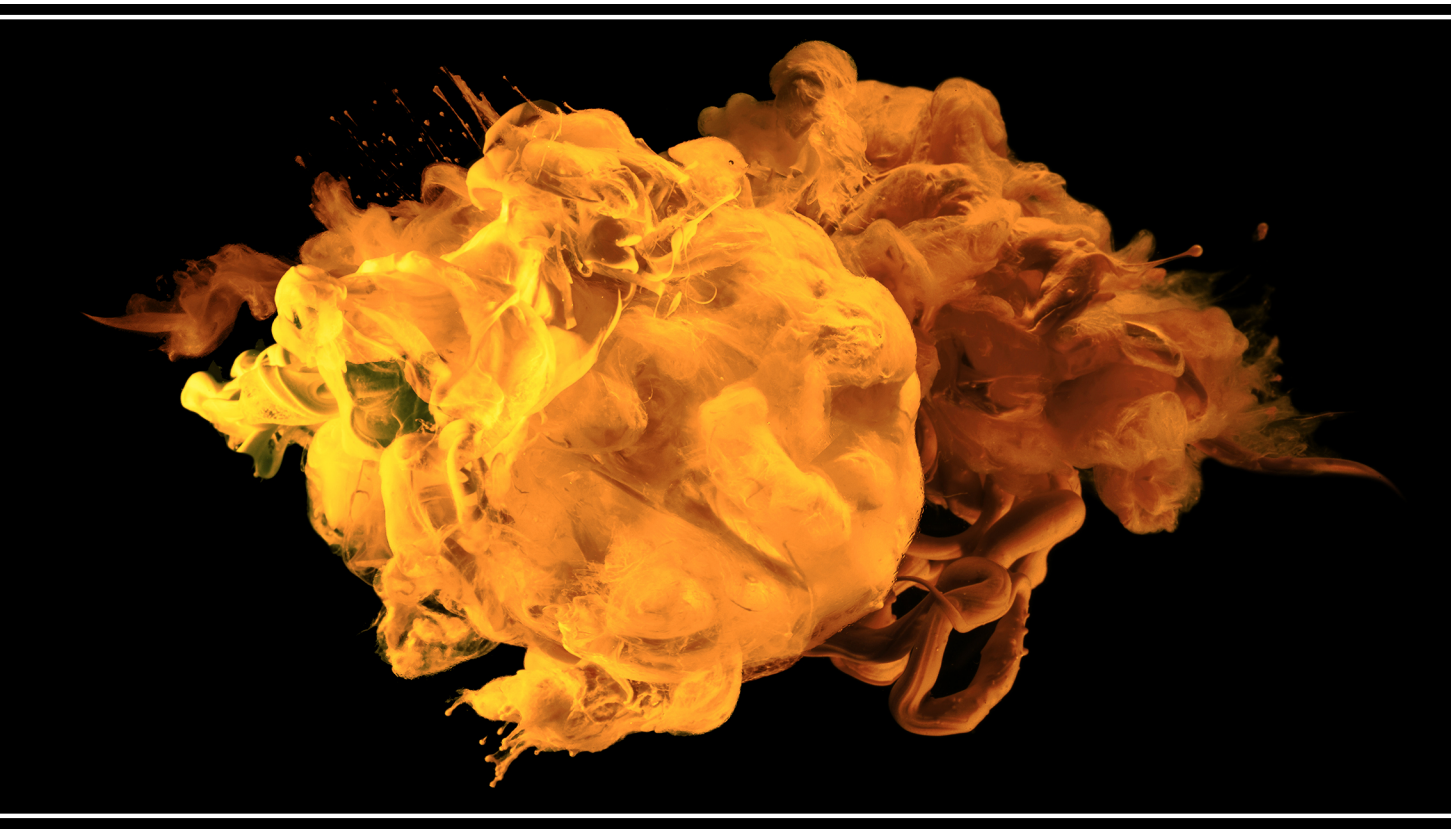
Link
User Guide
Link version 3.5



DYALOG

The Tool of Thought for Software Solutions

Link User Guide

Link version 3.5

Document Revision: 20211115_35

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

https://dyalog.com

Contents

1. Overview

1.1 Introduction

Link enables users of Dyalog to store their APL source code in text files. This is the documentation for Link Version 3.0, which will be released in the autumn of 2021 and included with the next release of Dyalog APL. If you have an earlier version of APL or Link, you might want to read one or more of the following pages before continuing:

- Link version 2.0 If you are actually looking for documentation of the version which was distributed with Dyalog APL versions 17.1 and 18.0.
- Migrating to Link 3.0 from Link 2.0: Dyalog recommends migrating to version 3.0 at your earliest convenience.
- Migrating to Link 3.0 from SALT: If you have APL source in text files managed by SALT that you want to migrate to Link.
- Installation instructions: If you want to download and install Link from the GitHub repository rather than use the version installed with APL, for example if you want to use Link 3.0 with Dyalog version 18.0.
- The historical perspective: Link is a step on a journey which begins more than a decade ago with the introduction of SALT for managing source code in text files, as an alternative to binary workspaces and files, and will hopefully end with the interpreter handling everything itself.

1.1.1 Audience

It is assumed the reader has a reasonable understanding of Dyalog and in particular workspaces, and namespaces.

1.1.2 What is Link?

Link allows you to use Unicode text files to store APL source code, rather than "traditional" binary workspaces. The benefits of using Link and text files include:

- It is easy to use source code management tools like Git or Subversion to manage your code.

> **Note**
>
> Although an SCM is not a requirement for Link, if you are not already using a source code management system, we ***highly*** recommend making the effort to learn about and install Git.
>
> If you choose not to use an SCM then you just need your own strategy for taking suitable copies of your source files, as you would with workspaces.

- Changes to your code are **immediately** written to file: there is no need to remember to save your work. The assumption is that you will make the record permanent with a *commit* to your source code management system, when the time is right.
- Unlike binary workspaces, text source can be shared between different versions of APL - or even with human readers or writers don't have APL installed at all.
- Source code stored in external files is preserved exactly as typed, rather than being reconstructed from the tokenised form.

1.1.3 Link is NOT...

- **A source code management system**: unlike it's predecessor SALT, Link has no source code management features. You will need to use a separate tool like Git to manage the text files that Link will maintain for you as you work with Dyalog APL.
- **A database management system:** although Link is able to store APL arrays using a pre-release of the *literal array notation*, this is only intended to be used for constants which you consider to be part of the source code of your applications. Although all functions and operators that you define will be written to source files by default, source files are only created for arrays by explicit calls to Link.Add or by specifying optional parameters to Link.Export. Application data should be stored in a database management system or files managed by the application.

1.1.4 Link fundamentals

Link establishes ***links*** between one or more **namespaces** in the active APL workspace and corresponding **directories** containing APL source code in Unicode test files. For example, the following user command invocation will link a namespace called `myapp` to the folder `/home/sally/myapp`:

```
]LINK.Create myapp /home/sally/myapp
```

A set of API functions is available in the session namespace `⎕SE`, for performing Link operations under programme control. Using the API, the above would be written:

```
⎕SE.Link.Create 'myapp' '/home/sally/myapp'
```

If `myapp` contains sub-directories, a namespace hierarchy corresponding to the directory structure will be created within the `myapp` namespace. By default, the link is bi-directional, which means that Link will:

- **Keep source files up-to-date:** Any changes made to code in the active workspace using the tracer and editor are immediately replicated in the corresponding text files.
- **Keep the workspace up-to-date:** Any changes made to the external files using a text editor, or resulting from an SCM action such as rolling back or switching to a different branch, will immediately be reflected in the active workspace.

You can invoke Link.Create several times to create multiple links, and you can also use Link.Import or Link.Export to import source code into the workspace or export code to external files *without* creating links that will respond to subsequent changes.

1.1.5 Functions vs. User Commands

With a few exceptions, each Link API function has a corresponding User Command, designed to make the functionality slightly easier to use interactively in the session.

User commands

The user commands have the general syntax

```
]LINK.CmdName arg1 [arg2] [-name[=value] ...]
```

where `arg2`'s presence depends on the specific command, `-name` is a flag enabling the specific option and `-name=value` sets that option to a specific value. Some options (like `codeExtensions` and `typeExtensions`) require an array of values: in these cases the user commands typically take the *name* of a variable containing that array.

For a list of installed user commands, type:

```
]LINK -?
```

API functions

The API is designed for use under program control, and options are provided in an optional namespace passed as the left argument. The general syntax of the utility functions is

```
options FnName arguments
```

where `options` is a namespace with variables, named according to the option they set, containing their corresponding values. The `-name=value` option can be set by `options.name←value`, and switches with values (e.g. `-name`) can be set by `options.name←1`. Unset options will assume their default value.

Options can also be provided as a character vector with the literal array representation of the option workspace, for example:

```
'(name: 1)' FnName arguments
```

The details of the arguments and options can be found in the API Reference.

1.1.6 Further reading

To get started using Link, please read:

- Basic Usage to see how to set up your first links, and learn about exporting existing application code to source files.
- Setting Up Your Environment for a discussion of how to set up Link-based development and runtime environments.
- Technical Details and Limitations if you want to know about the full range of APL objects that are supported, and some of the edge cases that are not yet supported by Link.

If you have an existing APL application that you want to move to Link, you might want to read one of the following texts first:

- Converting Your Workspace to Text Source: if you already have an existing body of APL code in binary workspaces.
- Migrating to Link 3.0 from SALT: if you are already managing text source using Link's predecessor SALT.

1.1.7 Frequently Asked Questions

- What happens if I save a workspace after creating Links?
- Are workspaces dead now?
- How is Link implemented?

1.2 Technical Details and Limitations

Link enables the use of text files as application source by mapping workspace content to directories and files. There are some types of objects that cannot be supported, and a few other limitations to be aware of.

1.2.1 Supported Objects

In the following, for want of a better word, the term *object* will be used to refer to any kind of named entity that can exist in an APL workspace - not limited to classes or instances.

Supported: Link supports objects of name class 2.1 (array), 3.1 (traditional function), 3.2 (d-function), 4.1 (traditional operator), 4.2 (d-operator), 9.1 (namespace), 9.4 (class) and 9.5 (interface).

Unscripted Namespaces: Namespaces created with `⎕NS` or `)NS` have no source code of their own and are mapped to directories. In Link version 3.0, one endpoint of a link is always an unscripted namespace and the other endpoint of the link is a directory.

Scripted Namespaces: So-called scripted namespaces, created using the editor or `⎕FIX`, have textual source and are treated the same way as functions and other "code objects". Link 3.0 does not support scripted namespaces, or any other objects that map to a single source file, as ***endpoints*** for a link, although they are supported as objects *within* a linked namespace.

It is likely that this restriction will be lifted in a future version of Link.

Variables are ignored by default, because most of them are not part of the source code of an application. However, they may be explicitly saved to file with `Link.Add`, or with the `-arrays` modifier of `Link.Create` and `Link.Export`.

Functions and Operators: Link is not able to represent names which refer to primitive or derived functions or operators, or trains. You will need to define such objects in the source of another function, or a scripted namespace.

Unsupported: Link has no support for name classes 2.2 (field), 2.3 (property), 2.6 (external/shared variable), 3.3 (primitive or derived function or train), 4.3 (primitive or derived operator), 3.6 (external function) 9.2 (instance), 9.6 (external class) and 9.7 (external interface).

1.2.2 Other Limitations

- Namespaces must be named. To be precise, it must be true that `ns≡(⎕NS⍬)⍎⍕ns`. Scripted namespaces must not be anonymous. When creating an unscripted namespace, we recommend using `⎕NS` dyadically to name the created namespace (for example `'myproject' ⎕NS ⍬` rather than `myproject←⎕NS ⍬`). This allows retrieving namespace reference from its display form (for example `#.myproject` rather than `#.[namespace]`).
- There must be exactly one file in the directory per named item to be created in the workspace. In particular, you must not have more than one file defining the same object: this will be reported as an error on `Link.Create` or `Link.Import`.
- Names which have source files should not have more than one definition within the same namespace. For example, if you have a global constant linked to a source file, you should not reuse that name for a local variable. If you were to edit the local variable while tracing, Link would be unable to distinguish it from the global name, and overwrite the source file.
- In a case-insensitive file system, Links created with default options cannot contain names which differ only in case, because the source files would have indistinguishable names. The `caseCode`

option can be used to get Link to generate file names which encode case information - see Link.Create for more information.

- Link does not support namespace-tagged functions and operators (e.g. `foo←namespace.{function}`).
- Changes made using `←`, `⎕NS`, `⎕FX`, `⎕FIX`, `⎕CY`, `)NS` and `)COPY` or the APL line `∇` editor are not currently detected. For Link to be aware of the change, a call must be made to Link.Fix. Similarly, deletions with `⎕EX` or `)ERASE` must be replaced by a call to Link.Expunge.
- Link does not support source files that define multiple names, even though `2⊃⎕FIX` does support this.
- The detection of external changes to files and directories is currently only supported if a supported flavour of .NET is available to the interpreter. Note that the built-in APL editor *will* detect changes to source files on all platforms, when it opens an editor window.
- Source code must not have embedded newlines within character constants. Although `⎕FX` does allow this, Link will error if this is attempted. This restriction comes because newline characters would be interpreted as a new line when saved as text file. When newline characters are needed in source code, they should be implemented by a call to `⎕UCS` e.g. `newline←⎕UCS 13 10 ⍝ carriage-return + line-feed`
- Although Link 3.0 will work with Dyalog version 18.0, Dyalog v18.1 or later is recommended if it is important that all source be preserved as typed. Earlier versions of APL may occasionally lose the source as typed under certain circumstances (and revert of source code generated from tokens, which may be formatted slightly differently).

1.2.3 How does Link work?

Some people need to know what is happening under the covers before they can relax and move on. If you are not one of those people, do not waste any further time on this section. If you do read it, understand that things may change under the covers without notice, and we will not allow a requirement to keep this document up-to-date to delay work on the code. It is reasonably accurate as of September 2021.

Terminology: In the following, the term *object* is used very loosely to refer to functions, operators, namespaces, classes and arrays.

What Exactly is a Link?

A link connects a namespace in the active workspace (which can be the root namespace `#`) to a directory in the file system.

When a link is created:

- An entry is created in the table which is stored in the workspace using an undocumented I-Beam, recording the endpoints and all options associated with the Link. Link.Status can be used to report this information. Earlier versions used `⎕SE.Link.Links`, but version 3.0 only stores information on links with an endpoint in `⎕SE` in that variable.
- Depending on which end of the link is specified as the source, APL source files are created from workspace definitions, or objects are loaded into the workspace from such files. These processes are described in more detail in the following sections.
- If .NET is available, a .NET File System Watcher is created to watch the directory for changes so that those changes can immediately be replicated in the workspace (unless an option it set to prevent this).

CREATING APL SOURCE FILES AND DIRECTORIES

Link writes textual representations of APL objects to UTF-8 text files (but can load source files created using any Unicode encoding). Most of the time, it uses the system function `⎕SRC` to extract the source form writing it to file using `⎕NPUT`. There are two exceptions to this:

- So-called "unscripted" namespaces, which contain other objects but do not themselves have a textual source, are represented as sub-directories in the file system (which may contain source files for the objects within the namespace).
- Arrays are converted to source form using the function `⎕SE.Dyalog.Array.Serialise`. It is expected that the APL language engine will, in the future, support the "literal array notation", and that `⎕SRC` will one day be extended to perform this function, but there is as yet no schedule for this.

LOADING APL OBJECTS FROM SOURCE

With the exception of variables stored in `.apla` files, which are processed by `⎕SE.Dyalog.Array.Deserialise`, Link loads code into the workspace using `2 ⎕FIX`.

When you are watching both sides of a link, Link delegates the work of tracking the links to the interpreter. In this case, editing objects will cause the editor itself (not Link) to update the source file. You can inspect the links which are maintained by the interpreter using a family of I-Beams numbered 517x. When a *new* function, operator, namespace or class is created, a hook in the editor calls Link code which generates a new file and sets up the link.

If .NET is available, Link uses a File System Watcher to monitor linked directories and immediately react to file creation, modification or deletion.

The Source of Link itself

Link consists of a set of API functions which are loaded into the namespace `⎕SE.Link`, when APL starts, from **$DYALOG/StartupSession/Link**. The user command file **$DYALOG/SALT/SPICE/Link.dyalog** provides access to the interactive user command covers that exist for most of the API functions. The code is included with installations of Dyalog version 18.1 or later.

If you want to use Link with version 18.0 or download and install Link from GitHub, see the installation instructions.

The Crawler

In a future version of Link, an optional and configurable crawler will be able to run in the background and occasionally compare linked namespaces and directories, using the same logic as Link.Resync, and deal with anything that might have been missed by the automatic mechanisms. This will be especially useful if:

- The File System Watcher is not available on your platform
- You add functions or operators to the active workspace without using the editor, for example using `)COPY` or dfn assignment.

The section on supported objects provides much more information about the type of APL objects that are supported by Link.

Breaking Links

If Link.Break is used to explicitly break an existing Link the namespace reverts to being a completely "normal" namespace in the workspace. If file system watcher was active, it is disabled. Any information that the interpreter was keeping about connections to files is removed using `5178⌶`. None of the definitions in the namespace are modified by the process of breaking a link.

If you delete a linked namespace using `)ERASE` or `⎕EX`, Link may not immediately detect that this has happened. However, if you call `Link.Status`, or make a change to a watched file that causes the file system watcher to attempt to update the namespace, Link will discover that something is amiss, issue a warning, and delete the link.

If you completely destroy the active workspace using `)LOAD` or `)CLEAR`, all links with an endpoint in the workspace will be deleted - but links to `⎕SE` will survive.

1.2.4 The Future

To summarise, the Link road map currently includes the following goals:

- Adding the crawler, which will automatically run Link.Resync in the background, in order to detect and help eliminate differences between the contents of linked namespaces and the corresponding directories. It may replace the File System Watcher in environments where it is not available.
- Replacing the use of SALT in Dyalog's tools, including a new implementation of user commands and other mechanisms for loading source code into the interpreter based on Link instead.
- Support for linking individual source files: Link 3.0 is only able to link a namespace to a directory. There are situations where it is practical to create a link to a single source file, particularly in the case of a scripted namespace.

Over time, it is a strategic goal for Dyalog to move more of the work done by Link into the APL interpreter, such as:

- Serialisation and deserialisation of arrays, using the literal array notation
- File system watching or other mechanisms for detecting changes to source at both ends of a link

1.3 Workspaces

1.3.1 Link versus Workspaces

As the *versus* in the heading is intended to imply, the main purpose of Link is to replace many uses of workspaces. Link is intended to make it possible for APL users to move away from the use of workspaces as a mechanism for storing APL source code.

Are Workspaces Dead Now?

No: Workspaces still have many uses, even if they are falling out of favour as mechanism for source code management:

- **Distribution:** For large applications, it will be inconvenient or undesirable to ship large collections of source files that are loaded at startup. The use of workspaces as a mechanism for the distribution of packaged collections of code and data is expected to continue.
- **Crash Analysis:** When an application fails, it is often useful to save the workspace, complete with execution stack, code and data, for subsequent analysis and sometimes resumption of execution. Dyalog will continue to support this, although we may gradually impose some restrictions, for example requiring the same version and variant of the interpreter in order to resume execution of a saved workspace.
- **Pausing work:** In many ways, this is similar to crash analysis: sometimes you need to shut down your machine in the middle of things and resume later, but you don't want to be forced to start from scratch because you have created an interesting scenario with data in the workspace. Saving a workspace allows you to do this.

With the exception of the scenarios mentioned above, Link is intended to make it unnecessary to save workspaces. All source code changes that you make while editing or tracing your code should immediately end up in text files and be managed using an SCM. The normal workflow is to start each APL session by loading the code into the workspace from source directories. You might want to save a "stub" workspace that contains a very small amount of code that loads everything else from text source, but from version 18.0 of Dyalog APL you can now easily set that up using text files as well, rendering workspaces obsolete as part of your normal development workflow.

Saving workspaces containing Links

If you `)SAVE` a workspace which has active links in it, this creates a potential conflict between the source code embedded in the workspace and any changes that may have been made to external source since the workspace was saved. If you `)LOAD` a saved workspace, Link will issue a warning along the lines of:

```
IMPORTANT: 1 namespace linked in this workspace: #.myapp
IMPORTANT: Link.Resync is required to use anything other than Link.Status and Link.Diff
```

In other words, most link user commands and API functions will be disabled, until you run Link.Resync, which will compare the contents of the workspace and the source directories, list the differences and propose actions to take in order to bring the contents of the workspace in line with the source folders.

> **Note**
>
> **Beware: If you continue working without doing a Resync, strange things may happen:** Link user commands and API functions will refuse to perform any actions, but names defined in the linked namespace contain references to external source files that the interpreter and editor will still honour. Using the built-in editor will read the external source file at the start of an editing session, and any changes made will be written to file, even though Link itself remains disabled.

In other words, you should **NOT** continue working without a Resync, unless you have a very good reason to do so and understand exactly what might happen.

Distribution

If you want to distribute a workspace created using Link to import code, note that if the workspace is loaded on a machine where the recorded source file names are not valid, this will lead to confusion. Application workspaces should always be built using Link.Import. Alternatively, use Link.Break to remove the links before you `)SAVE` the workspace.

> **Note**
>
> If you automate a build process using Link.Create rather than Link.Import, immediately followed by a `]SAVE`, there is a significant chance that a File System Watcher callback will be running in a separate thread, which will cause the `]SAVE` to fail.

See the discussion about setting up your environment for more tips on creating development and runtime environments.

1.4 History of source files as text in Dyalog

Link 3.0, released in 2021, is another step in the journey from binary workspaces to APL source in text files.

Workspaces

Historically, APL systems have used *saved workspaces* as the way to store the current state of the interpreter in a binary file which contains a collection of code and data. In many ways, a workspace is similar to a workbook saved by a spreadsheet application; a very convenient package that contains everything the application needs to run. Saving the workspace at the end of a run preserves updated data, as well as any code changes that might have been made.

Component Files and SQL Databases

Workspaces are very convenient, but the binary format makes them awkward if you want to compare, or otherwise manage, different versions of the source code - or the data, for that matter. Users preferred to store their data in component files or other storage mechanisms. As teams started writing larger systems, many development teams also created their own source code management systems (SCMs), typically storing multiple versions of code in component files or SQL tables.

These SCMs served large developer teams well for several decades. However, none of them became tools that were shared by the APL community, and they all suffered from the fundamental problem of using binary formats.

SALT - the Simple APL Library Toolkit

In 2006, Dyalog APL Version 11.0 introduced Classes and the ability to represent Namespaces as text "scripts". With that release, Dyalog APL included a tool known as SALT, which supported the use of Unicode text files as backing for the source of not only classes and namespaces, but functions, operators and variables as well. At the same time, a component named SPICE added user commands to Dyalog APL, using text source files which implemented a specific API, based on SALT's file handling.

SALT is Link's direct predecessor, and has many of the same features as Link:

- The ability to load entire directory structures into the workspace as namespaces
- A tool called "Snap", which would write all or selected parts of a workspace to corresponding source files.
- A hook in the APL system editor, which would update source files as soon as code was edited, without requiring a separate save operation.
- Startup processing of files with a `.dyapp` extension, to allow launching applications from text files without requiring a "boot workspace".

Link 2.0

After SALT had grown organically for more than a decade, it was decided that this functionality should be re-implemented in a new system: the Link project began. The first version of Link that was released to the general public was 2.0. The main differences between Link and SALT are:

- Link delegates the task of maintaining information about external source files to the APL interpreter, rather than using a trailing comment in functions and operators or "hidden" namespaces for classes and namespaces to track this information.
- New interpreter functionality based on `2 ⎕FIX` makes it possible for the interpreter to preserve source code exactly as typed, when an external source file is used.
- A file system watcher added support for using external editors and immediately replicating the effect of SCM system actions, such as a git pull or revert operation, inside the active workspace.
- Rather than using the extension `.dyalog` for all source, Link uses different extensions for different types of source, such as `.aplf` for functions, `.apln` for namespaces, and `.apla` for arrays.
- Use of a model of the proposed Literal Array Notation to represent arrays, rather than the notation used by SALT.
- We hope to add support for the array notation to the Dyalog APL interpreter in a future release.
- Link has no source code management features; the expectation is that users who require SCM will combine Link with an external SCM such as Git or SVN
- SALT included a simple mechanism for storing and comparing multiple versions of the source for an object by injecting digits into the file name.

Link 3.0

Link 3.0 is the first major revision of Link. It adds:

- Support for saving workspaces containing links and resuming work after a break.
- Support for names which differ only in case (for example, `FOO` vs `Foo`) in case-insensitive file systems, by adding "case coding" information to the file name.
- A new `Link.LaunchDir` API function, that makes it straightforward to replace the old SALT `.dyapp` files with new features in the interpreter, that make it possible to launch the interpreter using a configuration file or single APL source file.

A more complete description of the differences between Link 2.0 and 3.0 are described in the guide on upgrading from 2.0 to 3.0.

2. Install and Upgrade

2.1 Installation

Link 3.0 is included with Dyalog version 18.1 or later; no installation is required. The instructions on this page only apply if you want to:

- Use Link 3.0 with Dyalog version 18.0 (in place of Link 2.0).
- Participate in testing pre-releases of Link.
- Have some other reason for wanting to use a different version than that which is distributed with Dyalog APL.

Link is maintained as an open source project at github.com/dyalog/link. Start by downloading (or cloning) the latest release of Link and locating the subfolder called `StartupSession`. This folder contains the code required to run Link.

If you only rarely expect to update Link and you have the necessary permissions, you can **OVERWRITE the installed version of Link** by replacing the StartupSession folder that already exists in the main Dyalog program folder with the downloaded folder.

If you want to regularly update Link as new versions are made available, or you do not have permission to partially overwrite the Dyalog installation, you can keep the code outside the main program folder, so it can easily be updated with a `git pull`, or a copy operation that does not require special permissions. You will need to declare the location of the folder by setting the `DYALOGSTARTUPSE` parameter. You can add it to the command line when you start APL, but it is probably easier to use one of the following alternatives:

- **Set the `DYALOGSTARTUPSE` environment variable** to point to the StartupSession folder.
- **Update the configuration file (or the Windows registry)**, to set the parameter there. Typically, you would edit `~/.dyalog/dyalog.config` to make the change for all versions, or a specific file such as `~/.dyalog/dyalog.180U64.dcfg` for a specific version, to include a line:

```
DYALOGSTARTUPSE: "/Users/mkrom/link/StartupSession"
```

If you are using Dyalog version 18.0, you will also need to update the user command file used to invoke Link user commands. This only needs to be done once, because the new user command file is designed to pick user command definitions up from the current copy of Link.

The user command file is **SALT/spice/Link.dyalog**. If you have not done a complete checkout or clone of the repository, you will need to download this file from GitHub, as it is not included in the normal release package file.

- If you have permission, you can overwrite the installed version of the Link user commands by copying the file into **$DYALOG/SALT/spice/Link.dyalog**.
- Alternatively you can place a copy of the file in your **MyUCMDs** folder. This will cause it to take priority over the installed copy. Under Linux or Mac, you may need to create the folder yourself, under Windows the installation of Dyalog APL should have created it for you.

You will need to restart Dyalog APL each time you update any of the files mentioned above.

2.2 Upgrading to Link 3.0

If you are upgrading from Link 2.0 to 3.0, there are many new options and API functions (and corresponding user commands) that are available. The most significant changes are described here.

Breaking Changes

Some of the changes have the potential to break existing applications that use Link and require a review of existing code that calls the Link API:

- When specifying the name of a directory, Link 3.0 will not accept a trailing slash (this is reserved for possible future extensions)
- The `source=both` option has been removed from Link.Create.
- Link.List has been renamed Link.Status.
- When providing a new value for an array using Link.Fix, Link 3.0 expects the text source form of the array rather than the value of the array (bringing arrays into line with all other cases). To update using a new value, assign the value to the array and call Link.Add.
- If you have defined handlers for custom array representations, there have been significant changes to the arguments to the `beforeRead` and `beforeWrite` callback functions. Also, a new `getFilename` callback has been added. These callback functions are described in the documentation for Link.Create.
- **fastLoad:** When loading very large bodies of code (thousands or tens of thousands of functions), you may need to specify the new `fastLoad` option on Link.Create, in order to disable the checking of whether the names of items actually defined by source files correspond to the name of the file. Without this option, link creation may slow down so much that it could be considered a breaking change.

Other Significant Changes

The most important new features are:

- The addition of code to warn upon the loading of a workspace saved with active links, combined with the Link.Resync command which will assist in bringing a saved workspace in line with the external source, provide better support for resuming work after a break.
- "Case Coding" of file names, supporting the maintenence of source for names which differ only in case (for example, `FOO` vs `Foo`) in case-insensitive file systems.
- The addition of the Link.LaunchDir API function, which returns the name of the directory that the interpreter was started from, either using the `LOAD=` or `CONFIGFILE=` setting.

Change History

A detailed list of new features added to recent releases and a few behavioural changes can be found in the Link Change History.

2.3 Change History

This appendix details the changes made at each version of Link since the release of version 2.0

For a discussion of key differences between 2.0 and 3.0, see Upgrading to Link 3.0.

2.3.1 Version 3.0

For the version 3.0 project, most enhancements and bug fixes are recorded as GitHub Issues. The following lists only records the most important enhancements.

- When specifying a directory, a trailing slash is reserved for future extension
- API functions now throw errors rather than return an error message when they fail.
- Link.List has been renamed to Link.Status
- Link.Resync reports differences between the workspace and external source directories, and supports resumption of work from a saved workspace.
- Link.Create: `source=both` has been removed. It used to copy from namespace to directory, then the other way.
- Link.Create: `source=auto` has been added. It uses the non-empty side of the link as the source.
- Link.Break has a `recursive` flag to break all children namespaces if they are linked to their own directories and the `-all` modifier allows you to break all links in the active workspace but maintain links in the session namespace.
- Link.Import and Link.Export have an `overwrite` flag to allow overwriting a non-empty destination
- Link.Create and Link.Export have an `arrays` modifier to export arrays and a `sysVars` modifier to export namespace-scoped system variables
- Link.Create has a `fastLoad` flag to reduce the load time by not inspecting source to detect name clashes
- `beforeWrite` had been split into two callbacks : `beforeWrite` when actually about to write to file, and `getFilename` when querying the file name to use (see the Link.Create documentation for more details).
- `beforeWrite` and `beforeRead` arguments have been refactored into a more consistent set.
- Link.Fix now correctly expects text source for arrays (as produced by ⎕SE.Dyalog.Array.Serialise), as documented, whereas Link 2.0 expected the array itself. Similarly, the source (rather than the array itself) is correctly reported by the `beforeWrite` callback.
- If a variable has an existing source file, then the file will be updated if the variable is edited using the built-in editor.

Although Link 3.0 will work with Dyalog APL v18.0, many bug-fixes require version 18.2 or later - including but not limited to: - #155 Require keyword does not work - #149 Link induce status messages - #148: Fixing linked function removes all monitor/trace points in it - #144: Link can produce unloadable files

2.3.2 Version 2.1

Version 3.0 was labelled version 2.1 during most of its development, until the end of March 2021. It was renumbered just before the beginning of the distribution of official Beta releases of Dyalog Version 18.2. In other words: if you have version 2.1 installed, this is an early version of what became 3.0 and you should upgrade at your earliest convenience.

2.3.3 Version 2.0

- Link.Break has an `all` flag to break all links
- Link.Version reports the current version number
- Initial public release

3. Working with Link

3.1 Basic Usage

These sections cover the most commonly used commands. For more advanced usage, please consult the API documentation.

3.1.1 Starting from an existing folder containing text files

Use Link.Create to Link a directory containing text source to a namespace in the active workspace.

The following example loads APL code from the folder **/users/sally/myapp** into a namespace called `myapp`.

```
      ⎕SE.Link.Create myapp '/users/sally/myapp'
```

For every day use in the session, it might be more convenient to use the user command:

```
      ]LINK.Create myapp /users/sally/myapp
```

3.1.2 Importing code without creating a link

Sometimes you want to experiment and make modifications to your code without saving those changes. Use Link.Import to bring code from text source files into the active workspace without creating a link. The syntax of Import is almost identical to Create. The important difference being that changes to code in the workspace or in source files are not tracked or acted upon following an Import. For example:

```
      ]LINK.Import myapp /users/sally/myapp
```

3.1.3 Starting a new project

If you are starting a completely new project, create either a namespace in the active workspace or a folder on the file system (or both), and use Link.Create, naming the namespace and the folder, as in the example at the start of this page.

- If neither of them exist, Link.Create will reject the request on suspicion that there is a typo, in order to avoid silently creating an empty directory by mistake.
- If both of them exist AND contain code, and the code is not identical on both sides, Link.Create will fail and you will need to specify the `source` option, whether the namespace or the directory should be considered to be the source. Incorrectly specifying the source will potentially overwrite existing content on the other side, so use this with extreme caution!

To illustrate, we will create a namespace and populate it with two dfns and one tradfn, in order to have something to work with. In this example, the functions are created using APL expressions; under normal use the functions would probably be created using the editor, or perhaps loaded or copied from an existing workspace.

```
      'stats' ⎕NS ⍬ ⍝ Create an empty namespace
      stats.⎕FX 'mean←Mean vals;sum' 'sum←+⌿,vals' 'mean←sum÷1⌈⍴,vals'
      stats.Root←{⍺←2 ⋄ ⍵*÷⍺}
      stats.StdDev←{2 Root(+.×⍨÷⍴),⍵-Mean ⍵}
```

We could now create a source directory using Link.Export, and then use Link.Create to create a link to it. However, Link.Create can do this in one step: assuming that the directory `/users/sally/stats` is

empty or does not exist, the following command will detect that there is code in the namespace but not in the directory, and create a link based on the namespace that we just populated with our functions:

```
      ]LINK.Create stats /users/sally/stats
Linked: #.stats ↔ C:\tmp\stats
```

The double arrow `↔` in the output indicates that synchronisation is bi-directional. If .NET is not available, the default will be to only replicate changes in the namespace to file, which will be indicated by a `→`. We can check that the three expected files have been created:

```
      ls←⎕NINFO⍠1 ⍝ List files, allowing wildcards
      ls '/users/sally/stats/*'
 /users/sally/stats/Mean.aplf  /users/sally/stats/Root.aplf
 /users/sally/stats/StdDev.aplf
```

We can also verify that the new source directory can be used to re-build the original namespace::

```
      )CLEAR
clear ws
      ]LINK.Create stats /users/sally/stats
Linked: stats ↔ users/sally/stats
      stats.⎕NL -3 ⍝ Verify functions were loaded as expected
 Mean  Root  StdDev
```

3.1.4 Starting a project from a workspace

If your existing code is in a workspace rather than in text files, you should read the section on converting a workspace to source files before continuing.

3.1.5 Saving your work

Once a link is set up using Link.Create, you can work with your code using the Dyalog IDE exactly as you would if you were not using Link; the only difference being that Link will ensure that any changes you make to the code, using the Dyalog editor, within the `stats` namespace are instantly copied to the corresponding source file.

The use of a source code management system like Git is recommended. If you do that, then you effectively save your work by doing a commit.

> **Note**
>
> In the context of this document, the term *Dyalog IDE* includes both the Windows IDE and the Remote IDE (RIDE), which is tightly integrated with the interpreter.

Conversely, if you are new to Dyalog APL, and have a favourite editor, you can use it to edit the source files directly, and any change that you make will be replicated in the active workspace (assuming that a .NET File System Watcher is available).

If you use editors inside or outside the APL system to add new functions, operators, namespaces or classes, the corresponding change will be made on the other side of the link. For example, we could add a `Median` function to the namespace we created earlier:

```
      )ED stats.Median
```

In the Edit window, we complete the function:

```
Median←{
     asc←⍋vals←,⍵
     Mean vals[asc[⌈2÷⍨0 1+⍴vals]]
 }
```

When the editor fixes the definition of the function in the workspace, Link will create a new file:

```
      ls '/users/sally/stats/*'
 /users/sally/stats/Mean.aplf  /users/sally/stats/Root.aplf
 /users/sally/stats/StdDev.aplf /users/stats/StdDev.aplf
```

3.1.6 Viewing the status of links

The function (and corresponding user command) Link.Status will show namespaces that are currently linked and the folders to which they are linked. For example:

```
      ]link.status
Namespace  Directory           Files
#.stats    /users/sally/stats      4
```

3.1.7 Un-Linking a namespace

To continue using code in the active workspace without the risk of updating text source files or picking up changes made using external editors, use Link.Break.

Clearing the workspace, for example using `)CLEAR`, or exiting Dyalog, for example with `)OFF`, will also break all links in the active workspace.

See the technical details on breaking links for more information, for example about what happens when you delete a linked namespace from the active workspace.

3.1.8 Changes made outside the Editor

When changes are made using the editor which is built-in to Dyalog IDE (which includes RIDE), source files are updated immediately. Changes made outside the editor will not immediately be picked up. This includes:

- Definitions created or changed using assignment (`←`), `⎕FX` or `⎕FIX` - or the APL line "`∇`" editor.
- Definitions moved between workspaces or namespaces using `⎕CY`, `⎕NS` or `)COPY`.
- Definitions erased using `⎕EX` or `)ERASE`

If you write tools which modify source code under program control, it is a good idea to call the API functions Link.Fix or Link.Expunge to inform Link that you have made the change.

If you update the source files under program control and inbound synchronisation is not enabled, you can use Link.Notify to let Link know about an external change that you would like to bring into the workspace.

3.1.9 Arrays

By default, Link does not consider arrays to be part of the source code of an application and will not write arrays to source files unless you explicitly request it. Link is not intended to be used as a database management system; if you have arrays that are modified during the normal running of your application, we recommend that you store that data in an RDBMS or other files that are managed by the application code, rather than using Link for this.

However, if you have arrays that represent error tables, range definitions or other *constant* definitions that it makes sense to conside to be part of the source code, you can add them using Link.Add:

```
      stats.Directions←'North' 'South' 'East' 'West'
      ]Link.Add stats.Directions
Added: #.stats.Directions
```

Once you have created a source file for an array, Link *will* update that file if you use the editor to modify the array. Only if you modify the array using assignment or other means than the editor will you need to call Link.Add to force an update of the source file.

Changes made to source files, including the addition of new `.apla` files, will always be reflected in the workspace, if the link has been set up to watch the file system.

3.1.10 Setting up Development and Runtime Environments

We have seen how to use `]Link.Create` to load textual source into the workspace in order to work with it. As your project grows, you will probably want to split your code into modules, for example application code in one directory and shared utilities in another - and maybe also run some code to get things set up.

Next, we will look at Setting up Development and Runtime Environments, so that you don't have to type the same sequence of things over and over again to get started with development - or running the application.

3.2 Setting Up Your Environment

With a small project, you can get by using `Link.Create` and/or `Link.Import` to bring your source into the workspace in order to work with it. However, even in a small project, this quickly gets tedious, and as the project grows, you may want to load code from more that one directory, and perhaps run some code in order to set things up or even start the application. Fortunately, the Link API provides all the functions that you need to automate the setup.

3.2.1 Description of the functions and procedure for automating set up

3.2.2 Worked example

To illustrate, we will create a small application that uses the stats library that we created in the section on basic usage. We will put the application into a namespace called `linkdemo`:

```
      )clear
clear ws
      )ns linkdemo
      ]link.create linkdemo /users/sally/linkdemo
Linked: #.linkdemo ↔ /users/sally/linkdemo

      )ed linkdemo.Main
```

Our application is going to prompt the user for an input array and output the mean and standard deviation of the data, until the user inputs an empty array. Obviously, the code should be enhanced to validate the input and perhaps trap errors, but that is left as an exercise for the reader.

```
     ∇ Main;data
[1]   ⍝ Compute Mean and StdDev until user inputs an empty array
[2]
[3]    :Repeat
[4]        ⎕←'Enter some numbers:'
[5]        :If 0≠⍴data←⎕
[6]            ⎕←'Mean:   ',1↓⍕#.stats.Mean data
[7]            ⎕←'StdDev: ',1↓⍕#.stats.StdDev data
[8]        :EndIf
[9]    :Until 0=≢data
     ∇
```

We will need the `stats` code in the workspace as well, of course. Since we only intend to use it and don't want to risk making changes to its source code while testing our own application, we will use `]link.import` rather than `]link.create` to bring that code into the workspace. Note that after the import, `]link.status` still only reports a single link:

```
      ]link.import stats /users/sally/stats
Imported: #.stats ← /users/tmp/stats

      ]link.status
 Namespace    Directory            Files
 #.linkdemo   /users/sally/linkdemo     1

      linkdemo.Main
      Enter some numbers:
⎕:
      50+?100⍴100
Mean:    102.4
StdDev:  30.1
Enter some numbers:
⎕:
      ⍬
```

Automating Startup

Starting with Dyalog APL version 18.0, it is simple to launch the interpreter from a text file: either a source file defining a function, namespace or class using the LOAD parameter or from a configuration file using the CONFIGFILE parameter. Configuration files allow you to both set a startup expression and include other configuration options for the interpreter. For example, if we were to define a file `dev.dcfg` in the `linkdemo` folder with the following contents:

```
{
  Settings: {
      MAXWS: 100M,
```

```
      LX: "linkdemo.Start 0 ⋄ ⎕←⎕SE.Link.Create 'linkdemo' ⎕SE.Link.LaunchDir"
  }
}
```

This specifies an APL session with a MAXWS of 100 megabytes, which will start by creating the `linkdemo` namespace and calling `linkdemo.Start`. The namespace will be created using the directory named by the result of the function `⎕SE.Link.LaunchDir`; this will be the directory that the CONFIGFILE parameter refers to (or, if there is no CONFIGFILE, the directory referred to by the LOAD parameter).

The function `linkdemo.Start` will bring in the `stats` library using `Link.Import`: since we are not developers of this library, we don't want to create a bi-directional link that might allow us to accidentally modify it during our testing. It also creates the name `ST` to point to the stats library, which means that our `Run` function can use more pleasant names, like `ST.Mean` in place of `#.stats.Mean` - which also makes it easier to relocate that module in the workspace:

```
     ∇ Start run
[1]   ⍝ Establish development environment for the linkdemo application
[2]
[3]    ⎕IO←⎕ML←1
[4]    ⎕SE.Link.Import '#.stats' '/users/sally/stats' ⍝ Load the stats library
[5]
[6]    :If run
[7]        Main
[8]        ⎕OFF
[9]    :EndIf
     ∇
```

We can now launch our development environment using `dyalog CONFIGFILE=linkdemo/devt.cfg`, or on some platforms right-clicking on this file and selecting Run.

Development vs Runtime

The `Start` function takes a right argument `run` which decides whether it should just exit after initialising the environment, or it should launch the application by calling `Run` and terminate the session when the user decides that the job is done.

This allows us to create a second configuration file, `linkdemo/run.dcfg`, which differs from `dev.dcfg` in that we reserve a bigger workspace (since we'll be doing real work rather than just testing), and brings the source code in using `Link.Import` rather than `Link.Create`, which means that we won't waste resources setting up a file system watcher, and that accidental changes made by anyone running the application will not update the source files.

```
{
  Settings: {
      MAXWS: 1G,
      LX: "linkdemo.Start 1 ⋄ ⎕←⎕SE.Link.Import 'linkdemo' ⎕SE.Link.LaunchDir"
  }
}
```

Distribution Workspace

As we have seen, Link allows you to run your application based entirely on textual source files. However, if you have a lot of source files it may be more convenient for the users of your application to receive a single workspace file with all of the source loaded.

To prepare a workspace for shipment, we will need to:

- Set `⎕LX` in the so that it calls the `Start` function
- Use `Link.Break` to remove links to the source files. If you omit this step, you can create an extremely confusing situation.
- `)SAVE` the workspace

Scripted Applications

Recent versions of Dyalog APL support running APL from a script either by redirecting input to a normal APL interpreter or (recommended from version 18.2) using the new script engine. When the interpreter is running from a script, it intentionally provides you with a completely clean environment without any development tools loaded. This means that the session namespace is not populated, and Link is not loaded. If you add the following expression to the beginning of your script, it will (amongst other things) bring Link into the session so that the API becomes available:

```
(⎕NS⍬).({}enableSALT→⎕CY'salt')
```

Note that this depends on the interpreter being able to find the salt workspace (`salt.dws`). You may need to provide a full path name to that file, if you don't have a standard installation.

3.3 Converting an Existing Workspace to use Link

In order to start using Link to maintain code that resides in a workspace, you first need to export the code in the workspace to one or more folders.

The simplest way to do this is to use Link.Export. In principle, it should be possible to write the entire contents of any workspace to an empty folder called `/folder/name` using the following:

```
'options' ⎕NS ⍬
options.(arrays sysVars)←1
options ⎕SE.Link.Export # '/folder/name'
```

or equivalently, using the user command:

```
]link.export # /folder/name -arrays -sysvars
```

You can also use Link.Create with the same arguments, if you want an active link to exist after the export has been done.

3.3.1 Options

-arrays

By default, Link assumes that the "source code" only consists of functions, operators, namespaces and classes. Variables are assumed to contain data which is transient and thus not part of the source. The `-arrays` causes all arrays in the workspace to be written to source files as well. You can also write selected variables to file, see the documentation for Link.Create for more options.

-sysVars

By default, Link will assume that you do **not** wish to record the settings for system variables, because your source will be loaded into an environment that already has the desired settings. If you want to be 100% sure to re-create your workspace exactly as it is, you can use `-sysVars` to record the values of system variables from each namespace in source files.

Beware that this will add a *lot* of mostly redundant files to your repository. It is probably a better idea to analyse your workspace carefully and only write system variables to file if you really need them, using Link.Add.

3.3.2 Workspaces containing Namespaces

If your workspace is logically divided up into namespaces and you are happy for them all to end up in the same directory, you can use a single call to Link.Export or Link.Create like the one at the beginning of this section to write everything out at once. If you don't want the workspace to end up as a single directory tree, you can either restructure things afterwards using file explorers or command line tools, or you can make several separate calls to Export or Create to write the contents of individual namespaces to different locations.

Of course, if you create more than one source directory, you will need make more than one call to Link.Create or Link.Import in order to re-create the workspace in order to run your code.

3.3.3 Flat Workspaces and the `-flatten` Switch

If your workspace is not divided into namespaces, but all your code and data are in the root (or #) namespace, it probably still consists of more than logically distinct sets of code ("modules"), that you might wish to manage separately. If you Export such a workspace, all the source files will obviously end up in the same folder.

If you subsequently separate the source files into separate folders in order to make the source more manageable, you can still load it all into a single "flat" namespace using the `-flatten` switch. This allows the code to run unchanged, although you have created a structure for the source.

The mappings to source files will be recorded, so that synchronisation will work if you edit the code in the APL system or using an external editor. If you create a new name inside the workspace, Link will obviously not know which folder to write it to, and will prompt you to specify a target folder.

3.3.4 Recreating the Workspace

In order to recreate the workspace from source, you will need to make one or more calls to `Link.Create` or `Link.Import`, depending on the structure that you have created. For some ideas on how to set this up, see Setting up your Environment.

3.4 Migrating from SALT to Link

If you have been using SALT (Link's direct predecessor) to maintain the source of your application, the good news is that nearly all of your source files already have a format which can be used directly with Link (with the exception of source files containing arrays).

There are a few issues that may require some work to sort out.

3.4.1 No Version Control Features

SALT includes basic version control features, such as storing multiple versions of the source for a single function by creating file names containing sequence numbers, and providing tools to compare different versions. Link assumes that you will use an external SCM like Git or SVN, and contains no version control features of its own.

3.4.2 Different API

You need to replace all calls to SALT functions like `]SE.SALT.Load` with calls to `]SE.Link.Create` or `]SE.Link.Import`.

3.4.3 File Name Extensions

SALT uses the extension of `.dyalog` for all source files including arrays. Link will load `.dyalog` files and, if you edit existing objects the source file will be updated. However, if you create any new items, all new files will have extensions that vary by type, for example `.aplf` for functions, `.apln` for namespaces, or `.apla` for arrays.

You can rename all your source files to the Link defaults in one operation by creating a link using the `-forcefilenames` switch. Remember to take a backup before you cause such a sweeping change to your source files!

3.4.4 `.dyapp` Files

From Dyalog version 18.0, you can launch the APL interpreter using any APL source file, or a configuration file. As a result, `.dyapp` files are now deprecated. The section on setting up your environment contains examples of using the new mechanisms to launch your application.

3.4.5 Arrays

SALT uses a couple of different formats to represent arrays, either XML or executable APL expressions. Link uses the future literal array notation. You will need to load your arrays into the workspace using SALT and then use Link.Add to write them back out again.

3.4.6 Loading Individual Files

The SALT `Load` function supports loading individual source files and maintaining a link to the source file, so that editing the function will cause the source file to be updated. While Link.Import can load individual files, the files loaded in this way will not be synchronised. Link.Create only supports linking an entire directory to a namespace.

This may well change in a future release. Until then, you can use `2 ]FIX 'file://filename'` to achieve more or less the same effect as SALT's Load.

3.4.7 The `ꞏ:require` Comment

SALT was implemented before the interpreter added support for the `:Require` keyword, which can be used to manage the order in which dependencies are loaded. SALT used special comments to implement its own dependency management. Link does not support these comments; you must switch to using the keyword.

4. API & Command Reference

4.1 API Overview

4.1.1 API function syntax

The Link API functions all reside in `⎕SE.Link`.

The general syntax for Link API functions is as follows:

```
message ← options ⎕SE.Link.FnName args
```

where:

- `message` is a simple character vector or nested array containing messages related to the effects of the function call.
- `options` is a namespace containing optional parameters. Only certain functions accept an options namespace.
- `FnName` is the name of the API function
- `args` is either a character vector or a nested vector as described in the help section for that API function.

4.1.2 Option Namespaces

Some API functions accept an option namespace as the left argument. For example, to create a link with non-default `source` and `flatten` options, you would write:

```
options←⎕NS ⍬                                  ⍝ create empty namespace
options.(source flatten)←'dir' 1               ⍝ set two named options
options ⎕SE.Link.Create 'myapp' '/sources/myapp'  ⍝ namespace and director name on the right, options on left
```

Creating option namespaces will become more elegant once Dyalog APL is enhanced with a notation for namespaces. Until that time (no definite schedule has yet been set), Link API functions will accept a character vector left argument which represents an array in the proposed Literal Array Notation, for example:

```
'(source:''dir'' ⋄ flatten:1)' ⎕SE.Link.Create 'myapp' '/sources/myapp'
```

4.1.3 User commands

Most API functions have a corresponding user command, to make them a little easier to use interactively. The API functions with user command covers are indicated with `]` in the function reference tables. These user commands all take exactly the same arguments and options as the API functions, specified using user command syntax. The Link.Create call above would thus be written:

```
]LINK.Create myapp /sources/myapp -source=dir -flatten
```

Lowercase option names: Although option names are case sensitive and some of them contain uppercase letters when provided to API functions via option namespaces, the user command option names are entirely lowercase, to make interactive use more convenient.

Specifying extensions: Two options require arrays identifying file extensions: `codeExtensions`, `customExtensions` and `typeExtensions`. For convenience, the `]LINK.Create` user command accepts the *name* of a variable containing the array, rather than the array values.

4.1.4 Basic API Function reference

The following functions cover the vast majority of normal use-cases:

Function	User Command	Right Argument(s)	Left Argument(s)	Result
Add	`]Add`	items		message
Break	`]Break`	namespaces	options: `all` `recursive`	message
Create	`]Create`	namespace directory	options: `source` `watch` `arrays` (and many more)	message
Export	`]Export`	namespace directory	options: `overwrite` `caseCode` `arrays` `sysVars`	message
Expunge	`]Expunge`	items		boolean array
Import	`]Import`	namespace directory	options: `overwrite` `flatten` `fastLoad`	message
LaunchDir		none	none	directory name
Refresh	`]Refresh`	namespace	options: `source`	message
Resync	`]Resync`	namespace	options: `proceed` `arrays` `sysvars`	message
Status	`]Status`	namespace	options: `extended`	message
Version				version number as string

> **Note**
>
> The currently active version of Link is reported by `⎕SE.Link.Version` and found in the output of `]TOOLS.Version`

4.1.5 Advanced API Function reference

The "advanced" functions are typically used when building your own tools on top of Link, rather than simply using Link to maintain the source of an application:

Function	User Command	Right Argument(s)	Left Argument(s)	Result
CaseCode		filename	*<none>*	case-coded filename
Fix		source	array: namespace name oldname	boolean
GetFileName	`]GetFileName`	items	*<none>*	filenames
GetItemName	`]GetItemName`	filenames	*<none>*	items
Notify		event filename oldfilename	*<none>*	*<none>*
StripCaseCode		filename	*<none>*	filename without case code
TypeExtension		name class	option namespace used for Create	file extension (without leading `'.'`)

4.2 Link.Add

4.2.1 Syntax

```
]LINK.Add <items>
message ← ⎕SE.Link.Add items
```

This function allows you to add one or more existing APL items to the link, creating the appropriate representation in the linked directory. Note that a source file will only be created or updated if the item is in a linked namespace.

This is useful to write a new or modified array to a source file: arrays are normally not written to file by Link.

It is also useful when a change has been made to a linked item using any mechanism other than the APL editor, for example the definition of a new dfn using assignment, or the use of `)COPY` to bring new objects into the workspace.

> **Note**
>
> You can create or update an item from source while adding it to the Link by calling Link.Fix.

4.2.2 Arguments

- `items` is a simple character vector or nested vector of character vectors containing the names of items to be added to the link.
  In the user command, `<items>` is a space-separated list of names.

4.2.3 Result

- `message` is a simple character vector describing items that were:
  - Added (they belong in a linked namespace and were successfully added)
  - Not linked (they do not belong to a linked namespace)
  - Not found (the name doesn't exist at all)

4.3 Link.Break

```
]LINK.Break [<ns>] [-all={#|⎕SE|*}] [-recursive={on|off|error}]
message ← {options} ⎕SE.Link.Break namespace
```

Breaks an existing link: Does not affect the contents of the active workspace except to remove all traces of the link, preventing any further synchronisation from taking place.

> **Note**
>
> If you have enabled Pause Threads on Error and you have an application thread running which encounters an error, Link.Break can hang when the thread which is created to shut down the file system watcher becomes paused. If this happens, use IDE menu items to resume execution.

4.3.1 Arguments

`namespace` is a list of namespace names as character vectors or references

In the user command, `<ns>` is a space-separated list of namespace names

4.3.2 Options

all

{`#`|`⎕SE`|`*`}

By default (`-all` or `-all=#`), break links to all namespaces within the main workspace.

To break links to namespaces in the session space, use `-all=⎕SE`, and to break absolutely all links, `-all=*`.

Note that the list of namespaces is ignored when `-all` is used.

recursive

{on|off|**error**}

Break child namespaces too if they have separately defined links.

4.3.3 Result

`message` is a simple character vector describing namespaces that were: - effectively unlinked - not linked in the first place - not found

4.4 Link.CaseCode

```
names ← {options} ⎕SE.Link.CaseCode names
```

If case codes is on (default is off), each file name will have a case code (see below).

If you set up a `getFilename` hook when creating a Link, Link will prompt your hook for a file name whenever a new source file needs to be created. If case coding is also enabled, the file name should be correctly case coded. The *CaseCode* function is provided to add case coding to any file name.

What is a "case code"?

A reverse binary indication of the letter cases in the main part of the name, encoded in octal. For example

`HelloWorld` has the uppercase indication
`1000010000` which when reversed is
`0000100001` which is binary for
33_{10} which in octal is
41_{8} so the full name including case code is
`HelloWorld-41`

ARGUMENTS

`names` is a simple character vector or nested vector of character vectors giving the names of items for which to generate case coded results.

RESULT

- file name(s) with case code

4.5 Link.Create

4.5.1 Syntax

```
]LINK.Create <ns> <dir> [-source={ns|dir|auto}] [-watch={none|ns|dir|both}] [-casecode] [-forceextensions] [-forcefilenames] [-arrays] [-sysvars] [-
flatten] [-beforeread=<fn>] [-beforewrite=<fn>] [-getfilename=<fn>] [-codeextensions=<var>] [-typeextensions=<var>] [-fastload]

message ← {options} ⎕SE.Link.Create (namespace directory)
```

4.5.2 Arguments

- `namespace` is either a reference to, or a simple character containing the name of a namespace.
  In the user command `<ns>` is simply the name of the namespace. If a reference is used, it must refer to a namespace which has a display form which has name class 9 and can be used to locate the namespace (as opposed to an "anonymous" space with a name containing `[namespace]` or similar segments).
- `directory` is a simple character vector containing the path to a file system directory without any trailing slash or backslash.
  In the user command, `<dir>` is the path to the file system directory.

4.5.3 Result

- `message` is a simple character vector describing the established link, along with possible failures

4.5.4 Common options

source

Default: **auto**

The **source** option specifies whether to consider the namespace in the active workspace (ns) or directory on the file system (dir) as the source (also used by a subsequent Refresh).

`source` is a simple character vector, one of `'ns'`, `'dir'` or `'auto'`.

- **dir** means that the namespace must be non-existent or empty and will be populated from source files.
- **ns** means that the directory must be non-existent or empty and will be populated by source files for the items in the namespace.
- **auto** will use whichever of ns or dir that is not empty. If both are empty, it will use **dir** on a subsequent Refresh.

watch

Default: **both** if a file system watcher can be created, else **ns**

The **watch** option specifies which sides of the link to watch for changes (and synchronise). Watching a **dir** (or **both**) is currently only supported using the .NET Framework or .NET Core.

`watch` is a simple character vector, one of `'none'`, `'ns'`, `'dir'` or `'both'`.

- **none:** changes are not automatically propagated across the link in either direction.
- **ns:** changes made in a linked namespace changes (made with the editor) will be copied to files. Note that it will **not** reflect changes made using other mechanisms, such as assignment, `⎕FX`, `⎕FIX`, `⎕CY`, or `⎕NS`. If you want to programmatically change an item so that the change is reflected to files, you should use `⎕SE.Link.Fix`.
- **dir** will mirror changes made to files (using any mechanism) into the linked namespace. Note that there is a chance that updating a large number of files (e.g. git checkout, git pull or an unzip) may cause the file system watcher to miss changes and not report them to Link. If the source files are on a network drive, the file system watcher may be even more unreliable. Use `⎕SE.Link.Resync` if you suspect something is wrong.
- **both** will synchronise changes in both directions. This is the default, and is recommended except in very special circumstances.

> **Note**
>
> `Link.Refresh` can be used to force a wholesale update of everything based on the setting of the `-source` option, and `Link.Resync` can always be used to generate a list of differences between the workspace and linked directories if you are in doubt about the current state.

arrays

Default: **off**

The **arrays** flag will export arrays on link creation.

- if simply set (`options.arrays←1` for the function or `-arrays` for the user command), then all arrays are exported.
- if set to a comma-separated list of names (`options.arrays←'name1,name2,...'` for the function or `-arrays=name1,name2,...` for the user command) then arrays with specified names are exported.

This option takes effect only when **source** is **ns**, and only when the link is initially created.

sysVars

Default: **off**

The **sysVars** flag will export namespace-scoped system variables to file.

The exhaustive list of exported variables is: `⎕AVU` `⎕CT` `⎕DCT` `⎕DIV` `⎕FR` `⎕IO` `⎕ML` `⎕PP` `⎕RL` `⎕RTL` `⎕USING` `⎕WX`. They will be exported for all unscripted namespaces.

This option takes effect only when **source** is **ns**.

forceExtensions

Default: **off**

The **forceExtensions** flag forces correct file extensions.

If enabled, file extensions will be adjusted (if necessary), when an item is defined in the workspace from an external file, so that the file extension accurately reflects the type of the item according to **typeExtensions**.

forceFilenames

Default: **off**

The **forceFilenames** flag will force correct file names.

If enabled, file names will be adjusted so that they match the item name, when an item is defined in the workspace from an external file, so that the file name matches the name of the item.

By default, Link will always create new files with the same name as items created in the active workspace. However, it will not insist that file names match item names when importing items from a directory.

If **forceFilenames** is not set. Link will update to the same file that an item was loaded from, even though the file name does not match the item name.

4.5.5 Advanced Options

flatten

Default: **off**

The **flatten** flag prevents the creation of sub-namespaces in the active workspace.

The **flatten** option will load all items into the root of the linked namespace, even if the source code is arranged into sub-directories. This is typically used for applications that have source which is divided into modules, but still expects to run in a "flat" workspace.

Note that if **flatten** is set, newly created items need special treatment:

- If a function or operator is renamed in the editor, the new item will be placed in the same folder as the original item.
- If a new item is created, it will be placed in the root of the linked directory.
- It is also possible to use the **getFilename** setting to add application-specific logic to determine the file name to be used (or prompt the user for a decision).

A suggested workflow is to always create a stub source file in the correct directory and edit the function that appears in the workspace, rather than creating new functions in the workspace.

This option takes effect only when **source** is **dir**.

caseCode

Default: **off**

The **caseCode** flag adds a suffix to file names on write.

If your application contains items with names that differ only in case (for example `Debug` and `DEBUG`), and your file system is case-insensitive (for example, under Microsoft Windows), then enabling **caseCode** will cause a suffix to be added to file names, containing an octal encoding of the location of uppercase letters in the name.

For example, with caseCode on, two functions named `Debug` and `DEBUG` will be written to files named `Debug-1.aplf` and `DEBUG-37.aplf`.

> **Note**
>
> Dyalog recommends that you avoid creating systems with names that differ only in case. This feature primarily exists to support the import of applications which already use such names. You will probably also want to enable **forceFilenames** if you enable **caseCode**.

beforeWrite

If you specify a **beforeWrite** function, it will be called before Link updates a file or directory, allowing support of custom code or data formats.

`beforeWrite` is a simple character vector containing the name of a function relative to the linked namespace.

In the user command, simply give the name. For example:
`]LINK.Create -beforeWrite=Foo ns /tmp/folder`

Your function will be called with a nested right argument containing the following elements:

Index	Description
[1]	Event name ('beforeWrite')
[2]	Reference to a namespace containing link options for the active link.
[3]	Fully qualified filename that Link intends to write to (directories end with a slash)
[4]	Fully qualified APL name of the item that Link intends to write
[5]	Name class of the APL item to write
[6]	Old APL name (different from APL name if the write is due to a rename)
[7]	Source code that Link intends to write to file

> **Note**
>
> Do not assume a specific length, more elements may be added in the future.

Your callback function must return one of the following results:

- `0`: The **beforeWrite** function has completed all necessary actions. Link should not update any files.
- `1`: The **beforeWrite** function wishes to "pass" on this write: Link should proceed as planned.

beforeRead

If you specify a **beforeRead** function, it will be called before Link reads source from a file or directory, allowing support of custom code or data formats.

`beforeRead` is a simple character vector containing the name of a function relative to the linked namespace.

In the user command, simply give the name. For example:
`]LINK.Create -beforeRead=Foo ns /tmp/folder`

Your function will be called with a nested right argument containing the following elements:

Index	Description
[1]	Event name (`'beforeRead'`)
[2]	Reference to a namespace containing link options for the active link.
[3]	Fully qualified filename that Link intends to read from (directories end with a slash)
[4]	Fully qualified APL name of the item that Link intends to update
[5]	Name class of the APL item to be read

> **Note**
>
> Do not assume a specific length, more elements may be added in the future.

Your callback function must return one of the following results: - `0` : The **beforeRead** function has completed all necessary actions. Link should not update the workspace. - `1` : The **beforeRead** function wishes to "pass" on this read: Link should proceed as planned.

getFilename

If you specify a **getFilename** function, it will be called before Link updates a file or directory, allowing you to modify the name (or more likely the extension) of the file used to store the source for an APL item. Changing the file name this way allows you to override the **caseCode**, **forceFilenames** and **forceExtensions** options.

`getFilename` is a simple character vector containing the name of a function relative to the linked namespace.

In the user command, simply give the name. For example:
`]LINK.Create -getFilename=Foo ns /tmp/folder`

Your function will be called with a nested right argument containing the following elements:

Index	Description
[1]	Event name (`'getFilename'`)
[2]	Reference to a namespace containing link options for the active link.
[3]	Fully qualified filename that Link intends to use (directories end with a slash)
[4]	Fully qualified APL name of the item
[5]	Name class of the APL item
[6]	Old APL name (different from APL name if the write is due to a rename)

> **Note**
>
> Do not assume a specific length, more elements may be added in the future.

Your callback function must return a simple character vector which must be one of:

- empty: to signify that Link should proceed with the suggested file name.
- non-empty: to specify the name to be used.

codeExtensions

Default: `'aplf' 'aplo' 'apln' 'aplc' 'apli' 'dyalog' 'apl' 'mipage'`

Specify file extensions that are expected to contain source code. Link will only process changes to files with the specified extensions.

`var` is a nested vector of character vectors.

From a user command, the syntax is `-codeExtensions=var` where `var` holds the expected vector of extensions.

customExtensions

Default: `''  ⍝ an empty character vector meaning no custom extensions`

Specifies additional file extensions handled by **beforeRead** functions.

`customExtensions` is a nested vector of character vectors.

From a user command, the syntax is `-customExtensions=var` where `var` holds the expected vector.

If you have specified a **beforeRead** handler function, and your code supports the use of custom file extensions to store source data in application-specific formats, you need to set **customExtensions** so that Link does not ignore changes to these file types.

The reason for splitting the list of extensions into two parts (**codeExtensions** and **customExtensions**) is to avoid your code having to repeat the list of standard extensions, or update this list if it should be extended in the future.

typeExtensions

Default: `6 2⍴2 'apla' 3 'aplf' 4 'aplo' 9.1 'apln' 9.4 'aplc' 9.5 'apli'`

The **typeExtensions** table specifies the default extension that should be used when creating a new file to contain the source for an item of a given type.

typeExtensions is a two-column matrix with numeric name class numbers in the first column nd character vectors of corresponding file extensions in the second column.

From a user command, the syntax is `-typeExtensions=var` where `var` holds the expected array.

Note that the **forceExtensions** switch can be used to correct all extensions on pre-existing files when a link is created.

The default corresponds to:

Type	extension
2	apla
3	aplf
4	aplo
9.1	apln
9.4	aplc
9.5	apli

fastLoad

Default: **off**

The **fastload** flag will reduce the load time by not inspecting the source to detect name clashes.

This affects only initial directory loading, but not subsequent editor or file system watcher events. It is worth setting **fastLoad** for very large projects that don't produce name clashes (that is, two files defining the same APL name).

Side effects are (again, only at initial load time, not at subsequent events):

- good: load will be significantly faster because files won't be inspected to determine their true APL name.
- bad: clashing names won't be detected: files may silently overwrite each other's APL definition if they define the same APL name.
- bad: **forceFileNames**/**forceExtensions** won't be observed
- bad: **beforeRead** may report incorrect name class

This option takes effect only when **source** is **dir**.

4.6 Link.Export

```
]LINK.Export <ns> <dir> [-overwrite] [-casecode] [-arrays{=name1,name2,...}] [-sysvars]
msg ← {opts} ⎕SE.Link.Export (ns dir)
```

This function takes the same arguments as Link.Create but saves the contents of a namespace to directory without creating a Link.

If the source is an unscripted namespace, then the destination is interpreted as a directory.

If the source is anything else, then the destination is interpreted as a directory (and a correctly named file will be created there), *unless* it ends with a recognised extension (like `.aplf`), in which case it is interpreted as a file name.

ARGUMENTS

- ns : unscripted namespace or APL name
- dir: directory or file name

OPTIONS

- `overwrite` : Allow overwriting existing files in the destination directory
- other options have same effect as in Link.Create

RESULT

- String describing the exported source and destination, along with possible failures

4.7 Link.Expunge

```
]LINK.Expunge <item>
{available} ← ⎕SE.Link.Expunge items
```

This function is intended as a replacement for the system function `⎕EX` in tools that manage code. It removes an item from the workspace and also deletes the corresponding source file.

If you manually `)ERASE` items, you can subsequently call Expunge to remove the source file.

ARGUMENTS

- APL item name(s)

RESULT

- Simple Boolean vector with one element per name in the right argument.

The value of an element of the result is 1 if the corresponding name is now available for use. This does not necessarily mean that the existing value was erased for that name. A value of 0 is returned for an ill-formed name or for a distinguished name in the argument.

4.8 Link.Fix

```
{fixed} ← {namespace} {name} {oldname} ⎕SE.Link. Fix src
```

This function is intended as a replacement for `⎕FIX` or `⎕FX` in environments in which some or all namespaces are linked. It will allow you to add or modify an array, function, operator, or scripted namespace, class or interface, without worrying about whether the namespace is linked. The source will be fixed in the target namespace, and the corresponding file will be created/updated if there is an active link which specifies that the namespace is being watched.

For arrays, Fix expects the source to be Array Notation (you can use `⎕SE.Dyalog.Array.Serialise` to produce it from array values). For other items, it uses the source in the form that would be produced by `⎕NR` or `⎕SRC`. In all cases the source is a vector of text vectors.

Normally, one can use `⎕FIX` or `⎕FX` inside the target namespace, e.g. `myns.⎕FIX 'avg←{sum←+⌿⍵' 'sum÷≢⍵}'` but since `Link.Fix` exists only as `⎕SE.Link.Fix` then the target namespace must be explicitly specified as in `myns ⎕SE.Link.Fix 'avg←{sum←+⌿⍵' 'sum÷≢⍵}'`. The default namespace is the calling namespace.

Note: If the item has already been updated or created and you only need to trigger an update of the source file, you can also use Link.Add.

RIGHT ARGUMENT: SOURCE

- A vector of character vectors representing the source code of the item to be defined

LEFT ARGUMENT: {NAMESPACE} {NAME} {OLDNAME}

- namespace: The namespace (by name or by reference) within which the source shall be fixed. Defaults to `''` which means the calling namespace.
- name: The name of the item being defined. Defaults to `''` which means that the name will be defined by the source to be fixed. The name is required only for arrays, because their source doesn't contain their name.
- oldname: The old name of the fixed item, if this operation is a rename. Defaults to `name`, which means it is not a rename.

RESULT

- The name of the item that was defined.

4.9 Link.GetFileName

```
files ← ⎕SE.Link.GetFileName items
```

Returns the fully qualified name of the file containing the source of the given APL item. See also ⎕SE.Link.GetItemName.

ARGUMENTS

- A simple vector with an APL item name, or a vector enclosed names. The result will have the same structure as the argument.

RESULT

- For each APL item name:
- if item does not exist or does not belong to a linked namespace: empty vector
- otherwise: file name that the item is linked to

4.10 Link.GetItemName

```
items ← ⎕SE.Link.GetItemName files
```

Returns the name of the fully qualified APL item that is linked to a file. See also ⎕SE.Link.GetFileName.

ARGUMENTS

- A simple vector containing a file name, or a vector of enclosed names. The result will have the same structure as the argument.

RESULT

- For each file name:
- if file does not exist or does not belong to a linked directory: empty vector
- otherwise : item name that the file is linked to

4.11 Link.Import

```
]LINK.Import <ns> <dir> [-overwrite] [-flatten] [-fastload]
msg ← {opts} ⎕SE.Link.Import (ns dir)
```

This function takes the same arguments as Link.Create, but loads a directory containing source files into a namespace without creating an active link.

If source is a directory, then its contents are imported into the destination namespace.

If source is a single file, then the corresponding APL name is created in the destination namespace.

ARGUMENTS

- ns: namespace
- dir: directory or file name

OPTIONS

- `overwrite`: Allow overwriting APL names in the destination namespace
- other options have same effect as in Link.Create

RESULT

- String describing the imported destination and source, along with possible failures

4.12 Link.LaunchDir

```
dir ← ⎕SE.Link.LaunchDir
```

If APL was launched with a `LOAD` or `CONFIGFILE` parameter, `Link.LaunchDir` returns the fully qualified name of the directory in which the file used to start APL is located. If both were specified, the `LOAD`ed file takes priority.

If neither parameter is specified, the current working directory is returned.

This function is useful during the startup of applications loaded directly from source, and allows you to locate additional resources that are located relative to the source for the code used to start the application. For examples of usage, see setting up your environment

ARGUMENTS

- None

RESULT

- A character vector containing a fully qualified directory name.

4.13 Link.Notify

```
{name} ← ⎕SE.Link.Notify args
```

When synchonisation is active, Link will call Notify each time it detects a change to a linked source file. If synchronisation is not enabled, you can use this function to bring an external change into the active workspace, to notify the link system that an external file has changed.

If the workspace and directories become un-synchronised, you are probably better off using Link.Resync to get a list of differences, or Link.Refresh to completely re-load the external source.

ARGUMENTS

- **type** of event that happened
  - `'created'` : new file
  - `'changed'` : update to existing file
  - `'renamed'` : a file or subdirectory got a new name
  - `'deleted'` : a file or directory was erased
- **path** of affected file or directory
- **oldpath** is the previous path

can be omitted for all but a **rename** event

RESULT

- If link updated an APL item, its full name is returned as a string. Otherwise an empty string is returned.

4.14 Link.Refresh

```
]LINK.Refresh <ns> [-source={ns|dir|auto}]
msg ← {opts} ⎕SE.Link.Refresh ns
```

Refresh will break and re-create a link by using one one side of the link as source, and bringing the other side into line.

> **Note**
>
> **Refresh has the potential to lose changes:** Refresh will overwrite one end of the link without checking for changes. Link.Resync provides better control, allowing you to review the differences before selecting how they should be resolved, and is now recommended in place of Refresh in most scenarios.

Refresh is useful when you have decided not to watch one side of a link, but now want to pick up any changes that have happened since the link was created or most recently refreshed.

- To bring the workspace into line with the source directories, use `source=dir`.
- If you have made changes to linked namespaces using other mechanisms than the editor (such as using `⎕FIX`, `⎕FX`, `⎕NS`, `⎕CY` or assignment), you can Refresh with `source=ns` to update the directory.

ARGUMENTS

- namespace(s)

OPTIONS

- **source** {ns|dir|**auto**}

  Whether to consider the ns or dir as the source for the link. - `dir` means that items in the namespace will be overwritten by items in files. - `ns` means that items in files will be overwritten by items in the namespace. - `auto` re-uses the same source that was determined at Create time.

  The default is to use the setting that was specified at creation (`auto`).

RESULT

- String describing the established link, along with possible failures

4.15 Link.Resync

```
]LINK.Resync <ns>

msg ← {opts} ⎕SE.Link.Resync ⍵
```

`Link.Resync` will re-synchronise the contents of linked namespaces and the corresponding source directories. It is useful when:

- You know that you have made changes of a type which will not trigger updates, such as function assignments or the `)COPY` system command
- You have reason to believe that the file system watcher might have missed some updates
- You have loaded a workspace that was saved with active links. In this case, all Link functionality will be disabled until you do a Resync to ensure that the workspace content matches the external source.

By default, Resync takes no action, but outputs a list of differences found, with a recommendation of whether the the difference should be resolved by updating a file should be created or updated (`→`), or that a file should be read and the workspace updated (`←`). For example:

```
      ]link.resync
2 updates required: use -proceed option to synchronise
 Name          Direction  File             Comments
 #.badapp.Foo  →                           Item has no corresponding file
 #.badapp.Goo  ←          /myapp/Goo.aplf  File now dated 08:17 yesterday,
                                           WS copy is dated 03 Sep 2021
```

If you accept the recommendations, you can add the `proceed` switch, after which the link will be up-to-date and work can proceed normally.

```
      ]link.resync -proceed
1 file read, 1 file updated
```

If Link is not able to suggest an action, it will display `?` in the direction column, for example if the source file is now older than when it was loaded into the workspace, `-proceed` will be rejected, you will need to resolve the difference manually.

> **Note**
>
> **Beware: The recommendations are NOT necessarily the correct actions!** For example, if an item exists in the workspace but not on file, Resync will recommend creating the file (and vice versa if a file exists but the item is not found in the workspace). But if the file was intentionally removed or renamed in the source, the correct action is actually to delete or rename it in the workspace. Always review the recommendations carefully before `-proceed` ing. Of course, if you are using a source code management system like Git, you should easily be able to detect and recover from mistakes.

ARGUMENTS

- ns: namespace(s) to consider

OPTIONS

- **proceed**

Whether to execute the changes suggested by Resync without the `proceed` option.

- **arrays**, **sysvars**

Whether arrays and system variables should be included in the analysis. See Link.Create for details of these options.

RESULT

- String describing the changes made, if requested.

4.16 Link.Status

```
]LINK.Status [<ns>] [-extended]
status ← {opts} ⎕SE.Link.Status ns
```

This function provides information about existing links.

ARGUMENTS

- ns: namespace to look for links in (use `''` to list all links)

OPTIONS

- **extended** {**0**|1}

  Request additional information

RESULT

- Table of links

  First three columns are always:

- namespace reference
- directory name
- number of linked files and directories (excluding root directory)

  If `extended` was specified, options settings for each link:

- case code
- flatten
- force extensions
- force filenames
- watch
- paused

4.17 Link.StripCaseCode

```
files ← {opts} ⎕SE.Link.StripCaseCode files
```

If case codes is on (default is off), each file name will have a case code.

If you set up a `beforeRead` hook when creating a Link, Link will allow your prompt your hook take appropriate action before a file is imported. If the filename may have a case code. The *StripCaseCode* function is provided to remove case coding from any file name.

ARGUMENTS

- file name(s)

RESULT

- file name(s) without case code

4.18 Link.TypeExtension

```
ext ← opts ⎕SE.Link.TypeExtension nc
```

This function is used internally by Link, but is also available for use when implementing extensions to Link using exits like `beforeRead` and `beforeWrite`.

RIGHT ARGUMENT

- nameclass of item

LEFT ARGUMENT

- link options namespace used as left argument of Link.Create

RESULT

- character vector of the extension (without leading `'.'`)\ Note that extension will be (`,'/'`) for unscripted namespaces (name class `¯9`) because they map to directories

4.19 Link.Version

```
version ← ⎕SE.Link.Version
```

This niladic function returns the current Link semantic version number as a string in the format `'X.Y.Z'`, where X Y and Z are non-negative integers. Development or experimental versions will have a trailing hyphen and string such as `'X.Y.Z-alpha3'`.