DYALOG

The tool of thought for software solutions

# Parallel Language Features

## Version 16.0


Dyalog Limited

Minchens Court, Minchens Lane
Bramley, Hampshire
RG26 5BH
United Kingdom

tel: +44(0)1256 830030

fax: +44 (0)1256 830031
email: support@dyalog.com
http://www.dyalog.com




*Isolate version 1.0.1454*
*Dyalog Version 16.0*

*Revision: 20160627_160*

# Contents

# 1 Introduction

This document describes proposals for two new primitives and a new type of array, designed to allow APL developers to take advantage of using multiple processors to execute code in parallel:

- The function “isolate” (⍪) creates a special kind of namespace known as an *isolate* inside which all APL language statements are executed in parallel with the main APL process.
- The monadic operator “parallel” (⍦) derives a function that will execute the left operand in an empty isolate.
- A new type of array known as a “future” is a pointer to the future result of an expression currently being executed in an isolate. Futures can be passed around as function arguments and as items of nested arrays, but will “block” any expression which actually needs the value pointed to by the future.

Since Dyalog version 14.0, *futures* have been implemented as a new primitive array element type. However, the *isolate* function and the *parallel* operator are provided as models which are partly implemented in APL, provided in the distributed workspace `ws\isolate`.

Although the current implementation is "only a model", Dyalog Ltd. believes that the design is essentially final. The code has remained unchanged for the last couple of releases. However, at the moment, the performance of the model has been more than sufficient, and work on other features have taken priority over a fully “primitive” implementation of futures and isolates.

## 1.1 Release Notes

The following changes have been made to the future and isolate implementations since version 14.0:

**New features included in Dyalog version 14.1, May 2015**

- Interrupts should now always interrupt application threads, rather than any of the threads implementing the isolate model.

- Errors in application code will no longer suspend isolate handling threads, even when “suspend threads on error” is enabled.

- *Microsoft Windows only:* Support for applications shipped as bound executables: If isolates are used from a bound executable containing the isolate namespace, the executable is re-launched as the host for isolate processes.

- The function `isolate.isSlave` returns `1` if the current process is an isolate “slave” process. This is useful when the same workspace (or bound executable) is used as the client application AND as the base for isolate processes.

- The workspace explorer no longer crashes when viewing isolates. This fix has also been back ported to version 14.0. A future version may allow the viewing of the contents of isolates, at present the explore will declare that there is no viewable content.

- The experimental feature described in the manual *Dyalog Experimental Functionality – Shared Code Files* allows for faster initialisation of isolates which need to make use of large code bases already in use by the parent process.

Features in the new isolate workspace released **September 2014**

- Under Microsoft Windows, the caption of suspended isolate processes is changed to help identify the relevant process (see 4.7 on debugging).
- The `StartServer` function now takes a list of rules which filter the network addresses of clients that are allowed to make use of the server (see section 6.4 on running Isolate Servers).

# 2 Philosophical Preamble

This section contains a discussion of issues surrounding the parallelisation of APL code. If you are looking to quickly get started with using futures and isolates, skip to section 3.

## 2.1 APL is a Parallel Notation

The APL language contains many language features which are potentially parallel, although most implementations do not execute them in parallel:

- Most primitive functions work on arrays, performing operations on all the elements of argument(s).
- Many classical primitive operators from APL or APL2 such as each (`f¨`), inner product (`f.g`), outer product (`∘.g`), and some reductions (`f/`) and scans (`f\`), express repeated and potentially parallel execution of one or more functions.
- New operators added to Dyalog APL more recently, such as rank (`f⍤n`), key (`f⌸`), and even some uses of the power operator (`f⍣g`), also express potentially parallel operations.
- In Dyalog APL, the use of the dot to execute an expression within the scope of an array of objects or namespaces, expresses potentially parallel execution (`namespaces.foo`).

Dyalog Ltd. intends to give high priority to implementing actual parallel execution of APL language statements.

There are two main tracks to pursue:

- Automatic parallelisation, where the interpreter automatically infers that an APL language statement can be parallelised, and decides to execute code in parallel when the language processor believes this will improve the performance of an application.
- Explicit parallelisation, where the language provides new mechanisms that allow the programmer to declare that certain parts of the application could or should be executed in parallel.

## 2.2 Automatic Parallelisation

Given the number of parallel language constructs in the APL language, it might seem that automatic parallelisation should be easy to implement. In fact, parallel execution of a number of scalar dyadic functions on floating-point arguments was introduced in version 12.1 of Dyalog APL, and some user applications did realise noticeable speed-ups. However, automatic parallelisation of traditional APL expressions faces a number of significant challenges:

1) Most arguments are small: parallel execution has a set-up time to start parallel threads or processes, and a synchronization cost when the results computed by separate processes are coalesced into a single array result. In typical applications, the arguments to most primitive functions only have a very small number of elements. Any potential speed-up from parallel execution will be cancelled out by the setup cost.

2) Cost of memory access: although it is common for modern hardware to have multiple cores, the memory cannot fully support many cores reading or writing significant volumes of data simultaneously: a queue will form. Thus, operations which have a large number of

small operations on large quantities of data (such as adding or multiplying large arrays), cannot be parallelized efficiently on current hardware due to memory access costs.

3) Side-effects: when operator expressions are applied to user-defined functions, those functions can have side-effects. Many existing applications rely on the current order of execution of operator expressions and would fail or produce incorrect results if user-defined functions started executing in parallel.

Successful automatic parallelisation on existing hardware will require an APL language system which is able to coalesce multiple primitive APL operations into larger units which can be effectively executed in parallel – combined with an ability to detect that user-defined functions are free from side-effects (in other words, a "compiler").

Dyalog Ltd. is conducting and funding research into APL compilers and does expect to increase the amount of automatic parallelisation that Dyalog APL supports. However, in the short term, explicit parallelisation though futures and isolates promise to provide more "bang for the buck" for the application developer.

## 2.3 Explicit Parallelisation

As mentioned in the previous section, it can be difficult for the APL engine to determine whether parallelisation is both safe and worthwhile. However, in many cases the programmer knows where the potential for significant parallel execution lies in an application, and is able to identify sections of code that are free of side-effects (or have manageable side-effects) - and represent enough work for the setup and synchronization costs to make parallel execution worthwhile.

In the past, many languages have provided language extensions aimed at allowing parallel processing[1]. These have typically depended on the programmer to make use of explicit synchronization features such as semaphores, which allow threads to mutually exclude each other from interfering with each other as they modify shared data, notify each other of progress, and wait for one step to complete before a dependent task can be started. These features are notoriously difficult to use and often give rise to deadlocks and other difficult-to-detect defects in code, even when used carefully by quote trained professionals unquote.

The challenge is to come up with new language features which make it easy for APL users to introduce parallelism without reducing the readability of the code – and without making the application fragile and prone to synchronization of timing errors.

This has been the main design goal for futures and isolates.

[1] Including Dyalog APL, which provides a spawn operator (&), the ⎕TGET/⎕TPUT/⎕TSYNC family of system functions, and the :Hold … :EndHold control structure.

# 3 Introducing Futures and Isolates

*Futures* and *Isolates* are language extensions which are intended to provide the developer with ways to express the existence of potentially parallel sections of code in a *deterministic* fashion. Briefly:

- An *isolate* is a namespace which is semantically equivalent to a normal namespace, except that expressions which are executed inside an isolate can run in parallel (in a separate process) to the main application process[2]. The current proposal is to create isolates using a new primitive function ⍤ (isolate).

- A *future* is an array of unknown rank, shape and content, which is returned as the result of any expression that is executed inside an isolate. Futures can be passed as arguments to user-defined functions or "nested" to form arrays containing several futures, without blocking (waiting for the isolate to produce a result). Arrays containing futures can be subjected to structural transformations such as reshape or partitioned enclose, without blocking. Blocking only occurs when one or more futures are passed to a function which needs to know the actual value (for example, a mathematical primitive function).

Between them, futures and isolates make it straightforward to divide application code into sections which can run in parallel: if one function requires the result of code which is still executing in parallel, the dependent function will automatically wait until the data that it needs is available.

## 3.1 Isolates

An isolate created using ⍤ is very similar to a namespace created using ⎕NS. The contents of an isolate can be referenced using the same dot notation that is used to refer to the contents of a regular namespace. It is also possible for code running inside an isolate to refer back to the parent space using the ## (Parent) symbol. However, isolates reside in separate processes and do not share a common workspace with the main process, and this does introduce some differences.

The most important differences between isolates and namespaces are the features that make isolates a tool for simple implementation of parallel execution within an application:

- Any evaluation which occurs within an isolate is handled in a separate process: If more than one isolate exists, execution can proceed independently within each isolate. If more than one processor is available, expressions inside isolates will run in parallel with the main process.

- An expression executed within an isolate immediately returns a *future*, without waiting for the expression to finish execution. A future is a placeholder for an as yet un-computed

[2] This is different from the threading that results from the use of the & (spawn) operator: These threads all run inside a single APL process, and only one thread is actually running at any given time.

value (see section 3.2 for more information). The future automatically turns into a real value when the expression produces a result.
- Errors and interrupts are treated differently: In particular, errors are not reported to the calling process until the value is referenced. Note that this means that, if values are never referenced, errors in the code that (would have) produced them will go un-detected.

A number of differences are due to the fact that the isolate is actually inside a separate process. Although the contents – both code and data – are "only a dot away", there are some restrictions on making references between isolates (or they would not deserve the name).

- Because the isolate is running in a separate process, it does not share file ties or any process-related handles or resources with its parent process[3].

- Inside an isolate, the special symbol **##** (aka "Parent") refers to the root (**#**) of the parent process - and not, as one might expect, the space inside which the isolate was created (as would be the case for a normal namespace).

- Code running inside an isolate can ***only*** refer to spaces that are not contained within itself ***via*** the root of the parent workspace (referenced using **##**, as mentioned above). You cannot pass a reference to something contained within one isolate to another isolate.

A handful of differences are due to the current implementation. These may disappear or change as we collect feedback and the implementation improves and eventually becomes completely integrated with the interpreter. Examples of these differences include:

- Expressions which produce a shy result when executed in a namespace will no longer be shy when executed in an isolate, because a future was returned and the shy nature of a result cannot presently be preserved by futures. For example: `nsref.(X←42).`

For about the differences between isolates and namespaces, see section 7.

## 3.2 Futures

A future is an array of unknown rank, shape and content, which is returned as the result of any expression that is executed inside an isolate. Futures have the following characteristics:

- A future can be assigned to a variable, passed as the argument to a function or operator, stranded together with other elements to form an array, or be inserted into a nested array, while remaining a future (with unknown content).

- Arrays *containing* nested (enclosed) futures can be passed as arguments to user-defined functions or operators, or subjected to primitive structural functions such as reshape or compress, so long as no function which needs to know the shape or content of the future is encountered.

- If a primitive or system function which needs to reference the contents of the future is encountered, an attempt is made to display the future in the session, or pass it to an external program such as a Microsoft.NET method or a shared library function, the

[3] In fact, as we shall see later, an isolate *could* be running on a completely different machine, which might not even be running the same operating system or be able to see the same file system as the parent process.

current thread will suspend until the expression which produced the future either produces the result, or the expression fails.

In other words, future results of expressions that are currently being executed inside one or more isolates (which are running in separate processes) can be passed around an application until the actual values are needed for the next computational step, at which point the code (the thread) that needs it will automatically block until the value is available.

Together, isolates and futures are designed to make it straightforward to write applications which contain sections that can run in parallel. No semaphores or special mechanisms are required to synchronize the independent expressions: When a result that is being computed asynchronously is required, the consumer of the result will simply wait until the value becomes available.

## 3.3 Parallel Operator

Parallel ( ⫽ ) is a monadic operator which returns a derived function which will immediately return a future, and execute the operand function within an empty isolate; it provides a mechanism for parallel execution of a function without the overhead of explicitly creating an isolate using ⍠.

The parallel operator is typically combined with each (¨), in which case the operand function is invoked multiple times. For example, the elapsed time for the following expression should be slightly more than 3 seconds (without the parallel operator, it would be roughly 12 seconds):

```
      ⎕DL ⫽¨ 3 3 3 3
3.003 3.004 3.003 3.003
```

The temporary isolate is empty when the function starts running; the operand function may create global names for its own use, but they will be discarded together with the temporary isolate when execution completes.

# 4 The Version 14.*x* Implementation

The ability to deal with the new type of array known as a *future* is a primitive feature which was introduced with Dyalog APL version 14.0, built-in to all primitive functions. However, the mechanisms for the creation of *isolates* and the execution of expressions within them are currently modelled in APL with the help of an undocumented I-Beam.

The model can be found in the distributed workspace `ws\isolate`, in the form of a namespace called `isolate`, and a set of functions and operators that have been given names that are intended to be similar to future primitive expressions. The following functions and operators expose models of future primitive functions or operators:

| Syntax | Primitive Equivalent | Alternative Name | Description |
|---|---|---|---|
| `ø arg` | `¤ arg` | `isolate.New` | Creates a new isolate |
| `f II` | `f ‖` | `isolate.ll` | Parallel operator |
| `f IÏ` | `f ‖¨` | `isolate.llEach` | Equivalent of Parallel Each |
| `f o_II` | `∘.(f‖)` | `isolate.llOuter` | Parallel outer product |
| `f IIö ranks` | `(f‖)⍤ranks` | `isolate.llRank` | Parallel rank operator[4] |
| `f IIÐ` | `(f‖)⌸` | `isolate.llKey` | Parallel key operator (see rank footnote) |

In the rest of this document, most parallel examples will make use of the above shortcuts: In particular, `ø` will be used in place of future `¤` primitive and `IÏ` in place of `// ¨`. Note that alternative, more traditional names are available for the names exposed in the root namespace.

The `isolate` namespace also exposes a set of functions which can be used to set configuration options, query the state of the isolate framework, start and use "remote" isolate servers. These functions will be introduced in different sections of this document, and are listed in Appendix A.

## 4.1 Parallel Each

A trivial example, which illustrates the potential of isolates very well, is the behaviour of parallel invocations of the system function `⎕DL`, which consumes almost no system resources, but takes a fixed amount of elapsed time. In "normal" Dyalog APL:

```
      T←⎕AI ⋄ ⎕←⎕DL¨8⍴3 ⋄ ⎕←⎕AI-T
3.008 3.004 3.005 3.004 3.007 3.009 3.007 3.007
0 7125 24054 0
```

[4] In the current model, to avoid blocking in order to "mix" the results into the complete result matrix, `IIö` (rank) and `IIÐ` return nested arrays containing futures, which need a final "mix" to produce the same result that the primitive operator would have returned. See Appendix A for details.

The second line of output shows that the elapsed time for this expression is very slightly more than 24 seconds, which should be no surprise: the each operator applies the ⎕DL function sequentially to each of the 8 elements in the right argument.

With futures and isolates, we are able to run the 8 delays in parallel using the operator named ⍝⍝, which represents the proposed primitive operator parallel used in conjunction with the each operator: ∥¨:

```
      T←⎕AI ⋄ ⎕DL ⍝⍝ 8⍴3 ⋄ ⎕AI-T
3.002 3.002 3.002 3.004 3.004 3.003 3.004 3.003
0 3000 3051 0
```

We can see from the timings that there is a slight overhead due to the cost of creating the isolates and communicating with them – a small number of milliseconds per function invocation.

Calls to ⎕DL parallelise exceptionally well, since they consume virtually no resources – but they are perhaps not THAT useful in practice. Let us examine what happens if we use a (very slightly) more interesting example. Consider the following function which loops a given number of times, does some fairly hard (if useless) work, and returns the number of elapsed milliseconds as its result:

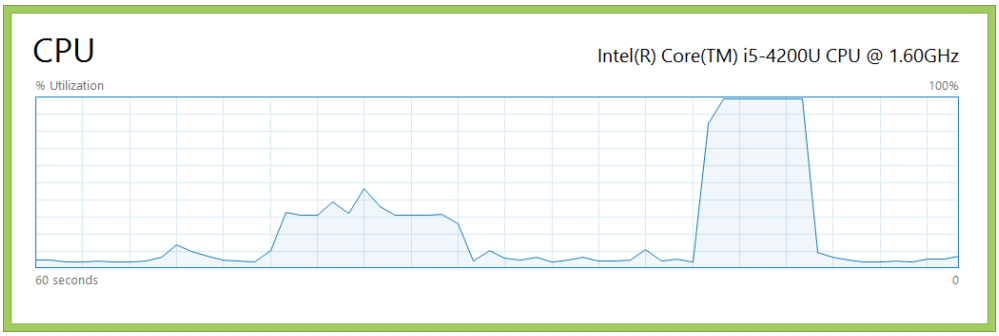
```
    ∇ r←loop n;work
[1]    r←3⊃⎕AI
[2]    :Repeat ⋄ n←n-1
[3]        work←≠∪10000?10000
[4]    :Until n≤0
[5]    r←(3⊃⎕AI)-r
     ∇
```

Unlike ⎕DL, this function consumes a large amount of CPU time – it is in a tight CPU loop. Let us repeat an experiment similar to the one above:

```
      T←⎕AI ⋄ loop¨ 4⍴1E4 ⋄ ⎕AI-T
2958 3036 3039 3002
0 12047 12040

      T←⎕AI ⋄ loop ⍝⍝ 4⍴1E4 ⋄ ⎕AI-T
6561 6621 6631 6611
0 5829 6664 0
```

The following Windows Task Manager screen shot shows the profiles of these two executions: The first run taking 12 seconds at about 30% reported utilization, the second 6 seconds at 100%.




We can see that the parallel invocations come fairly close to running twice as fast, completing the four calls to loop in 55% of the time (6.6 vs 12.0 seconds). This is not a bad result on the

machine where this was run, a fairly typical laptop of the current era – with a 1.6GHz Intel Core i5-4200. The machine is reported as having 4 logical processors, but there are only two independent cores.

## 4.2 Explicit or Persistent Isolates

When a function is invoked under one of the parallel operators, an empty isolate is created for each function invocation, and the isolate is destroyed when execution completes. If you need to create persistent state that needs to allow several function invocations to share the same code or data, it makes sense to create one or more isolates ahead of time, and use them to execute more than one expression.

Isolates are created using the function ⌀, which represents the future primitive function ⍤. The right argument can be a namespace to clone, or an empty vector to create an empty namespace. For example:

```
      ns←⎕NS ''          ⍝ Create an empty namespace
      ns.X←1 2 3 4       ⍝ Give the name X a value within the NS
      is1←⌀ ''           ⍝ Create an empty isolate
      is1.X←5 6 7 8 9    ⍝ Give X a different value in the isolate
      is2←⌀ ns           ⍝ Clone the namespace as another isolate
      isolates←is1 is2   ⍝ References to isolates can form an array
      isolates.(+/X)     ⍝ Execute (+/X) in 2 isolates in parallel
10 35
      T←⎕AI ⋄ isolates.(⎕DL 3) ⋄ ⎕AI-T
3.003 3.003
0 3000 3010 0
```

Since the dot notation allows us to execute any expression within an isolate, we can also easily transfer code to an isolate “on demand”. We can repeat the example with our loop function.

```
      isolates←isolates,⌀¨'' ''⍝ Add another 2 isolates (total is 4)
      isolates.⎕FX ⊂⎕CR 'loop' ⍝ Fix loop function in each isolate
 loop  loop  loop  loop
      T←⎕AI ⋄ isolates.loop 1E4 ⋄ ⎕AI-T
6714 6783 6801 6788
0 5890 6814 0
```

Performance note: calls to existing isolates are generally very slightly faster, since no isolate needs to be created first. The above example happened to take a little longer runtime than the earlier example using `I⍤`, but this will have been due to “other things going on” in the background.

## 4.3 Blocking

An expression executed within an isolate *immediately* returns a future, while the expression continues to execute in the background. Arrays containing futures can be manipulated using structural primitives, and passed as arguments to user-defined functions. Your application will only wait (or *block*) for the result if you encounter an expression which needs to know the actual *value* of the future.

This is hard to illustrate in a printed document, so it is recommended that you perform the following experiments in an APL session:

```
      delays←⎕DL I⍤ 3 10 20 ⍝ Returns immediately
      2↑delays              ⍝ Returns after about 10 seconds
3.002 10.006
```

```
      delays                ⍝ Waits until 20 seconds have passed
3.002 10.006 20.021
```

As can be seen in the preceding example, each item of the resulting array is a completely separate future, and blocking only occurs if you reference an item which has not yet been produced.

In section 4.5, we will learn how to track the execution and know which values are now available, in case we need to write applications which are guaranteed not to block (this might be a requirement for an application which provides services to others and needs to remain responsive at all times).

## 4.4 Errors in Asynchronous Expressions

If an asynchronous expression encounters an error, this is not revealed until an attempt is made to use the value. In the following slightly convoluted example, we invoke a function three times: the first function call will succeed, the second will fail with a `DOMAIN ERROR` after approximately ten seconds, and the third with a `LENGTH ERROR` after about 30 seconds:

```
      reciprocals←5 10 30 {z←⎕DL ⍺ ⋄ 1 2÷⍵} ⍤¨ 2 0 (3 4 5)
      ⎕DL 15           ⍝ Wait for more than ten seconds

      reciprocals
FUTURE ERROR: 11: DOMAIN ERROR: {z←⎕DL ⍺ ⋄ ÷⍵}
      reciprocals
                ^

      reciprocals[1] ⍝ This is fine
0.5 1

      reciprocals[2] ⍝ This failed after 10 seconds
FUTURE ERROR: 11: DOMAIN ERROR: {z←⎕DL ⍺ ⋄ ÷⍵}
      reciprocals[2]
                  ^

      reciprocals[3] ⍝ Fails after 30 seconds
FUTURE ERROR: 5: LENGTH ERROR: {z←⎕DL ⍺ ⋄ 1 2÷⍵}
      reciprocals[3]
                  ^

      ⍪⎕DMX.(EN EM Message)
```

| |
|---|
| 86 |
| FUTURE ERROR |
| 5: LENGTH ERROR: {z←⎕DL ⍺ ⋄ 1 2÷⍵} |

Important points:

1) Although the first error occurred after 10 seconds, while the main process was performing a (`⎕DL 15`), no error was reported at that time. The error is only when a reference is

made to a failed future. Note that this means that if you never reference the result, the error will never be reported[5].

2) If we are blocking on several futures, an error is reported as soon as the *first* one of them fails.

3) Errors are directly associated with the corresponding future: Note that the errors reported for item [2] and [3] are different (`DOMAIN ERROR` and `LENGTH ERROR`), and that item [1] can be extracted despite the fact that the array contains failed futures.

4) Like any item of an array, a future can be replaced – see below.

5) All failures in asynchronous expressions signal a `FUTURE ERROR`, event number 86. Information about the underlying error is provided in `⎕DMX.Message`.

**Beware:** The exact format of the error messages produced by failures of asynchronous expressions is quite likely to change as the implementation matures. It is recommended that you write applications in a way that allows you to easily adapt to changes in error reporting.

As mentioned above, a future can be overwritten at any time - it is just an item of an array. Thus, we can "repair" an array containing failed futures by replacing the failed future with any value we like:

```
      reciprocals[2 3]←⊂'[failed]' ⍝ Overwrite the failed future
      reciprocals
 0.5 1  [failed]  [failed]
```

If you overwrite a "live" future (where the isolate is still at work), the result of the asynchronous expression will simply be discarded when it eventually completes.

**Beware:** Even if you overwrite a "live" future, the isolate will keep running until it has finished the calculation that was launched. The isolate remains in a busy state, and will not be able to process another expression until the current one has finished executing. There is currently no way to stop an isolate from continuing execution of an asynchronous expression.

## 4.5 Tracking the Status of Asynchronous Expressions

It can be useful to know whether an array contains un-computed futures, in order to avoid blocking. In particular, any code which needs to remain responsive in order to provide services to others should try to avoid unexpected blocking. The isolate namespace provides four functions to help you monitor the status of futures without the risk of blocking. The right argument to each function must be the `name` of an array:

| Function | Return Value |
|---|---|
| `Values` | An array of future values with the same size as the named array, with unfulfilled futures replaced with the value given as the left argument. |
| `Available` | A Boolean array with 1 marking values which are available. |
| `Failed` | A Boolean array with 1 marking futures which have encountered errors. |
| `Running` | 1s identify futures where the isolate is still running. |

[5] If a tree falls in the forest, and nobody is there, does it make a sound? In this case, the answer is "no".

For example:

```
      z←99, ⎕DL ⍳3 6 9 ¯1
      ⎕DL 7 ⍝ Wait until an "interesting" point in time
      isolate.((¯1 Values'z'),(Available'z'),(Running'z'),⍪Failed'z')
99      1 0 0
 3.009  1 0 0
 6.015  1 0 0
¯1      0 1 0
¯1      0 0 1
```

Explanations:

`Values:` the first item (99) was never a future so the value is immediately available. 3.009 and 6.015 are the first two ⎕DL results. `¯1` is returned for "unfulfilled" futures.

`Available:` the first three values are available, the last two are not.

`Running:` the fourth item is a future which is still in process after 7 seconds (the `⎕DL 9`).

`Failed:` the last item of `z` is a failed future, because `¯1` is not a valid argument to ⎕DL.

## 4.6 Interrupting Asynchronous Expressions

In version 14.1, interrupts should always be issued to application threads that are waiting for future results, rather than interfere with the threads handling isolates. In both the two examples below, an interrupt issued while waiting under v14.1 will simply cause a return to the six-space prompt. The wait for the future is abandoned, and processing continues in the background. If a new attempt is made to reference the future value, it will either succeed or, if the value has not yet been produced, block again.

If similar expressions were used in a defined function, the wait would be cut short and an interrupt would be signalled on the following line of code, in the same way as when using ⎕DL.

**Version 14.0 Interrupt Handling**

This section has been retained for users of v14.0 who happen to be reading the v14.1 documentation – and just in case v14.1 interrupt handling doesn't always do what is intended.

In version 14.0, if you issue an interrupt, there will be a significant likelihood that it will be caught by one of the threads which is waiting for a future result. In this case, you will be asked to confirm whether you would like to continue waiting. In the first example below, we respond *Y*, resumed waiting and received the result. At the second prompt we answered *N*: in this case the future is treated in the same way as if the calculation had encountered an error.

```
      ⎕DL ⍳3 6
ISOLATE: Interrupt - continue waiting (Y/N)? Y
3.003 6.009

      ⎕←z←⎕DL ⍳3 6
ISOLATE: Interrupt - continue waiting (Y/N)? N
FUTURE ERROR: USER INTERRUPT 1002
      ⎕←z←⎕DL ⍳3 6
     ^

      isolate.Values 'z'
3.006  [Null]
```

The final result shows us that the first item of the result had been produced before we interrupted the wait.

Note that interrupting a wait for a future does NOT interrupt the code running within the isolate. The computation will continue and the result, when it is available, will be discarded. In the above example, the second element of `z` will remain a future forever. Since calls to isolates are processes sequentially, the next calculation will be held in a queue until the previous function call is completed.

In version 14.1, interrupts will not impact the processing of future results: The second element of `z` will eventually be filled in, if `z` still exists at that time.

## 4.7 Debugging Asynchronous Expressions

As we have seen, the default behaviour is to trap all errors occurring within expressions that are executed in an isolate, and return the error to the calling environment:

```
      ⎕FX 'R←A CALC B' 'R←A÷2×B'
      1 CALC 2
0.25
      1 CALC IÏ 2 3 0
FUTURE ERROR: 11: DOMAIN ERROR: CALC[1] R←A÷2×B
      1 CALC IÏ 2 3 0
                   ^
```

During development of code which is to run in isolates, trapping of errors can make debugging rather tedious. Fortunately, we can set the `onerror` option to `debug`:

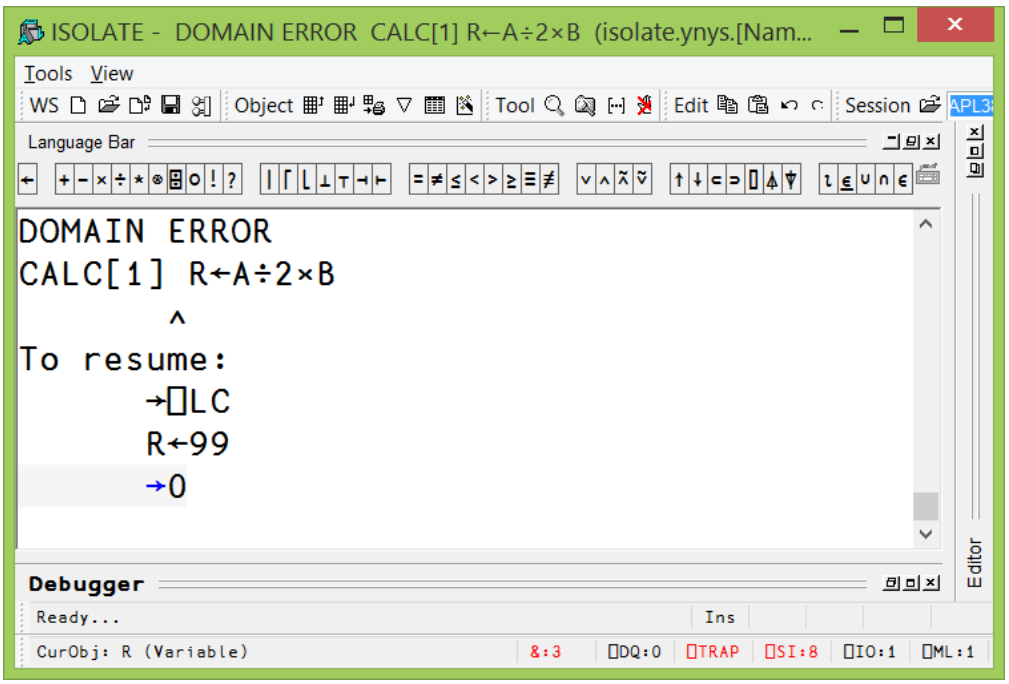
```
      isolate.Config 'onerror' 'debug' ⍝ Returns previous value
signal

      isolate.Reset 0 ⍝ All processes must be restarted[6]
Reset: 2 isolates, 6 processes

      1 CALC IÏ 2 3 0
```

… at this point the APL session will hang indefinitely, due to one of the isolate processes being suspended with an error. Selecting `debug` overrides the runtime option, forcing the use of the full development environment to host isolate processes. The session caption for failed sessions should have been changed to make it easy to identify which session to select. This is a completely normal Dyalog APL development session: if you wish to resume normal operations after you have corrected the error, you must take care not to cut the stack back past your function (or if you do, try to carefully trace back into it). Ideally, you should fix your problem and continue execution of your function in such a way that it returns a result. If everything comes unstuck, return to the main session and call (`isolate.Reset 0`) again; this will terminate all the processes.

[6] A reset is necessary after most configuration changes, because isolate processes need to be restarted in order to pick up the new configuration.




We can return to the client session and see that the result has arrived:

```
0.25 0.1666666667 99
```

Of course, you should ensure that any code changes you make to your own application code are saved in your source code repository before you return control to the isolate framework.

# 5 Calling Back from Isolates

In all the examples we have seen so far, we have issued calls from the main process into isolates – either to persistent isolates created using `⌀`, or to temporary isolates created using one of the operators like `I⍤`.

## 5.1 Calling Back to the Parent

Sometimes, there are resources which it is impractical to copy to into each isolate, either because they change frequently and all isolates need to share the values in real time, or because the cost of replicating the resource make it impractical. For example, if the main process has an open database connection, there might be good reasons for isolates to call back and share that connection (or current transaction), rather than making a new database connection from each isolate.

For code running in an isolate, the special `##` (parent) is a reference to the root namespace of the initiating process, and any reference to the contents of `##` represent a call back to the main process. In the APL model, call-backs are not enabled by default, because there is significant overhead in establishing the additional communications channel that is required.

The `listen` property is used to enable callbacks:

```
      isolate.Config 'listen' 1 ⍝ Enable callbacks
0
      isolate.Reset 0 ⍝ Restart all services after config change
0 isolates, 4 processes reset
```

Imagine that we have a shared counter that all isolates need to increment from time to time:

```
      COUNTER←1
      {##.COUNTER←##.COUNTER+⍵} I⍤ 1 1 1 1
2 2 2 2
      COUNTER
2
```

Our intention was that each parallel function would have incremented the counter once, but something went wrong. The reason is that each expression makes two references to `##.COUNTER` – first to read it and then to set it, after adding 1. When the example runs, it is almost guaranteed that all four isolate processes will manage to make the read request before any one of them manages to perform an update. Therefore, each isolate sees a value of 1, adds one, and all four of them set `COUNTER` to 2. We need a way to ensure that each function manages to read AND update `COUNTER` as an atomic operation, without giving the other isolates a chance to interfere.

It is not possible to use tokens (`⎕TPUT` / `⎕TGET`), or `:Hold` control structures to achieve this, because the isolates are separate processes and do not share token pools or any other state with each other.

The simplest way to implement this kind of *transaction* is to use a server-side function, similar to a *stored procedure* in most SQL database servers. Call-backs to the main process are guaranteed to run sequentially. The same is true if several threads in the main process make

calls to the same isolate: if there are simultaneous calls they will be queued and executed one by one, to avoid race conditions.

Let us define a function in the root namespace to modify COUNTER:

```
     ∇ R←INCR X
[1]    COUNTER←COUNTER+X
[2]    R←COUNTER
     ∇
```

Now, we can make atomic updates:

```
      COUNTER←1
      {##.INCR ⍵} ⍤1 ⍳4
2 5 3 4
      COUNTER
5
```

Note that the result illustrates that the four call-backs were not run in order from left to right: the order of execution of isolates is not guaranteed – they run completely independently. The important thing is that COUNTER was incremented four times – as expected.

## 5.2 Call-backs versus Interrupts

**Beware:** In version 14.0, Dyalog APL does not provide a way to select which thread is interrupted. If you have enabled call-backs using the "listen" option (see section 5.1), and you interrupt execution, there is a risk that you will interrupt a call-back, or terminate the listening thread while it is waiting for the next call-back. In version 14.1 the listening thread is immune to weak interrupts, but if you issue a strong interrupt, you will also run the risk of interrupting the listener.

If you unintentionally interrupt the listener, you will see warnings displayed in the session, similar to the ones coloured blue in the following. The first one is a warning from the listener that it is shutting down, and the second message tells us that the listener is being restarted:

```
      is.⎕DL 6
Weak interrupt received, RPC Server shutting down
6.011

      DIVISOR←10
      {⍵÷##.DIVISOR} ⍤1 ⍳3
ISOLATE: Callback server restarted
0.1 0.2 0.3
```

We can see from the above that the interrupt failed to stop the first function call, but instead it caused the listener to shut down. If you issue weak interrupts, you should never see this message in version 14.1.

## 5.3 Calling From One Isolate to Another

The full primitive implementation may allow calls to be chained, so that one isolate can call another by going via the parent process, as in:

```
      is1.(##.is2.(⎕DL 1))
```

However, this is ***not*** currently supported, and it is not clear that the APL-coded model will ever be able to support it. A primitive implementation may eventually also allow an isolate to create child isolates and make calls to them, but this is also unsupported in the model.

Note that, even when calls between isolates become enabled, it will probably not be possible for one isolate to contain a reference to another – references cannot cross isolate boundaries.

# 6 Isolate Processes

By default, the framework will create one process for each processor that is available on your machine. As isolates are created, they will initially be allocated to free processes. If no free processes are available, each new isolate will be allocated to the process which is managing the smallest number of isolates. If several processes have the same number of isolates and some of them contain "busy" isolates, the least busy process will be selected to host the new isolate.

## 6.1 isolate.State

The function `isolate.State` can be used to monitor isolate allocation and activity:

```
      isolates←ø¨6ρc''
      isolate.State ''
Host       Port  Isolates  Busy
----       ----  --------  ----
localhost  7052          2     0
           7053          2     0
           7054          1     0
           7055          1     0
```

We can see from the above that we have 4 processes available, each listening on a separate port, and that the two first ones have been allocated two isolates each. The "Busy" column shows how many isolates are currently running code:

```
      z←⎕DL IÏ ι6 ⋄ ⎕DL 3
      isolate.State ''
Host       Port  Isolates  Busy
----       ----  --------  ----
localhost  7052          2     0
           7053          3     1
           7054          2     1
           7055          2     1
```

We created another six isolates for a total of 12, so each processor had 3 each – but the temporary isolates are disappearing as each delay completes. After 3 seconds, 9 isolates remain, as documented above.

## 6.2 Limiting the number of Isolates per Process

By default, several isolates may share a single APL process. If you are running code with no side-effects, the only problem will be that two or more isolates share a single operating system thread and the workspace that is available within it – which may limit performance.

However, if your isolates use component files, shared libraries or other "per process" system resources, these will now be shared – not with the main process, but between "random" isolates that happen to share the same process. This may cause problems unless your code is written to deal with that situation.

If your code needs to run in a single process, you should set the `isolates` configuration option to 1. Obviously, imposing a limit creates a risk that you will run out of processes:

```
      isolate.Config 'isolates' 1
99
      isolates←ø¨4⍴⊂'' ⍝ OK to create four isolates

      fifth←ø ''
ISOLATE ERROR: All processes are in use
      fifth←ø ''
     ^
```

See the next section for information on how to increase the number of processes.

## 6.3 Changing the number of Processes

By default, the system detects the number of processes available, and sets the `processors` option to this value. The `processes` option, which controls the number of processes to start per processor, defaults to `1`. The default values are expected to give a reasonable utilisation of multi-core hardware for fairly CPU-intensive isolates. However, depending on whether your application is memory- or CPU-bound, or limited by file or network I/O, the "best" choice might be very different. It might make sense to run only one or two processes – or dozens – depending on what your isolates are going to be doing.

You can set these two options to any integer value – and then you must call `isolate.Reset` in order to get the new settings to take effect. Note that in the current version, every one of the (`processors×processes`) APL tasks are started as soon as the isolate framework is initialised or reset. A potential future enhancement would be to allow dynamic allocation.

```
      isolate.Config 'processors' 3
4
      isolate.Config 'processes' 2
1
      isolate.Reset 0 ⍝ Get new config to take effect
Reset: 6 isolates, 4 processes

      z←⎕DL ⍸⍳6 ◇ ⎕DL 2
      isolate.State ''
 Host       Port  Isolates  Busy
 ----       ----  --------  ----
 localhost  7052         0     0
            7053         0     0
            7054         0     0
            7055         1     1
            7056         1     1
            7057         1     1
```

## 6.4 Running Isolate Servers

By default, isolates are hosted in processes that are started on your local machine, according to the `processors` and `processes` options described in the previous section. If you have processing capacity available on other machines, you can load the `isolate` workspace on these machines and use the function `isolate.StartServer` to turn the current APL session into an isolate server which can be used to run slave processes.

The right argument to `StartServer` is a list of access rules, separated by commas, which control which client machines will be allowed to connect to the server. If the right argument is empty, the server will only be usable from clients on the same machine.

Each rule is of the form `'keyword=value'`. At the moment, the only keyword which is accepted is "`ip`". The value must be a full or partial IP address. If the address is a prefix of the address of a client, the connection will be accepted. For example, `'ip=192.168.1'` would allow all machines with an address starting with 192.168.1 to connect.

The number of processes started is sensitive to the same option settings as for local use:

```
      isolate.Config 'Processors' 8
4
      isolate.StartServer 'ip=192.168.1,ip=192.168.6'
  Machine Name:  MortensYoga
  IP Ports:      7052 7053 7054 7055 7056 7057 7058 7059

 Enter the following in the client session:

       #.isolate.AddServer 'MortensYoga' (7052-⎕IO-⍳8)

  Full IP address list:
   IPv4    192.168.193.1      IPv6    [fe80::545f:770f:dab:368b]
           192.168.247.1              [fe80::ccbd:aa34:3cfb:6e24]
```

You can use (`isolate.Reset 0`) to shut the server down again – or simply end the session.

## 6.5 Using an Isolate Server

As suggested in the output of the `StartServer` function, you need to call `AddServer` to make a remote server available as a host for new isolates. The argument is a 2-element vector containing the IP address of the server, and the range of port numbers that is exposing. For example:

```
      #.isolate.AddServer 'calcsrv' (7052 7053)
 Host         Port  Isolates  Busy
 ----         ----  --------  ----
 calcsrv      7052         0     0
              7053         0     0
```

If a server that you are using is shut down, or the connection to it is lost, errors can occur without warning if you continue using isolates hosted on a server which is no longer accessible. You may see failures within the isolate infrastructure, along the lines of:

```
      ⎕DL ⍳4
ISOLATE: Connection to calcsrv:7053 failed: 1111
ERR_CONNECT_DATA  /* Could not connect to host data port */
      ⎕DL ⍳4
          ^
```

The same kind of errors *may* occur if your local isolate servers are terminated for any reason, but this is much less likely than the loss of a network connection.

## 6.6 Combining Local and Remote Isolates

As we have seen: if you start creating isolates without calling `AddServer`, local processes are started. If you call `AddServer` after the first use of isolates, the new processes will be

added to those which are already accessible. You may also explicitly add local processes using `AddServer` with an empty vector as the right argument:

```
      #.isolate.AddServer ''
 Host         Port  Isolates  Busy
 ----         ----  --------  ----
 calcsrv      7052         0     0
              7053         0     0
 localhost    7052         0     0
              7053         0     0
```

In the above example, the `calcsrv` (which we added in the previous section) - and the local machine both have 2 processes. Both machines are using the same port numbers (on different IP addresses).

The function `RemoveServer` can be used to remove a server from your list of remote servers. Note that this function will allow you to remove processes which are hosting persistent isolates. If you subsequently attempt to use such isolates, an error will result:

```
      is←⌀ '' ⍝ Created on calcsrv
      #.isolate.RemoveServer 'calcsrv'
 Host         Port  Isolates  Busy
 ----         ----  --------  ----
 localhost    7052         0     0
              7053         0     0

      is.(2+2)
ISOLATE: No longer accessible
      is.(2+2)
     ^
```

# 7 Differences between Namespaces and Isolates

This chapter drills down on some of the less obvious differences between isolates and namespaces that may be encountered when converting existing code to make use of isolates.

## 7.1 File Ties are not shared with Isolates

Unlike namespaces, isolates reside in separate processes, so they do not share any resources that belong to the current process, such as file ties, loaded libraries, TCP sockets, etc. The following example illustrates the issue using component file ties:

```
      ns←⎕NS '' ⋄ is←⌀ ''  ⍝ A namespace and an isolate

      'file1' ⎕fstie 0     ⍝ Tie a file in main process
1
      'file2' ns.⎕fstie 0  ⍝ Tie a file "in" a namespace
2
      'file1' is.⎕fstie 0  ⍝ Tie first file in the isolate
1

      ns.(⎕fnames,[⍪]⎕fnums) ⍝ root and ns are in the same process
file1  1
file2  2

      is.(⎕fnames,[⍪]⎕fnums) ⍝ isolate is a separate process
file1  1
```

## 7.2 Namespace "Refs" and Isolates

In addition to the resources described in section 7.1, it is also not possible to share pointers such as namespace refs with an isolate. The process of passing the reference to an isolate, either through assignment or as an argument to a function, will cause the referenced space to be copied.

```
      ns←⎕NS '' ⋄ ns.Value←42 ⍝ Create namespace containing variable
      is←⌀ ''           ⍝ Create an isolate
      otherns←⎕NS ''    ⍝ Create a 2nd namespace
      otherns.ns←ns     ⍝ Within same process: create a ref to ns
      is.ns←ns          ⍝ Separate processes: ns is copied

      ns.Value←43       ⍝ Update value
      otherns.ns.Value  ⍝ Same process: shared value
43
      is.ns.Value       ⍝ Separate process: copied value
42
```

## 7.3 Classes versus Isolates

As mentioned in 7.2, namespace references cannot be shared, data will be serialised and copied when passed between isolates. Classes, Interfaces and Instances cannot be serialised, so they cannot even be copied:

```
      is.MyClass←MyClass
ISOLATE: Transmission failure: Object oriented namespace cannot
be serialised
```

You can fix the source of a class in an isolate using ⎕FIX, but beware: if you execute an expression that returns a class or an instance, this will fail because that result cannot be passed back:

```
FUTURE ERROR: 11: ISOLATE ERROR: Result cannot be returned from
isolate
      is.⎕FIX ⎕SRC MyClass
                          ^
```

The solution is to ensure that you return a result which is not a class, instance or interface. For example:

```
      is.{≠⎕SRC ⎕FIX ⍵}⎕SRC MyClass ⍝ Count lines of source
21
```

## 7.4 Unexpected output of Shy Results

Futures are not currently able to preserve the shy nature of results – any shy result which is returned as a future becomes non-shy:

```
      ns←⎕NS ''           ⍝ Create an empty isolate
      is←isolate.New ''   ⍝ Create an empty isolate
      ns.(X←10?10)        ⍝ Shy result in ns: output suppressed
      is.(X←10?10)        ⍝ With an isolate, this produces output
6 9 3 4 8 2 7 1 5 10
```

The parenthesised assignment within a namespace in the current process is shy and does not produce any output; when the same assignment is made within an isolate, a future is returned and the result of the assignment is displayed in the session.

## 7.5 Function calls into Isolates must return results

Due to the way that calls into isolates are implemented as remote procedure calls that return futures, any expression executed in an isolate MUST return a result, or a FUTURE ERROR will be signalled.

Note that when an error is signalled because there is no result, or because the result is a class, function or anything else that cannot be returned, the expression may well have been executed.

# 8 Option Settings

The isolate namespace has a number of configuration options which control how isolates are started and managed. Options are queried and set using the Config function, for example:

```
      isolate.Config 'workspace' 'c:\temp\myapp\myapp.dws'
c:\temp\myapp\myapp.dws
```

Most of the options will only take effect following a call to (`isolate.Reset 0`) in order to shut down all existing servers and re-initialise the system.

An empty argument to the `Config` function gives a list of all current settings:

```
      isolate.Config ''
drc                    #
homeport            7051
homeportmax         7151
isolates              99
listen                 0
maxws              64000
onerror           signal
processes              1
processors             4
protocol             IPv4
runtime                1
workspace        isolate
```

The current set of supported options is listed below.

| Option Name | Default | Description |
|---|---|---|
| `drc` | # | Location of CONGA namespace to use |
| `homeport` | 7051 | The lowest port number that will be used |
| `homeportmax` | 7151 | The highest port number to try listening on |
| `isolates` | 99 | Number or isolates allowed per process |
| `listen` | 0 | 1 to allow isolates to issue callbacks to parent process |
| `maxws` | '64000' | By default, uses the same setting as the current APL session |
| `onerror` | 'signal' | Signal errors to the line waiting for results |
| `processes` | 1 | The number of processes to start per process |
| `processors` | 4 | Number of processors (under Windows, determined automatically) |
| `runtime` | 1 | Whether to run isolates using the runtime engine |
| `workspace` | 'isolate' | Workspace to load when starting new isolates |

## 8.1 drc

This option can be used to specify a non-default location of the Conga DRC namespace, which must be present for isolates to work. If `drc` is `#` (the default) we check to see if `#.DRC` exists and copy DRC from the CONGA workspace (which must be on the workspace path or in the same folder as the APL executable) if it not. Otherwise `drc` must be a reference to an existing DRC namespace.

## 8.2 homeport and homeportmax

Each isolate - and if the `listen` option is set to 1 the main process as well – will open a listening TCP port in order to receive commands to be executed. These two options control the first and last port number that will be attempted used. The system will use the configured number for the main process listener, and the next (`processors×processes`) numbers for the isolates.

If the creation of the main process listener fails, the port number will be incremented by (`1+processors×processes`) and the listener creation will be retried – up to the upper limit set by `homeportmax`.

## 8.3 isolates

This parameter makes it possible to set a cap on the number of isolates that each available process is allowed to contain (within a process, separate APL threads will be used to execute code for different isolates). By default, the limit is 99 – but if you are running isolate code which needs to have complete control of process resources such as component files or shared libraries, you can set this value to 1.

## 8.4 listen

When this option is set to 1, a listening thread is launched in the main APL process, and the isolate will be able to call back into the root of the main workspace by referring to the symbol `##`. The default is 0 because there cost of creating isolates is significantly increased when an additional connection is required. The cost of using isolates, once they have been created, is not affected.

## 8.5 maxws

If you want to start processes using a different workspace size than your main process, you can set this option before processes are launched. The option value is a string which has to follow the same rules as the `maxws` environment variable or registry entry for APL.

## 8.6 onerror

This option controls the handling of errors that occur in expressions that are executed in isolates. Allowed values are:

`signal`: a FUTURE ERROR will be signalled in an expression which depends on the result of the expression.

`debug`: do not trap errors in the isolate: let the isolate server suspend. This will force the use of the full development environment for isolate processes, even if the `runtime` option is set to 1.

## 8.7 processors

The number of processors available for use for isolate servers (default determined automatically using `1111⌶⍬`).

## 8.8 processes

The number of processes that will be started per available processor (default 1).

## 8.9 workspace

This specifies the name of a workspace to be used at start-up by the processes which will host isolates. By default, it is set to "`isolate`", and the system will use the distributed workspace by that name. Note that, if you use the runtime interpreter, you must either use the full path to the workspace or place the workspace in the same folder as the Dyalog executable, or the runtime interpreter will not be able to find it.

See section 9.2 for details on how to create a workspace which contains your own application code, and use it as the host.

## 8.10 runtime

Controls whether the runtime interpreter or the full development system is used to run isolate server processes. The default is `1` (use the runtime), which gives faster startup of isolates and no visible "clutter" caused by additional sessions. This mechanism works by adding "rt" to the end of the name of the executable that is running the current process. If you are using a bound executable, it should always be set to 0.

Note: The development system will always be used if the `onerror` option is set to `debug`, regardless of the `runtime` setting.

# 9 Further Reading

## 9.1 Tools and Samples on GitHub

The Parallel folder in the Dyalog GitHub repository at http://github.com/Dyalog/Samples/ contains tools that can be used to extend the functionality provided by the isolate namespace, with code examples showing how to use the tools.

At the time that this document was written, the samples included:

`PEACH` – an enhanced implementation of Parallel Each which uses a fixed set of isolates and keeps them busy (rather than creating a new empty isolate for each item of the arguments), and displays an optional progress bar for the computation.

`IIPageStats` – an example which computes letter frequencies on a collection of newspaper web pages in parallel (using `PEACH`).

See the README.md files in the repository for more information.

## 9.2 Serving Isolates from Your own Application Workspace

If your isolate is going to need all the code that is in your application workspace, the most efficient way to start isolate processes may be to load that workspace. To set this up, you need copy the `isolate` namespace into your workspace, also either include the `DRC` namespace from the Conga workspace, or place the Conga workspace in a location where the expression (`'DRC' ⎕CY 'ws\conga'`) will succeed during start-up.

The latent expression (`⎕LX`) in your workspace should invoke the `isoStart` function before your application is started. In a slave process, the `isoStart` function will start serving calls, but in a normal process it will exit immediately and allow your application to start. For example:

```
⎕LX←'isolate.ynys.isoStart ⍬ ⋄ Run'
```

Note: If your application is shipped as a bound executable under Microsoft Windows, you ***must*** set the workspace property to an empty character vector (`''`), in order to avoid attempts to load the workspace.

The following example function is an application which can both run as a bound executable and a normal workspace.

```
     ∇ Run;iss;t;F;⎕TRAP
[1]    ⎕TRAP←(999 'C' '→ERR')(0 'E' '#.LOG') ⍝ Trap & report all
errors
[2]
[3]    →isolate.isSlave⍴0 ⍝ Exit if this is a slave process
[4]
[5]    {}isolate.Reset 0 ⍝    Nor any zombie isolates
[6]
[7]    :If ''≡⎕WSID
[8]        ⍝ Looks like we are a bound executable
[9]        {}isolate.Config'runtime' 0    ⍝ do not add "rt" to exe name
[10]       {}isolate.Config'workspace' '' ⍝ no workspace to load
[11]   :EndIf
[12]
[13]   iss←isolate.New¨''' '' '' ⍝ Make 3 isolates
[14]   t←iss.⎕DL 1 1 1           ⍝ Do 3 short delays
[15]   'F'⎕WC'Form'(⍕t)          ⍝ An ugly form with results in caption
[16]   'F.T'⎕WC'Text'(⍕,'.'⎕WG'APLVersion')(50 50) ⍝ Display version
[17]   ⎕DQ'F'
[18]   →0
[19]
[20]  ERR:⎕TRAP←0 'S'            ⍝ Trap no more errors
[21]   'F'⎕WC'Form' 'Error'      ⍝ Create a form
[22]   'F.T'⎕WC'Text'(⍕⍪#.DMSI)(50 50) ⍝ Put ⎕DM and ⎕SI in the body
[23]   ⎕DQ'F'
[24]   →0
     ∇
```

The companion function LOG which collects information about errors in the environment where they occur is defined as follows.

```
     ∇ LOG
[1]    #.DMSI←⎕DM,⎕SI
[2]    ⎕SIGNAL 999
     ∇
```

## 9.3 Topics Yet to Come

The following sample is under development, contact support@dyalog.com if you need it:

- How to run an isolate server as a Windows Service.

# Appendix A: Function and Operator Reference

## A.1 Models of Future Primitives

Each syntax block shows the model syntax on the left. On the right, in the first row the proposed primitive symbol and in the 2nd row the alternative name exposed by the isolate namespace:

```
isolate←ø init                                  r←¤⍵
                                        isolate.New ⍵
```

Creates an isolate. The right argument can take one of the following forms:

| Right argument | Meaning |
|---|---|
| `''` | Empty vector: empty isolate |
| `'wsname'` | A simple, non-empty character vector is interpreted as a workspace name, the workspace is copied to create the isolate |
| `'fn' 'var'` | A vector of names is used to create an isolate containing named objects from current workspace |
| `Namespace reference` | Creates clone of argument space |

The result is a reference to the isolate which has been created.

```
f II                                              f⫴
                                        isolate.ll
```

Model of (`f⫴`): A function derived from `f`, which will create an isolate empty except for a copy of the operand function, start execution of the function and immediately returns a future.

```
f IÏX                                            f⫴¨
                                    isolate.llEach
```

Model of (`f⫴¨`): A function derived from `f`, which will create one isolate containing a copy of `f` for each required invocation (using the same rules as for `f¨`) of `f`, start all the function calls, and return an array of futures of the same shape as would be returned by `f¨`.

```
f o_II                                          ∘.(f⫴)
                                    isolate.llOuter
```

Model of `∘.(f⫴)`: A function derived from `f`, which will create one isolate containing a copy of `f` for each required invocation (using the same rules as for `∘.f`) of `f`, start all the function calls, and return an array of futures of the same shape as would be returned by `∘.f`.

| f IIö ranks | f‖ö ranks<br>isolate.llRank |
|---|---|

Model of f‖ö: A function derived from f, which will create one isolate containing a copy of f for each required invocation (using the same rules as for fö ranks) of f, start all the function calls, and return an array of futures of the same shape as the outer shape of the rank invocation.

Note that, in the current model, the result is missing the final mix that would be performed by the rank operator

```
      ρ□←(,⍤ 2)2 3 4ρ⍳12
1 2 3 4 5 6 7 8 9 10 11 12
1 2 3 4 5 6 7 8 9 10 11 12
2 12

      ρ□←(, IIö 2)2 3 4ρ⍳12
 1 2 3 4 5 6 7 8 9 10 11 12  1 2 3 4 5 6 7 8 9 10 11 12
2
```

The reason for the missing mix is so that an array of futures can be returned. If the mix was included, the derived function would be forced to block in order to know the shapes of the individual results.

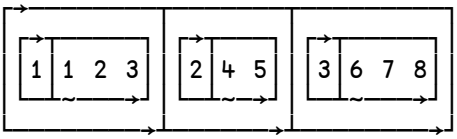
| f IIÐ | f‖⌸<br>isolate.llKey |
|---|---|

Model of f‖⌸: A function derived from f, which will create one isolate containing a copy of f for each required invocation (using the same rules as for f⌸) of f, start all the function calls, and return an array of futures of the same shape as the outer shape of the key invocation.

As with the model of rank, in the current model, the result is missing the final mix, in order to avoid blocking:

```
      {⍺ ⍵}⌸1 1 1 2 2 3 3 3
1   1 2 3
2   4 5
3   6 7 8

      ]disp {⍺ ⍵} IIÐ 1 1 1 2 2 3 3 3
```




## A.2 Isolate Server Management

| r←AddServer address ports |
|---|

Prepare to use an isolate server which must already be running on a machine with the specified IP address and has started isolate processes that are listening on the specified port numbers. The result of State after the isolate processes have been added. For example:

```
      #.isolate.AddServer 'calcsrv' (7052 7053)
 Host         Port  Isolates  Busy
 ----         ----  --------  ----
 calcsrv      7052         0     0
              7053         0     0
```

An empty right argument makes local processes available:

```
      #.isolate.AddServer ''
 Host         Port  Isolates  Busy
 ----         ----  --------  ----
 calcsrv      7052         0     0
              7053         0     0
 localhost    7052         0     0
              7053         0     0
```

## r←isSlave

Returns `1` when called from within an isolate server process, else `0`. This function is useful when the same workspace is used to run the client application and the host for isolate processes; it allows application startup code to easily determine which role it is expected to play.

```
      #.isolate.isSlave
0
```

## r←RemoveServer address

Removes a server which was added using `AddServer`. The result is the state after the isolate processes have been added. For example:

```
      #.isolate.RemoveServer 'calcsrv'
 Host         Port  Isolates  Busy
 ----         ----  --------  ----
 localhost    7052         0     0
              7053         0     0
```

An empty right argument removes local processes:

```
      #.isolate.RemoveServer ''
[no servers defined]
```

## r←StartServer access

Starts an isolate server on the current machine. The right argument is a list of access rules, separated by commas, which control which client machines will be allowed to connect to the server. If the right argument is empty, the server will only be usable from clients on the same machine.

Each rule is of the form `'keyword=value'`. At the moment, the only keyword which is accepted is "`ip`". The value must be a full or partial IP address. If the address is a prefix of the address of a client, the connection will be accepted. For example, `'ip=192.168.1'` would allow all machines with an address starting with 192.168.1 to connect.

It is anticipated that future versions of StartServer will support filtering on client SSL certificates IDs, and probably also the use of an external rule file.

The function produces output containing information that should be useful when providing arguments to the `AddServer` function:

```
      isolate.StartServer 'ip=192.168.1,ip=192.168.6'
  Machine Name:  calcsrv
  IP Ports:      7052 7053

 Enter the following in the client session:

       #.isolate.AddServer 'calcsrv' (7052-⎕IO-⍳2)

  Full IP address list:
   IPv4    192.168.193.1      IPv6    [fe80::545f:770f:dab:368b]
           192.168.247.1              [fe80::ccbd:aa34:3cfb:6e24]
```

## A.3 Configuration and Process State

### r←Config property [value]

Report or set configuration options (see section 8 for details). With an empty right argument, all current option settings are returned:

```
      isolate.Config ''
 drc                    #
 homeport            7051
 homeportmax         7151
 isolates              99
 listen                 0
 maxws              64000
 onerror           signal
 processes              1
 processors             4
 protocol            IPv4
 runtime                1
 workspace        isolate
```

With a right argument consisting of an option name, the current setting is returned:

```
      isolate.Config 'processors'
4
```

To set an option, follow the name with the new value. The previous setting is returned:

```
      isolate.Config 'processors' 8
4
```

### r←Reset 0

Resets the isolate mechanism, shutting down any isolate processes. The right argument currently has to be zero, additional options may be added in the future. The result is a brief report on the activity of the function:

```
       isolate.Reset 0
Reset: 0 isolates, 4 processes
```

## r←State ''

Produces a report on the current state of isolate processes. The right argument must currently be an empty character vector:

```
      isolate.State ''
[not initialised]

      is←⍬'' ⍝ Create an isolate: Initialises processes

      isolate.State ''
 Host       Port  Isolates  Busy
 ----       ----  --------  ----
 localhost  7052         1     0
            7053         0     0
            7054         0     0
            7055         0     0
            7056         0     0
            7057         0     0
            7058         0     0
            7059         0     0
```

Following the headers, there is one row of output for each isolate process. The four columns of output contain:

1. The host name or IP address (argument to `AddServer`) on which the process is running. For readability, empty if the name is the same is the preceding row.
2. The port number that the process is listening on.
3. The number of isolates currently hosted by this process.
4. The number of isolates that are currently executing an expression.

## A.4 State of Futures

The following functions all take the name of an array containing futures as the right argument, and return information regarding the state of each item of the array:

```
r←Available name
```

Returns a Boolean array with the same shape as the variable named in the right argument:

1: The corresponding item is not a future (either it was never a future, or the computation has succeeded).

0: The computation of the corresponding item is in progress, or has failed.

```
r←Failed name
```

Returns a Boolean array with the same shape as the variable named in the right argument:

1: The computation of the corresponding item has failed.

0: The corresponding item is not a future, is still being computed, or has completed successfully.

```
r←Running name
```

Returns a Boolean array with the same shape as the variable named in the right argument:

1: The computation of the corresponding item is in progress.

0: The corresponding item is not a future, has failed, or has completed successfully.

```
r←[null] Value name
```

Returns and with the same shape as the variable named in the right argument. If the computation is in process or has failed, the corresponding item of the result contains a ⍬NULL, or the optional left argument if it was provided. Otherwise, the item contains the corresponding value from the named variable.

# Appendix B: Troubleshooting

At the time that this document was written, the actual usage of isolates was essentially limited to synthetic experiments designed to test the code, or used during the writing of the document itself. Nonetheless, some problems were encountered; this section is a collection of explanations and tips to help you move forward when things stop working.

If you experience that the system frequently hangs while using futures and isolates, try the *Threads|Resume all Threads* menu item. Dyalog recommends that you do not have "Threads|Pause on Error" enabled when working with the experimental parallel features.

This should no longer be an issue from version 14.1: if you still encounter hangs that you suspect might be caused by this issue, please contact support@dyalog.com.

## B.1 "Normal" Errors

The following error messages may appear during normal operations:

**ISOLATE: Callback server restarted**
The system detected that listening was enabled, but no listener process was running.

**ISOLATE: Interrupt – continue waiting (y/n)?**
An interrupt was issued while the system was waiting for isolates to produce results. You can choose whether to issue an error for missing values, or resume waiting.

This should no longer happen in version 14.1: if you encounter this prompt in v14.1 please contact support@dyalog.com.

**ISOLATE: Transmission failure**
An attempt was made to transmit an unserialisable value (such as a class or an instance) to an isolate.

**FUTURE ERROR: 11: ISOLATE ERROR: Result cannot be returned from isolate**
An attempt was made to return an unserialisable value from an isolate.

## B.2 Initialisation and Communication Failures

**ISOLATE: Connection to address:port failed: 1111 ERR_CONNECT_DATA /* Could not connect to host data port */**
**ISOLATE: Unable to connect to isolate process: PROC10584797 192.168.17.105 7052**
**ISOLATE: Unable to connect to started processes (attempt n of m)**
**FUTURE ERROR: 86: COMMUNICATIONS FAILURE 1119 ERR_CLOSED : /* Socket Closed while receiving */**

These errors are most likely when using remote isolate servers: they suggest that the ip addresses or port numbers are not valid; the network has failed or the remote server has been shut down or crashed after it was added. The errors could also indicate that the started

isolates are crashing immediately. A common cause of this is the inability to locate the CONGA workspace.

Suggestions:

1) Check whether the processes are running or the network has failed
2) Try switching the ”runtime” property off to see whether any error messages are visible

**Unable to create listener on port 7051**
The system was unable to create a listener to receive callback requests. This probably means there is a defunct APL process which still has the port open. Use the “netstat” command to identify and kill it (a future enhancement to the Reset function will hopefully be able to do this automatically).

## B.3 Internal Errors

The following errors indicate serious internal logic errors, and should not occur during normal operations:. Please report the circumstances, ideally with instructions which reproduce the error, to the Dyalog Helpdesk:

**ISOLATE: Connection OK but incorrect handshake response received**
**ISOLATE: New process did not respond to handshake**
**ISOLATE: Unknown command**
**ISOLATE: Internal error**