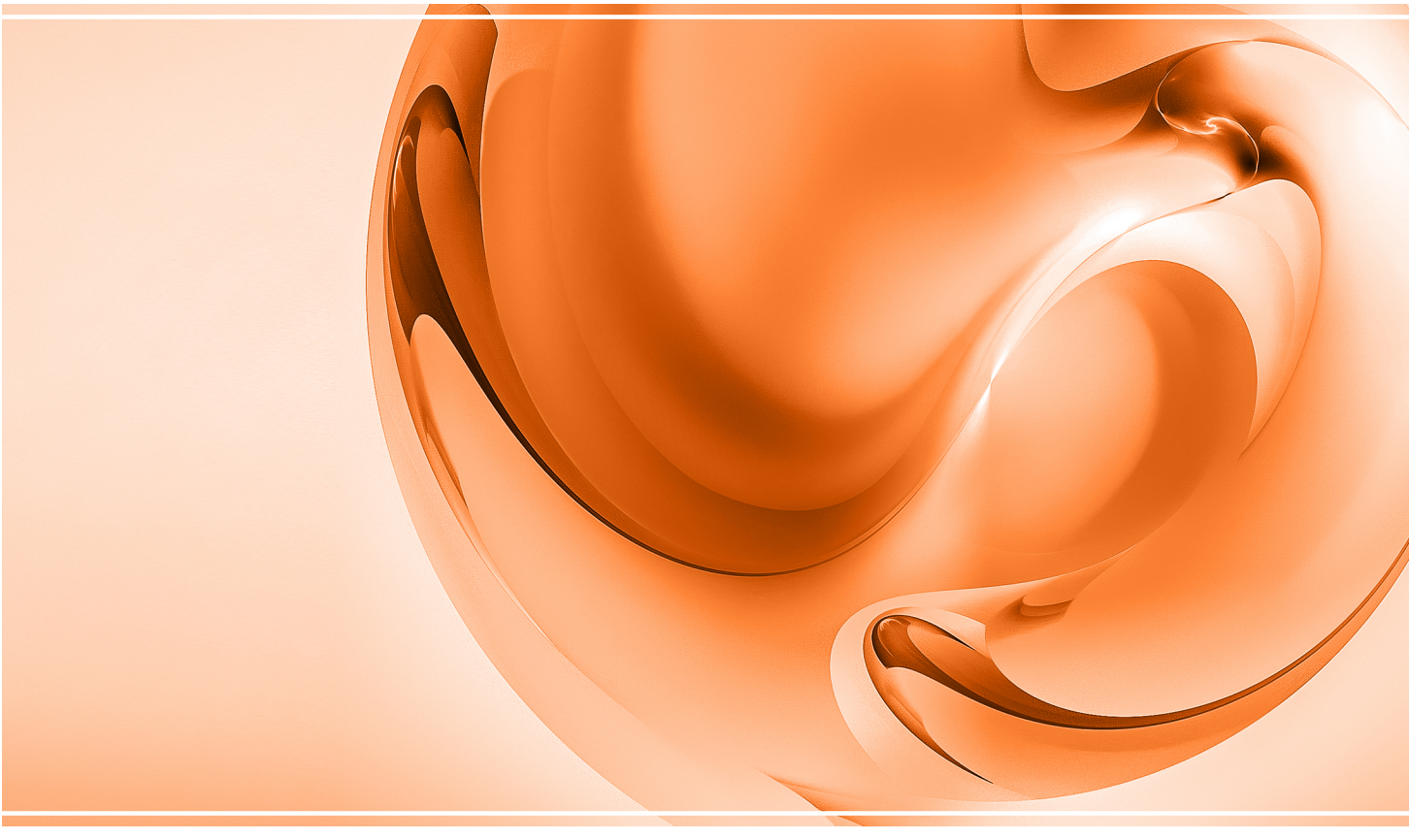
Shared Code Files User Guide

Dyalog version 19.0



DYALOG

The tool of thought for software solutions

Shared Code Files User Guide

Dyalog version 19.0
Document Revision: 20240129_190

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

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

Contents

1 About This Document

This document is intended as an introduction to shared code files, introduced for the purpose of improving the performance of large applications while reducing their memory consumption and initialisation time.

The exact specification of shared code files is subject to change from time to time. Dyalog Ltd recommends that code that loads and creates shared code files is embedded in utility functions that can easily be modified as required.

1.1 Audience

It is assumed that the reader has a reasonable understanding of Dyalog.

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

1.2 Conventions

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

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

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

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

Material of particular significance or relevance.

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information pertaining to behaviour in earlier releases of Dyalog or to functionality that still exists but has been superseded and is no longer recommended.

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

A full list of the platforms on which Dyalog version 19.0 is supported is available at https://www.dyalog.com/dyalog/current-platforms.htm. Although the Dyalog programming language is identical on all platforms, differences do exist in the way some functionality is implemented and in the tools and interfaces that are available. Differences in behaviour between operating systems are identified with the following icons (representing macOS, Linux, Microsoft Windows and UNIX respectively):

2 Introduction

While a standard Dyalog workspace (a **.dws** file) needs to be read by the interpreter and loaded in its entirety, a *shared code file* (a **.dwx** file) has a structure that allows it to be attached to the active workspace with a minimum of file operations. This significantly decreases the start time of application processes, especially when several processes run on the same machine.

When a shared code file is attached to the active workspace, a list of all the *names* of the functions, operators and variables that it contains is loaded into the active workspace. However, the definitions and values of these names are only paged into virtual memory the first time that the names are referenced, and they are not loaded into the active workspace unless their content is modified. In addition, because shared code files are memory-mapped by the operating system, the definitions and values are shared by concurrent processes. This means that, if a shared code file is already in use by one Dyalog application, then the name list can be loaded from shared memory by another application; the same is true for any of the names that are already in use (unless the system is low on memory, in which situation memory mapped pages are flushed from memory).

The use of shared code files is only supported on 64-bit Unicode interpreters and there are no current plans to extend this support. Shared code files are memory mapped, and they can only be attached by interpreters that use exactly the same memory layout as the system that generated them.

2.1 Benefits Offered by Shared Code Files

Many large applications are currently forced to load more code than is necessary because it is difficult to predict precisely what code will be used. The main benefit of shared code files is that applications only load code and data on demand.

BENEFIT 1: *Significantly reduced start-up times for applications*

Similarly, computational sub-processses (such as isolates) can be launched in a fraction of the time that would otherwise be required.

BENEFIT 2: *Reduced workspace usage*

Workspace size can be reduced or more space can be used to execute code more efficiently. A shared code file is materialised one page at a time, as memory is referenced. This means that the contents of comments, which are typically in a separate part of the file from the actual code lines, are usually only read from file if the code is edited. Similarly, running a compiled function is unlikely to require loading the source of the function into virtual memory.

BENEFIT 3: *Reduced file I/O and memory consumption*

This is most apparent on machines that run several processes using the same code and is due to the sharing of memory-mapped files.

BENEFIT 4: *More efficient application execution*

Objects residing in a shared code file remain outside the dynamic portion of the workspace unless they are modified. In some applications, this means that the complexity of the active workspace is significantly reduced. As a result, memory allocations are generally cheaper and less memory needs to be inspected and moved around when compactions occur.

2.2 Fundamental Limitations

Despite the benefits offered by shared code files, they will not replace the standard Dyalog workspace due to some fundamental limitations.

RESTRICTION 1: *Shared code files are read only*

Multiple processes can memory-map shared code files simultaneously; each process that uses a shared code file is using the contents of that shared code file directly as memory. This means that a shared code file cannot be updated while it is in use.

Instead, when an application modifies an object that resides in a shared code file, a copy of the relevant part of the shared code file is made in workspace memory and the original file is not modified.

RESTRICTION 2: *Shared code files each have a fixed virtual memory address*

A shared code file contains pointers to absolute memory locations contained within it. This means that the virtual memory address to which it is memory mapped must be fixed when the memory-mapping occurs. When a shared code file is created, a parameter specifies the virtual memory address at which it will be loaded and all pointers contained within it are adjusted to fit this address. If an application uses more than one shared code file, each shared code file must have a different address.

RESTRICTION 3: *Shared code files cannot be shared across architectures*

A shared code files cannot be converted in any way when it is used. This means that, unlike workspaces, component files and arrays transmitted using TCP objects or CONGA, shared code files cannot be shared between different platforms or versions of Dyalog.

RESTRICTION 4: *Shared code files are not workspaces*

A shared code file is not a workspace; it is not possible to `]CY` or `)COPY` from them.

RESTRICTION 4: *64-bit Unicode only*

The benefits of memory mapping are only realisable in 64-bit address spaces. This, combined with the other fundamental limitations, means that shared code files are only supported for 64-bit Unicode systems; there are no current plans to extend this support.

2.3 Temporary Limitations

There are several restrictions when using shared code files that could be removed in future Dyalog versions.

RESTRICTION 1: *All objects saved must be visible from the root of the current workspace*

The following cannot be saved in a shared code file:

- GUI namespaces and their derivatives
- Functions created by starting an auxiliary processor
- External functions created using name association (⎕NA)
- ⎕SM

RESTRICTION 2: *It is not possible to have more than 8 shared code files simultaneously attached*

A maximum of 8 virtual memory addresses are available for shared code files; these *slots* have identifiers 1 to 8. In a future release this limit could be significantly increased, but the fundamental issue of needing a separate slot for each shared code file simultaneously will remain.

RESTRICTION 3: *Cannot)SAVE or ⎕SAVE a workspace that has shared code files attached*

It is not possible to)SAVE or ⎕SAVE the current workspace if any shared code files are attached (they must be assimilated or detached first – see *Section 4.3* and *Section 4.4* respectively).

RESTRICTION 4: *Attaching a shared code file containing namespaces copies all the namespaces (functions and arrays remain in the shared space)*

Attaching shared code files results in data being copied from the shared code files as needed (see *Section 4.4*). However, namespaces are always copied.

RESTRICTION 5: *Only certain content can be saved in a shared code files*

The content of a shared code file is limited to namespaces, nested arrays, simple arrays, tradfns, tradops, dfns, dops and derived functions (futures and external variables are instantiated and become arrays). If other content (for example, .NET objects, shared variables and COM objects) is present in a workspace then that workspace cannot be saved as a shared code file (see *Section 4.1*).

2.4 Summary of Limitations

Fundamental limitations:

- Shared code files are read only
- Shared code files have a fixed virtual memory address
- Shared code files cannot be shared across architectures
- Shared code files are not workspaces
- 64-bit Unicode only

Temporary limitations:

- All objects saved must be visible from the root of the current workspace
- It is not possible to have more than 8 shared code files simultaneously attached
- Cannot `)SAVE` or `]SAVE` a workspace that has shared code files attached
- Attaching a shared code file containing namespaces copies all the namespaces (functions and arrays remain in the shared space)
- Only certain content can be saved in a shared code file

3 Performance

The main purpose of shared code files is to reduce the execution time and memory consumption of APL applications.

3.1 Loading

Unless an application uses a large proportion of its constituent code soon after start-up, it is significantly faster to start that application in a nearly empty workspace and attach shared code files containing the rest of the code. This also reduces the memory footprint of the application.

The performance improvement is most noticeable in an environment where several processes run the same application on a single machine, for example, applications using isolates and/or running on Citrix servers or other servers. This is because:

- shared code files are memory-mapped; once one process has caused a part of the application to be paged in, subsequent processes have very fast access to the same part of that application.
- as the memory-mapped files are shared, only a small part of a function needs to be copied into each active workspace that shares them; this reduces the overall memory usage across all processes.

3.2 Code Execution

Code or data that is located in a shared code file is paged into virtual memory the first time that it is used. This incurs a performance overhead; however, subsequent calls to that code or data (or anything else on the same page) by the same or any other process do not experience the same performance impact.

Similarly, the first time that the content of a name (function, operator or variable) in a shared code file is amended also involves a performance overhead (the content of a shared code files is read-only; modifying the content of a name causes it to be copied into the active workspace). However, subsequent writes to that the content of that name by the same process do not experience the same performance impact.

Not only do subsequent calls/writes not experience the same performance impact, their performance is often improved when compared with performing the same operations without shared code files. This is due to the workspace memory manager running more efficiently when it has a smaller set of data in the main workspace than if everything was in the main workspace. Specifically:

- more workspace is available for application data, making it easier for memory manager algorithms to allocate memory.
- the contents of the shared code files are ignored by compaction and garbage collection algorithms.

4 Technical Reference

The operations that comprise the shared code file mechanism are implemented using three I-Beams – `8659⌶`, `8666⌶` and `8667⌶`. Specifically:

- `[names](8667⌶){slot}{file}`
  save shared code file – see *Section 4.1*
- `[nameclasses](8666⌶){file}`
  attach shared code files – see *Section 4.2*
- `(8666⌶)⍬NULL`
  assimilate shared code files – see *Section 4.3*
- `(8666⌶)0⍴⊂''`
  detach shared code files – see *Section 4.4*
- `R←(8659⌶)⍬`
  list shared code files – see *Section 4.5*
- `R←{slot}(8659⌶){ncs}`
  list attached names – see *Section 4.6*

4.1 Save Shared Code File

Purpose: Saves a shared code file.

Syntax: `[names] (8667⌶) {slot} {file}`

where:

- `names` is a vector of character vectors or a matrix specifying the names to save; this list of names of functions, operators and variables restricts the names in the shared code file that are saved.
- `slot` is the slot identifier (an integer in the range 1-8) for the unique fixed virtual memory address at which to load the shared code file.
- `file` is a character vector of the filename for the shared code file. If a filename extension is not provided, then **.dwx** is used. If a file of this name already exists, or the file cannot be created for any reason, then the operation will fail.

A multi-user development team might need a strategy for creating (and attaching) cycles of shared code files as shared code files could remain in use for some time by members of the development team. This should not be an issue with distributed applications.

This creates a shared code file, optionally based on a list of names of functions, operators or variables. Restrictions apply to the location and structure of objects that can be placed into a shared code file; most importantly, the names must all be visible in the root (#) of the active workspace. For a complete list of restrictions, see *Section 2.4*.

4.2 Attach Shared Code File

Purpose: Attaches one or more shared code files to the active workspace.

Syntax: `[nameclasses] (8666⌶) {file}`

where:

- `nameclasses` is a list of nameclass identifiers to be brought over (cannot include sub-classes). The default is `2 3 4 9` (variables, functions, operators and namespaces respectively).
- `file` is a vector of character vectors of shared code files to load, or a single character vector (a character scalar is not acceptable). If filename extensions are not provided, then **.dwx** is used. The path can be absolute or relative to current location; there is no sensitivity to WSPATH.

The effect of attaching shared code files is analogous to performing a `)PCOPY` (protected copy) from the shared code files, that is:

- names that already have a definition are preserved unaltered; if the same name appears in more than one shared code file, then the files are searched in the specified order and the first occurrence of the name is used.
- names in attached files immediately affect the results of system functions that provide metadata, such as `⎕NL` or `⎕NC`.

If any shared code files are already attached, then they are detached from the active workspace before new shared code files are attached (see *Section 4.4*). Multiple shared code files cannot be attached using separate calls to `8666⌶`.

4.3 Assimilate Shared Code Files

Purpose: Copies referenced objects in the shared code files into the active workspace.

Syntax: `(8666⌶)⎕NULL`

When the right argument to `(8666⌶)` is `⎕NULL`, all referenced objects in the shared code files are copied into the active workspace. The active workspace then contains all the code and data that was visible to it when the shared code files were attached; it can then be saved and used independently of the shared code files. The shared code files that are attached to the active workspace are then disconnected from the active workspace.

Significant space might be required to assimilate all the code in the shared code files into the active workspace. If a WSFULL error occurs then the operation will fail; it cannot be rolled back and leaves the workspace in an indeterminate (but consistent) state. In this situation, the shared code files are not disconnected from the active workspace as doing so could result in further errors.

Only things that are the current referent copied. The process is driven from the data not the name; this means that if multiple shared code files that include the same names are attached, then only the first of these names is external and that is the one that gets copied.

4.4 Detach Shared Code Files

Purpose: Detaches all shared code files from the active workspace.

Syntax: `(8666⌶)0⍴⊂''`

When the right argument to `(8666⌶)` is `0⍴⊂''` (that is, a zero-length list of names), any existing attached shared code files are detached.

Detaching shared code files results in data being copied from the shared code files as needed. However, namespaces are always copied when a shared code file is first attached.

Before a shared code file is disconnected from the active workspace:

- if a name that was brought into the active workspace when the shared code file was attached has not had its associated code/data changed, then the name is expunged from the active workspace.

- if a name in the active workspace embeds references to objects residing in a shared code file, then the entire definitions of the referenced objects are copied (assimilated) into the active workspace. This includes (for example), tacit functions that are derived from functions in a shared code file and arrays that contain references to data in a shared code file. These objects must still be functional following the disconnect.

As shared code files are read-only, they cannot be updated while they are in use. Instead, if a shared code file needs to be updated, it must be rebuilt. When a new version of a shared code file becomes available, anyone using the old version should detach it and attach the new one instead as soon as is practical.

4.5 List Shared Code Files

Purpose: Lists the shared code files that are attached to the current workspace.

Syntax: `R←(8659⌶)⍬`

where:

- `R` is a 2-column matrix listing the shared code files that are attached to the current workspace:
  - `[;1]` is the slot identifier for the fixed virtual memory address of the shared code file.
  - `[;2]` is the name (including the filename extension) of the shared code file that was loaded.

The rows of the matrix (one row for each shared code file) are ordered to correspond to the order in which the shared code files were specified when they were originally attached, that is, in the right argument to `8666⌶` (see *Section 4.2*).

4.6 List Attached Names

Purpose: Lists the names in the shared code file identified by the specified memory address.

Syntax: `R←{slot}(8659⌶){ncs}`

where:

- `slot` is the slot identifier (an integer in the range 1-8) for the unique fixed virtual memory address of the shared code file.
- `ncs` is an integer vector that would be a valid right argument to ⎕NL; it identifies the nameclasses and subclasses for which the names should be listed.

R lists the names in the shared code file identified by slot. If any element of `ncs` is negative, then positive values in `ncs` are treated as if they were negative and R is a vector of character vectors. Otherwise, R is a simple character matrix.

5 Technical Details

This section contains discussions intended to clarify the functionality of shared code file support.

5.1 Shared Code Files are Read Only

A shared code file is a read-only repository. Items within it can be modified, but doing so can result in data being copied into the main workspace.

Consider these cases where item A is modified:

1. `A` is a function
   - `B←A` will introduce a new name `B` into the main workspace but no new data.
   - When `A` is edited or otherwise re-fixed, the new version will be stored in the main workspace.
2. `A` is a simple array such as `1 2 3 4`.
   - `B←A` will introduce a new name `B` into the main workspace but no new data.
   - `C←A,1` will introduce a new name `C` and new data into the main workspace.
   - `A,←1` will create new data in the main workspace.
3. `A` is a nested array such as `'AB' 'CD'`.
   - `B←A` will introduce a new name B into the main workspace but no new data.
   - `A[1]←⊂'XY'` will introduce some new data into the main workspace.

In each of these cases, the content of the attached shared code file remains unaltered. This means that, if names of items in a shared code file are expunged using `⎕EX` and the shared code file(s) are detached and reattached, then the items in the shared code file will be restored to their original values. The only way to change the values in a shared code file is to recreate the entire file.

Although a shared code files can contain data, these values should either be constants or initial values for structures that will be copied into the workspace as soon as the application modifies them.

5.2 Attaching, Assimilating and Detaching Shared Code Files

When one or more shared code files is attached, the following rules apply:

- When items with the same name exist in multiple workspaces, the one that is used in the active workspace is the first one found when going through the workspaces in the following order:
  1. the active workspace
  2. the shared code file specified first when attaching (see *Section 4.2*)
  3. the shared code file specified second when attaching, etc.
- When the shared code files are assimilated:
  - all references to each shared code file are resolved by copying data from the shared code file to the active workspace as required.
- When the shared code files are detached:
  - names in the active workspace that reference data in a shared code file are deleted (namespace references are not deleted).
  - all remaining references to the shared code file are resolved by copying data from the shared code file to the active workspace as required.

EXAMPLE

The active workspace MAIN is populated using the following assignments:

```
FN1 ← {⍵ × 1}
FN2 ← {⍵ × 2}
NS1 ← ⎕NS ''
NS1.A ← 1
```

Name	Parent	Value
FN1	#	{⍵ × 1}
FN2	#	{⍵ × 2}
NS1	#	Namespace ref
A	NS1	1

Shared code files DWS1 is populated using the following assignments:

```
FN1 ← {⍵ × 1.1}
FN3 ← {⍵ × 3}
V ← 'AB' 'CD'
NS1 ← ⎕NS ''
NS1.A ← 2
NS1.B ← 3
```

Name	Parent	Value
FN1	#	{⍵ × 1.1}
FN3	#	{⍵ × 3}
V	#	'AB' 'CD'
NS1	#	Namespace ref
A	NS1	2
B	NS1	3

Shared code files DWS2 is populated using the following assignments:

```
FN3 ← {⍵ × 3.1}
FN4 ← {⍵ × 4}
NS2 ← ⎕NS ''
NS2.A ← 4
NS3 ← ⎕NS ''
NS3.A ← 5
```

Name	Parent	Value
FN3	#	{⍵ × 3.1}
FN4	#	{⍵ × 4}
NS2	#	Namespace ref
A	NS2	4
NS3	#	Namespace ref
A	NS3	5

After attaching DWX1 and DWX2 (in that order) to MAIN the following will be accessible:

Name	Parent	Value	Location of Value	Notes
FN1	#	{⍵ × 1}	WS	FN1 in DWX1 is inaccessible
FN2	#	{⍵ × 2}	WS	
FN3	#	{⍵ × 3}	DWX1	FN3 in DWX2 in inaccessible
FN4	#	{⍵ × 4}	DWX2	
V	#	'AB' 'CD'	DWX1	
NS1	#	Namespace ref		
A	NS1	1	WS	NS1.A and NS1.B in DWX1 are inaccessible
NS2	#	Namespace ref		
A	NS2	4	DWX2	
NS3	#	Namespace ref		
A	NS3	5	DWX2	

Following these assignments:

```
FN3 ← {⍵ × 3.2}
FN5 ← FN4
V[1] ← ⊂'XY'
NS2.B ← 6
```

The following are now accessible:

Name	Parent	Value	Location of Value	Notes
FN1	#	{⍵ × 1}	WS	
FN2	#	{⍵ × 2}	WS	
FN3	#	{⍵ × 3.2}	WS	Updated value
FN4	#	{⍵ × 4}	DWX2	

Name	Parent	Value	Location of Value	Notes
FN5	#	{ω × 4}	DWX2	
V	#	'XY' 'CD'	Split between WS and DWX1	
NS1	#	Namespace ref		
A	NS1	1	WS	
NS2	#	Namespace ref		
A	NS2	4	DWX2	
B	NS2	6	WS	New value
NS3	#	Namespace ref		
A	NS3	5	DWX2	

The shared code files are now disconnected. This is achieved either by assimilating them into the active workspace or by detaching them; the result of each of these operations is shown below.

Following assimilation of the shared code files, the main workspace will contain:

Name	Parent	Value	Notes
FN1	#	{ω × 1}	
FN2	#	{ω × 2}	
FN3	#	{ω × 3.2}	
FN4	#	{ω × 4}	Copied into WS
FN5	#	{ω × 4}	Copied into WS
V	#	'XY' 'CD'	Partially copied into WS
NS1	#	Namespace ref	
A	NS1	1	
NS2	#	Namespace ref	

Name	Parent	Value	Notes
A	NS2	4	Copied into WS
B	NS2	6	
NS3	#	Namespace ref	
A	NS3	5	Copied into WS

Alternatively, following detachment of the shared code files, the main workspace will contain the following values:

Name	Parent	Value	Notes
FN1	#	{⍵ × 1}	
FN2	#	{⍵ × 2}	
FN3	#	{⍵ × 3.2}	
FN5	#	{⍵ × 4}	Copied into WS
V	#	'XY' 'CD'	Partially copied into WS
NS1	#	Namespace ref	
A	NS1	1	
NS2	#	Namespace ref	
A	NS2	4	Copied into WS, namespace has changed
B	NS2	5	
NS3	#	Namespace ref	All namespaces in shared code files are retained (see *Section 2.3*)
A	NS3	5	Copied into WS

A Worked Example

This appendix comprises an annotated example that demonstrates the use of some of the cases of the I-Beam functions described in *Chapter 4* (examples of *assimilate* and *detach* are not included).

First, load the dfns workspace:

```
      )LOAD dfns
C:\...\ws\dfns.dws saved Sun Apr 12 17:18:38 2015

An assortment of D Functions and Operators.

      tree #                A Workspace map.
      ↑¯10↑↓attrib ⎕nl 3 4  A What's new?
      ⍝notes find 'Word'    A Apropos "Word".
      ⎕ed'notes.contents'   A Workspace overview.
```

Now compute the size of all the functions, variables and namespaces in the workspace (approximately 6 MB):

```
      +/⎕SIZE ⎕NL ⍳10
5947936
```

Define a helper function called saveDWX to create a shared code file:

```
      saveDWX←8667⌶
```

Create a shared code file containing everything in the dfns workspace, mapped at virtual memory address 1:

```
      saveDWX 1 'dfns.dwx'
```

If this fails due to the file already existing, then erase it and try again:

```
      saveDWX 1 'dfns.dwx'
DOMAIN ERROR: Shared code file already exists
      saveDWX 1 'dfns.dwx'
     ^
      ⎕NDELETE 'dfns.dwx'
      saveDWX 1 'dfns.dwx'
```

Clear the workspace and define two new helper functions, attachDWX and listDWX:

```
      )CLEAR
clear ws

      attachDWX←8666⌶
      listDWX←8659⌶
```

Attach the shared code file to the active workspace and compute how much workspace was consumed in doing so:

```
      wa←⎕WA

      attachDWX 'dfns.dwx'

      ⎕WA-wa
¯64320
```

(rather than consuming space, 64 KB was released due to workspace reorganisation)

Check how many names are now visible in the workspace and call the `easter` function to find the date for Easter Sunday in 2015 (to prove that functions in the attached shared code file can be run successfully):

```
      ≢⎕NL ⍳10
273

      easter 2015
20150405
```

List the shared code files that are attached to the active workspace (the first column shows the slot identifier). Next, display the first 5 names made available by the shared code file in slot identifier 1:

```
      listDWX ⍬
1  dfns.dwx

      5↑1 listDWX ⍳10
Cholesky
NormRand
UndoRedo
X
_fk
```

Finally, verify that the result of ⎕NL and the names exposed by the shared code file are identical (the only difference should be the three names defined since the)CLEAR operation):

```
      (⎕NL ⍳10)≡1 listDWX ⍳10
0

      ⍴1 listDWX ⍳10
270 12

      ⍴⎕NL ⍳10
273 12

      (↓⎕NL ⍳10)~↓1 listDWX ⍳10
attachDWX     listDWX       wa
```

Index