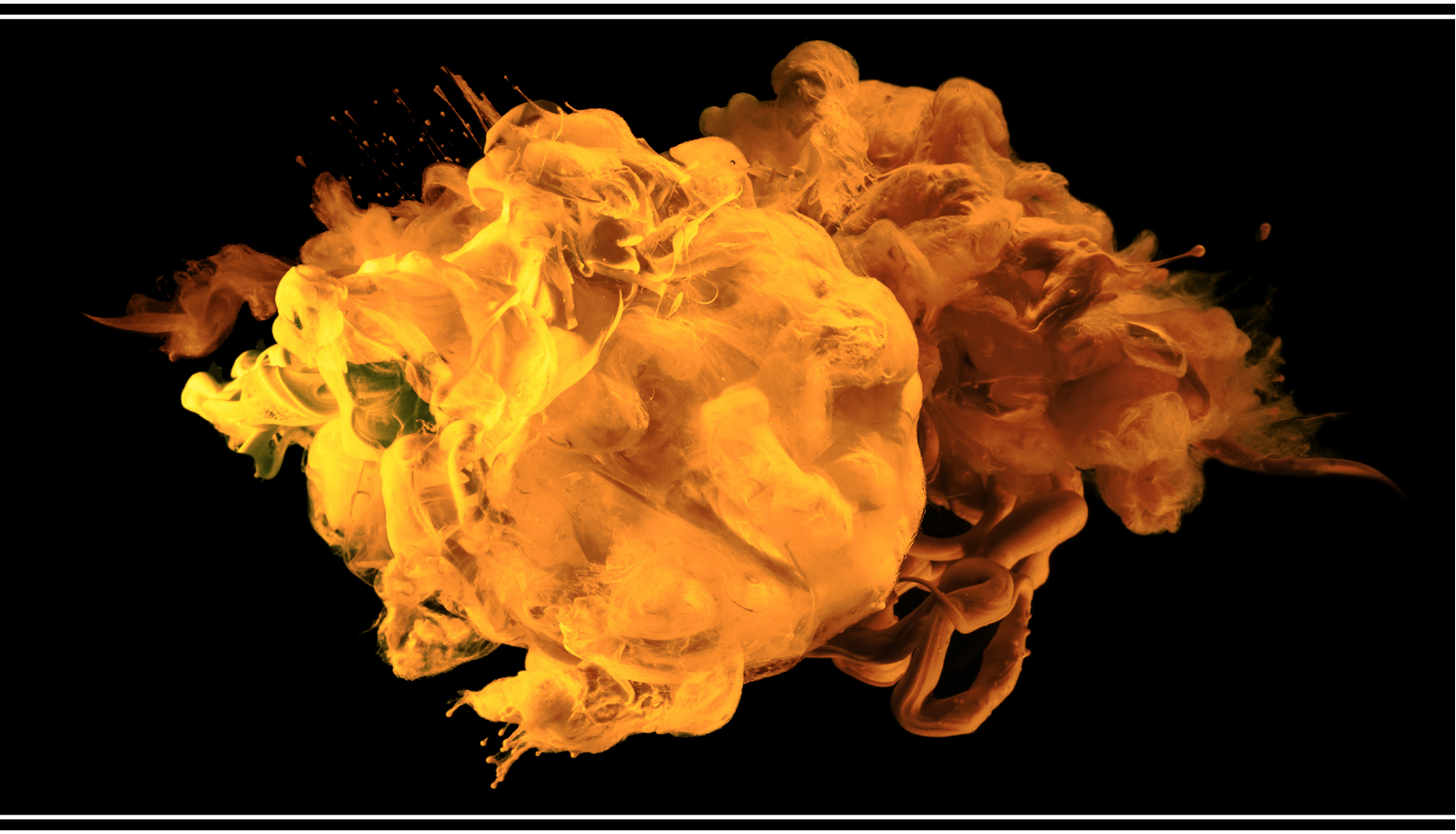
Link
User Guide

Link version 3.0



DYALOG

The Tool of Thought for Software Solutions

Link User Guide

Link version 3.0

Contents

1. Introduction

Link version 3.0 is included with Dyalog version 18.1. If you have an earlier version of APL or Link, you may want to check out one or more of the following pages before continuing:

- Migrating to Link 3.0 from from Link 2.0: If you are already using an earlier version of Link.
- 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 pick Link up directly 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 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 that an SCM is not a requirement for using Link; without an SCM you just need your own strategy for taking suitable copies of your source files, as your 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.

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.

1.2 Link is NOT...

- **A source code management system**: we recommend using Git to manage the text files that Link will help you create and edit using 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, arrays are only written to source files upon request using 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.

1.3 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
```

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 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.

1.4 Functions vs. User Commands

Before we move on to look at creating some links, a few words about the two ways that Link functionality can be accessed. With a few exceptions, each Link API function has a corresponding User Command, designed to make the functionality slightly easier to use interactively.

1.4.1 User commands

The user commands have the general syntax

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

where `arg2` 's presence depends on the specific command, and `-name` is a flag enabling the specific option and `-name=value` sets the specific 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 the needed array.

For a list of installed user commands, type:

```
]link.?
```

1.4.2 API Functions

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

```
options FnName arg
```

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 (just like omitted modifiers in the user command).

The details of the arguments to the functions and the user commands can be found in the API Reference.

1.5 Further Reading

To continue your journey towards getting set up with Link, you will want to read:

- Getting Started, 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 may 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 that is not in Link, you may want to read).
- Migrating to Link 3.0 from SALT, if you are already managing text source using Link's predecessor SALT.

1.6 Frequently Asked Questions

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

2. Getting Started

2.1 Getting Started

Linking to an Existing Directory

In daily use, the assumption is that your source already exists in one or more file system folders that you are managing using an SCM like Git, or taking copies of at suitable intervals using some other mechanism. Link.Create loads the source code into the workspace so that you can work with it, keeping the content of the workspace synchronised with the source files, no matter which side of the link you make changes.

For example, the following user command load all the code in the folder `/home/sally/myapp` into a namespace called `myapp` , creating the namespace if it does not exist:

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

The first argument to the user command, and to the corresponding API function `SE.Link.Create` is the name of a namespace (in the case of the function you could also use a reference to an existing namespace). The second argument is a directory name.

Starting with a Workspace

If your existing code is in a workspace rather than in source files, you may want to skip forward and read the section on converting a workspace to source files before continuing.

Starting a New project

Of course, you might be one of those lucky people who is starting a completely new project. In this casem you can still use `Link.Create` , but you will need to create either the namespace or the folder (or both) first. For the command to succeed, at least one of them needs to exist and exactly one of them needs to contain some code.

- 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(+⌿(x←⍵-⍺)×⍵)÷⍴⍵}
```

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 `/tmp/stats` is empty or does not exist, the following command will detect that there is code in the workspace but not in the directory, and create a link based on the namespace that we just created:

```
      ]LINK.Create stats /tmp/stats -source=ns
Linked: #.stats ←→ C:\tmp\stats
```

The double arrow `←→` in the output indicates that synchronisation is bi-directional. We can verify that the three expected files have been created:

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

Let's verify that our source directory can be used to re-build the original namespace::

```
      )CLEAR
clear ws
      ]LINK.Create stats /tmp/stats
Linked: stats ←→ C:\tmp\stats
      stats.⎕NL -3 ⍝ Verify functions were loaded as expected
 Mean  Root  StdDev
```

If you have an existing workspace containing several namespaces, code in the root of the workspace, or variables, you will want to read about converting your workspace to text source.

Working with the Code

Once the link is set up, 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 within the `stats` namespace are instantly copied to the corresponding source file.

NB: 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. If you do not have a File System Watcher available on your platform, it may be a few seconds before the Crawler kicks in and detects external changes.

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:

```
      )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 '/tmp/stats/*'
 /tmp/stats/Mean.aplf  /tmp/stats/Median.aplf  /tmp/stats/Root.aplf  /tmp/stats/StdDev.aplf
```

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.

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.

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
```

Note that by default, source files for arrays are presumed to define the initial value of the array when the application starts. Note that changes made to arrays will not be picked up by a crawler even if the array was originally loaded from a source file. You always need to use the built-in editor or explicitly call Link.Add to update the source file with a new value.

Although changes to the array in the workspace are not automatically written to file, changes made to the source files will always be considered to be updates to the source code and reflected in the workspace if synchronisation is active.

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.

2.2 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.

To illustrate, we will create a small application that uses the stats library that we created in the introduction. We'll 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.Run
```

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:

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

      linkdemo.Main
      Enter some numbers:
⎕:
      5?100⍴100
Mean:    102.4
StdDev:  30.1
Enter some numbers:
⎕:
          0
```

Automating Startup

Starting with 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' '/home/sally/stats' ⍝ Load the stats library
[5]   ST←#.stats
[6]
[7]   :If run
[8]       Main
[9]       ⎕OFF
[10]  :EndIf
    ∇
```

We can now launch out 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 a potentially confusing situation.
- `)SAVE` the workspace

2.3 WStoLink

Converting an Existing Workspace to use Link

In principle, it should be possible to write the entire contents of any workspace to an empty folder called `/folder/name` using the following command:

```
]Link.Create # /folder/name -arrays -sysVars
```

-ARRAYS

By default, Link assumes that the "source code", only includes functions, operators, namespaces and classes. Variables are assumed to contain data which can either be erased or stored in some kind of database, rather than being part of the source. The `-arrays` causes all arrays in the workspace to be writte to source files as well (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 might 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 the Link.Add.

WORKSPACE 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.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 calls to Link.Create to write the contents to different locations.

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.

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 wish to manage separately. In this case, all the source files will end up in the same folder.

If you subsequently separate the code into separate folders in order to make the source more manageable, a `-flatten` switch allows you to bring them everything back into the original "flat" workspace structure, so that the application will run without requiring code changes.

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

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`. For some ideas on how to set this up, see Setting up your Environment.

2.4 SALTtoLink

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 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.

NO VERSION CONTROL FEATURES

SALT includes primitive 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.

DIFFERENT API

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

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!

.DYAPP FILES

With 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.

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.

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.

THE ⍝V: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.

2.5 Installation

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

Link is maintained as an open source project at https://github.com/dyalog/link. Once you have downloaded or cloned a particular tag or release of the source code to a folder on your machine, you have two alternatives:

- **OVERWRITE the installed Link** by replacing the contents of the **$DYALOG/StartupSession** in your installation with the contents of the StartupSession folder that you have downloaded.
- **Set the `DYALOGSTARTUPSE` environment variable** to point to the StartupSession folder that you have downloaded. This route is recommended if you think you will regularly be updating your copy of Link; if you have done a `git clone` of the repository, then all you need to do is a `git pull` and restart APL.

If you are using Dyalog APL version 18.0, you also need to update the user command file used to invoke Links user commands. This should only need to be done once, even if you subsequently download new versions of Link. Again, there are two options:

- Overwrite the installed file **$DYALOG/SALT/spice/Link.dyalog** with the corresponding file that you have downloaded.
- Place a copy of the downloaded file into your `MyUcmds` folder, which will cause it to take priority over the installed copy.

You will need to restart Dyalog APL each time you update these files, in order to pick up the new definitions.

3. 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.

3.0.1 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 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.

3.0.2 Other Significant Changes

The most important new features are:

- The addition of Link.Pause and Link.Resync which provide better support for resuming work after a break, especially if the active workspace has been saved and reloaded.
- "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.

3.0.3 Release Notes

A detailed list of new features added to recent releases and a few behavioural changes can be found in the Link 3.0 Release Notes.

4. Release Notes

This document provides a list of differences between recent versions of Link. For a discussion of differences between 3.0 and 2.0 that you need to be aware of, see Upgrading to Link 3.0.

4.1 Version 3.0

- When specifying a directory, a trailing slash is reserved for future extension
- Public functions now throw errors rather than return an error message when they fail.
- Link.Pause has been added
- Link.List has been renamed to Link.Status
- 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
- 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.
- Dyalog APL v18.1 or newer is required for the fixes to the following issues
- #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

4.2 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 18.1. 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.

4.3 Version 2.0

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

5. Discussion

5.1 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 that are worth discussing:

5.1.1 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.

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 - because the APL interpreter does not have a source form for such items. 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).

5.1.2 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 from (for example `#.myproject` rather than `#.[namespace]`).
- 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 under .Net and .Net Core. Note that the built-in APL editor *will* detect changes to source files on all platforms, but not before the editor is opened.
- 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 version 18.0, Dyalog v18.1 is recommended if it is important that all source be preserved as typed. Earlier versions of APL have small glitches that occasionally lose the source as typed under certain circumstances.

5.2 Workspaces

5.2.1 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 no longer a recommended 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 do `)SAVE` a workspace which has active links in it, this creates a potentially confusing situation. Objects defined in the workspace will still contain the information that was created by `2⌶FIX`, which means that after a re-load, editing may cause the editor itself to update the source files, even though Link has not been activated. If you reload the workspace on a different machine, the source files may not be available, leading to confusing error messages.

You should always Link.Break or Link.Resync after loading a workspace which was saved with links in it.

5.3 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 April 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 directory #) 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 (earlier versions used `⎕SE.Link.Links`, but version 3.0 only uses this for links with an endpoint in `⎕SE`), recording the endpoints and all options associated with the Link. The command `]Link.Status` can be used to report this information.
- 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.
- By default, a .NET File System Watcher is created to watch the directory for changes, so they can immediately be replicated in the workspace (if .NET is available)

CREATING APL SOURCE FILES AND DIRECTORIES

Link writes textual representations of APL objects to UTF-8 text files. 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 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

As a general rule, Link loads code into the workspace using `2⎕FIX'file://...'`.

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

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 pick Link up from GitHub, see the installation instructions.

The Crawler

In a future version of Link, hopefully available during 2021, an optional and configurable Crawler will run in the background 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 document Technical Details and Limitations provides much more information about the type of APL objects that are supported by Link.

5.3.1 Breaking Links

If Link.Break is used to explicitly break an existing Link, the entry is removed from `⎕SE.Link.Links`, and the namespace reverts to being a completely "normal" namespace in the workspace. If file system watch was active, the watcher 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 watch file that causes the file system watched 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 will be deleted.

5.4 History

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. Data quickly ended up in component files or other storage mechanism. As teams started writing larger systems, many development teams also created their own source code management systems, typically storing multiple versions of code in component files or SQL tables.

These SCM's 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 felt like time for a fresh start, and Link was born. 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 code 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.

The Future

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 and can replace the File System Watcher in environments where it is not available.
- Eliminating the use of SALT, with a new implementation of user commands and other mechanisms for loading source code into the interpreter, based on Link rather than SALT.
- 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 namespace.
- Improving integration with the APL Interpreter so that the editor will honour a Pause.

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

6. API

6.1 Index

The Link API functions are all found in ⎕SE.Link. Typically, API functions take a character vector or a nested vector as a right argument. The left argument may either be a namespace containing option values, or an array of character vectors. Namespaces may be specified by reference. For more details on setting options, look below the following table:

Basic API Function reference

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

] These functions have user command covers.

Advanced API Function reference

Function	Right Argument(s)	Left Argument(s)	Result
CaseCode	filename	*<none>*	case-coded filename
Fix	source	array: namespace name oldname	boolean
GetFileName]	items	*<none>*	filenames
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 '.')

] These functions have user command covers.

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
```

User commands

Some 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 above 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
```

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.

6.2 Link.Add

```
]LINK.Add <items>

msg ← ⎕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. A source file will be created/updated whether the linked namespace is watched or not.

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

ARGUMENTS

- APL item name(s)

RESULT

- String 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)

6.3 Link.Break

```
]LINK.Break [<ns>] [-all] [-recursive={on|off|error}]

msg ← {opts} ⎕SE.Link.Break ns
```

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.

ARGUMENTS

- namespace name(s) or reference(s)

OPTIONS

- `all` : Break all existing links (arguments are ignored)
- `recursive` {on|off|**error**}: Break child namespaces too if they have separately defined links.

RESULT

- String describing namespaces that were:
- effectively unlinked
- not linked in the first place
- not found

6.4 Link.CaseCode

```
names ← {opts} ⎕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

- file name(s)

RESULT

- file name(s) with case code

6.5 Link.Create

```
]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]

msg ← {opts} ⎕SE.Link.Create (ns dir)
```

ARGUMENTS

- **namespace** name of - or reference to - a namespace
- **directory** name of a file system directory (without trailing slash or backslash)

RESULT

- String describing the established link, along with possible failures

COMMON OPTIONS

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

Whether to consider the ns or dir as the source (also used by a subsequent Refresh). - `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.

Defaults to `auto`.

- **watch** {none|ns|dir|**both**}

  Specifies which sides of the link to watch for changes (and synchronise). - `ns` will mirror namespace changes (done with the editor) 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 namespace. Note that there is a chance that massive file changes (e.g. git checkout, git pull or an unzip) may cause the file system watcher to miss changes. It is recommended to Link.Pause the link before doing massive changes to files, then Link.Resync to resume file watching. - `both` will do both.

  Watching a `dir` (or `both`) is currently only supported using the .Net Framework or .NetCore, but a Crawler is planned to perform watching on all platforms, and to recover from cases where changes are not picked up by other mechanisms.

  The default is `both` where supported, else `ns`.

- **caseCode** (default off) 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 i s 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. Also note that you will probably want to enable **forceFilenames** if you enable **caseCode**.

- **forceExtensions** (default off) Force correct 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) Force correct filenames

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 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.

- **arrays** (default off) Export arrays

  - if simply set (to 1) (e.g. `-arrays`), then all arrays are exported
  - if set to a comma-separated list of names (e.g. `-arrays=name1{,name2,...}`) then arrays with specified names are exported

This option takes effect only when **source** is **ns**, and only when the link is initially created. Arrays will not be monitored for changes during operation of the application.

- **sysVars** (default off) 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**.

- **flatten** (default off) Do not create sub-namespaces

  flatten 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 which have source which is divided into modules, but still expects to run in a "flat" workspace.

  Note that if **flatten** is set, new 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 simple work-around is to always create a stub source file in the correct directory and editing the function that appears in the workspace, rather than creating new functions in the workspace.

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

- **beforeWrite** `ns.hookname` name of function to call before writing to file

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.

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** `ns.hookname` name of function to call before before reading a file

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.

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** `ns.hookname` name of the function to call to decide the file or directory name linked to an APL item

  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.

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

- **codeExtensions** File extensions that are expected to contain source code

When reacting to changes in a watched directory, Link will only process files if the changed file has one of the listed extensions.

The default is `'aplf' 'aplo' 'apln' 'aplc' 'apli' 'dyalog' 'apl' 'mipage'`

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

- **customExtensions** Specifies additional file extensions handled by **beforeRead** functions

  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.

  default is `''` - no custom extensions

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

  The reason for splitting the list of extensions into two parts 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** Specify the default file extensions to use when creating files

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 and corresponding file extensions in the second column.

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

The default is:

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

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

- **fastLoad** (default off) Flag to reduce the load time by not inspecting 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 with users that don't produce name clashes (i.e. 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**.

6.6 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 maintaining 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 interpreter as a directory (and a correctly named file will be created there), *unless* it ends with a recognised extension, in which case it is interpreted as a file name.

ARGUMENTS

- source : unscripted namespace or APL name
- destination: 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

6.7 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.

6.8 Link.Fix

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

This function is intended as a replacement for `⎕FIX` or `⎕FX` in code which manipulates linked namespaces. It will allow you to add or modify an array, function, operator, or scripted namespace, class or interface within a linked namespace. The source will be fixed in the target namespace, and the corresponding file will be created/updated.

For arrays, Fix uses the Array Notation from `⎕SE.Dyalog.Array`. For other items, it uses the source provided 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←+⌿ω' 'sum÷≢ω}'` 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←+⌿ω' 'sum÷≢ω}'`. The default namespace is the calling namespace.

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

When an item is edited and the namespace is being watched, a call is made to this function by the APL interpreter.

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 is 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

- 1 if the item was fixed in a linked namespace, else 0 (and the source code wasn't fixed)

6.9 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

- APL item name(s)

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

6.10 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

- file name(s)

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

6.11 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 a permanent 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

- destination: namespace
- source: 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

6.12 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 was 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.

ARGUMENTS

- None

RESULT

- A character vector containing a fully qualified directory name.

6.13 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.

Note: The Link.Refresh function can be used to synchronise all extant changes between a linked directory and namespace.

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.

6.14 Link.Pause

```
]LINK.Pause

msg ← ⎕SE.Link.Pause ⍬
```

Pause will temporarily suspend all synchronisation activities performed by Link. The recommended way to resume after a pause is to use Link.Resync, unless you are sure that you want to resume by using one side of the link as the definitive source, ignoring all changes on the other side, in which case Link.Refresh is the right choice.

Obviously, for links created with no synchronisation, Pause will have no effect. To change the `watch` setting, the link needs to be broken then created again.

Pause may be useful when doing batch code updates (such as `svn update` or `git pull`), copying or unzipping large quantities of source files, or other operations which may overload the file system watcher or cause unnecessary "thrashing" to occur.

It may also be useful when running application code, to guarantee that the code will not change while running.

Editing Code While Paused

While Link itself will not perform synchronisation while links are paused, the editor which is integrated with the Dyalog APL IDE will update source files in some situations.

In the default scenario, where Link is watching both sides of a link, the information which links each code object in the workspace to a source file is maintained by the interpreter. If code is edited using the built-in editor, the editor *will* automatically also update the source file (prompting the user for confirmation, depending on how the APL system is configured). The only way to avoid this is to use Link.Break. In the longer term we hope to convince the editor to honour paused links.

ARGUMENTS

- None

RESULT

- String stating whether Pause was successful or not.

6.15 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.

BEWARE: Refresh has the potential to lose changes: if there are un-synchronised changes on both sides of the link, then Refresh will destroy one set of changes (the non-source side will be overwritten by the source side). 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 occurred:

- 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

6.16 Link.Resync

```
]LINK.Resync

msg ← {opts} ⎕SE.Link.Resync ⍵
```

`Link.Resync` will re-synchronise your workspace and source directories. It is the best way to resume work if you have used Link.Pause to temporarily stop watching the file system, or you have loaded a checkpoint workspace that might contain obsolete code, or you have any other reason to suspect that the contents of the active workspace no longer match the source directories.

If you had previously used Link.Pause, your links will no longer be in a paused state following a Resync - unless you explicitly set the `pause=yes` option.

WARNING: Resync is one of the most recent items of functionality added to Link, and should be considered somewhat experimental in Link 3.0. While this is the case, the default value for the `confirm` option will be `list`, which means that Resync will display output documenting the updates that it intends to make. If there are any outstanding differences, you need to explicitly set `confirm=yes` to execute the synchronisation.

The current plan is that, once Resync reaches maturity, the default will become `confirm=yes`, and an optional Crawler will become available. The *Crawler* will run Resync in the background, from time to time, to keep. It is likely that this will happen in Link version 3.1, hopefully during the summer of 2021.

ARGUMENTS

- Currently unused, reserved for future enhancements

OPTIONS

- **confirm**

  Whether to execute the synchronisation, list the changes required, or both. - `list` means that a list of actions that would be performed will be displayed. - `yes` means that the actions will be performed. - `copy` means that the actions will be performed and the list of actions will also be returned.

  Defaults to `list` in 3.0, this is expected to change in Link 3.1.

- **pause**

  Whether the link should be in a paused state following the resync.

  Defaults to `no`.

RESULT

- String describing the changes made, if requested.

6.17 Link.Status

```
]LINK.Status [<ns>] [-extended]

status ← {opts} ⎕SE.Link.Status ns
```

This function provides details of existing links.

ARGUMENTS

- 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, link options settings:

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

6.18 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

6.19 Link.TypeExtension

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

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

6.20 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. Unstable versions will have a trailing hyphen and string such as `'X.Y.Z-alpha3'`.