.[classname]`.

4.7.1 Syntax

```
]SE.SALT.New '{path/filename}[.ext] [-version{=vers}]' ['arg|
(args)']
```

where:

- `path/filename` is the full path and filename of the class to instantiate.
- `-version` must have a modifier value (*`vers`*) that specifies the version number of the **.dyalog** file to instantiate the object from. If no version number is specified and the file containing the class to instantiate is a versioned file, then the latest (highest numbered) version is used.
- `arg` specifies any arguments needed to instantiate the class (in object oriented terminology this specifies the arguments that are passed to the constructor of the class). If more than one argument is required, then the list of arguments must be contained within parentheses.

4.7.2 Use

To instantiate a class in the **ComponentFile** file in the **study\files** directory using the argument **c:\temp\cfile** without naming the ComponentFile class in the namespace, either the `Load` function or the `New` function can be called. The following statement performs this action by calling the `Load` function (see *Section 4.6*):

```
      ]NEW (]SE.SALT.Load 'study\files\ComponentFile -NoName')
'c:\temp\cfile'
```

Alternatively, the following statement performs this action in one step rather than two by calling the `New` function:

```
      ]SE.SALT.New 'study\files\ComponentFile' 'c:\temp\cfile'
```

4.8 RemoveVersions Function

The `RemoveFunctions` function has been deprecated in favour of third-party version control software.

Editing an APL object that has been saved within a versioned file results in SALT saving a new version of the file (unless specifically instructed not to). This can result in numerous file versions being created. Once a stable version of the file has been achieved, these superfluous versions can be deleted using the `RemoveVersions` function.

The `RemoveVersions` function returns the number of versions that have been deleted.

4.8.1 Syntax

```
SE.SALT.RemoveVersions '{path/filename}[.ext] [-version
{=vers}|-all] [-collapse] [-noprompt]'
```

where:

- `path/filename` specifies the full path and filename (without the version number) of the versioned file that has superfluous versions.
- `extension` indicates the file's entension. If no extension is specified then an extension of **.dyalog** is assumed.
- `-version` must have a modifier value (*`vers`*) that specifies the version or range of versions to delete:
  - *`n`* only version n is deleted
  - *`>n`* all versions higher than *`n`* are deleted
  - *`<n`* all versions lower than *`n`* are deleted
  - *`n-m`* all versions in the range *`n`* to *`m`* (inclusive) are deleted
- `-all` removes all versions except the latest version.
- `-collapse` renumbers the latest version of the file with the lowest available version number following the specified deletion. Only relevant in either of the following situations:
  - all versions except the latest one are deleted, either by specifying the `-version` modifer with a modifier value of *`=>0`* or by specifying the `-all` modifier.
  - trailing versions except the last one are deleted by specifying the `-version` modifer with a modifier value of *`=>N`* – in this situation the remaining file is assigned the lowest available version number and versioning resumes from this number.

  If all the versions are removed (either by specifying the `-all` modifier or by specifying `-version=>0`) but the `-collapse` modifier is not specified, then this has the effect of switching off versioning for the file.
- `-noprompt` implicitly accepts all the changes that the call to the `RemoveVersions` function makes – omitting this modifier means that the user is prompted to confirm the deletion.

4.8.2 Use

Inclusion of the `-version` modifier with the `-range` modifier value deletes a specified version (or range of versions) of that file. In this situation, SALT deletes all versions of the file within the specified range. For example:

```
]SE.SALT.RemoveVersions 'path/MyClass -version=<5'
```

deletes all versions of the **MyClass.dyalog** file that have a version number less than 5. If there were only five versions of the **MyClass.dyalog** prior to the deletion, then the single remaining file retains its name of **MyClass.5.dyalog**. To rename this file so that it has a version number of 1, the `-collapse` modifier can be specified:

```
]SE.SALT.RemoveVersions 'path/MyClass -version=>0
-collapse'
```

The single remaining file is now called **MyClass.1.dyalog** – versioning is still switched on for this file, so the next time it is saved a new **MyClass.2.dyalog** version is created.

If the `-all` modifier had been specified instead of the `-version` modifier then specifying the `-collapse` modifier has the same effect as when specifying `-version` to remove all versions except the latest one, that is:

```
]SE.SALT.RemoveVersions 'path/MyClass -all -collapse'
```

results in a single remaining file called **MyClass.1.dyalog** – versioning is still switched on for this file, so the next time it is saved a new **MyClass.2.dyalog** version is created. However, if the `-collapse` modifier is not specified with the `-all` modifier (or with the `-version=>0` modifier) then the version number is removed from the single remaining file and versioning is switched off.

4.9 Save Function

When an APL object is ready to be saved, the `Save` function can be called to save it in a native text file.



The `Save` function cannot save APL objects of certain nameclasses – for a list of the types of nameclass that can be saved see *Section 3.5*.

The first time that an APL object is saved, the location must be specified. If the APL object has already been saved by calling the `Save/Snap` function, then subsequent saves of that APL object do not need to specify a location – by default, it is saved in the same location as it was previously (SALT achieves this using the APL object's tag information). If a different location is specified and the file is versioned, then a new

version number must be specified for versioning to continue. For non-scripted namespaces a location must be specified every time the `Save` function is called as SALT cannot retain tag information on non-scripted APL objects.



When saving a SALTed file, Dyalog Ltd recommends that:

- the chosen filename is restricted to alphanumeric characters as non-alphanumeric characters can cause issues on some operating systems.
- the filename is the same as the name of the SALTed item it contains (for ease of use)

The `Save` function returns the full path and name of the file that it saves.



When defining an APL object, it is good practice to define any system settings that could affect the object (for example, ⎕IO and ⎕ML) at the start of the script. If this is not done then the script picks up these values from the environment, which could result in unexpected behaviour.

4.9.1 Syntax

```
⎕SE.SALT.Save '{objectname} [[path/]filename][.extension]
[-version[=vers]] [-convert] [-banner{=top}][-noprompt]
[-makedir] [-format[=APL|XML]]'
```

for an unnamed namespace:

```
⎕SE.SALT.Save {objectref} '[[path/]filename][.extension]
[-version[=vers]] [-convert] [-banner{=top}][-noprompt]
[-makedir] [-format[=APL|XML]]'
```

where:

- `objectname` is the name of the APL object that is to be saved.
- `objectref` is the APL object reference of the APL object that is to be saved.
- `path/filename` is the full path and filename (without an extension) under which to save the script file. If the file has previously been saved through SALT, then this can be omitted; in this situation the file will be saved to the same location as before by default.
- `extension` indicates the file's entension. If no extension is specified then an extension of **.dyalog** is assumed.
- `-version` turns on versioning for the file (see *Section 3.10*). Optionally it can take a modifier value (*`vers`*) that identifies a specific version number to use

(this is included in the file's name) – if this modifier value is not included then a value one greater than the highest value currently saved is used.

- `-convert` retains the scripted format given to a previously unscripted namespace by SALT. Only relevant when saving a previously unscripted namespace.
- `-banner` adds a banner to the top of a namespace when it is saved, irrespective of whether `-convert` is specified. Must have a modifier value (*top*) that either specifies the text to use or executes (`⍎`) a variable containing the text to use. If the banner includes space characters then the entire modifier value must be contained within " marks. Only relevant when saving unscripted namespaces.
- `-noprompt` specifies that SALT is not to prompt the user for confirmation before saving the file each time its content is amended. Specifying this modifier means that the file (or a new version of the file is versioning is on) will be saved automatically every time the content is amended. This modifier can be specified with unversioned or versioned files.
- `-makedir` creates any necessary directories to satisfy the specified path.
- `-format` identifies the format in which to save the APL object. By default, APL objects are saved in XML format, but a modifier value (*APL*) can be specified to save the APL object in APL format.

4.9.2 Use

Inclusion of the `-version` modifier when saving a file turns on versioning for that file. In this situation, SALT saves the file as a new file with the specified name and adds a version number immediately before the **.dyalog** extension – if the modifier value *number* is included then the number specified becomes the version number, otherwise 1 is used. For example:

```
]SE.SALT.Save 'MyClass path\MyClassDir -version=3'
```

saves the APL object in the specified path as a script file called **MyClass.3.dyalog**. If a file of that name already exists and the `-noprompt` modifier has not been specified then SALT will ask for confirmation to overwrite the file; if `-noprompt` has been specified then the file will be overwritten automatically.

When saving an unscripted namespace, the `Save` function constructs a temporary script that is discarded after the namespace has been saved (unless the `-convert` modifier is specified). This script is used to save the namespace as a scripted namespace. Specifying the `-convert` modifier retains the constructed script; this means that SALT can identify (and save) subsequent changes made to the namespace through the editor.

The `-banner` modifier adds the specified text to the top of the converted namespace when saving it. For a single line banner, the text can be entered directly as a modifier value, for example, `-banner=text`. If the required banner text is multiple lines in length then it must be defined as a variable and the modifier value must be set to execute that variable. For example, a variable called TITLE can be defined in the workspace and assigned to be:

```
**********************
* Copyright ABC XYZ *
*    2000 - 2013    *
**********************
```

Setting the modifier `-banner=⍎TITLE` makes the defined text block appear at the top of the namespace in the file.

If the APL object being saved is a variable, then the format in which it is saved can be a valid consideration. Serialising variables using the APL format can result in executable expressions that exceed Dyalog's limit for executing an APL statement, especially if the variable comprises a nested array. As an alternative in this situation, the XML format can be used. Changing from the default XML format to APL format is achieved by specifying the `-format` modifier with the `APL` modifier value.

4.10 Settings Function

Some of SALT's functions take values from global parameters. These are retrieved from the external repository and loaded into SALT at the start of a Dyalog session. They remain active for the session unless they are modified by calling the `Settings` function.



The external repository stores configuration settings and options and is operating-system-dependent:

- On Microsoft Windows, it is the registry (global functions can also be modified in the **Configuration** dialog box – see *Appendix A*).
- On UNIX it is the **$HOME/.dyalog/SALT.settings** file.
- On Mac OS it is the **Users/<name>/.dyalog/SALT.settings** file (only created the first time a settings change is made).

4.10.1 Syntax

```
⎕SE.SALT.Settings '[parameter] [value] [-reset] [-permanent]'
```

where:

- `parameter` specifies the session parameter to retrieve/update (see *Section 4.10.2.1*).
- `value` specifies a value for the session parameter.
- `-reset` reloads the values from the external repository, replacing the session parameter values with the global parameter values.
- `-permanent` saves the values of the session parameters to the external repository, replacing the global parameter values.



`]SE.SALT.Set` can be used as an alias for `]SE.SALT.Settings`.

4.10.2 Use

Calling the `Settings` function without any arguments or modifiers returns a list of all the session parameters and their current values. For example:

```
]SE.SALT.Settings ''
```

Calling the `Settings` function with a single argument (one parameter only) returns the current session value for that parameter.

A session parameter can be modified by calling the `Settings` function with a single argument that comprises a parameter and a value. For example:

```
]SE.SALT.Settings 'editor \myprogs\vi.exe'
```

This modified session parameter is active throughout the Dyalog session but is not saved for subsequent Dyalog sessions unless the value is propagated to the global parameter in the external repository by specifying the `-permanent` modifier. For example:

```
]SE.SALT.Settings 'editor -permanent'
```

The session parameter can be replaced with the global parameter using the `-reset` modifier. For example:

```
]SE.SALT.Settings 'editor -reset'
```

Whenever a setting is changed, the `Settings` function returns the previous value of that setting.

4.10.2.1 Parameters

The possible session parameters are:

- *cmddir* – specifies the full path to the directory or list of directories from which to retrieve user commands. Multiple directories are specified using an operating-system-specific character:
  - On Linux the separator is `:` (that is, a colon)
  - On Mac OS the separator is `:` (that is, a colon)
  - On Microsoft Windows the separator is `;` (that is, a semi-colon)

  If multiple directories are specified, then SALT searches them in order and retrieves the first user command it finds with the specified name.

  Earlier versions of Dyalog allowed the use of the ∘ character as a separator – this has been superseded by the operating-system-specific characters and should no longer be used.

  To add a new directory to the list of directories, precede its path with a comma (`,`) character. For example:

  ```
  ]SE.SALT.Settings 'cmddir ,\ucmd1\c1'
  ```

  This adds the new directory to the start of the list of directories and it becomes the default location for fetching user commands.

  To remove a directory from the list of directories, precede its path with a tilde (~) character. For example:

  ```
  ]SE.SALT.Settings 'cmddir ~\ucmd1\c1'
  ```

- *compare* – states the full path to the comparison program to use.
- *debug* – specifies the level of debugging that SALT should use. Possible values are:
  - 0 : no debugging and report errors in the environment. This is the default value.
  - >0 : stop if an error is encountered
- *editor* – states the full path to the editing tool to use.
- *edprompt* – specifies whether a user is prompted for confirmation to overwrite the file when modifying a script or remove a file when deleting versions. Possible values are:
  - 0 or n : the user is never prompted for confirmation
  - 1 or y: the user is prompted for confirmation each time a script is modified or a version is deleted. This is the default value.
- *fndels* – specifies whether ∇ characters are used to enclose a new tradfn/tradop when saving it (a useful way of identifying tradfns/tradops).

Possible values are:

- 0 : do not use ∇ characters to enclose a new tradfn/tradop when saving it. This is the default value.
- 1 : use ∇ characters to enclose a new tradfn/tradop when saving it.



When saving a tradfn/tradop:

- setting *fndels* to 1 means the tradfn/tradop cannot be loaded in a version of Dyalog prior to version 17.0 using the `⎕SE.SALT.Load` function or the `]Load` user command, but can be loaded using `2⎕FIX`.
- setting *fndels* to 0 means the tradfn/tradop cannot be loaded in a version of Dyalog prior to version 17.0 using `2⎕FIX` but can be loaded using the `⎕SE.SALT.Load` function or the `]Load` user command.

- *mapprimitives* – specifies whether the glyphs that cannot be displayed in classic mode (⌸, ⌹, ⍤, ⍮, ⊆, ⍬) are automatically translated from Unicode into their `⎕Uxxxx` classic mode equivalent forms when loading/saving scripts. Possible values are:
  - 0 : do not translate the glyphs – the APL interpreter will fail if these Unicode glyphs are present in a script in classic mode or if their `⎕Uxxxx` form is used in a Unicode interpreter.
  - 1 : automatically translate the glyphs, making code fully portable between Unicode and classic versions of Dyalog. This is the default value.
- *newcmd* – specifies when new user commands become effective in the user interface. Possible values are:
  - auto : new user commands are detected automatically. This is the default value.
  - manual : new user commands do not become effective until the user command `]UReset` is run. For more information on user commands, see the *User Commands User Guide*.
- *track* – specifies the element tracking mechanism to use (basic information is always tracked). Possible values are:
  - compiled : preserves the `400⌶` state of SALTed objects (even though `400⌶` directives are not a visible part of a function's code). The information is reinstated when an object is loaded into the workspace by SALT.
  - atinfo : retrieves the function, user and timestamp information (as recorded by the monadic system function `⎕AT`) pertaining to the last time that the function was saved. The information is reinstated when a function is loaded into the workspace by SALT. Can only be used for traditional functions and operators.

    - new : tracks unSALTed objects; when these objects are edited using the default editor, they are saved in the first directory named in the *workdir* session parameter.

  Multiple values can be selected using the comma character as a separator – in this situation the list of values should be enclosed in single or double quotation marks.

  When returning a list of multiple objects, the space character is used as a separator.

- *varfmt* – specifies the format in which variables are saved. Possible values are:
  - APL
  - XML : this is the default value.
- *workdir* – specifies the full path to the directory or list of directories from which to retrieve files. Multiple directories are specified using an operating-system-specific character:
  - On Linux the separator is `:` (that is, a colon)
  - On Mac OS the separator is `:` (that is, a colon)
  - On Microsoft Windows the separator is `;` (that is, a semi-colon)

  If multiple directories are specified, then SALT searches them in order and retrieves the first user command it finds with the specified name.

  Earlier versions of Dyalog allowed the use of the ∘ character as a separator – this has been superseded by the operating-system-specific characters and should no longer be used.

  To add a new directory to the list of directories, precede its path with a , character. For example:

  ```
  ]SE.SALT.Settings 'workdir ,\proj\p1'
  ```

  This adds the new directory to the start of the list of directories and it becomes the default location for storing files.

  To remove a directory from the list of directories, precede its path with a ~ character. For example:

  ```
  ]SE.SALT.Settings 'workdir ~\proj\p1'
  ```

  SALT's files are always assumed to be in **[SALT]** (by default, this is **[DYALOG]/SALT**) even if that directory is not explicitly included in the list of working directories (that is, *workdir*).

4.11 Snap Function

Although the `Save` function enables individual APL objects to be saved, saving all the APL objects in a workspace using the `Save` function would be a repetitive process. Instead, the `Snap` function can be called to perform a bulk save of every APL object in the workspace in individual files – all new APL objects are saved to the specified directory and all modified APL objects are saved to the appropriate location.



The `Snap` function cannot save APL objects of certain nameclasses – for a list of the types of nameclass that can be saved see *Section 3.5*. It is the user's responsibility to ensure that unscripted namespaces do not include APL objects of these nameclasses; failure to do so can result in a loss of data.

To do this, the `Snap` function identifies all APL objects that need to be saved. It then determines which ones have been modified and which ones are new by reviewing the special tag associated with each APL object (see *Section 3.6* for tag information). If an APL object needs to be saved, or if SALT cannot determine if an APL object needs to be saved (for example, a non-scripted namespaces), then the `Snap` function calls the `Save` function to save that APL object (see *Section 4.9* for `Save` function information).



When saving a SALTed file, Dyalog Ltd recommends that the chosen filename is restricted to alphanumeric characters as non-alphanumeric characters can cause issues on some operating systems.

The `Snap` function returns a list of the names of the APL objects that have been successfully saved. If the `Snap` function stops for any reason, then everything in the same `Snap` call that has already been saved remains saved and a list of the names of the APL objects that have been successfully saved is returned.



When defining an APL object, it is good practice to define any system settings that could affect the object (for example, `⎕IO` and `⎕ML`) at the start of the script. If this is not done then the script picks up these values from the environment, which could result in unexpected behaviour.

4.11.1 Syntax

```
⎕SE.SALT.Snap '[fullpath] [-class{=nameclass}] [-convert]
[-clean] [-banner{=top}] [-fileprefix{=prefix}] [-loadfn[=
```

```
[+]path]] [-nosource] [-noprompt] [-makedir] [-show[=details]]
[-⍙⍙{=chars}] [-patterns{=string}] [-version[=vers]] [-format
[=APL|XML]]'
```

where:

- `fullpath` specifies the full path under which to save the new script files (modified versions of previously saved files are saved in their original location). If a full path is not included, then the first directory named in the *workdir* session parameter is used (for details of this session parameter, see *Section 4.10.2.1*).

  If this modifier is not included and the first directory named in the *workdir* session parameter is the **[SALT]** directory, then the `Snap` function will generate an error message and neither the new nor the modified files will be saved. This is to prevent the creation of extraneous files in the SALT directory.

- `-class` selects APL objects of the nameclass or nameclasses specified by the mandatory modifier value (*`nameclass`*). The modifier value can be 2 (variables), 3 (functions), 4 (operators) or 9 (namespaces) – finer granularity values are also accepted (see *Section 3.5* for information on valid nameclasses and subclasses). Multiple nameclasses can be included using a comma as a separator.

  Specific nameclasses/subclasses can be excluded by using the `~` prefix.

- `-convert` retains the scripted format given to a previously unscripted namespace by SALT. Only relevant when saving a previously unscripted namespace. Specifying this modifier means that the `-banner` modifier can, optionally, be included.
- `-clean` runs the `Clean` function on the objects to be saved.
- `-banner` adds a banner to the top of a namespace when it is converted from an unscripted namespace and saved as a scripted namespace. Must have a modifier value (*`top`*) that either specifies the text to use or executes (`⍎`) a variable containing the text to use. If the banner includes space characters then the entire modifier value must be contained within `"` marks. Only relevant if the `-convert` modifier is also included in the `Snap` function call.
- `-fileprefix` must have a modifier value (*`prefix`*) that specifies the string with which to prefix to APL object names when saving them to file (by default the filenames used are the same as each APL object's name followed by **.dyalog**).
- `-loadfn` generates a <load_ws> function that, when executed, redefines every APL object in the current workspace and runs the `⎕LX` for the workspace. By default, the function is called **load_ws.dyalog** and it is stored in the same location as the new script files. Optionally, a modifier value (*`path`*) can be

specified that identifies the full path to a different directory or **.dyalog** file in which to store the <load_ws> function. If the APL objects include class dependencies, this modifier automatically takes account of those dependencies (see *Section 3.8*

Specifying the `-loadfn` modifier means that the `-nosource` modifier can, optionally, be included.

- `-nosource` instructs SALT that the <load_ws> function being created should exclude scripts from namespaces when used to recreate a workspace. Only relevant if the -loadfn modifier is also included in the `Snap` function call.
- `-noprompt` specifies that SALT is not to prompt the user for confirmation before saving the file each time its content is amended. Specifying this modifier means that the file (or a new version of the file is versioning is on) will be saved automatically every time the content is amended. This modifier can be specified with unversioned or versioned files.
- `-makedir` creates any necessary directories to satisfy the specified path.
- `-show` does not save any APL objects but returns a list of the APL objects that would be saved by calling the `Snap` function with the specified modifiers. Optionally, can include the modifier value *`details`* to display the full path for each APL object that would be saved.
- `-ΔΔ` must have a modifier value (*`chars`*) that specifies the two characters to use in filenames instead of the Δ and Δ in the APL object's name. By default, **%** and **=** are used.
- `-patterns` only selects APL objects of the specified pattern. Must have a modifier value (*`string`*) that is an APL object name and can contain the wildcard *`*`*, for example, a modifier value of *`GUI*`* would select all APL objects with names starting with **GUI**. The modifier value can include multiple APL object names each separated by a space character (in which case the entire modifier value must be contained within " marks) – each APL object name can include multiple wildcards.

  Specific patterns can be excluded by using the ~ prefix.
- `-version` turns on versioning for the file (see *Section 3.10*). Optionally it can take a modifier value (*`vers`*) to identify a specific version number to include in the file's name – if this modifier value is not included then a value of 1 is used. If a modifier value is specified then this number is used as the version number for all the APL objects being saved.
- `-format` identifies the format in which to save the APL object. By default APL objects are saved in XML format, but a modifier value (*`APL`*) can be specified to save the APL object in APL format.

4.11.2 Use

Each new APL object is saved with the filename **<objectname>.dyalog**, where the name of the file is the same as the APL object's name:

- Any letter that has an accent in the APL object's name will not have the accent in the file's name.
- Any ∆ or ⍙ character in the APL object's name will be replaced by `%` and `=` respectively unless alternative characters have been specified using the `-∆⍙` modifier.

Multiple new APL objects could have the same filename, for example, if a namespace contains a new class called FOO and a new function called Foo, then the `Snap` function would try to assign each the filename **foo.dyalog**. To avoid this contention, the `Snap` function appends numbers preceded by a dash to the filenames:

- version numbering example: **myclass.3.dyalog**
- `Snap` function numbering example: **myclass-1.dyalog**
- both: **myclass-1.3.dyalog**

If the `-convert` modifier is specified, then the `Snap` function saves an unscripted namespace by converting it into a scripted namespace (replacing the unscripted version in the workspace with the scripted one) and then tracking changes made to it. If the `-convert` modifier is not specified, then the `Snap` function creates a directory in the specified location and gives it the same name as the unscripted namespace. The APL objects within the unscripted namespace are then saved in individual (scripted) files in this directory.



The treatment of unscripted namespaces is the only way in which the `Save` and `Snap` functions differ when saving APL objects.

With the `-convert` modifier specified:

- `Save` function: saves as scripted namespace and tracks changes
- `Snap` function: saves as scripted namespace and tracks changes

Without the `-convert` modifier specified:

- `Save` function: saves as scripted namespace but cannot track changes
- `Snap` function: saves as directory containing files for individual APL objects and cannot track changes

The `-banner` modifier adds the specified text to the top of the namespace when saving it. For a single line banner, the text can be entered directly as a modifier value, for example, `-banner=text`. If the required banner text is multiple lines in length

then it must be defined as a variable and the modifier value must be set to execute that variable. For example, a variable called TITLE can be defined in the workspace and assigned to be:

```
**********************
* Copyright ABC XYZ *
*    2000 - 2013    *
**********************
```

Setting the modifier `-banner=⍎TITLE` makes the defined text block appear at the top of the namespace in the file.

A prefix can be applied to the names of all the new files by specifying the required prefix as a modifier value of the `-fileprefix` modifier. If the prefix should only be applied to a subset of the new files, then those files should be saved first using an appropriate pattern/class. For example:

```
]SE.SALT.Snap '\ws\utils -patterns=GUI* -fileprefix=Win'
```

This saves all the new APL objects that have names starting with 'GUI' to files starting with 'Win', therefore the function GUImenu is saved in the **\ws\utils** directory as a file called **Winguimenu.dyalog**. If the requirement was that all APL objects except dfns should be prefixed with 'nonDFN', then the function call could have been:

```
]SE.SALT.Snap '\ws\utils -class=~3.2 -fileprefix=nonDFN'
```

Specifying the `-loadfn` modifier creates a new <load_ws> script file called (by default) **load_ws.dyalog**. When executed, this script redefines every APL object in the current workspace and runs the `]LX` for the workspace. A modifier value can be included to define a different location/name for the **load_ws.dyalog** file, although the file must have the extension **.dyapp** or **.dyalog**. For example:

```
]SE.SALT.Snap '\ws\utils -loadfn'
```

creates a file called **load_ws.dyalog** in the same directory as the other new files created by the `Snap` function call (that is, **\ws\utils**), whereas:

```
]SE.SALT.Snap '\ws\utils  -loadfn=\ws\ldscpts\ldit.dyalog'
```

creates a file called **ldit.dyalog** in the **\ws\ldscpts** directory.



The script created by the `-loadfn` modifier can be used with the `Boot` function to automatically start Dyalog with the workspace and all its constituent APL objects loaded. For more information on the `Boot` function, see *Section 4.2*.

Inclusion of the `-version` modifier turns on versioning for all files included in the `Snap` function. In this situation, SALT saves each file as a new file with a version number immediately before the **.dyalog** extension – if the modifier value *`number`* is included then the number specified becomes the version number, otherwise 1 is used. For example:

```
]SE.SALT.Snap '\ws\utils -version=3'
```

saves each APL object as a script file called **<objectname>.3.dyalog**. If a file of that name already exists and the `-noprompt` modifier has not been specified then SALT will ask for confirmation to overwrite the file; if `-noprompt` has been specified then the file will be overwritten automatically.

If the APL object being saved is a variable, then the format in which it is saved can be a valid consideration. Serialising variables using the APL format can result in executable expressions that exceed Dyalog's limit for executing an APL statement, especially if the variable comprises a nested array. As an alternative in this situation, the XML format can be used. Changing from the default XML format to APL format is achieved by specifying the `-format` modifier with the `APL` modifier value.

A Configuration Options

The global parameters that SALT takes as session parameters can be amended by defining new values through the `Settings` SALT function (see *Section 4.10*).

OS WIN In the Microsoft Windows operating system, some of these values can also be amended in the **SALT** tab of the **Configuration** dialog box (see *Section A.1*).

Table A-1 details the configuration options that are available.

Although the values can also be amended by editing the external repository strings directly, Dyalog Ltd does not recommend this method.

Table A-1: *Configuration options for global/session parameter values*

	`Settings` Function Parameter Name	Configuration Dialog Box Field
Enable/disable SALT	*n/a*	**Enable SALT** check box
User command location	*cmddir*	**User Command** tab
Comparison program	*compare*	**Compare command line**
Debugging level	*debug*	*n/a*
Editing tool	*editor*	**Editor command line**
Frequency of overwrite prompts	*edprompt*	*n/a*
Tradfn/Tradop ∇ delimitation	*fndels*	*n/a*
Mapping of primitives to ⎕xxxx for classic users	*mapprimitives*	*n/a*

Table A-1: *Configuration options for global/session parameter values (continued)*

	Settings Function Parameter Name	Configuration Dialog Box Field
New user command detection	*newcmd*	*n/a*
Element tracking mechanism	*track*	*n/a*
Variable storage format	*varfmt*	*n/a*
SALT source files' search paths	*workdir*	**Source folders**

A.1 Configuration Dialog Box

OS WIN The **Configuration** dialog box is only available when using the Microsoft Windows operating system.

To amend the options in the Configuration dialog box

1. In the Dyalog session window, select **Options > Configure...**.

   The **Configuration** dialog box is displayed.
2. In the **SALT** tab of the **Configuration** dialog box, amend the required settings.
3. Click **OK** to save your changes and return to the session window.

   The amendments take effect immediately.

The settings that can be amended in the **SALT** tab of the **Configuration** dialog box are:

- **Enable Salt** – select this check box to enable SALT or uncheck it to disable SALT.
- **Compare command line** – the full path to the comparison program to use.
- **Editor command line** – the full path to the editing tool to use.
- **Source folders** – the full path to the directory (or list of directories) from which to retrieve SALT files.

B SALT Function Syntax Summary

`Boot` function syntax:

for a **.dyalog** file:

```
]SE.SALT.Boot '{path/filename}{.dyalog} [-xload]' ['argument']
```

`Clean` function syntax:

```
]SE.SALT.Clean '[objects] [-deletefiles]'
```

`List` function syntax:

```
]SE.SALT.List '[directory|.dyalog file] [-folders] [-versions]
[-extension[=ext]] [-full[=value]] [-recursive] [-raw] [-type]'
```

`Load` function syntax:

```
]SE.SALT.Load '{path/name} [-target{=namespace}] [-noname]
[-disperse[=objects]|-nolink] [-protect] [-version{=vers}]
[-source[=no]]'
```

`New` function syntax:

```
]SE.SALT.New '{path/filename}[.ext] [-version{=vers}]' ['arg|
(args)']
```

`Save` function syntax:

```
]SE.SALT.Save '{item} [[path/]filename][.extension] [-version
[=vers]] [-convert] [-banner{=top}][-noprompt] [-makedir]
[-format[=APL|XML]]'
```

for objects:

```
]SE.SALT.Save {objectref} '[[path/]filename][.extension]
[-version[=vers]] [-convert] [-banner{=top}][-noprompt]
[-makedir] [-format[=APL|XML]]'
```

`Settings` function syntax:

```
]SE.SALT.Settings '[parameter] [value] [-reset] [-permanent]'
```

`Snap` function syntax:

```
]SE.SALT.Snap '[fullpath] [-class{=nameclass}] [-convert]
[-clean] [-banner{=top}] [-fileprefix{=prefix}] [-loadfn[=
[+]path]] [-nosource] [-noprompt] [-makedir] [-show[=details]]
[-ΔΔ{=chars}] [-patterns{=string}] [-version[=vers]] [-format
[=APL|XML]]'
```

C Example: SALT in Use

Although SALT is still fully supported, Dyalog Ltd expects that *Link* and third-party tools will replace SALT as the mechanism for using and managing text files as APL source code an recommends migrating from SALT to Link and third-party tools as soon as is convenient to do so. For more information, see the *Link User Guide*.

This example has been created as an illustration of SALT's source code management capabilities and the flexibility of its functions. To achieve this it does not necessarily follow an efficient workflow process or best coding practice.

Three employees of a company are working on the same project. All have access to the shared directory in which SALT saves APL objects.

John opens Dyalog and creates a function:

```
      ∇report
[1]   doWork
      ∇
```

John saves the `report` function as version 1 in a new directory called **project**:

```
      ]SE.SALT.Save 'report \project\report -makedir -version'
\project\report.1.dyalog
```

Dan opens Dyalog and creates a namespace called `utils` within the root namespace:

```
      )NS utils
#.utils
```

Dan retrieves the `report` function from the **project** directory and adds it into the new `utils` namespace:

```
      ⊢]SE.SALT.Load '\project\report -target=utils'
report
```

Dan creates and edits a class in the `utils` namespace:

```
      )ED ⍥utils.regex
```

Dan saves all changes in the `utils` workspace to the **project** directory:

```
      ⎕SE.SALT.Snap '\project'
#.utils.regex
```

Only the `regex` class is new, so that is the only APL object saved.

Dan checks the entire contents of the **project** directory:

```
      ⎕SE.SALT.List '\project -recursive'
```

```
Type    Name                Versions  Size  Last Update
        project\report      1           19  2013/06/07 15:12:19
<DIR>   project\utils                       2013/06/07 15:16:48
        project\utils\regex             31  2013/06/07 15:16:48
```

Brian opens Dyalog and does not want to be prompted when changes are made to files that have been saved using SALT:

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

This confirms that no prompts will now be given.

Brian sets his working directory to the **project** directory:

```
      ⎕SE.SALT.Settings 'workdir \project'
\project
```

Brian brings **report.dyalog** into his workspace, calling the `Load` function with an argument of `r*` as there are no other files in the directory with a name starting with the letter r:

```
      ⎕SE.SALT.Load 'r*'
report

      ⎕VR'report'
      ∇ report
  [1]    doWork
  [2]   ⍝∇¨§\project\report.dyalog§1§ 2013 6 7 15 12 19 822
§aaaaúö§0
      ∇
```

Brian edits the `report` function in his workspace several times, which produces a new file each time (as versioning is on).

Meanwhile, John edits the `report` function. Upon completion the following message is displayed:

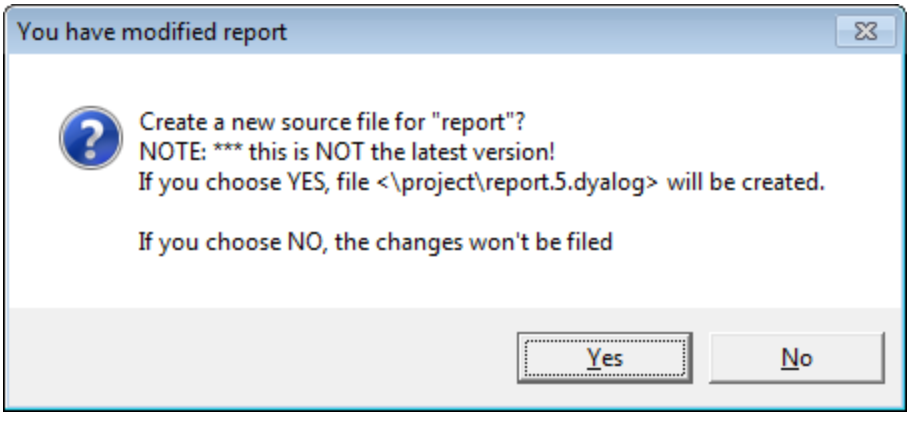


Figure C-1: *Changed function message*

John was not previously aware that the file had been worked on since he saved it. He clicks **No** and compares his version with the latest version:

```
      ]SE.SALT.Compare 'report -version=ws'
Comparing function <report> in the ws with the one in
\project\report.4.dyalog

[0]     report
+       do some more Work
-[1]    doWork for John
+       and again
```

As]AT tracking is not turned on, John does not know who made the modification. He talks to his teammates and finds out Brian made the modifications – they agree that John should merge his changes with Brian's changes using the editor. John does this, but before SALT saves the new version the following message is displayed:

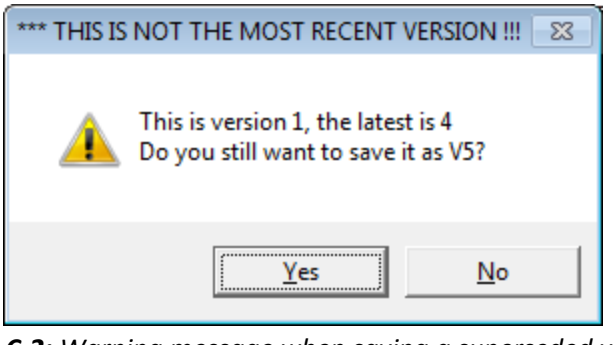


Figure C-2: *Warning message when saving a superseded version*

John clicks **Yes** and SALT saves **report.5.dyalog**.

John wants to clear up the unnecessary versions, so he checks what exists:

```
      ]SE.SALT.List '\project -recursive -versions'
Type     Name                  Version    Size Last Update
         project\report        [5]         21  2013/06/09 12:43:37
         project\report        [4]         21  2013/06/08 22:32:30
         project\report        [3]         21  2013/06/08 22:19:18
         project\report        [2]         21  2013/06/08 22:13:05
         project\report        [1]         19  2013/06/08 21:16:13
<DIR>    project\utils                         2013/06/08 21:17:57
         project\utils\regex                32  2013/06/08 21:17:56
```

John removes all but the latest version:

```
      ]SE.SALT.RemoveVersions \project\report -all -collapse
```

SALT prompts for confirmation of the removal:

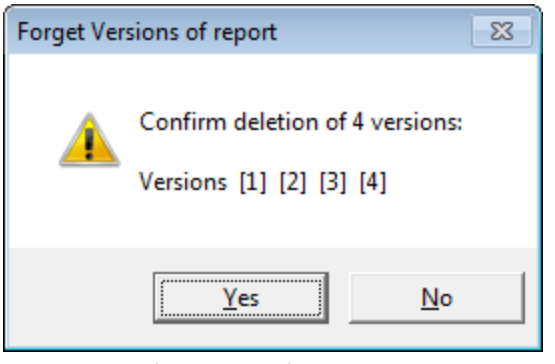


Figure C-3: *Confirmation of version removal request*

John clicks **Yes** and SALT deletes version 1, 2, 3 and 4.

```
4 versions deleted.
```

John instantiates the `regex` class anonymously and checks what has become available by doing this:

```
      reg←]SE.SALT.New '\project\utils\regex'
      reg.]NL ¯3
run
```

The `run` function is available (a *method* in object oriented programming).

John tests this function to check whether it works:

```
      reg.run
33
```

The returned value indicates that the `run` function is working correctly.

Dan clears his workspace and loads the contents of the project directory:

```
      )CLEAR
CLEAR WS

      ⎕SE.SALT.Load \project\*
report  #.utils.regex
```

SALT loads two files (the `report` function and the `regex` class) in the `utils` namespace.

Dan creates a function to load the contents of the project directory:

```
      ⎕SE.SALT.Snap '\project -loadfn=projX.dyapp'
** WARNING: ⎕LX is empty
```

This warning tells Dan that although the **projX.dyapp** file will recreate the workspace as it is now, nothing in the workspace will be executed as ⎕LX has not been set.

Dan tests whether the **projX.dyapp** file works on a clear workspace:

```
      )CLEAR
CLEAR WS

      ⎕SE.SALT.Boot '\project\projX.dyapp'
Loaded: report
Loaded: #.regex
```

As ⎕LX was empty, nothing is executed. However, the APL objects have been successfully imported:

```
      )FNS
report

      )CLASSES
regex
```

Index

.

.dyalog files 9
.dyapp files 9
  Autostarting 11

B

Boot function 19

C

Clean function 20
Compare function 21
Configuration 47

D

Directory structure 6
DYAPP Environment Variable 11

E

Environment Variables
  DYAPP 11
  SALT 6

F

File extensions
  .dyalog 9
  .dyapp 9
File format 8
Flags See *Modifiers*

I

Installation 5

L

List function 24
Load function 27

M

Modifiers 17

N

New function 30

R

RemoveVersions function 31

S

SALT Environment Variable 6
SALT functions 16
  Boot function 19
  Clean function 20
  Compare function 21
  List function 24
  Load function 27
  Modifiers 17
  New function 30
  Notation when calling 17
  RemoveVersions function 31
  Save function 33

Settings function 36
Snap function 41
Save function 33
Session parameters 37
Configuration 47
Set function See *Settings function*
Settings function 36
Snap function 41
Summary of syntax 49
Switches See *Modifiers*

T

Tags 10

V

Version management 15