



▶ Multi Threading

▶ Programming

Programming with threads requires care

This document covers the following topics

- Thread Switching
- Name scope
- Debugging
- Using Threads
- Stock considerations
- Globals and the order of execution
- Threads and niladic functions
- Threads and objects
- Threads and external functions (`□NA`)

Thread Switching

The interpreter may switch between running threads at the following points:

- Between any two lines of a defined (or dynamic) function or operator.
- While waiting for a `□DL` to complete.
- While waiting for a `□FHOLD` to complete.
- While awaiting input from:
 - `□DQ`.
 - `□SR`
 - `□ED`
 - The session prompt or `□:` or `□`.
- While awaiting the completion of an external operation:
 - A call on an external (AP) function.
 - A call on a `□NA` (DLL) function.
 - A call on an OLE function.

At any of these points, the interpreter might execute code in other threads. If such threads change the global environment; for example by changing the value of, or expunging a name; then the changes will appear to have happened while the thread in question passes through the switch point. It is the task of the application programmer to organise and contain such behaviour!

You can prevent threads from interacting in critical sections of code by using the `:Hold` control structure.

Dyalog Ltd

South Barn
Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom

Phone:

+ 44 (0) 1256 830 030

Fax:

+ 44 (0) 1256 830 031

e-mail:

sales@dyalog.com



▶ Multi Threading

▶ Programming

Dyalog Ltd

South Barn
Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom

Phone:

+ 44 (0) 1256 830 030

Fax:

+ 44 (0) 1256 830 031

e-mail:

sales@dyalog.com

Name Scope

APL's name scope rules apply whether a function call is synchronous or asynchronous. For example when a defined function is called, names in the calling environment are visible, unless explicitly shadowed in the function header.

Just as with a synchronous call, a function called asynchronously has its own local environment, but can communicate with its parent and 'sibling' functions via local names in the parent.

This point is important. It means that siblings can run in parallel without danger of local name clashes. For example, a GUI application can accommodate multiple concurrent instances of its call-back functions.

However, with an asynchronous call, as the calling function continues to execute, both child *and parent functions* may modify values in the calling environment. Both functions see such changes immediately they occur.

If a parent function terminates while any of its children are still running, those children will thenceforward 'see' local names in the environment that called the parent function. In cases where a child function relies on its parent's environment (the setting of a local value of `⎕IO` for example), this would be undesirable, and the parent function would normally execute a `⎕TSYNC` in order to wait for its children to complete before itself exiting.

If, on the other hand, after launching an asynchronous child, the parent function calls a *new* function (either synchronously or asynchronously), names in the new function are beyond the purview of the original child. In other words, a function can only ever see its calling stack decrease in size – never increase. This is in order that the parent may call new defined functions without affecting the environment of its asynchronous children.

Debugging

If a thread sustains an un-trapped error, its execution is *suspended* in the normal way. At the same time, all other thread activity in the workspace is *paused* until the suspension is cleared or restarted. A consequence of this is that, at any one time, only a single thread can be suspended.

The session is attached to the suspended thread, so that you can:

- Examine and modify local variables.
- Trace and edit functions.
- Clear suspensions.
- Restart execution.

When the last suspension in a thread is cleared or restarted, the session is reattached to the base thread, and any paused threads are resumed.



▶ Multi Threading

▶ Programming

From the session in a suspended thread, you can spawn a new thread, but its execution is immediately paused until the parent's suspension is removed.

The error message from a thread other than the base is prefixed with its thread number:

```
260:DOMAIN ERROR
Div[2] rslt←num÷div
      ^
```

State indicator displays: `)SI` and `)SINL` have been extended to show threads' tree-like calling structure.

```
      )SI
.    Calc[1]
&5
.    .    DivSub[1]
.    &7
.    .    DivSub[1]
.    &6
.    Div[2]*
&4
Sub[3]
Main[4]
```

Here, *Main* has called *Sub*, which has spawned threads *4* and *5* with functions: *Div* and *Calc*. *Div*, after spawning *DivSub* in each of threads *6* and *7*, has been suspended at line [2].

Removing stack frames using 'Quit' from the tracer or `→` from the session affects only the current thread. When the final stack frame in a thread (other than the base thread) is removed, the thread is expunged.

`)RESET` removes all but the base thread.

Dyalog Ltd

South Barn
Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom

Phone:

+ 44 (0) 1256 830 030

Fax:

+ 44 (0) 1256 830 031

e-mail:

sales@dyalog.com



▶ Multi Threading

▶ Programming

Using Threads

Put most simply, multithreading allows you to *appear to* run more than one APL function at the same time, just as Windows (or UNIX) *appears to* run more than one application at the same time. In both cases this is something of an illusion, although it does nothing to detract from its usefulness.

Dyalog implements an internal timesharing mechanism whereby it shares processing between threads. Although the mechanics are somewhat different, APL multithreading is rather similar to the multitasking provided by Windows 95/98 and NT. If you are running more than one application, Windows switches from one to another, allocating each one a certain *time slice* before switching. At any point in time, only one application is actually running; the others are paused, waiting.

If you execute more than one Dyalog thread, only one thread is actually running; the others are paused. Each APL thread has its own State Indicator, or SI stack. When APL switches from one thread to another, it saves the current stack (with all its local variables and function calls), restores the new one, and then continues processing.

Stack Considerations

When you start a thread, it begins with the SI stack of the calling function and sees all of the local variables defined in all the functions down the stack. However, unless the calling function specifically waits for the new thread to terminate (see `⌈TSYNC`), the calling functions will (bit by bit, in their turn) continue to execute. The new thread's view of its calling environment may then change. Consider the following example:

Suppose that you had the following functions: `RUN[3]` calls `INIT` which in turn calls `GETDATA` but as 3 separate threads with 3 different arguments:

```

▽ RUN;A;B
[ 1]   A←1
[ 2]   B←'Hello World'
[ 3]   INIT
[ 4]   CALC
[ 5]   REPORT
      ▽
      ▽ INIT;C;D
[ 1]   C←D←0
[ 2]   GETDATA&'Sales'
[ 3]   GETDATA&'Costs'
[ 4]   GETDATA&'Expenses'
      ▽
    
```

Dyalog Ltd

South Barn
Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom

Phone:

+ 44 (0) 1256 830 030

Fax:

+ 44 (0) 1256 830 031

e-mail:

sales@dyalog.com



▶ Multi Threading

▶ Programming

When each *GETDATA* thread starts, it immediately sees (via *ISI*) that it was called by *INIT* which was in turn called by *RUN*, and it sees local variables *A*, *B*, *C* and *D*. However, once *INIT[4]* has been executed, *INIT* terminates, and execution of the root thread continues by calling *CALC*. From then on, each *GETDATA* thread no longer sees *INIT* (it thinks that it was called directly from *RUN*) nor can it see the local variables *C* and *D* that *INIT* had defined. However, it *does* continue to see the locals *A* and *B* defined by *RUN*, until *RUN* itself terminates.

Note that if *CALC* were also to define locals *A* and *B*, the *GETDATA* threads would still see the values defined by *RUN* and not those defined by *CALC*. However, if *CALC* were to modify *A* and *B* (as globals) without localising them, the *GETDATA* threads would see the modified values of these variables, whatever they happened to be at the time.

Globals and the Order of Execution

It is important to recognise that any reference or assignment to a global or semi-global object (including GUI objects) is **inherently dangerous** (i.e. a source of programming error) if more than one thread is running. Worse still, programming errors of this sort may not become apparent during testing because they are dependant upon random timing differences. Consider the following example:

```

▽ BUG;SEMI_GLOBAL
[1]   SEMI_GLOBAL←0
[2]   FOO& 1
[3]   GOO& 1
      ▼
      ▼ FOO
[1]   :If SEMI_GLOBAL=0
[2]       DO_SOMETHING SEMI_GLOBAL
[3]   :Else
[4]       DO_SOMETHING_ELSE SEMI_GLOBAL
[5]   :EndIf
      ▼
      ▼ GOO
[1]   SEMI_GLOBAL←1
      ▼

```

Dyalog Ltd

South Barn
Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom

Phone:

+ 44 (0) 1256 830 030

Fax:

+ 44 (0) 1256 830 031

e-mail:

sales@dyalog.com



Multi Threading

Programming

In this example, it is formally impossible to predict in which order APL will execute statements in *BUG*, *FOO* or *GOO* from *BUG[2]* onwards. For example, the actual sequence of execution may be:

```
BUG[1] → BUG[2] → FOO[1] → FOO[2] →
DO_SOMETHING[1]
```

or

```
BUG[1] → BUG[2] → BUG[3] → GOO[1] →
FOO[1] → FOO[2] → FOO[3] →
FOO[4] → DO_SOMETHING_ELSE[1]
```

This is because APL may switch from one thread to another between any two lines in a defined function. In practice, because APL gives each thread a significant time-slice, it is likely to execute many lines, maybe even hundreds of lines, in one thread before switching to another. However, you must not rely on this; **thread-switching may occur at any time between lines in a defined function.**

Secondly, consider the possibility that APL switches from the *FOO* thread to the *GOO* thread after *FOO[1]*. If this happens, the value of *SEMI_GLOBAL* passed to *DO_SOMETHING* will be 1 and not 0. Here is another source of error.

In fact, in this case, there are two ways to resolve the problem. To ensure that the value of *SEMI_GLOBAL* remains the same from *FOO[1]* to *FOO[2]*, you may use diamonds instead of separate statements, e.g.

```
:If SEMI_GLOBAL=0 ♦ DO_SOMETHING SEMI_GLOBAL
```

Even better, although less efficient, you may use *:Hold* to synchronise access to the variable, for example:

```
▽ FOO
[1]   :Hold 'SEMI_GLOBAL'
[2]       :If SEMI_GLOBAL=0
[3]           DO_SOMETHING SEMI_GLOBAL
[4]       :Else
[5]           DO_SOMETHING_ELSE SEMI_GLOBAL
[6]       :EndIf
[7]   :EndHold
▽
▽ GOO
[1]   :Hold 'SEMI_GLOBAL'
[2]       SEMI_GLOBAL←1
[3]   :EndHold
▽
```

Dyalog Ltd

South Barn
Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom

Phone:

+ 44 (0) 1256 830 030

Fax:

+ 44 (0) 1256 830 031

e-mail:

sales@dyalog.com



▶ Multi Threading

▶ Programming

Now, although you still cannot be sure which of *FOO* and *GOO* will run first, you can be sure that *SEMI_GLOBAL* will not change (because *GOO* cuts in) within *FOO*.

Note that the string used as the argument to *:Hold* is completely arbitrary, so long as threads competing for the same resource use the same string.

A Caution

These types of problems are inherent in all multithreading programming languages, and not just with Dyalog.

If you want to take advantage of the additional power provided by multithreading, it is advisable to think carefully about the potential interaction between threads.

Threads and Niladic Functions

In common with other operators, the spawn operator *&* may accept monadic or dyadic functions as operands, but not niladic functions. This means that, using spawn, you cannot start a thread that consists only of a niladic function

If you wish to invoke a niladic function asynchronously, you have the following choices:

- Turn your niladic function into a monadic function by giving it a dummy argument which it ignores.
- Call your niladic function with a dynamic function to which you give an argument that is implicitly ignored. For example, if the function *NIL* is niladic, you can call it asynchronously using the expression:

```
{NIL}& 0
```

- Call your function via a dummy monadic function, e.g.

```
▽ NIL_M DUMMY
[1]  NIL
▽
NIL_M& ''
```

Dyalog Ltd

South Barn
Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom

Phone:

+ 44 (0) 1256 830 030

Fax:

+ 44 (0) 1256 830 031

e-mail:

sales@dyalog.com



Multi Threading

Programming

- Use execute, e.g.

```
⚡ & 'NIL'
```

Note that niladic functions *can* be invoked asynchronously as callback functions. For example,

```
□WS'Event' 'Select' 'NIL&'
```

will execute correctly as a thread, even though *NIL* is niladic. This is because callback functions are invoked directly by `□DQ` rather than as an operand to a the spawn operator.

Threads and Objects

The following rules apply when using threads and objects together.

1. All events generated by an object are reported to the thread that *owns* the object and cannot be detected by any other threads. A thread *owns* an object if it has created it or inherited it. If a thread terminates without destroying an object, the ownership of the object and its events passes to the parent thread.

2. The Root object `'.'` and the Session object `□SE` are owned by thread 0. Events on these objects will be only be detected and processed by `□DQ` running in thread 0, or by the implicit `□DQ` that runs in the Session during development.

3. Several threads may invoke `□DQ` concurrently. However, each thread may only use `□DQ` on objects that it owns. If a thread attempts to invoke `□DQ` on an object owned by another thread, it will fail with *DOMAIN ERROR*.

4. Any thread may execute the expression `□DQ '.'`, however:

4.1 In thread 0, the expression `□DQ '.'` will detect and process events on the Root object and on any Forms and other top-level objects owned by thread 0 or created by callbacks running in thread 0. The expression will terminate if there are no active and visible top level objects **and** there are no callbacks attached to events on Root.

4.2 In any other thread, the expression `□DQ '.'` will detect and process events on any Forms and other top-level objects owned by that thread or created by callbacks running in that thread. The expression will terminate if there are no active and visible top level objects owned by that thread.

Dyalog Ltd

South Barn
Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom

Phone:

+ 44 (0) 1256 830 030

Fax:

+ 44 (0) 1256 830 031

e-mail:

sales@dyalog.com



Multi Threading

Programming

5. A thread may use `□NQ` to *post* an event to an object owned by another thread, or to invoke the default processing for an event, or to execute a method in such an object. This means that the following uses of `□NQ` are allowed when the object in question is owned by another thread:

`□NQ object event...`

0 `□NQ object event...`

2 `□NQ object event...`

2 `□NQ object method...`

3 `□NQ ole_object method...`

4 `□NQ activexcontrol event...`

The only use of `□NQ` that is prohibited in these circumstances is

1 `□NQ object event...`

which will generate a *DOMAIN ERROR*.

6. While a thread is waiting for user response to a *strictly modal* object such as a `MsgBox`, `FileBox`, `Menu` or `Locator` object, any other threads that are running remain suspended. APL is not able to switch execution to another thread in these circumstances.

Threads and External Functions

External functions in dynamic link libraries (DLLs) defined using the `□NA` interface may be run in separate **C** threads. Such threads

- **take advantage of multiple processors** if the operating system permits.
- allow APL to **continue processing in parallel** during the execution of a `□NA` function.

Note that this feature does not apply to Version 7 because Windows 3.x does not support multi-threading.

When you define an external function using `□NA`, you may specify that the function is to be run in a separate C thread by appending an ampersand (&) to the function name, for example:

```
'beep'□NA'user32□MessageBeep& i'
a MessageBeep will run in a separate C
thread
```

Dyalog Ltd

South Barn
Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom

Phone:

+ 44 (0) 1256 830 030

Fax:

+ 44 (0) 1256 830 031

e-mail:

sales@dyalog.com



▶ Multi Threading

▶ Programming

When Dyalog first comes to execute a multi-threaded `⎕NA` function, it starts a new C-thread, executes the function within it, and waits for the result. Other APL threads may then run in parallel.

Note that when the `⎕NA` call finishes and returns its result, its new C-thread is retained to be re-used by any subsequent multithreaded `⎕NA` calls made *within the same APL thread*. Thus any APL thread that makes any multi-threaded `⎕NA` calls maintains a separate C-thread for their execution. This C-thread is discarded when its APL thread finishes.

Note that there is no point in specifying a `⎕NA` call to be multi-threaded, unless you wish to execute other APL threads at the same time.

In addition, if your `⎕NA` call needs to access an APL GUI object (strictly, a window or other handle) it should normally run within the same C-thread as APL itself, and not in a separate C-thread. This is because Windows associates objects with the C-thread that created them. Although you *can* use a multi-threaded `⎕NA` call to access (say) a Dyalog Form via its window handle, the effects may be different than if the `⎕NA` call was not multi-threaded. In general, `⎕NA` calls that access APL (GUI) objects should **not** be multi-threaded.

If you wish to run the same `⎕NA` call in separate APL threads at the same time, you **must** ensure that the DLL is *thread-safe*. Functions in DLLs which are not thread-safe, **must be prevented from running concurrently** by using the `:Hold` control structure.

Note that all the standard Windows API DLLs *are* thread safe.

Notice that you may define two separate functions, one single-threaded and one multi-threaded, associated with the same function in the DLL. This allows you to call it in either way. For example:

```
'mbeep'⎕NA'user32|MessageBeep& i'
⌘ Multi-threaded MessageBeep
'sbeep'⎕NA'user32|MessageBeep i'
⌘ Single-threaded MessageBeep
```

Dyalog Ltd

South Barn
Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom

Phone:

+ 44 (0) 1256 830 030

Fax:

+ 44 (0) 1256 830 031

e-mail:

sales@dyalog.com