ASWI programming standards for APL programs

by

F.H.D. van Batenburg
T.R.C. Bonnema
L.van Geldrop
H. van Loon
B. Smoor

version 2; changes by F.H.D. van Batenburg

1. INTRODUCTION

This paper describes a set of standards proposed by the participants of the "APL-Share-Ware-Initiative", ASWI for short. These standards are an integration of the various standards applied by BSO/Management Support (Utrecht, Netherlands), Philips/PASS-PC'S (Eindhoven, Netherlands), and the Institute for Theoretical Biology (Leiden, Netherlands).

The objective of this standard is two-fold.

1. The first objective is to define a set of rules according to which APL programming should be carried out, in order to improve program maintainability.
   It is recognized that in a professional APL programming environment, where different people work for different clients, a need exists for a concise and standardized use of APL.
2. The second objective is to smooth the introduction to new programs for potential users. To fulfil this objective a set of standard names are proposed which should be available in every workspace and which act and inform according to preconceived, general accepted notions. This ensures that a novice will always know how to move around in the on-line documentation of a foreign workspace.

We ask all programmers participating in ASWI to adhere to these standards for the following reasons.

- First of all, standards are a boon for not too experienced programmers because -hopefully- standards reflect the accumulated wisdom of experts.
- Secondly, standards are useful because the ***writers*** can fall back to automatisms and can focus their thinking on the real hard nuts in the program.
- And finally, ***readers*** can faster access the information when they do not need to puzzle things out but can rely on standards instead.

This holds for good standards, but it holds for mediocre standards too.

So, even a non-superb standard is better than very clever incidently used constructions. For this reason, we urge programmers to deviate from this standard in exceptional cases only and then to document these deviations carefully in order to ensure program maintainability.

2. NAMING CONVENTIONS

This paragraph outlines the various standards for naming of systems, workspaces, functions, variables and labels.

In this chapter we talk about capitals and lowercast characters. If your system has 2 other charactersets (for example capitals and underscored characters), please read "normal APL text-characters" for capitals and "the other characterset" for lowercast characters.

We will discuss here the names of...
1. systems
2. workspaces
3. user functions
4. sub-functions
5. imported sub-functions
6. Special (sub-)functions
7. standard names
8. groups
9. labels
10. arguments
11. local variables
12. global variables
13. sub-global variables
14. shared variables

2.1 Systems & System IDentifier

An APL system is a collection of one (or more) APL workspace(s) and files which conceptually form a whole. Together they perform the functions described in the functional specification of the system.

The system is identified by a 3 character identifier (no more, no less). Each of the characters should be alphabetic capitals.

Valid identifiers are in the range of 'AAA' till 'ZZZ'.

We will refer to the system identifier as '***sid***' hereafter.

Examples:
- DLG - Dialogue system
- PLT - Plotting system
- RNA - RNA analysis
- SCR - Screen utilities
- TED - Text editing system
- WHL - Whaling population growth simulation

2.2 Workspaces

The workspace is identified by a name of 3 or more characters of which the first three are the ***sid***.

All characters should be capitals and digits may be used (not in the ***sid***).

Preferably a system is contained in 1 workspace. If this is not feasible (for example for reasons of memory requirements) the system should be divided in functionally cohesive subsystems distributed over different workspaces. The names of those workspaces all start with the same ***sid***.

Example of *WHL* system:
- *WHLDATA* - file which contains all data of whale simulation.
- *WHLSIM* - workspace which performs population simulation.
- *WHLANAL* - workspace which statistically analyses results.

2.3 Functions

This paragraph will outline the naming conventions for *user-functions*, *sub-functions* and *imported sub-functions*. Finally we will list some standard names.

2.3.1 User-functions

By user functions we mean:
- driver or main functions (the ones that start up the system).
- functions to be activated by the end-user directly.

User functions can be recognized by:
1. an easy-to-remember name.
2. error-proof and with error-messages in the language of the end-user.
3. in general no parameters.
4. in general no explicit result.

Use at least 5 characters for the name of a user function. The first three are the ***sid*** and the next two are alphabetic capitals. Remaining characters may be alphanumeric capitals.

Examples:
- *PLTHISTGRAM*, *PLTPIE* & *PLTGRAPH*
- *WHLSETPARMS*, *WHLSIMULATE*, *WHLANALYSE* & *WHLGRAPH*

2.3.2 Sub-functions

Sub-functions are functions that are not used by the end-user directly. In general such functions are invoked from other APL functions.

A sub-function has a name of at least 6 characters for the name of a sub-function. The first three are the ***sid***, followed by the underscore "__". The rest of the characters may be alphanumeric capitals.

Try to recognize functional related sub-functions, and characterize them with common characters in position 5 and 6 of their names.

(For systems with many functions, you can apply this principle recursively for remaining characters.)

Examples:
- *SIM_*... all simulation programs of system with ***sid***=SIM.
- *SIM_DS*... all discrete sub-functions.
- *SIM_CN*... all continuous sub-functions.
- *SIM_CNAS*... all continuous sub-functions for "assignment".
- *SIM_CNTRL*... all continuous sub-functions for "transfer".
- *SIM_CNSP*... all continuous sub-functions for "splitting".

An alternative to using underscore for ***sid***-separation is applying capitals for separation.

Examples:
- *Sim*... all simulation programs of system with ***sid***=Sim.
- *SimDis*... all discrete sub-functions.
- *SimCnt*... all continuous sub-functions.
- *SimCntAss*... all continuous sub-functions for "assignment".
- *SimCntTrf*... all continuous sub-functions for "transfer".
- *SimCntSplt*... all continuous sub-functions for "splitting".

An exception can be made for subfunctions that are used very frequently. Having 4 characters in front of the name of subfunctions that are called in many other functions is awkward and makes reading harder.

Use for those frequently used functions a 1-character lowercase prefix. For example "***u***..." instead of "*UTL_*" for utilities from the *UTL*...-workspace; "***t***..." instead of "*TLS_...*" for personal tools in the *TLS...*-workspace and "***m...***" instead of "*MDF_...*" for Machine Dependent Functions from the *MDF*-workspace.

2.3.3 Special (sub-)functions

Four special groups of functions can be recognised: Information-, tools, utilities and machine-dependant sub-functions.

2.3.3.1 INFORMATION FUNCTIONS

Two useful information functions are demonstration functions and help functions.

Demonstration functions have the name of its object, followed by "*D*". For example *uPRINTD* demonstrates the use of function *uPRINT*.

Help functions describe the use of a specific function or a specific variable. They have the name of their object followed by "∆" or "*H*". For example *uPRINTH* or *uPRINT*∆ describes the use of function *uPRINT*. and *RNAstructure*∆ describes global variable *RNAstructure*.

2.3.3.2 MACHINE DEPENDENT FUNCTIONS

Machine dependent functions are cover functions that hide code that is unique for a device, for a particular machine or for a particular APL implementation. We will elaborate on this in chapter coding. Use naming convention "***m****...*". For example mprint to send data to printer, *mPENCLOSE* for "⊂" if some of you APL implementations do not have the partioned enclose, or use different character, *mGUIO* for graphical user input/output, *mFILE* for writing and reading data.

2.3.3.3 TOOLS

Tools are functions that help you as a programmer. For example a program that lists all functions, or a program that lists all calls made by one of more functions. Use naming convention "***t****...*". For example *tFNLOCK* to automatically lock functions, *tXREF* to show all used objects in a function *tREPLACE* to replace strings in all functions *tRENAME*2 to rename functions.

2.3.3.4 UTILITIES

Utilities are also functions that help you, but not as tools, but to insert in implementations. A collection of sub-functions that your programs use. Use naming convention "***u****...*". For example *uBOX* to convert a vector to matrix, *uDMB* to delete multiple blancs in a string, *uKIT*1 and *uKIT*2 to glue objects under, respectively alongside each other, *uSWAP* to swap characters (lowercase to uppercase).

2.3.4 Imported sub-functions

Functions with the same names in different workspaces should be identical. Sub-functions which are copied from other workspaces are "imported sub-functions". They retain their original name as long as they are not changed. As their ***sid*** will differs from that of all the other functions in that new workspace their foreign origin remains obvious.

If you introduce any change in the code of that function, change the name according to the new workspace in which the function now resides. (If necessary, you can explain the origin in the comment.)

2.3.5 Standard names

The following niladic functions should always be present in the workspace whenever applicable: *LX*, *BEGIN*, *RECOVER*[1] and *TESTALL*.

2.3.5.1 LX

Niladic program *LX* will either start the main program *BEGIN* or present the most general information which is provided by *ABSTRACT* (see paragraph DOCUMENTATION).

For example in workspace *WHLSIM*:

```
⎕LX←'LX'
⎕FX 'Z←LX' 'Z←ABSTRACT'
```

and in workspace *DLGUTILS*:

```
⎕LX←'LX'
⎕FX ⊃ 'LX' 'BEGIN'
```

2.3.5.2 BEGIN

If a workspace has one user function only, then this function is invoked by function *BEGIN*.

In workspaces with more than one user function or with sub-functions (utilities) only, function *BEGIN* only presents those alternative functions.

For example in workspace *WHLSIM*:

```
⎕FX ⊃ 'BEGIN' 'WHLSIMULATE'
```

2.3.5.3 RECOVER

Any system which might be corrupted by unexpected errors or crashes should provide a function *RECOVER* to repair the damage as good as possible.

If automatic recovery is awkward or impossible, *RECOVER* should direct the user to the proper information, or describe the best procedure to do so.

---

1) The original ASWI standard prescribes the use of: ***sid*** resulting in names: ***sid****LX*, ***sid****BEGIN* and ***sid****RECOVER*

A smart construction proposed by Holls is to have ⎕*LC* local in the main function and assign it '⎕*LC*←'→*RECOVER*''. Program *RECOVER* should not only recover, but also return "1↓⎕*LC*" or another line number where the main program should continue.

For example in workspace *WHLSIM*:

```
  ∇Z←RECOVER;ERR
[1] ERR←CHECKINTEGRETIY
[2] ERR←REPAIR ERR
[3] ⎕←(1+×ERR)⊃ 'Restart'('Unrecoverable error',⍕ERR)
[4] Z←(1+×ERR)⊃(1↓⎕LC)(0)∇
```

2.3.5.4 TESTALL

Every big application needs a set of automatic tests to ensure a minimal quality. Assemble those tests in function *TESTALL*.

2.4 Groups

Groupnames start with the same name as the main item in the group; at the end *GP* is added.

Examples:

- item *RNAPREDGP* is the group with main item *RNAPRED*.
- item *WHLSIMULGP* is the group with main item *WHLSIMUL*.

2.5 Labels

The only permissable label names are *B#*, *E#* (with # a number), *BE*, *XX* and *EXIT*. See "branching" in next chapter.

2.6 Variables

This paragraph will outline the conventions for the names of arguments, shared, global, sub-global and local variables.

2.6.1 Standard documentation

Some standard names that can be functions or variables are:

- *ABSTRACT*
- *DEMO*
- *DESCRIBE*
- *HELP*
- *README*
- *WS*
- *∆FILES*
- *∆GLOBALS*
- *∆HISTORY*
- *∆SHARED*
- *∆SYSTEM*

These variables (or functions) are described in chapter DOCUMENTATION.

2.6.2 Function arguments

The name of the arguments and the explicit result of a function are *L*, *Z* and *R* or short mnemonic names which start with *L*, *Z* or *R*.

- *explicit result*: use name *Z* for the explicit result.
- *left argument*: use name *L* for the left argument of a function.
- *right argument*: use name *R* for the right argument of a function.
- use *F* for the left function argument of an operator.
- use *G* for the right function argument of an operator.

Do not reassign variables *L* and *R*. If you adhere to this rule, you can always restart a function with "→1".

2.6.3 Local variables

A local variable is a variable which is local to that function. The name of a local variable may be *A-E* or *H-K* or *M-Q*, *S-Y*. Or alternatively formulated, all capitals except *F*, *G*, *L*, *R*, or *Z*.

Use *I*, *J* and *K* for loopcounters and important indices.

Use *T* for *T*emporary line-locals. A line-local is a variable which gets its value somewhere else in the same line.

Use *S* for temporary line-*S*pans. A line-span is used for a break of long lines into smaller ones where the result of the first smaller one is used in the next one.

Use *Z* as early as possible in the program. Use *Z* if there is in the beginning of the program a preliminary result, even though it will be updated and/or extended in a major way later on.

When you want to use more mnemonic names, keep the following quidelines:

1. do not use names longer than 4 characters. Longer names might conflict with names of user functions and will diminish readability of the whole.
2. use *L*, *R*, *Z* as the first –or only-- character if that variable is the left argument, right argument, or explicit result.
3. do not use the standard label names for variables.

For clarity, longer mnemonic names could be used up to 4 characters in order to improve the readability of the programs. Do not make the names to long, however; what you gain in variable documentation, you loose in function clarity.

2.6.4 Global variables

A global variable is a variable which is not local to a function, or a variable which is stored in an APL file. Such a global variable is always available for reference by every APL function in the workspace.

Use at least 5 characters for the name of a global variable. The first three characters are the ***sid***. The next characters are lowercast alphanumerics.

Examples:

- *SCRappl*, *SCRfrs*, *SCRsubglbls*, *SCRmiss*.
- *RNAenergyrules*, *RNAsequence*, *RNAstructure*, *RNAparams*.

Restrict the use of global variables as much as possible! If complex global structures are needed, access them through cover functions and document them in variables …∆ .

2.6.5 Sub-global variables

A sub-global variable is a variable, which is not a global variable but is accessed by more than one function.

Use at least 6 characters for the name of a sub-global variable. The first three are the ***sid***, followed by "__". The remaining characters are all lowercast character.

You might observe that the difference between sub-global variables and global variables is similar to the difference between user-functions and sub-functions, namely the underscore.

Examples:

- *WHL__menu*, *WHL__params*, *WHL__select*.
- *RNA__options*, *RNA__choice*.

Restrict the use of sub-global variables as much as possible!

2.6.6 Shared variables

The names of shared variables are still under discussion. We welcome any suggestions.

3. CODING & DETAIL DESIGN

The following standards are a mixture of code and detail-design considerations. As the distinction is sometimes hard to make, we present them all here.

In this chapter we will discuss the following topics in alphabetical order.

1. branching
2. comments
3. consistency
4. cover functions
5. defaults
6. efficiency
7. execute
8. function arguments
9. function size
10. input
11. messages
12. output
13. error messages
14. restartability
15. restricted number of variables
16. robustness
17. short variable span
18. short detection span

3.1 Branching

There are numerous ways in APL to branch to function lines. This excessive number of possibilities has contributed to give APL a bad name. Imagine an APL function in which different ways of branching are used; for example: `→L×⍳C`, `→C⍴L`, `→C↑L`, `→C↓L`, `→C/L` (where `C` is a condition). It is clear that in reading such a function, a substantial amount of time is needed to analyse the flow in that function.

For reasons of structured programming ***do not think in branches*** but in ***conceptual blocks of code*** which are executed once, a multiple times or not at all. Each block is marked by a label `B#` at the first statement and by a label `E#` at the last statement, with # a unique number for each block.

Following this proposal, you will recognise the following 6 types of construction:

- Condition:
```
B3:→(...)/1+E3
 ...
E3:...
```
- If/else:
```
B5:→(...)/1+B5b
B5a:...
 ...
→1+E5
B5b:...
 ...
E5:...
```
- Case:
```
B4:→(...)/B4a,B4b,B4c,1+E4
B4a:...
 ...
 →1+E4
B4b:...
 ...
 →1+E4
B4c:...
 ...
E4:...
```
- For-loop:
```
B1:→(...I←I+1)/1+E1
 .....
E1:→B1
```
- Repeat...while:
```
B8:→(...)/1+E8
 ...
E8:→B8
```

- Repeat...until (only use this construction if you are absolutely sure that the first iteration will always run flawlessly):
```
B3:...
 ...
E3:→(...)/B3
```

Remember that the numbers identify different blocks in one function, not different types of construction.

Short conditions or repetitions can be coded in one line. As the conceptual code-block begins and ends in this line, we use label *BE*.

Examples are:

- `BE:→(...)/1+⎕LC ⋄ expression`
- `BE:→(...)/1+⎕LC ⋄ expr. ⋄ →⎕LC`
- `BE: expression ⋄ →(...)/⎕LC`

Observe the following rules:

1. Consider if you can code without a loop.
2. Never branch to results of a calculation or to absolute line numbers!
3. Never branch away from the middle of a line as in `..⋄→(CONDITION)/FARAWAY⋄...`
4. To exit a function prematurely, do not use "`→0`", but always "`→EXIT`" [2]. Set label "*EXIT*" at the very end of a function. This results in the following type of construction:
```
XX:→(...)/EXIT
   ...
XX:→(...)/EXIT
   ...
EXIT: last, empty(!) program line
```
5. Think again if you can code without a loop.

3.2 Consistency

The most important quidance for good coding is consistency. Be consistent as much as possible. Most bugs that I met were caused by inconsistencies. Bad examples:

1. all subfunctions had "sequence,structure" as parameter but one had "structure,sequence".
2. all subfunctions returned explicit results which the main program printed except a few that did the printing themselves.
3. all subfunctions got "controlsettings" as parameter except one that used global variable "controlsettings" directly.
4. one sub-function changed the order of numbers in a variable, used this througout the various statements; in another case the main function changed the order before passing these shuffled values to (another) subfunction.
5. one piece of information had different names in different sub-functions: *STRUCTURE*, *STR*, *STRUC*, *STRUCT*, *SEC*, and *SECONDARY* (so *SEC* was often confused with the often occurring variable *SEQ*).
6. one piece of information had different names in different name-parts. Sequence and SecondaryStructure where called *SEQ* and *STR*, but their filenames *FILSEQ* and *FILSEC*.

3.3 Comments

Use comments to describe ***what*** is done and ***why***, but ***not how*** it is done (that's what APL is for).

Any APL function should at least contain the following comments:

```
[1] ⍝⍝ description what the function does
[2] ⍝ R: description of the right argument
[3] ⍝ L: description of the left argument
[4] ⍝ Z: description of the result
[5] ⍝ E: example of program call; preferably an
      executable expresion that shows what happens
[6] ⍝ F: names of files called
[7] ⍝ G: names of used globals and subglobals, if
      any, but preferably none are present!!!!!!
[8] ⍝ M: info of machine-dependent constructions
[9] ⍝ P: name of programmer, dates of changes
      (extra lines for important changes)
```

Example:

2) The original ASWI standard prescribes the use of *OUT* instead of *EXIT*.

```
[1] ⍝ Find positions of nucleotides A
[2] ⍝ R[]: Characters RNA sequence
[3] ⍝ Z[]: stem positions
[4] ⍝ E: 1 2 8 9 ←→RNA_FINDA'AACUCCUAA'
[5] ⍝ M: ⎕AT in [17]
[6] ⍝ P: EvBatenburg, 5-95
[7] ⍝            8-95: Add RNA_test
[8] ⍝   M.v.Baan 1-96: Check stems l.5-10
```

Apply 2 types of comment: section comment and line comment.

1. Section comments explain several lines of code. This type of comment uses a whole line in the function.
2. Line comments illuminate the code of a line. This type of comment trails at the right of that particular line.

Example:

- ⍝ This explaines the following segment
- .....code....⍝ This explaines code at its left

3.4 Cover functions

Special vicious things should not be addressed directly in your functions, but be hidden in specially designed cover-functions. Problem areas are:

1. system dependent code.
2. machine dependent constructions.
3. accessing global data.

Make those modules easily recognisable. You do that by starting their name with ***m*** (machine dependent functions).

Build a workspace with such functions for different platforms. For example both the Macintosh workspace *MEDT.mac* and the PC workspace *MEDT.AWS* contain *mEDIT*. The application always calls *mEDIT*, but if you use a Macintosh you copy *mEDIT* from *MEDT.mac*, and if you work on a PC you copy it from workspace *MEDT.AWS*.

A similar reasoning for hiding complex things goes for big and complex data structures. Use functions like *WHLSIZEGET* to get a data-item from a variable and *WHLSIZESET* to change a data-item from the variable. If you change your structure later, or if you move from storage in a variable to storage in a file, you only need to change *WHLSIZEGET* and *WHLSIZESET*.

3.5 Default settings

Localise non-default settings. For example if you want to use ⎕*IO*←0, set that system-variable local in the main function.

3.6 Efficiency

Good APL functions are not necessarily (CPU) efficient. What is efficient today may very well be inefficient tomorrow (and vice versa). As maintenance often costs much more than computertime, the readability of functions comes first.

3.7 Execute

Limit the use of execute ⍎. Not only is it inefficient, but more important is that statements can not be analysed if they are enclosed in stringquotes. For example programs that check for globals or programs that build the hierargical calling tree cannot see that other programs are called within a characterstring.

3.8 Function arguments

1. If you wonder what data to present to the right and what to the left of a function: give control to the left argument and real data to the right argument and .
2. Think what is the most used control for a function and use for this 0 in the left argument. Make the function substitute 0 as default when the left argument is missing. The same holds for the right argument: think of a sensible default which will start if the right argument is `''`.

3.9 Function size

Functions should perform one easy expressable task. Do not program vertically (many supershort statements), nor horizontally (few superlong statements), but rectangular.

The size of an APL function should be within working conditions and preferably less. This means that if one is working with a screen that displays 20 lines of 80 characters each, the size of APL functions should preferably be less than, or equal to 20 lines of 80 characters.

The listing of a program should fit both on the screen, as well as on an A4 page, whichever is smaller. For APL readers:

```
FUNCTIONSIZE←PAPERSIZE⌊SCREENSIZE
```

If you write an utility function, try to keep everything done in one function, so you do not need to copy (or delete) a whole tree of functions. This rule might conflict with the previous rule that size should not exceed that of screen⌊paper. Use your own judgment.

3.10 Input

1. Prompt for user input via a special function (for example via *uIN*).
2. Always supply a default that is used if the user only hits [CR]. Choose the least harmful option as default, or the most frequently used alternative.

3.11 Messages

1. Try to avoid error messages. Ask yourself what you can do to prevent the need for an error message. If you know enough to detect an error, try to prevent it
2. Try to prevent confirmation messages. Mostly you need them to warn the user that he is doing something dangerous. Rather then spend your time on confirmations, invest it to make the operation reversible.

3.12 Output

1. Do not program output directly using '⎕←', but route output via a special function (for example via *uOUT*).
2. Always consider if output can be given as explicit result, rather than writing directly to the screen. This enables the user (or the parent function) to decide later whether to print, or to analyse the result. When interrupting the output, the user only interrupts output, not the program which might remain pending.
3. Use cover functions for device dependent output.

3.13 Restartability

Make your code restartable.

1. Do not reassign the argument variables *L*, and *R*. This makes your function restartable by "→1".
2. Do not reassign used variables in a line until the very end. This makes your expression restartable. . So do not specify:

   ```
   Z←....Z.....Z←.....Z
   ```

   but use "line-local" *T*:

   ```
   Z←...T....T←....Z
   ```
3. Consider to set:
   - ⎕*LC* local in main function
   - assign '⎕*LC*←'→*RECOVER*'"
   - program function *RECOVER*

3.14 Restricted variable number

Restrict the number of variables that you (your reader) has to remember. Only use special variable names for information that is needed all over the program; but reuse "neutral" variable names for information that is needed in a short segment only.

The contenth of variables with neutral names can easily be documented in the comment.

A good approach is the escalator approach: identify where the initial nucleus of result starts, assign it to variable Z (this signals to the reader that here is the preliminary result) and continue in the sequel by updating Z step by step as "Z←....Z...."

3.15 Robustness

1. Write functions that can process arguments of variable shape (or even rank). In particular empty arguments.

2. Design functions such that they work for missing left argument (default) and do not crash for empty arrays.

3.16 Short variable span

Always keep variable span short (that is: use a variable immediately after its value is computed, not several lines later). So, do not initialise counter `I` to 0 in line 5 if you use it only in a loop that spans lines 20-25. In such a case, initialise `I` in line 19.

3.17 Short detection span

Always deal with errors and other conditions right after detection. For example do not cluster all error-handling "nice" together at a far-away spot.

4. DOCUMENTATION

This paragraph outlines the proposed on-line documentation 1) for programmers and 2) for end-users.

Use capitals and lowercase characters as in normal text (not capitals only).

Our convention is simple and straightforward: document anything you like by an APL variable that has the same name as the item you are documenting, extended[3] with "∆" or "H" (for Help). Use "*–H*" for documentation to help users and *-∆* for documentation to help APL programmers.

For example, if you document the global variable "*WHLxyz*" its programmer documentation is found in "*WHLxyz∆*", and its end-user documentation in "*HWHLxyzH*".

This chapter will discuss:
1. programmer documentation in WS
2. user documentation in WS
3. paper documentation.

4.1 Programmers documentation

By on-line programmers-documentation we mean a description of all components of an APL system in the workspace or on file so that it can be used rapidly at all times by a programmer. This documentation is intended for system maintenance and development.

The following items should be documented and on-line available: *∆SYSTEM*, *∆GLOBALS*, *∆SHARED*, *∆FILES*, *∆HISTORY* [4].

4.1.1 System

If any general information exists which should be available for the programmer, then it is documented in variable *∆SYSTEM*.

4.1.2 Files

Files are described briefly by *∆FILES*. gain, one line per file with name (optionally tie-number) and purpose. Again more detailed information about name, purpose, structure and acces in *-∆*.

Example: file *ACTS* is described *∆FILES* and could be described in more detail in *ACTS∆*.

For component files we suggest the following extract from Bergquist,G. & Cannon,R.

4.1.2.1 NAMING

- Use a 2 a 3 character prefix to identify the system to which it belongs.
- Convey the type of data in the file by one of the following suggested suffixes: *TXT* for ASCII text, *TMP* for temporary files, *DAT* for main production data, *BIN* or *HEX* for machine code.
- If you want/can you might express the following information in the name: owner, date, version/release, status.

4.1.2.2 COMPONENT USE

Use the first 10 components of component files for:

Component 1: a text with application identifier. Specify application, version number.

Component 2: a text with purpose of file and how it is structured.

Component 3: component index with appropriate names and purposes for each component. Use a matrix where each row can be executed to get meaningful variables for each component.

---

3) The original ASWI standard suggested prefixing ("*∆WHLxyz*" instead of "*WHLxyz∆*"), however extending keeps the object and its documentation together in an alphabetical list.

4) The original ASWI standard prescribes the use of: ***sid***, resulting in names: *∆**sid**SYSTEM*, *∆**sid**GLOBALS*, *∆**sid**SHARED* and *∆**sid**HISTORY*.

For example:

```
FNAMES←11 ⍝ Names of deptms
FSALARIES←12 ⍝ incomes.....
FUPDATS←13 ⍝ monthly change
```

Component 4-10: spare components for updates. Use for secondary indexes, audit trail, etc.

4.1.3 Global variables

Global variables must be documented briefly (1 line per variable) by *∆GLOBALS*. Report at least name and purpose of each.

Describe complicated variables in *-∆*, where *-* stands for the name of the variable to be described. Report at least name, purpose and structure.

Examples: global variables *RNAenergyrules*, *RNAsequence*, *RNAstructure* and *RNAparams* should at least be described in *∆GLOBALS*, and could be described in more detail in *RNAenergyrules∆*, *RNAsequence∆*, *RNAstructure∆* and *RNAparams∆*.

4.1.4 Local variables

Explain the meaning of local variables when they get the first assignment. Do not overcomment; when the previous prompt explains enough or when the variable is standard such as *T* for temporary variables.

4.1.5 Shared variables

Shared variables must be described briefly (1 line per variable) by *∆SHARED*. Similar to the global variables, more extensive information can be presented separately.

Example: shared variable *CTL* can also be described in *∆CTL*.

4.1.6 Functions

Programming documentation of functions is done in comment. Apply the following 3 types of comment in a function:

1. main comment: document in the very first lines of each function:
   `[1] ⍝⍝` description what the function does
   `[2] ⍝` *R*: description of the right argument
   `[3] ⍝` *L*: description of the left argument
   `[4] ⍝` *Z*: description of the result
   `[5] ⍝` *E*: example of program call
   `[6] ⍝` *F*: names of files called
   `[7] ⍝` *G*: names of used globals and subglobals, if any, but preferably none are present!!!!!!
   `[8] ⍝` *M*: machine-dependent constructions
   `[9] ⍝` *P*: name of programmer, dates of changes (extra lines for important changes)
2. section comments: document purpose, do not paraphrase code.
3. line comment: document purpose, do not paraphrase code.

4.1.7 Change-control

If you do not have a proper version-control system we suggest you apply the variable *∆HISTORY* to document any changes.

Furthermore keep track of changes in the comment of functions. For example:

```
[7] ⍝ P: EvBatenburg, 5-95
[8] ⍝            8-95: Add RNA_test
[9] ⍝   M.v.Baan 1-96: Check stems l.5-10
```

4.2 User documentation

On-line user documentation is useful for experienced users who have forgotten some particular details and want to refresh their memory. This requires that the information is short and compact and can be found easily following standard procedures.

On-line user documentation is also useful for the novice, but here it can have two different purposes. First the user wants to learn if the workspace will attend to his needs (investigation process) and second how to use it or where to find more elaborate information (application process).

The first purpose -investigation- also requires short compact information; only the second purpose needs more extensive information.

Both types of user are properly served with standard names to find their information. For this the following names serves: *ABSTRACT*, *WS*, *DESCRIBE*, *DEMO*, *-D*, *HELP* and *-H* (Help for end-users) or *-∆* (for APL programmers) and optionally *README*. [5]

4.2.1 ABSTRACT

ABSTRACT returns 1 (or at most 2) lines which describes the purpose of the system very briefly. Compare this with the more extensive descriptions in *DESCRIBE* and *HELP*.

If function *LX* does not start function *BEGIN*, then it activates *ABSTRACT*.

For example:

```
⎕FX⊃'Z←ABSTRACT' 'Z←''Simulate whale hunting for....'''
⎕FX⊃'Z←LX' 'Z←ABSTRACT'
```

4.2.2 WS

WS generates 6 lines with the following basic information:

1. ***sid***.
2. complete name of the workspace.
3. version number, release and/or last modification date; anything which uniquely distinguishes this workspace from earlier or newer versions.
4. copyright holder and/or author(s).
5. name, address or telephone number of person or company to be contacted for information.
6. manuals, books and other documentation of the system.

4.2.3 DESCRIBE (for "what?")

Program *DESCRIBE* describes what the Workspace will do: the purpose of the Workspace. Program *DESCRIBE* should inform the reader in a nutshell if the workspace is appropriate for his problem.

It does ***not*** teach the reader ***how*** to work with the workspace (that is the function of *HELP*), but only tells ***what*** the workspace does.

A 1-line summary of this information is given in *ABSTRACT*.

4.2.4 HELP (for "how?")

Program *HELP* describes ***how*** to work with the workspace. It describes the various options of the user-functions and the required syntax.

Keep the size of *HELP* limited to one screen⌊page, whichever is smallest. Leave detailed description to the *-H* and *-∆* descriptions (see nex paragraph).

If the workspace contains several user functions which can be activated by the user independently, than *HELP* at least summarises each user function.

4.2.5 *H-* or *∆-*

If *HELP* is too brief, then you add for each function a separate description in an objects of which the name ends with "*-H*" (if the reader is an application user) or "*-∆*" (if the reader is an APL programmer). Here you explain the particulars of that function. So the syntax of each user function ***sid****xyz* (which should at least briefly be described in "*HELP*") is described more elaborately in ***sid****xyz∆* (for programmers) or ***sid****xyzH* (for end-users).

This documentation is the only elaborate one, because it is primaraly intended for the novice to learn how to operate the various functions.

If *∆GLOBALS* is too brief then you add for each global variable a "*-∆*".

Examples:

- function *WHLSIMULATE* has its syntax described in *WHLSIMULATE∆* (or *WHLSIMULATEH*),
- function *RNAFOLD* in *RNAFOLD∆*.
- global variable *RNAparams* is described in *RNAparams∆*.

---

5) The original ASWI standard prescribes the use of: ***sid***, resulting in names: *∆**sid**ABSTRACT*, *∆**sid**WS*, *∆**sid**DESCRIBE* and *∆**sid**HOW*.

4.2.6 DEMO and *-D*

If possible, add program *DEMO* that demonstrates the main features of the workspace. Although this takes a lot of work in general, such an investment is often the best documentation.

A similar argument holds for user-functions. A *H*- or ∆-description is fine, but a demonstration often quicker conveys both the purpose and the required syntax.

If possible add a *D*-function that *demonstrates* the user-function. Such a demonstration function could have 3 purposes together:

1. its execution show the various possibilities
2. its listing shows the required syntax
3. after each change, it can be used to test if everything is still working as it should

4.2.7 README

Any recent changes which are not documented in the appropriate places can be explained by *README*.

4.3 Paper documentation

Documentation on paper can be very elaborate or very brief. This depends on many things (personal favour, size of organisation, habits of organisation). My personal minimum preference is:

1. A drawn diagram of the functional design.
2. Workspace status:
)FNS, *)VARS*, *)GRPS*, *)SYMBOLS*, *)SI* (should be empty), *⎕WA*, *⎕PW*, *⎕IO*, *⎕LX*.
   Give special attention to deviations of normal status such as *⎕IO*=0 or *⎕PP*=5 or *⎕CT*=0.
3. Print of the on-line user documentation: *ABSTRACT*, *WS*, *DESCRIBE*, *DEMO*, *HELP* and *README*.
4. Print of the on-line programmer documentation: *∆SYSTEM*, *∆GLOBALS*, *∆FILES*, *∆SHARED*, *∆HISTORY*, component 0 of each component file.
5. Function-function-matrix that shows for each function, which functions it calls.
6. Function-variable-matrix that shows for each function the variables it uses.
7. Summary of function information (one line for each function with: name, size, monadic/niladic/dyadic, result/not, fixing date.
8. Indented listing of function call structure with name and purpose (1th comment line?) of each function.
9. For each function one page:
   - *H*-variable of ∆-variable with description,
   - listing (preferably blocks indicated),
   - *XREF* and
   - *NESTING* of that function
   - *D*-function listing for syntax-demonstration
10. A list of error-messages to ensure consistency. This can be generated automatically when *uERRMSG* was used.
11. A list of all 1-line prompts to ensure consistency. This can be generated automatically when *uIN* was used..

5. MESSAGES

Messages are still under discussion. We welcome anybody who has suggestions. The following is a potpourri of opinions that are not yet integrated.

6

1. As mentioned in "coding and detail design": try to avoid error messages. Ask yourself what you can do to prevent the need for an error message. If you know enough to detect an error, try to prevent it
2. As mentioned in "coding and detail design": try to prevent confirmation messages. Mostly you need them to warn the user that he is doing something dangerous. Rather then spend your time on confirmations, invest it to make the operation reversible.
3. Distinguish between two different types of readers for your messages: application-users and application-builders (programmers).
   Builders need to know what went wrong and where. Do not use APL-like message like "DOMAIN ERROR" which might confuse the reader into thinking that an APL-domain error occurred. Instead produce a messages like "***mFILEINOUT: domain error in right argument".

   The whole purpose of your message to ***users*** should be to tell them ***what to do***, rather than telling what went wrong (who cares!). So never generate messages like

   OSMOO7E BOX IDENTIFICATION IN USE

   Rather say something like:

   Unable to access box, so check if another program uses box and remove that link.
4. If an error is recoverable by your program, do so immediately where the error is discovered. Do not jump to a general error-section. If a test discovers an error, this is the place where the first action should be taken. Here the reader should find the problem and its reaction together.
   So, always deal with errors and other conditions right after detection.
5. A sub-function may well be unhappy with an exceptional condition, but its caller might better be suited to react on such a condition. For this reason:

---

6) The original ASWI stated the following:
Also messages fall in different categories; we find:
1. Information.
2. Questions.
3. Warnings. A warning is issued when an abnormal condition is encountered but processing continues (may be in a slightly different way).
4. Errors. An error is a situation which makes it impossible to continue processing. So processing stops and user action is required. Usually an error can be corrected by the user.
5. Severe errors. A severe error is an error that cannot be corrected by this system nor the user. Further processing is impossible.

Another way to look at messages is to recognize that for an expert user the messages should be as short as possible, while for the novice user long, explanatory messages may be required.
Taking all of the above into consideration we propose the following message identification:
1. first three characters: ***sid***.
2. next three characters: unique message number
3. next one character
   - I-information
   - Q-questions
   - W-warning
   - E-error
   - S-severe error
4. next one character
   - blank - only one text
   - S-short text
   - L-long text
5. next one character: blank
6. The rest is the message text, which is free format.
   An example of such a message is:

   OSMOO7E BOX IDENTIFICATION IN USE

- If a sub-function cannot correct an error, ***avoid printing*** of an error message too soon (too low in the functional hierarchy). Choose return error-information as explicit result instead. For example: `''` (empty vector) instead of computed data, scalar error number instead of computed array, or error message.
- Avoid to let a sub-function halt the system by "→".
- Use errorcode 0 (zero) or `⍳0` for a correct process.

6. Route all your messages through a cover function in order to maintain consistency in the layout. For example "*L uERRMSG R*" has the severity of the error in *L* and the text in *R*. For example:
   uOUT `3` *uERRMSG*'No file; restart.'

6. MISCELLANEOUS

Several design choises are based on human experience with handling your program. This is very important and the list could (should?) be very long. This is a selection of what I considered the most important, but any suggestions for additions are welcome.

---

1. User-invoked functions should have short names that are meaningful and mnemonic.
2. Make user-functions niladic and request input via prompting messages. So the user doesn't need to remember any syntax.
3. Formulate the choise between alternatives in a positive style so that result of Yes or No is clear.
4. Indicate the required answers (Y/N or other) in the input request.
5. Always supply defaults so the user can hit [CR] if he is satisfied with the default. Indicate the default taken in the message. Choose for the defaults...
   - the previous choosen alternative,
   - the most common choise, or
   - the least dangerous choise.
6. Limit keyboard input to numbers and characters that are on any keyboard, do not use APL keyboard if you can avoid it. Process capitals- and lowercast indisciminately.
7. Prefix error messages with "***" so that they stand out clearly. Use *uERRMSG* to do this.
8. Do not let the user wait many seconds (or even minutes) between each question. Better ask all questions first and then start computing.
9. For lengthy computations, provide progress information such that the user knows the program is working. Preferably supply information about the expected time that is left.
   Examples are
   - a counter running backwards to zero
   - report of remaining expected computation time
   - removing a series of dots one by one.
10. For big applications that should be very reliable, you should design quality control functions that test each important function and returns ‘OK’ if the test works out well.. For example a main test function could be:

```
  ∇TSTALL
[1] TSTRNA_READ
[2] TSTRNA_FOLD
[3] TSTRNA_DISP1
[4] TSTRNA_PUT∇
```

7. SUMMARY

The main points are summarised very brief and sketchy here.

7.1 NAMING CONVENTIONS

- Systems & System IDentifier: 3 capitals; related workspaces identical ***sid***.
- Workspaces: ***sid***+≤5 capitals.
- User functions: ***sid***+≥2 capital.
- Sub-functions: ***sid***+"_"+≥2 capital.
- Imported sub-functions: no name change unless changed.
- Standard names: *LX* (with *BEGIN* or *ABSTRACT*), *BEGIN* (calls main function), *RECOVER* (repairs returns 1↑⎕*LC*), *TESTALL* (executes a test-bed) and *ABSTRACT DEMO DESCRIBE HELP WS* and *README* (documentation for users of the workspace) Δ*FILES* Δ*GLOBALS* Δ*HISTORY* Δ*SHARED* Δ*SYSTEM* and –Δ (documentation for programmers) –*H* (help for end-users) and –*D* (demonstration)..
- Groups: name of main item + *GP*
- Labels: *B*#:..*E*#: *BE*: *XX*: or *EXIT*:
- Arguments: *Z F L R G*
- Global variables: ***sid***+≥3 lowercast
- Sub-global variables: ***sid***+"_"+≥3 lowercast
- Local variables: ≤4 capitals, *T* for line-locals, *S* for line-spans, *I*,*J*,*K* for counters, never *Z F L R G*.

7.2 CODING

- Branching:
  think in blocks (delimited by *B*#:...*E*#:), not in jumps;
  never "→()/0" but always "*XX*:→()/*EXIT*".
- Comments: use line comment and section comment. Always start function with documentation of
  [1] ⍝⍝ purpose,
  [2] ⍝ *R*: right argument
  [3] ⍝ *L*: left afgument
  [4] ⍝ *Z*: result
  [5] ⍝ *E*: tiny example
  [6] ⍝ *F*: called files
  [7] ⍝ *G*: used globals
  [8] ⍝ *M*: machine-dependent constructions
  [9] ⍝ *P*: programmer and last modification date
- Cover functions: hide...
  1) system-dependency
  2) machine dependency
  3) global data (file or variable).
- Default settings: localise deviations of defaults.
- Efficiency: important, but readability comes first.
- Error-messages: routed (via *uERRMSG*).
- Errors: return info as explit result.
  Terminate with "→*EXIT*", not with "→".
- Error/condition span: keep it as short as possible.
- Execute: limit use as much as possible.
- Function arguments: control in left argument (always try applying defaults), data at the right.
- Function size: screensize ⌊papersize.
- Globals and subglobals: avoid if possible.
- ISO APL: stick to ISO APL.
- Input: preferably routed (*uIN*). Supply defaults.
- Output: preferably via explicit result. If not, preferably routed (via *uOUT*).

- Restartability:
  - do not reassign *L* and *R*
  - do not assign used variables in middle of line but use line-local *T*.
- Robustness: consider all ranks and shapes, especially 0.
- Variable span: keep it as short as possible.

7.3 DOCUMENTATION

7.3.1 Programmers documentation in WS

- System: ∆*SYSTEM*.
- Files: ∆*FILES* and -∆ for each file. For component files: comp.-1: identifier, application name, version, comp.-2: file purpose and structure, comp.-3: component names, number and purpose, comp.4-10: spare.
- Global variables (preferably absent): ∆*GLOBAL* and -∆ for each global variable.
- Local variables: document at first assignment.
- Shared variables: ∆*SHARED* and -∆ for each shared variable.
- Version-control: ∆*HISTORY* and in "[]⍕ *P*:"-line of each function.
- Functions:
  1) heading commentlines
  2) section comments
  3) line comments.

7.3.2 User documentation in WS

- *ABSTRACT*: 1 line with WS purpose.
- *DESCRIBE*: elaborated purpose.
- *HELP*: brief description of syntax.
- -∆: programmer help : purpose + syntax.
- -*H*: enduser help: purpose + syntax.
- *DEMO*: illustrates main functions.
- -*D*: function demonstration.
- *README* (preferably absent): last minute advice
- *WS*:
  1) sid
  2) Wsname
  3) version+release+modif.date
  4) copyrightholder+authors
  5) adresses for ordering and for information
  6) references to documentation.

7.3.3 Programmer documentation on paper

- A drawn diagram of the functional design.
- Workspace status:)*FNS*,)*VARS*,)*GRPS*,)*SYMBOLS*,)*SI* (should be empty), ⎕*WA*, ⎕*PW*, ⎕*IO*, ⎕*LX*
- User documentation: *ABSTRACT*, *WS*, *DESCRIBE*, *HELP*.
- Programmer documentation (∆*SYSTEM*, ∆*GLOBALS*, ∆*FILES*, ∆*SHARED*, ∆*HISTORY*, component 0 of each component file).
- Xref between functions calling functions.
- Xref between variables called in functions.
- Functions summary
- Indented function call structure (name+purpose).
- Summary of all global variables
- Function listings.
- Listing of all 1-line prompts.
- Listing of all error-messages.

7.4 MESSAGES

- Report to end-user what *to do*, rather than what happened.
- Report to programmer *what* error and *where*.
- Deal with errors right after detection.
- Print as late (high in functional hierarchy) as possible.
- Do not end with "→", but with "→*EXIT*".
- Route all error messages through cover function (for example *uERRMSG*).
- Report 0 or "⍳0" for correct termination.

7.5 MISCELLANEOUS

- user-function: short, meaningful names
- user-function input: via prompting, routed through cover function, defaults (indicated in message), no capitals/lowercase distinction
- user-function output: routed through cover function
- programmer-function input: via parameters, use defaults, accept wide range of ranks and lengthes
- programmer-function output: explicit result
- waiting time: give progress info, no long waiting between input requests, backwards counter.
- in errors: inform user what to do (not what happened).
- quality: design test functions

8. EXAMPLE

Examples can sometimes clarify what is still unclear in lengthy descriptions. For this reason we show several examples as well as anti-examples.

Workspace: MCURSOR.W3:

```
      )LOAD 1 MWIN
1 MWIN SAVED 10/15/1998 19:30:38
APL⋆PLUS/W Machine/vendor-Dependant-Functions for menus:
mWININI mWINMENGET mWINEND mWINOUT.
See DEMO, DESCRIBE and HELP for more information.

============================================================

      ⎕LX
LX

============================================================

     ⎕CR'LX'
Z←LX
Z←ABSTRACT

============================================================

     ABSTRACT
APL⋆PLUS/W Machine/vendor-Dependant-Functions for menus:
mWININI mWINMENGET mWINEND mWINOUT.
See DEMO, DESCRIBE and HELP for more information.

============================================================

     ⎕CR'WS'
WS;J;T
'WSID: mWIN'
'WSname: MWIN'
'WSdate/version: ',,J[T⍳⌈/T←0 12 31 24 60 ⊥⍉J←5↑[2]2 ⎕AT ⎕NL 3;]
'WSauthor: F.H.D. van Batenburg    '
'          Inst.Theoretical Biology'
'          Kaiserstraat 63         '
'          Leiden, The Netherlands '
'WSdocumentation: DESCRIBE DEMO HELP'

============================================================

     ⎕CR'BEGIN'
         ⍝ not present because not one main program

============================================================

     ⎕CR'README'
         ⍝ not present because no last minute info

============================================================

     ⎕CR'RECOVER'
         ⍝ not present because no vulnerable parts in WS

============================================================

     ⎕CR'TESTALL'
⎕←'Use demonstration programs to test functions.'

============================================================
```

```
      DESCRIBE
Calling "Z←mWINMENGET R" displays menu R on screen and returns the
number of the choosen menu-item in Z. Calling "mWINOUT R" sends R
to display window.

The machine-dependent functions in this workspace are all for menu
and output control: mWININI mWINMENGET mWINEND mWINOUT.
They can be embedded in other programs whenever you need menus.

They use standard arguments which are described in HELP and are
illustrated by program DEMO.

This workspace contains only the APL2000/W programs.
If one switches to another machine, the parent programs do not need
to be changed. One should only import the mWINGP that is written
for that other machine.
The alternative workspaces are MWINMAC for Macintosh,
                               MWIN.AWS for PC/FreeAPL,
                               MWIN.DWS for PC/Dyalog
                               MWIN.W3 for APL2000/W.

============================================================

      ⎕CR'DEMO'
DEMO
⍝ Demonstrate window with menus and output
⍝ E: DEMO
⍝ P: EvBatenburg 9-9-98
mWIND
'End of DEMO.'

============================================================

      ⎕CR'mWIND'
mWIND;HEAD;M;T
⍝ Demonstrate window with menus and output
⍝ E: mWIND
⍝ P: EvBatenburg 9-9-96
⍝               11-9-97: removal gloab variables
M←'1 &INFO?' '2 &About /aA' '1 &File' '2 &Quit /sQ'
M←M,'1 C&heck' '2 Flip/cF' '2XFlop/CF'
M←⊃M,'1 Visual' '2 Changevision' '2--------' '2-invisible' '2 Visible'
⎕←'-------------------------------------------------'
⎕←'We are going to set the windows title and a menu'
⎕←'First enter small text for windows title:'
HEAD←⍞
⎕←'-------------------------------------------------'
⎕←'OK, now the menu. The matrix used is: '
(M[1 2;])(M[3 4;])(M[5 6 7;])(M[8 9 10 11 12;])
⎕←'-------------------------------------------------'
T←⍞,∈⎕←⊃'Press return for...'(,(⎕CR 'DmWIN')[2+1↑⎕LC;])
T←HEAD 10000 mWININI M
⎕←'-------------------------------------------------'
⍞←⊃'Now we will execute...'(,(⎕CR 'DmWIN')[2+1↑⎕LC;])
B1:mWINOUT⊃'' 'Click any menu item, terminate with [File/Quit].'
T←mWINMENGET M
mWINOUT∈'You clicked on item: ',T,': ',2↓M[T;]
BE:→(~T∈6)/⎕LC+1 ⋄ M[6;2]←' X'[1+' '=M[6;2]] ⋄ mWINOUT 'Notice toggle change '
BE:→(~T∈7)/⎕LC+1 ⋄ M[7;2]←' X'[1+' '=M[7;2]] ⋄ mWINOUT 'Notice toggle change '
BE:→(~T∈9)/⎕LC+1 ⋄ M[11 12;2]←'- '[1+'-'=M[11 12;2]] ⋄ mWINOUT 'Notice visibility cha
nge.'
mWINOUT 'Observe that intermediary-menu is active for 1 second;'
mWINOUT 'normally this would be a time of computation.'
0⍴⎕DL 1
E1:→(4≠T)/B1
mWINOUT⊃'' '-------------------------------------------------'
mWINOUT 'Now a different menu with the same windowtitle. Menu is:'
⎕←M←DEMOmenu
B2:mWINOUT⊃'' 'Click any menu item, terminate with [File/Quit].'
T←mWINMENGET M
mWINOUT∈'You clicked on item: ',T,': ',2↓M[T;]
0⍴⎕DL 1
E2:→(9≠T)/B2
mWINOUT '-------------------------------------------------'
```

```
B3:mWINOUT⊃'' 'Now the same but please,  terminate by closing window.'
T←mWINMENGET M
mWINOUT∈'You clicked on item: ',T
0⍴⎕DL 1
E3:→(¯1≠T)/B3
mWINOUT '-------------------------------------------------'
mWINOUT 'Now look at the window; we are going to test mWINOUT.'
T←⍞,∈⎕←⊃'Press return for...'(,(⎕CR 'DmWIN')[2+1↑⎕LC;])
1 mWINOUT 'This sentence must be in ' ◇ 1 mWINOUT 1 ◇ mWINOUT ' line.'
mWINOUT '-------------------------------------------------'
T←⍞,∈⎕←⊃'Press return for...'(,(⎕CR 'DmWIN')[2+1↑⎕LC;])
0 mWINOUT 'One' ◇ T←⎕DL 1 ◇ 0 mWINOUT 'Two' ◇ T←⎕DL 1 ◇ 0 mWINOUT 'Three'
mWINOUT '-------------------------------------------------'
T←⍞,∈⎕←⊃'Press return for...'(,(⎕CR 'DmWIN')[2+1↑⎕LC;])
¯1 mWINOUT 'This line should be on a new, clean page.'
mWINOUT '-------------------------------------------------'
T←⍞,∈⎕←⊃'Press return for...'(,(⎕CR 'DmWIN')[2+1↑⎕LC;])
mWINOUT 'This is line 1',[1](99 13⍴' '),1+⍳99 ◇ mWINOUT 'Can you see line 1?'
mWINOUT '-------------------------------------------------'
T←⍞,∈⎕←⊃'Press return for...'(,(⎕CR 'DmWIN')[2+1↑⎕LC;])
mWINOUT(99 13⍴' '),100+⍳99 ◇ mWINOUT 'Can you see line 1?'
mWINOUT '-------------------------------------------------'
T←⍞,∈⎕←⊃'Press return for...'(,(⎕CR 'DmWIN')[2+1↑⎕LC;])
mWINOUT(99 13⍴' '),200+⍳99 ◇ mWINOUT 'Can you see line 1?'
mWINOUT '-------------------------------------------------'
mWINOUT 'Now we restore the original menu and situation.'
T←⍞,∈⎕←⊃'Press return for...'(,(⎕CR 'DmWIN')[2+1↑⎕LC;])
mWINEND

============================================================

     ▽HELP
Z←HELP
⍝ Help for mWIN workspace
⍝ E: HELP
⍝ P: EvBatenburg 15-10-98
Z←mWIN△

============================================================

     mWIN△
Create window and handle its menus and output.
L mWININI R
Z←mWINMENGET R
  mWINOUT R
  mWINEND
Prog.: F.H.D. van Batenburg 8-95.
       Institute for Theoretical Biology,
       Leiden University, The Netherlands.

First you should create a window and set the menu by mWMENINI
Next
-you can activate and let user choose a menuoption by calling
 mWMENGET
-you can send output to "menu"-frame using mWOUT
Finally you restore the original situation using mWMENEND

L mWININI M
Initialise window with caption and buffersize L and menu M.
Also set normal keyboard on.
Example: 'Example' 30000 mWININI M

Z←mWINMENGET M
Whenever you are interested  in  user  choise, activate
mWINMENGET. It gives you ¯1 if user has closed window,
otherwise a number that refers to the option of menu-item M[Z;].

mWINOUT R
Send text R to window created by mWININI.

mWINEND
Program mWINEND negates everyting done in mWININI.

A menu variable M as used in above programs is an array that
looks like:
1: '1 &Desk    ' <=Main item
```

```
2: '2 &About    ' <= option of item Desk
3: '1 &File /aO' <=Second main item
4: '2 &Save     ' <= 1th option of item File
5: '2 Save &as  ' <= 2th option
6: '2-separate  ' <= desactivated text
7: '2----------' <= separator
8: '2XNormal    ' <= 3th option with checkmark
So...
...column 1 determines level.
...column 2 determines normal(=blanc)/toggle on(=X)/
                       disable(=-).
...columns 3-.. itemtext ('-' is reserved for separator).
...inclusion of & for default (not in Mac).
...addition of / K for shortcut key K; /xK where x is a/c/A/C
   for alt/control/shift-alt/shift-control (limited in Mac)

See DEMO for demonstration.
See mWINGP for list of required programs.

============================================================

Z←mWINMENGET R;F;S;T;WN
⍝ APL⋆PLUS/W: return menu-itemnumber which was selected
⍝ R[;]: Menu-matrix (ignored in APL⋆PLUS/W)
⍝ Z: Choosen menu-itemnumber
⍝ G: mWINglobal[1]=Formname [2]=Option [3]=buffersize
⍝ E: 2 ↔ mWINMENGET ⊃'1 &Main' '2 Sub&1⍋sA' '2 Sub&2' '1 Main2' '2 Sub3'
⍝ P: EvBatenburg 11-95
⍝             JNg 01-97: / for shortcut changed to ⍋
⍝                 01-98: use ('separator' 1) instead of ('caption' '----')
((,'⍋'=R)/,R)←⎕TCHT
WN←1↓1⊃mWINglobal
B1:→((⍴WN)=1↑⍴R)/E1+1   ⍝ ---------- Replace menue
T←((1=∈+/¨WN='.')/WN) ⎕WI¨⊂'Delete'
F←1 1⊃mWINglobal
WN←T⍀(T←R[;1]≠'1')≠'.','O',⊃⍕¨⍳↑⍴R          ⍝ Object Fam.tree
WN←(⊂F,'.M'),¨(⊂[2](⍕+⍀R[;,1]='1'),WN)~¨' '
T←WN ⎕WI ¨(↑⍴R)⍴⊂'New' 'Menu'               ⍝ Menu properties
E1: mWINglobal[1]←⊂(⊂F),WN
                          ⍝  ---------- Menu properties
T←(⊂'caption'),[1.5](+/∨\⍎R≠' ')↑¨⊂[2]0 2↓Z←R
BE:→(0=⍴S←(∧/'-'=(⊃T[;2])[;⍳3])/⍳1↑⍴T)/⎕LC+1 ⋄ T[S;1]←⊂'separator' ⋄ T[S;2]←1
T←WN ⎕WI¨⊂[2]T
T←WN ⎕WI¨⊂[2](⊂'onClick'),[1.5](⊂'mWINglobal[2]←'),¨⍕¨⍳↑⍴R
T←WN ⎕WI¨⊂[2](⊂'enabled'),[1.5] ('-'≠R[;2])⍝ Active/disabled
T←WN ⎕WI¨⊂[2](⊂'value'),[1.5]~R[;2]∈' -'   ⍝ Toggle
S[;1]←' scCaA'⍳(T,1)↑S←((T←1↑⍴R),2)↑⊃(1-+/∨\⎕TCHT=R)↑¨⊂[2]R
T←(T/WN)⎕WI¨(T←S[;2]≠' ')/⊂[2](⊂'shortcut'),S[;2 1]

mWINglobal[2]←0          ⍝ ---------- Get user-respons
(1 1⊃mWINglobal)⎕WI'Focus'
BE:⎕WGIVE 0 ⋄ →(0=Z←2⊃mWINglobal)/⎕LC
T←WN ⎕WI¨⊂[2](⊂'enabled'),[1.5]R[;1]≠'1'

============================================================

other listings skipped

============================================================

      ⍙HISTORY
------+-----------+-------------------------------------------------
090998|EvBatenburg|DEMO → DEMO+DmWIN
  0198|JNg        |mWINMENGET: use ('separator' 1) instead of ('caption' '------')
  1297|JNg        |mWININI: Maximized Window ('where' 0 0 30 84 → 'visible' 3)
  0197|JNg        |mWINMENGET, mWININI:'/' for shortcut changed to '⍋'
  1095|EvBatenburg|mWMENGET shows interruptmenu and apple
  0895|EvBatenburg|mMENRPL mMENACT mMENCHK and part of mWMENINI into mWMENGET
      |           |mMENGET returns not 0 but waits for proper number.
      |           |ABSTRACT DEMO DESCRIBE mMENUGP updated.
  ..93|EvBatenburg|Initial design in MMMENU and MMMENU2

============================================================
```

```
      mWINGP
mWINEND
mWINMENGET
mWININI
mWINOUT

=============================================================

      ∆FILES    ⍝ not present because no files used
VALUE ERROR
      ∆FILES
      ^

=============================================================

      ∆GLOBALS
mWINglobal: [1-3] N.A. variable made by mWININI, used by
    other mWIN-functions and removed by mWINEND.
    It stores Window information
    mWINglobal[1]=Formname [2]=Option [3]=buffersize

=============================================================

      ∆SHARED   ⍝ not present because no shared variables
VALUE ERROR
      ∆SHARED
      ^

=============================================================

      ∆SYSTEM   ⍝ not present because no special system info
VALUE ERROR
      ∆SYSTEM
      ^
```

9. REFERENCES

Many ideas in this standard were copied or at least inspired by Holls and to a lesser account Bergquist. Component file ideas were taken from Cannon.

- Bergquist,G.A.(1987): Advanced techniques and utilities. ©Zark.
- Cannon,R. (1992): Suggested standards for component file systems at a commercial APL site, Vector 8(2)105-110.
- Holls,T.P. (1981): APL programming guide: programming conventions for APL application workspaces. IBM Poughkeepsie NY G320-6735-0