The tool of thought for expert programming

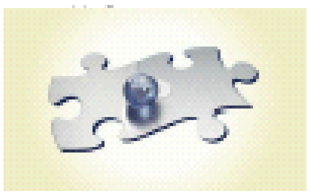
Dyalog™ for Windows

APLMON User Guide

Experimental Functionality

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



What is APLMON

APLMON is a functionality in the interpreter that, when enabled, will monitor APL primitives, and output the data into a CSV file (Comma Separated Values : a standard way to store a table of data in a plain text file, where each line is a row, and cell values are separated by one comma).

It was added to Dyalog APL Version 12.0. Note that it is still an experimental feature, and its specifications might change over time. However, be assured that, when not enabled (which is the default), it has no impact on the interpreter.

When running an APL application with APLMON enabled, it will log every primitive usage: how many times it was called, how much time was spent running it, and which arguments it was processing.

So this tool will point out which primitive usages your application hammers the most, to help Dyalog implementers decide where optimizations are worth doing.

Unfortunately, APLMON will not give you any application-side information, which is why APL programmers will prefer functions such as `⎕PROFILE` or `cmpx` from the dfns workspace to determine which parts of the application eats the CPU.

However, APLMON might be useful for APL programmers to check that their application use primitives as expected - for example, a mathematical application could be expected to spend most of its time using arithmetic function - or that it doesn't do useless computation such as disclosing simple scalars. Dyalog too would be interested in receiving APLMON output in order to determine where to concentrate performance enhancement work.

How to use APLMON

To enable APLMON, use the APLMON Root Method, passing as argument the name of the file where the data will be output. That file will be overwritten whether it exists or not. An empty file name disables APLMON.

Every time you call the method, if it was previously enabled, it outputs the data to the file that was set. It returns its name as a shy result. The data is also output when signing off the interpreter.

```
      +2 ⎕NQ '.' 'APLMON' 'C:\aplmon\myapplication.csv'

      ⍝ We can see that APLMON was disabled
      ⍝ so no output file was created.
      ⍝ Let’s run the application to monitor :
      MyApplication
      ⍝ Now, let’s output the data and disable APLMON
      +2 ⎕NQ '.' 'APLMON' ''
C:\aplmon\myapplication.csv
```

```
        ⍝ The data has been output to that file
```

This system allows you to log applications in separate files :

```
        ⍝ Enable APLMON for my first application :
        +2 ⎕NQ '.' 'APLMON' 'C:\aplmon\myapplication1.csv'

        MyApplication1
        ⍝ Flush the data of my first application
        ⍝ and disable APLMON :
        +2 ⎕NQ '.' 'APLMON' ''
C:\aplmon\myapplication1.csv
        ⍝ Do some non-monitored stuff
        InitialiseMyApplication
        ⍝ enable APLMON for my second application :
        +2 ⎕NQ '.' 'APLMON' 'C:\aplmon\myapplication2.csv'

        MyApplication2
        ⍝ Signing off will flush the data
        ⍝ of my second application :
        )OFF
```

Notes about time mesurement

On modern computers, accurate time measurements not trivial, due to several reasons :

- most operating systems are not real-time systems
- complex hardware (and software) generates lots of interruptions
- multi-core and variable-frequency processors invalidates clock count measurements

This is why, to help accuracy, any time measurement (be it with APLMON, ⎕MONITOR, or any other tool) should be done with as few other running processes as possible, and on a process that has a higher priority than them.

However, when doing enough measurements, even though they are still not perfectly accurate, their bias tends to level out so that the relative accuracy gets better. As long as we are interested in relative rather than absolute CPU consumption, it makes time measurements acceptable. This is why measurements get very accurate when in the order of magnitude of a second, be it in one or multiple shots.

Finally, it should be noted that the APLMON monitoring overhead will make the interpreter run about 3 times slower on Windows and about 2 times slower on UNIX (which is spent outside the measurement and therefore will not affect relative accuracy of the measurements).

After you start the interpreter, launch the task manager: Start → Execute... → type “taskmgr” and hit enter. Then in the “Processes” tab, select “dyalog.exe”, right-click on it and set its priority to “AboveNormal”, which is what we need because all other user processes are in priority “Normal” by default. We don't need higher priority though, and it is not desirable because the interpreter might hang the operating system and make user control difficult.

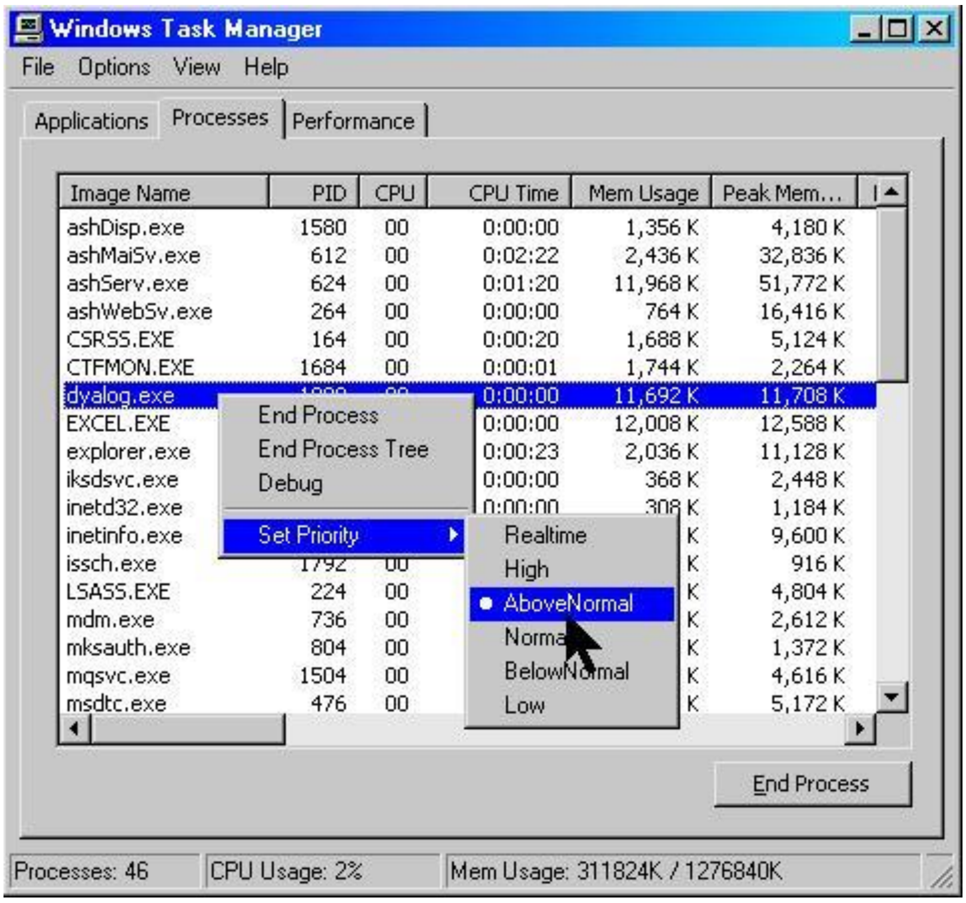


Image Name	PID	CPU	CPU Time	Mem Usage	Peak Mem...
ashDisp.exe	1580	00	0:00:00	1,356 K	4,180 K
ashMaiSv.exe	612	00	0:02:22	2,436 K	32,836 K
ashServ.exe	624	00	0:01:20	11,968 K	51,772 K
ashWebSv.exe	264	00	0:00:00	764 K	16,416 K
CSRSS.EXE	164	00	0:00:20	1,688 K	5,124 K
CTFMON.EXE	1684	00	0:00:01	1,744 K	2,264 K
dyalog.exe	[illegible]	[illegible]	0:00:00	11,692 K	11,708 K
EXCEL.EXE			0:00:00	12,008 K	12,588 K
explorer.exe			0:00:23	2,036 K	11,128 K
iksdsvc.exe			0:00:00	368 K	2,448 K
inetd32.exe			0:00:00	308 K	1,184 K
inetinfo.exe				K	9,600 K
issch.exe	1792	00		K	916 K
LSASS.EXE	224	00		K	4,804 K
mdm.exe	736	00		K	2,612 K
mksauth.exe	804	00		K	1,372 K
mqsvc.exe	1504	00		K	4,616 K
msdtc.exe	476	00		K	5,172 K

Processes: 46 CPU Usage: 2% Mem Usage: 311824K / 1276840K

Details on the APLMON output

The columns of the APLMON output file are :

token	Name of the primitive's token (symbol, if you prefer)
lfn, rfn	Name of the left and right operand's token, if the primitive was an operator
ltype, rtype	Data type of the left and right arguments of the function
lrank, rrank	Rank of the left and right arguments of the function. Arrays of rank greater or equal to 3 will be logged as « 3+ »
lshape, rshape	Range of the left and right argument sizes. A range is made of two numbers separated by a « < » sign. The real size is between these two numbers. So arguments with similar sizes are grouped together.
hitcount	Number of calls
time	Time spent by the primitive, in seconds

These fields are empty if not applicable (which will appear in the CSV file as two consecutive commas). Examples :

APL expression	token	lfn	rfn	ltype	rtype	lrank	rrank	lshape	rshape
`⎕AV`	[]AV								
`⍳1000`	iota				int16		0		1<1
`1↑¨'A' 'Poetic' 'Litterature'`	each	take		int8	nested	0	1	1<1	2<4

APLMON measures time only for the “atomic” calls (i.e. not complex expressions or user-defined functions). Primitives that evaluate APL expressions (such as ⍎, ⎕FIX, ⎕NEW) don't appear. For example :

– `foo←{1+⍵} ⋄ foo¨(N⍴1)` : the each will not appear in the log, but only the primitives sub-called by foo : so + will appeared as being called N times

– `+.×/(N⍴1)` : the / will not be measured, but `+.×` will appeared as called (N-1) times.

This ensure that time measurements don't overlap and therefore can be compared.

Some non-atomic expressions will be logged though, and the non-primitive parts will be logged as “non-primitive”, e.g. the left operand in `1∘+`.

Special syntaxes

APL has some special syntaxes that are, unavoidably, special-cased in APLMON.

Simple Indexing (where the indexed is semicolon-separated simple arrays)
The token is “simple indexing”, there are no lfn or rfn, the left argument is the indexed array, and the right argument is as representative of the indices as possible. We (arbitrarily) chose to make it appear as an array, the rank of which is the number of specified subscripts, and the shape of which is the number of items in these subscripts.
Examples :

APL expression	token	lfn	rfn	ltype	rtype	lrank	rrank	lshape	rshape
`'abcd'[3 3⍴3]`	simple indexing			unicode8	int8	1	1	2<4	5<9
`(2 2⍴4)[1;]`	simple indexing			int8	int8	2	1	2<4	1<1

Reach and Choose Indexing (where the indexer is a nested array)
The token is “reach indexing”, the left argument is the indexed array, and the right argument is the indexer array.

Axis specification
The token is “axis”, lfn is the function whose axis is specified, rfn is “non-primitive”, as it is the axis specification constant.

Assignment
The token is “left arrow” and it is considered as a monadic function.

Modified Assignment
The token is “left arrow”, lfn is the modifier function, rfn is empty, larg is the modified value and rarg the modifier value.

Indexed Assignment
The token is “indexed assignment”, lfn and rfn are empty, larg is the indexed array, rarg is the new value.

Modified Indexed Assignment
The token is “indexed assignment”, lfn is the modifier function, rfn is empty, larg is the indexed array, rarg is the new value.

Selective Assignment
The token is “selective assignment”, lfn and rfn are empty, larg is the indexed array, rarg is the new value.

Modified Selective Assignment
The token is “selective assignment”, lfn is the modifier function, rfn is empty, larg is the indexed array, rarg is the new value.

Stranding
The token is "stranding", larg and rarg are the arrays being stranded together.

Name Lookup

The token is "name lookup". This represents time spent in the parser looking up names in namespaces.

Defined Function Initialisation

The token is "tradfn init". This represents time spent on entry to a traditional defined function.

Selective Assignment Pre-scan

The token is "selassi pre-scan". This represents extra time spent in the parser to analyse the syntax of a selective assignment expression.

Control Structures

The token is "control structures". This represents time spent processing control structures in a traditional defined function.

Namespace dot syntax

The tokenis “nsref”, lfn and rfn are right and left operands respectively, larg and rarg are as expected.

Example : `⎕SE.⎕EX 'var'` : token is “nsref”, lfn is “[]EX”, rfn is “non-primitive”, larg is empty, rarg is the character vector.