

Link User Guide

Link version 3.0

The Tool of Thought for Software Solutions

Dyalog is a trademark of Dyalog Limited
Copyright © 1982-2021 by Dyalog Limited
All rights reserved.

Link User Guide

Link version 3.0

1. Introduction	4
1.1 What is Link?	4
1.2 Link is NOT...	4
1.3 Link Fundamentals	4
1.4 Functions vs. User Commands	4
1.5 Further Reading	5
1.6 Frequently Asked Questions	5
2. Getting Started	6
2.1 Getting Started	6
2.2 Setting up your Environment	8
2.3 WStoLink	10
2.4 SALTtoLink	11
2.5 Installation	12
3. Upgrading to Link 3.0	13
4. Release Notes	14
4.1 Version 3.0	14
4.2 Version 2.1	14
4.3 Version 2.0	14
5. Discussion	15
5.1 Technical Details and Limitations	15
5.2 Workspaces	16
5.3 How does Link work?	17
5.4 History	18
6. API	20
6.1 Index	20
6.2 Link.Add	22

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 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.

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 text 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:

```
jlink.?
```

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 case 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←{α←2 ⍉ ω*÷α}
stats.StdDev←{2 Root{+,×÷÷p},ω←Mean ω}
```

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←ω
  Mean vals[asc[[2÷~0 1+pvals]]]
}
```

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 than 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≠pdata◀□
[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: #stats ← c:\tmp\stats

linkdemo.Main
Enter some numbers:
□:
50+?100p100
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 -i □+□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 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 -i []+[]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 written 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!

.DYAPPFILES

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 `⍷: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 maintenance 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^o `Ⓜ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, `.apIn` 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 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

6.1 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 ¹	items		message
Break ¹	namespaces	options: <code>all</code> <code>exact</code>	message
Create ¹	namespace directory	options: <code>source</code> <code>watch</code> [and many more]	message
Export ¹	namespace directory	options: <code>overwrite</code> <code>caseCode</code> <code>arrays</code> <code>sysVars</code>	message
Expunge ¹	items		boolean array
Import ¹	namespace directory	options: <code>overwrite</code> <code>flatten</code> <code>fastLoad</code>	message
LaunchDir	none	none	directory name
Pause ¹			message
Refresh ¹	namespace	options: <code>source</code>	message
Resync ¹		options: <code>confirm</code> <code>pause</code>	message
Status ¹	namespace	options: <code>extended</code>	message
Version			version number as string

¹ These functions have [user command covers](#).

Advanced API Function reference

Function	Right Argument(s)	Left Argument(s)	Result
CaseCode	filename	<code><none></code>	case-coded filename
Fix	source	array: namespace name oldname	boolean
GetFileName ¹	items	<code><none></code>	filenames
GetItemName ¹	filenames	<code><none></code>	items
Notify	event filename oldfilename	<code><none></code>	<code><none></code>
StripCaseCode	filename	<code><none></code>	filename without case code
TypeExtension	name class	option namespace used for Create	file extension (without leading <code>'.'</code>)

¹ 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 "
options.(source flatten)+['dir' 1
options SE.Link.Create 'myapp' '/sources/myapp'

Ⓜ create empty namespace
Ⓜ set two named options
Ⓜ 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 ¹ 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)