The NAPA Primer

a Guide to Reuse your Solutions

NAPA Version 5.02

Yves Leduc.



(*)

(*) Logo designed by Gary Anderson

Table of Contents

Table of Contents....2
Introduction....5
Built-in extension hooks....6
Cells....8
WHEN?....8
A FIRST EXAMPLE....8
A SECOND EXAMPLE....10
ADVANCED USERS....11
HOW IT WORKS....13
Data cells....15
WHEN?....15
A FIRST EXAMPLE....15
A SECOND EXAMPLE....18
HOW IT WORKS....23
Generators....25
WHEN?....25
A FIRST EXAMPLE....25
HOW IT WORKS....28
ADVANCED USERS....28
User-defined C macros....29
WHEN?....29
A FIRST EXAMPLE....29
A SECOND EXAMPLE....29
HOW TO DEFINE A DEFAULT VALUE?....30
User-defined C functions....31
WHEN?....31
A FIRST EXAMPLE....31
A SECOND EXAMPLE....32
METHODOLOGY....32
HOW IT WORKS....33
“algebra”, “dalgebra”, “ialgebra”....34
WHEN?....34
AN EXAMPLE....34

HOW IT WORKS....36
“napa_init( )”, “napa_close( )”....37
WHEN?....37
AN EXAMPLE....37
HOW IT WORKS....38
“napa_loop_option(*long*)”....39
WHEN?....39
AN EXAMPLE....39
HOW IT WORKS....40
“call”....41
WHEN?....41
AN EXAMPLE....41
HOW IT WORKS....42
“duser”, “iuser”....44
WHEN?....44
A FIRST EXAMPLE....44
A SECOND EXAMPLE....46
HOW IT WORKS....49
ADVANCED USERS: OPTIONS....50
“dtool”, “itool” or “tool”....52
WHEN?....52
AN EXAMPLE....52
HOW IT WORKS....56
ADVANCED USERS....58
“post”....59
WHEN?....59
AN EXAMPLE....59
HOW IT WORKS....61
ADVANCED USERS....61
Options....62
WHEN?....62
AN EXAMPLE....62
HOW IT WORKS....63
Multiple Output User’s Functions....65
WHEN?....65
AN EXAMPLE....65

HOW IT WORKS....68
Passing Parameters by Addresses....69
WHEN?....69
AN EXAMPLE....69
HOW IT WORKS....70
Conclusion....72

Introduction

NAPA is open to extensions. This guide describes how users can customize the simulator to their applications and how they can build libraries of REUSABLE SOLUTIONS. Several examples are given and commented.

> We encourage the users to build libraries of reusable solutions and make them available to others.



Figure 1 - Make your work REUSABLE

Built-in extension hooks

Several hooks are built-in to extend ***NAPA***. Customization is allowed at several levels:

1. at the netlist level, using cells or data cells
2. at the netlist level, using generators
3. in *C* expressions, by using user-defined C functions
4. at the node level, using node "*algebra*", "*dalgebra*" or "*ialgebra*"
5. adding code to functions '*napa_init( )*', '*napa_close( )*' and '*napa_loop_option( )*'
6. at the variable level, using *C* functions through "*init*" and "*call*"
7. at the node level, using node "*iuser*" or "*duser*"
8. at the output level, using instruction "*post*"
9. at the node level, using node "*itool*" or "*dtool*" or instruction *"tool"*

It is important to understand what type of solution to choose and where to place the code. This booklet will explain in detail how the ***NAPA*** simulator interacts with the user-defined extensions and how to build solutions as powerful and as user-friendly as possible. Depending on the choice, the effort and the return on investment vary widely:

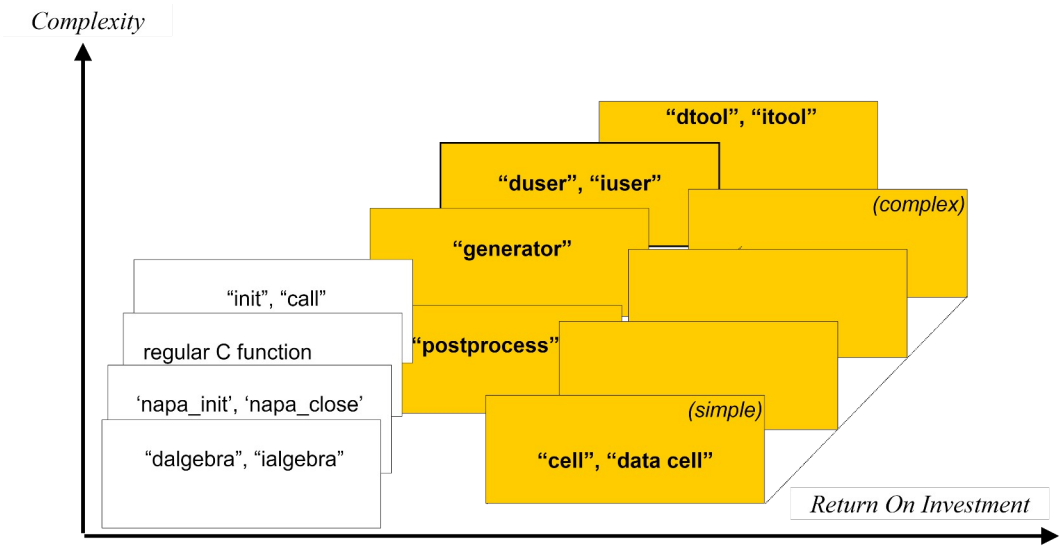


Figure 2 - Return On Investment

NOTE:

There is a general rule to remember. During the parsing, the ***NAPA*** transpiler has full access to the content of the netlists (main file, cells or data cells), but it has no way to parse the content of the header files. Using macro definitions, the ***C*** simulator built by the ***NAPA*** transpiler can communicate information to the header files. With proper switches, the macro preprocessor can configure the code located inside the

header files. But there is no way to obtain from the ***NAPA*** transpiler the smallest understanding of what is inside the header files. Therefore, the building of the ***C*** code of the simulator itself cannot be affected by the code of these headers. We will see that this unidirectional flow of information does not limit the power of the simulators we are building.

Please refer to the ***NAPA*** user's guide for further information (instructions, primitives, syntax, ...) concerning the ***NAPA*** transpiler.

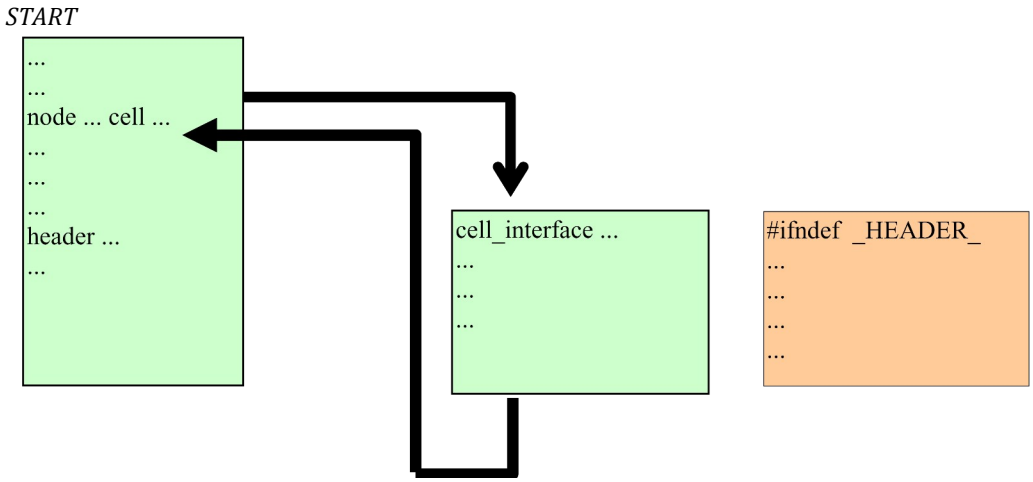


Figure 3 - Cells are parsed by NAPA, Headers are not!

Cells

The user can describe a reusable netlist using the pseudo-node "*cell*". The cell is instantiated building a hierarchical description. There is virtually no limit to the hierarchical depth, as long as it makes sense to do it.

WHEN?

You should consider partitioning your netlist if your problem can be described as a set of existing ***NAPA*** primitives (like nodes, variables...) and it represents a solution reusable immediately (multiple instantiations in your netlist) or is generic enough to be possibly reused in the future.

Collect the reusable network and write a cell (pseudo-node "*cell*"). The cell is ideally suited to describe the topology of the structure. As the same cell could describe several variants differing only by the values of their parameters, it is sometimes preferable to move the values of the cell parameters to a specialized primitive, the "*data cell*" (the "*data cell*" is described in the next chapter).

A FIRST EXAMPLE

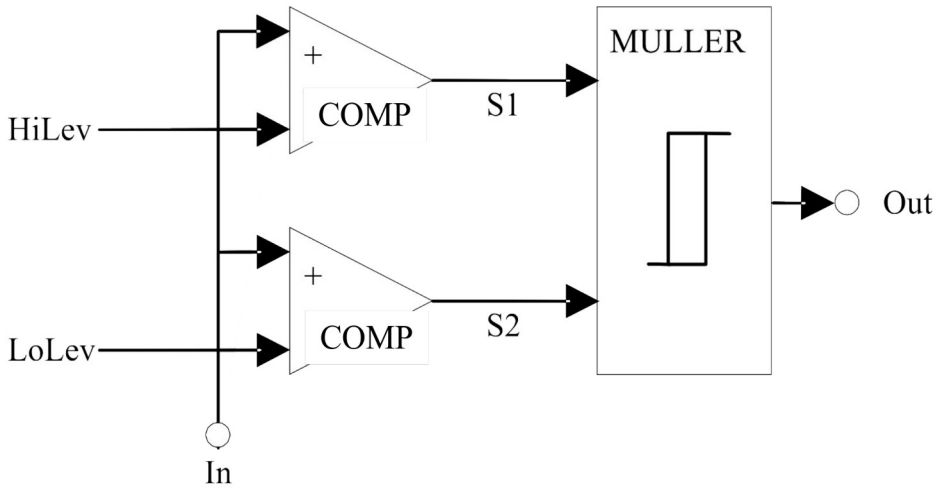


Figure 4 - The hysteresis comparator

File "hyst.net"

```
cell_interface $out  $in  $hilev $lolev

declare () $in $hilev $lolev

#* single-ended synchronous hysteresis comparator
node $s1   comp   $in  $hilev
node $s2   comp   $in  $lolev
node $out  muller $s1  $s2                  // digital hysteresis
```

This cell is described in a file, "hyst.net", which will be placed either in the project directory, if we consider that this is not a generic cell, or in the user's directory or the team's cell directory to be available for other applications. A declaration is used to document the cell and for better error messages if any.

The cell can be used in another cell or inside the main netlist:

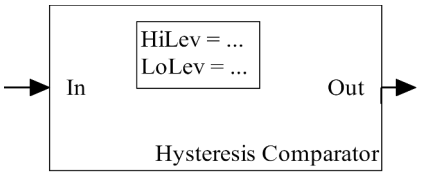


Figure 5 - The hysteresis comparator as a cell

File "ex01.nap"

```
title "the hysteresis comparator"

header <napa.hdr>

fs  1.0e6

dvar freq 1000.0

node s1 sine 0.0 1.0 freq 0.0
node s2 cell cmph "hyst.net" s1 0.20 0.30

output "stdout"  s1 s2
terminate  TIME > (1.50 / freq)
```

The ***NAPA*** transpiler parses the cell in a process transparent to the user. However, there is a way to understand how this parsing is done, with an option of the transpiler. Option **'-e'** asks for an expansion of the netlist without transpilation (this option exists only for this purpose):

```
%   NAPA ex01.nap -e > ex01a.nap
```

The resulting netlist "ex01a.nap" contains (here and below, the banner lines are shortened):

File "ex01a.nap"

```
#* *****  EXPANSION OF FILE  ex01.nap  ***** START ***** *#


title "the hysteresis comparator"
header <napa.hdr>
fs  1.0e6
dvar freq 1000.0
node s1 sine 0.0 1.0  freq 0.0

#* >>  node s2 cell cmph "hyst.net" s1 0.20 0.30

declare () s1 0.20 0.30
node cmph__s1   comp   s1  0.20
node cmph__s2   comp   s1  0.30
node s2  muller cmph__s1  cmph__s2

#* <<

output "stdout"  s1 s2
terminate  TIME > (1.50 / freq)
```

```
#* *****  EXPANSION OF FILE  ex01.nap  ****** END ****** *#
```

In this example, the flattening process is straightforward. The output of the simulation is:

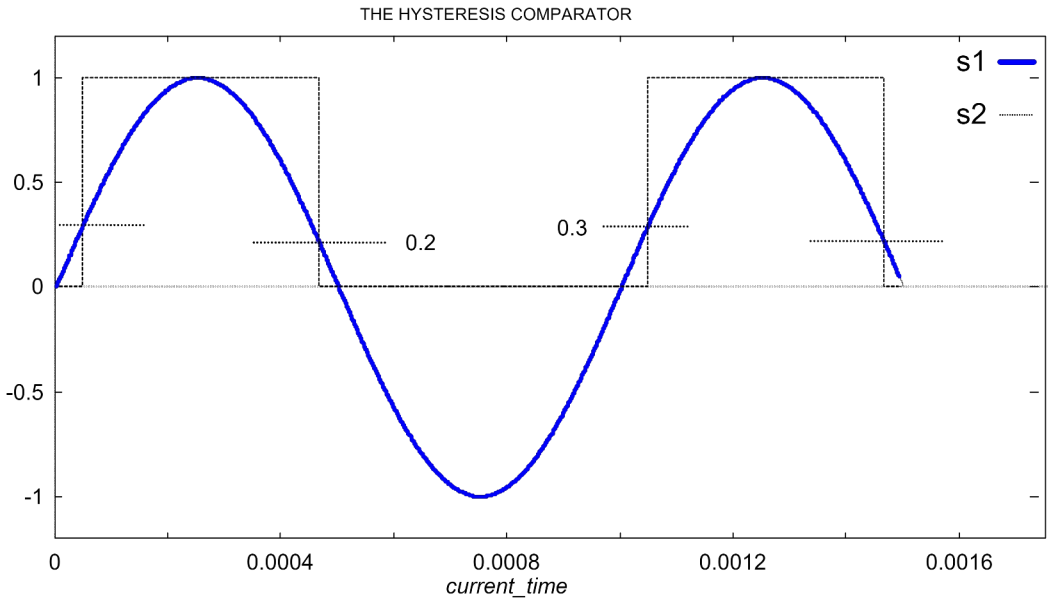


Figure 6 - The simulation of the hysteresis comparator

A SECOND EXAMPLE

The previous example shows a cell with a single output. If we have to describe a more complex cell with multiple outputs, it is recommended to follow another syntax:

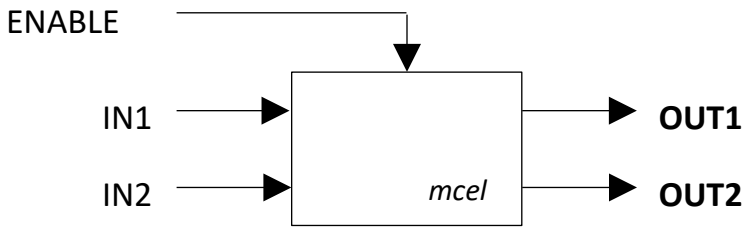


Figure 7 - An example7 of a cell with multiple outputs

File "mcel.net"

```
cell_interface $unused $out1 $out2 $enable $in1 $in2

#* this cell has 3 inputs and 2 outputs

node $s1    and  $enable $in1
node $s2    and  $enable $in2
node $out1  nor  $s1 $s2
node $out2  nand $s1 $s2
```

Here is an example of an instantiation of this cell. Note the usage of the specific identifier "*void*" corresponding to the dummy formal parameter "$unused" of the cell. This style is not mandatory but stresses the fact that the cell has multiple outputs.

File "ex02.nap"

```
header <napa.hdr>

fs  100.0e3

node a clock "10"
node b clock "1100"
node e clock "11110000"

node void cell pls "mcel.net" c d e a b

output "stdout" c d e a b
terminate LOOP_INDEX > 16LL
```

Again, let us look at the flattened description:

```
%   NAPA ex02.nap -e > ex02a.nap
```

The resulting netlist contains:

File "ex02a.nap"

```
#* *****  EXPANSION OF FILE  ex02.nap  ***** START ***** *#


header <napa.hdr>
fs  100.0e3
node a clock "10"
node b clock "1100"
node e clock "11110000"

#* >>  node void cell pls "mcel.net" c d e a b

node pls__s1   and e a
node pls__s2   and e b
node c nor  pls__s1 pls__s2
node d nand pls__s1 pls__s2

#* <<

output "stdout" c d e a b
terminate LOOP_INDEX > 16LL


#* *****  EXPANSION OF FILE  ex02.nap  ****** END ****** *#
```

ADVANCED USERS

The way the transpiler flattens the cell opens the door to several surprising variants. The transpiler behaves like a macro preprocessor. At the cell opening, it creates a table of correspondence between the instantiation parameters and the formal parameters of the cell (parameters beginning with the character "$"). During the parsing, the table is used to replace every formal parameter by its corresponding

instantiation value. This cell preprocessing is not limited by any syntax checks. It is, therefore, possible to replace *any* token of the cell!

It is for instance perfectly legal to write a cell like:

File "mycel.net"

```
cell_interface $out $in1 $in2 $function

#* polymorphic node: node kind is a parameter of the cell

node $out  $function  $in1 $in2
```

The cell could be used in the netlist:

File "ex03.nap"

```
header <napa.hdr>

fs 1.0e6

node a sine      0.0 1.0 1000.0 0.0
node b triangle 0.5 0.5 1234.0 9.87e-6

node out1 cell pls1 "mycel.net" a b prod
node out2 cell pls2 "mycel.net" a b sum

output "stdout"  a b out1 out2
terminate  TIME > 0.002
```

To check how the transpiler processes this netlist, we ask again for the flattened description:

```
%   NAPA ex03.nap -e > ex03a.nap
```

The resulting file contains:

File "ex03a.nap"

```
#* *****  EXPANSION OF FILE  ex03.nap  ***** START ***** *#


header <napa.hdr>
fs 1.0e6
node a sine      0.0 1.0 1000.0 0.0
node b triangle 0.5 0.5 1234.0 9.87e-6

#* >>  node out1 cell pls1 "mycel.net" a b prod

node out1  prod  a b

#* <<


#* >>  node out2 cell pls2 "mycel.net" a b sum

node out2  sum  a b

#* <<

output "stdout"  a b out1 out2
```

```
terminate  TIME > 0.002


#* *****  EXPANSION OF FILE  ex03.nap  ****** END ****** *#
```

The node kind is correctly replaced by the value given at the cell instantiation. The cell preprocessing brings up unexpected capabilities!

There is however a limitation: the resulting flattened netlist must be syntactically correct.

HOW IT WORKS

The parser builds a correspondence between the parameters of the instantiation (pseudo-node "cell") and the parameters of the "*cell_interface*". The instantiation name is used to construct a prefix. This prefix is built from the instantiation name preceded by the current prefix. The current prefix is a cascade of prefixes reflecting the hierarchy (null if the cell is instantiated from the main netlist).

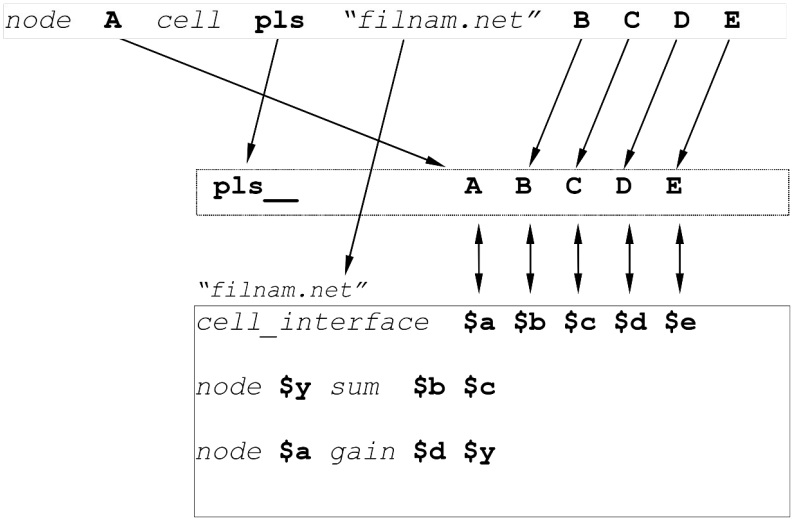


Figure 8 - The parsing mechanism of a cell (a)

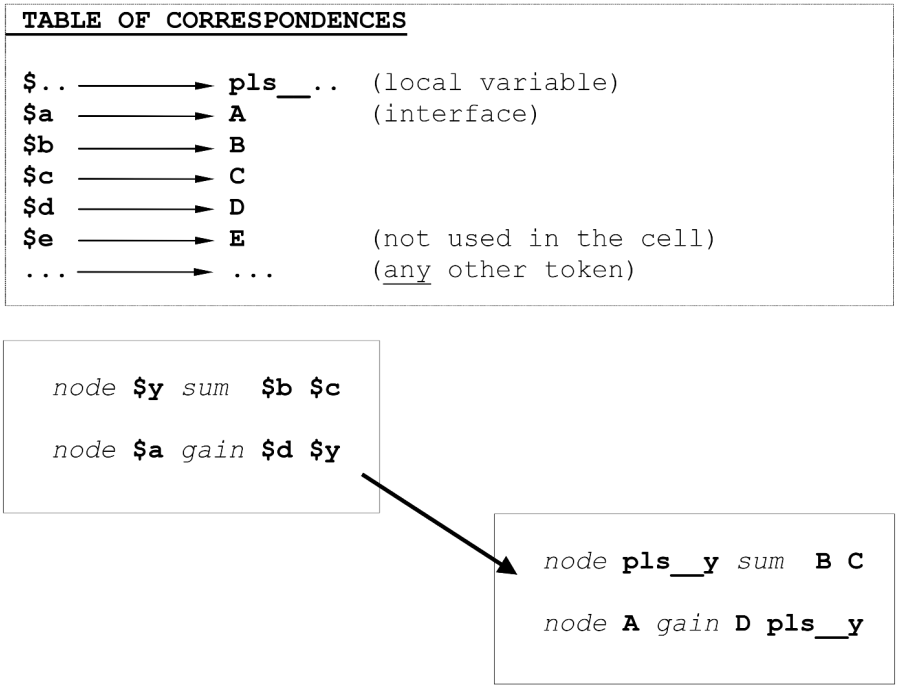


Figure 9 - The parsing mechanism of a cell (b)

Data cells

The user can place the values associated with the parameters of a cell or a netlist in a "*data cell*". This data cell may contain variable definitions and other simple initialization. Neither simulation control, nor nodes are allowed in this kind of file.

WHEN?

To make a cell reusable, it is interesting to separate the topology of a structure from its parameters. The data cell (instruction "*data*") provides a convenient way to collect the parameters in a file. The cell contains the topology; the data cell gives or computes the value of the associated parameters.

A FIRST EXAMPLE

Let us consider a low-pass FIR:

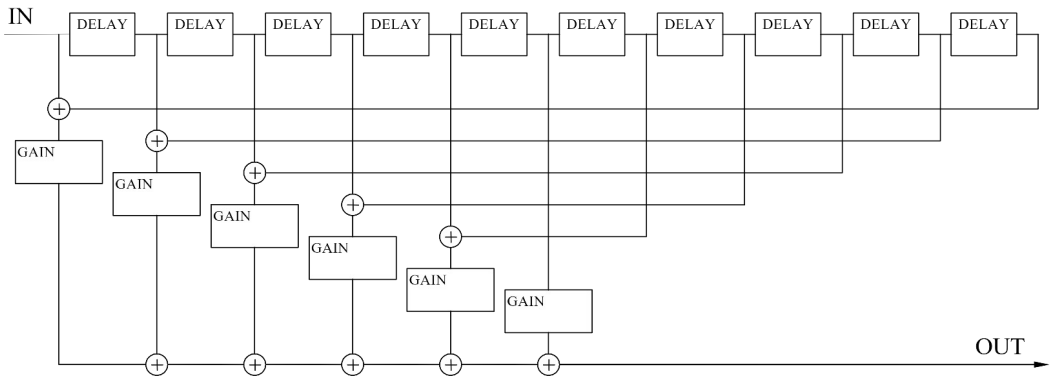


Figure 10 - A digital FIR

The netlist corresponding to this FIR is a ***NAPA*** cell. A parameter is reserved to pass the name of a file:

File "fir.net"

```
cell_interface   $y  $x  $filnam

#*  Symmetrical FIR structure
#*  Coefficients stored in the data file "$filnam"
data $filnam  $h0 $h1 $h2 $h3 $h4 $h5

node $d1  delay $x
node $d2  delay $d1
node $d3  delay $d2
node $d4  delay $d3
node $d5  delay $d4
node $d6  delay $d5
node $d7  delay $d6
node $d8  delay $d7
node $d9  delay $d8
node $d10 delay $d9

node $s0  sum   $x   $d10

node $s1  sum   $d1  $d9
node $s2  sum   $d2  $d8
node $s3  sum   $d3  $d7
```

```
node $s4  sum   $d4  $d6

node $g0  gain  $h0  $s0
node $g1  gain  $h1  $s1
node $g2  gain  $h2  $s2
node $g3  gain  $h3  $s3
node $g4  gain  $h4  $s4
node $g5  gain  $h5  $d5

node $y0  sum   $g0  $g1
node $y1  sum   $y0  $g2
node $y2  sum   $y1  $g3
node $y3  sum   $y2  $g4
node $y   sum   $y3  $g5
```

A data file “fir.dat” contains a set of coefficients compatible with this cell:

File “fir.dat”

```
data_interface $cf0 $cf1 $cf2 $cf3 $cf4 $cf5

#* coefficients for a low pass FIR to be used with cell "fir.net"

dvar $cf0      3.0510e+03
dvar $cf1      5.0980e+03
dvar $cf2      7.0090e+03
dvar $cf3      9.8440e+03
dvar $cf4      1.0950e+04
dvar $cf5      1.2043e+04
```

Here is an example of a main netlist instantiating the cell:

File “ex04.nap”

```
title "transfer function of a FIR"

header <napa.hdr>
header <toolbox.hdr>

fs 100.0e3

ivar npts POWEROF2(14)
directive NTF 10

node a noise 0.0 1.0
node b cell fir "fir.net" a "fir.dat"

tool tf "stdout" a 1.0 b 1.0 50000.0 npts

terminate TOOL_INDEX >= 1
```

Let us look again at the flattened description:

```
%   NAPA ex04.nap -e > ex04a.nap
```

The resulting file “ex04a.nap” contains:

File “ex04a.nap”

```
#* *****  EXPANSION OF FILE  ex04.nap  ***** START ***** *#


title "transfer function of a FIR"
header <napa.hdr>
header <toolbox.hdr>
fs 100.0e3
ivar npts POWEROF2(14)
directive NTF 10
node a noise 0.0 1.0

#* >>  node b cell fir "fir.net" a "fir.dat"


#* >>  >>  data "fir.dat"  fir__h0 fir__h1 fir__h2 fir__h3 fir__h4 fir__h5

dvar fir__h0      3.0510e+03
dvar fir__h1      5.0980e+03
dvar fir__h2      7.0090e+03
dvar fir__h3      9.8440e+03
dvar fir__h4      1.0950e+04
dvar fir__h5      1.2043e+04

#* <<  <<

node fir__d1  delay a
node fir__d2  delay fir__d1
node fir__d3  delay fir__d2
node fir__d4  delay fir__d3
node fir__d5  delay fir__d4
node fir__d6  delay fir__d5
node fir__d7  delay fir__d6
node fir__d8  delay fir__d7
node fir__d9  delay fir__d8
node fir__d10 delay fir__d9
node fir__s0  sum   a   fir__d10
node fir__s1  sum   fir__d1  fir__d9
node fir__s2  sum   fir__d2  fir__d8
node fir__s3  sum   fir__d3  fir__d7
node fir__s4  sum   fir__d4  fir__d6
node fir__g0  gain  fir__h0  fir__s0
node fir__g1  gain  fir__h1  fir__s1
node fir__g2  gain  fir__h2  fir__s2
node fir__g3  gain  fir__h3  fir__s3
node fir__g4  gain  fir__h4  fir__s4
node fir__g5  gain  fir__h5  fir__d5
node fir__y0  sum   fir__g0  fir__g1
node fir__y1  sum   fir__y0  fir__g2
node fir__y2  sum   fir__y1  fir__g3
node fir__y3  sum   fir__y2  fir__g4
node b   sum   fir__y3  fir__g5

#* <<
```

```
tool tf "stdout" a 1.0 b 1.0 50000.0 npts
terminate TOOL_INDEX >= 1


#* *****  EXPANSION OF FILE  ex04.nap  ****** END ****** *#
```

The coefficients of the filter are properly transmitted. The FIR structure is generic and can be stored in a library of reusable cells. The cell is customized through specific data cell files.

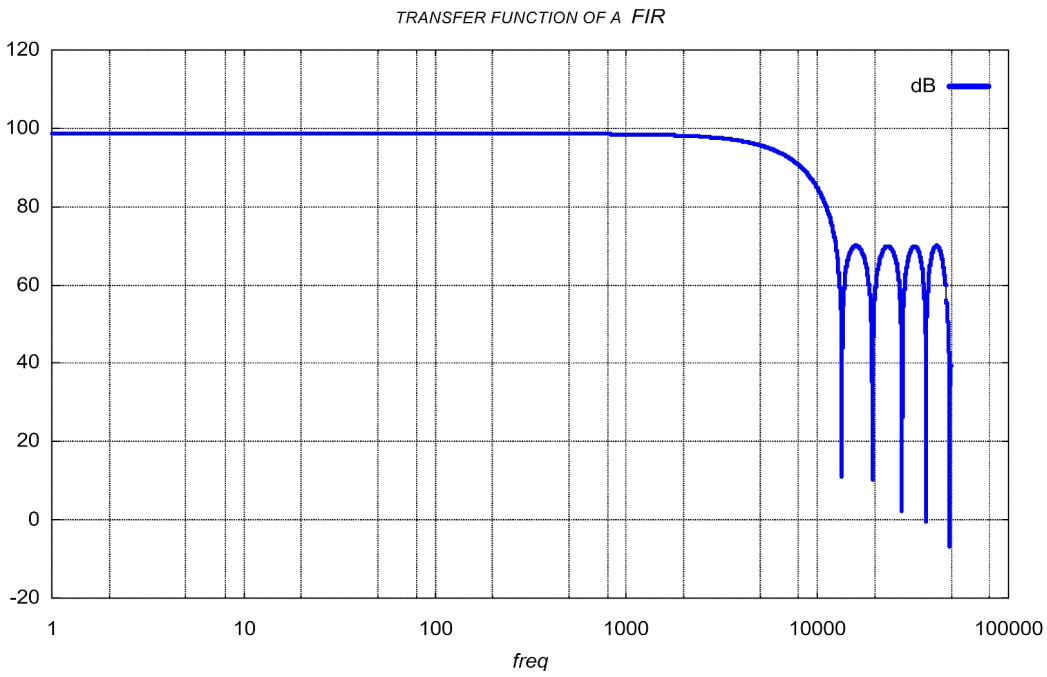


Figure 11 - Results of the simulation of the FIR

The data cell may be used not only to give coefficients but also to perform computations.

A SECOND EXAMPLE

In this example, we will describe a switched-capacitor integrator with a limited-gain operational amplifier. We imagine that this gain is a function of the bias current and that this function itself could depend on the type of amplifier we are using. We use a data file to store the data related to the operational amplifier. This data file computes the gain of the amplifier from the value of the bias current placed in the main netlist.

We will first build a model of the delayed switched capacitor integrator and put it in a cell file “intd.net”.

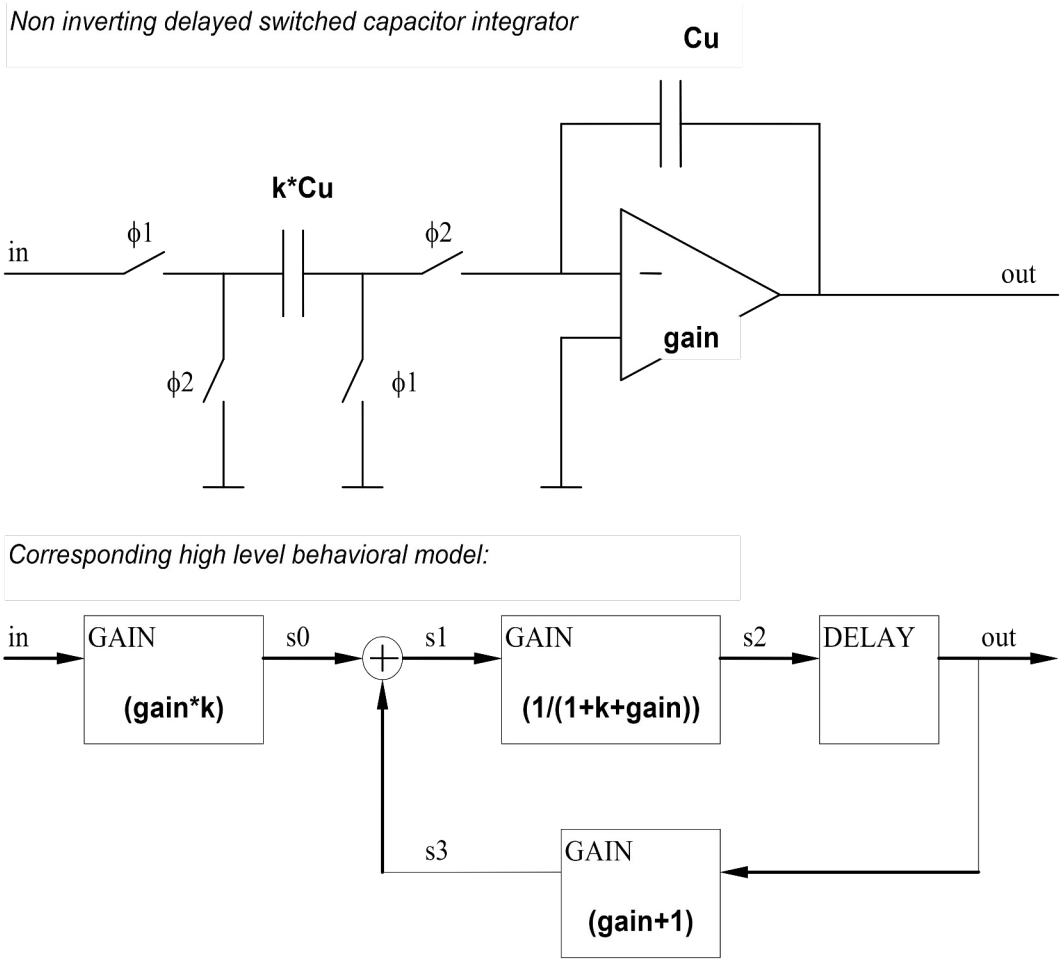


Figure 12 - The SWC integrator

File "intd.net"

```
cell_interface $out $in $k  $start $ibias $opfile

declare (analog)  $in            // input voltage
declare (analog)  $k             // capacitor ratio
declare (analog)  $start         // initial value

data $opfile  $gain $ibias

#* DELAYED INTEGRATOR WITH FINITE GAIN

init $out $start

dvar $g0  $gain * $k
dvar $g1  1.0/(1.0 + $k + $gain)
dvar $g2  $gain + 1.0

node $s0  gain  $g0 $in
node $s1  sum   $s0 $s3
node $s2  gain  $g1 $s1
node $out delay $s2
node $s3  gain  $g2 $out
```

In file “opamp0.dat”, the description of a simplistic operational amplifier:

File “opamp0.dat”

```
data_interface  $gain $i

#* OPAMP OP0 - SIMPLISTIC

dvar $gain  10000.0                        // constant gain
```

In file “opamp1.dat”, a more accurate behavior based on the curve fitting of a SPICE simulation: the file has the same interface, but the content is more sophisticated. The gain is computed from the formula placed in the data cell:

File “opamp1.dat”

```
data_interface  $gain $i

#* OPAMP OP1 - TYPIC

#* $i bias current in uA
#* curve fitting valid in interval [100 uA, 150 uA]

dvar $gds 2.0*(-13.2e-9 + 1.19e-9*$i + 2.32e-12*$i*$i)        &update
dvar $gm  2.0*(-296.0e-6 + 233.0e-6*sqrt($i) - 1.22e-6*$i)    &update

dvar $gain  $gm / $gds                                        &update
```

The main netlist could be:

File “ex05.nap”

```
title "comparison between 2 SWC integrators"

header <napa.hdr>

fs     2.0e6

dvar   ib 150.0             // bias current of opamp (uA)

node   in sine 0.0 1.0 123.0 0.0

node out0 cell int0 "intd.net" in 3.0 0.0 ib "opamp0.dat"
node out1 cell int1 "intd.net" in 3.0 0.0 ib "opamp1.dat"

node delta sub out0 out1

output "stdout" in out0 out1 delta

terminate TIME > 0.05
```

(Neither the opamp nor the input signal has an offset, as the non-inverting integrator is only conditionally stable.)

To understand the parsing mechanism, we flatten once more the description:

```
%   NAPA ex05.nap -e > ex05a.nap
```

File “ex05a.nap”

```
#* *****  EXPANSION OF FILE  ex05.nap  ***** START ***** *#


title "comparison between 2 SWC integrators"
header <napa.hdr>
fs     2.0e6
dvar   ib 150.0
node   in sine 0.0 1.0 123.0 0.0

#* >>  node out0 cell int0 "intd.net" in 3.0 0.0 ib "opamp0.dat"

declare (analog)  in
declare (analog)  3.0
declare (analog)  0.0

#* >>  >>  data "opamp0.dat"  int0__gain ib

dvar int0__gain  10000.0

#* <<  <<

init out0  0.0
dvar int0__g0  int0__gain * 3.0
dvar int0__g1  1.0/(1.0 + 3.0 + int0__gain)
dvar int0__g2  int0__gain + 1.0
node int0__s0  gain  int0__g0 in
node int0__s1  sum   int0__s0 int0__s3
node int0__s2  gain  int0__g1 int0__s1
node out0 delay int0__s2
node int0__s3  gain  int0__g2 out0

#* <<


#* >>  node out1 cell int1 "intd.net" in 3.0 0.0 ib "opamp1.dat"

declare (analog)  in
declare (analog)  3.0
declare (analog)  0.0

#* >>  >>  data "opamp1.dat"  int1__gain ib

dvar int1__dat_1__gds 2.0*(-13.2e-9 + 1.19e-9*ib + 2.32e-12*ib*ib) &update
dvar int1__dat_1__gm 2.0*(-296.0e-6 + 233.0e-6*sqrt(ib) - 1.22e-6*ib) &update
dvar int1__gain  int1__dat_1__gm / int1__dat_1__gds &update

#* <<  <<

init out1 0.0
dvar int1__g0  int1__gain * 3.0
dvar int1__g1  1.0/(1.0 + 3.0 + int1__gain)
dvar int1__g2  int1__gain + 1.0
node int1__s0  gain  int1__g0 in
node int1__s1  sum   int1__s0 int1__s3

node int1__s2  gain  int1__g1 int1__s1
node out1 delay int1__s2
node int1__s3  gain  int1__g2 out1
```

```
#* <<

node delta sub out0 out1
output "stdout" in out0 out1 delta
terminate TIME > 0.05


#* *****  EXPANSION OF FILE  ex05.nap  ****** END ****** *#
```

By stepping the bias current from 100 to 150 μA, we can analyze the behavior of the integrator for different values of bias current:

File "ex06.nap"

```
title "SWC integrator, absolute gain vs. bias current"

#* analysis of the attenuation caused by the opamp gain

header <napa.hdr>
header <toolbox.hdr>

fs    2.0e6

dvar   ib  linsweep(ctr, 100.0, 150.0, 21)      &update
dvar freq  123.0

export ib  freq

interlude  1000

node   in  sine 0.0 1.0 freq 0.0
node   out cell int "intd.net" in 3.0 0.0 ib "opamp1.dat"

node   ctr itool sinewave "stdout" out 1.0 freq 100000

terminate ctr > 20
```

File "ex06a.nap"

```
#* *****  EXPANSION OF FILE  ex06.nap  ***** START ***** *#


title "SWC integrator, absolute gain vs. bias current"
header <napa.hdr>
header <toolbox.hdr>
fs    2.0e6
dvar   ib  linsweep(ctr, 100.0, 150.0, 21)     &update
dvar freq  123.0
export ib freq
interlude  1000
node   in  sine 0.0 1.0 freq 0.0

#* >>  node   out cell int "intd.net" in 3.0 0.0 ib "opamp1.dat"

declare (analog)  in
declare (analog)  3.0
declare (analog)  0.0
```

```
#* >>  >>  data "opamp1.dat"  int__gain ib

dvar int__dat_0__gds 2.0*(-13.2e-9 + 1.19e-9*ib + 2.32e-12*ib*ib) &update
dvar int__dat_0__gm 2.0*(-296.0e-6 + 233.0e-6*sqrt(ib) - 1.22e-6*ib) &update
dvar int__gain  int__dat_0__gm / int__dat_0__gds &update

#* <<  <<

init out  0.0
dvar int__g0  int__gain * 3.0
dvar int__g1  1.0/(1.0 + 3.0 + int__gain)
dvar int__g2  int__gain + 1.0
node int__s0  gain  int__g0 in
node int__s1  sum   int__s0 int__s3
node int__s2  gain  int__g1 int__s1
node out delay int__s2
node int__s3  gain  int__g2 out

#* <<

node   ctr itool sinewave "stdout" out 1.0 freq 100000

terminate ctr > 20


#* *****  EXPANSION OF FILE  ex06.nap  ****** END ****** *#
```

HOW IT WORKS

The parser builds a correspondence between the parameters of the instantiation and the parameters of the "*data_interface*". The mechanism is similar to the parsing of the pseudo-node "*cell*". There is no explicit instantiation name here. For the instantiation of a data cell, the parser builds an unambiguous and unique instantiation name from a counter of instances.

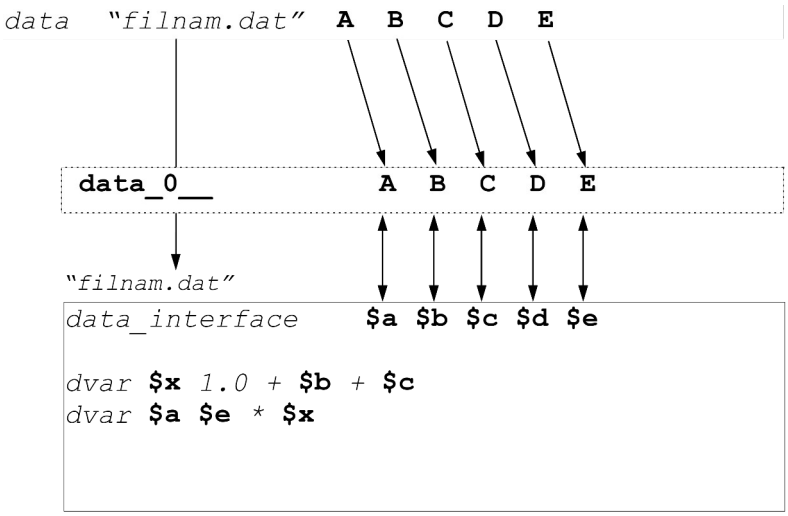


Figure 13 - The parsing mechanism of a data cell (a)

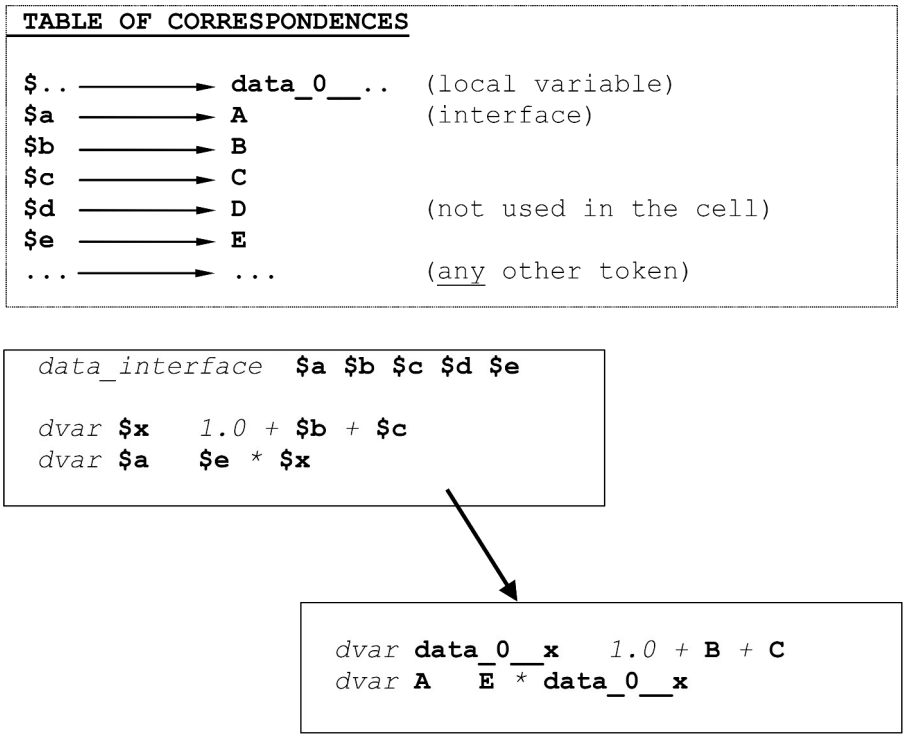


Figure 14 - The parsing mechanism of a data cell (b)

Generators

There is a large category of applications where the basic topology comes in many variants. In the previous example, the FIR could use another number of coefficients. The netlist is similar but must be adapted to the new number of parameters. It is, of course, possible to generate the cell by a stand-alone program or a script. ***NAPA*** provides a way to call the program on the fly during the parsing and to use the newly written cell immediately as if it were a usual cell.

WHEN?

Each time you have a reusable cell where the structure is regular but accepts many variants.

A FIRST EXAMPLE

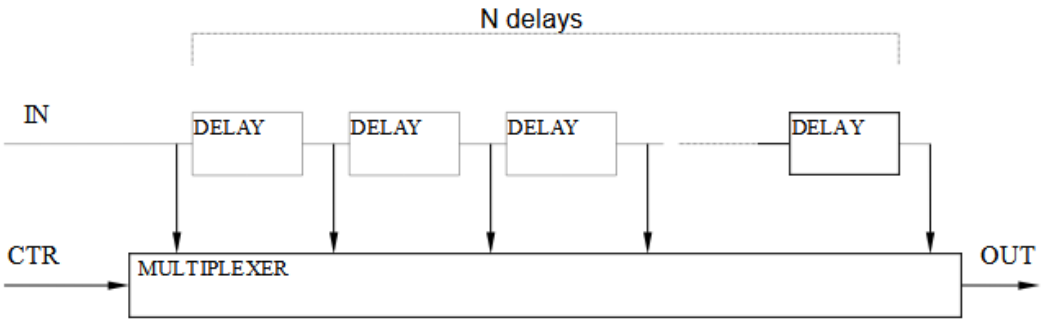


Figure 15 - Programmable delays

It is possible to describe this structure with a cell when the number of delays in the chain is fixed. Here is an example of a sequence of 4 delays:

File "delay4.net"

```
cell_interface  $out  $ctr $in

#* programmable delays 0 to 4 controlled by 'ctr'

declare (digital) $ctr

node  $d1 delay $in
node  $d2 delay $d1
node  $d3 delay $d2
node  $d4 delay $d3

node $out mux $ctr $in $d1 $d2 $d3 $d4
```

If the number of delays is variable, we have to write as many cells as we have variants. It is possible to use a generator of a cell. The example shows an ANSI-C program but you can imagine using any way to generate the text of the netlist: scripts, ***C***, Pascal...

File “delay.c”

```
/* *** Generator of programmable delays ************************* */

#include "./Header/generator.h"  /* specific header for NAPA generator */

int main(int argc, char **argv) {

  int  i, num;
  char ofnam[256];
  FILE *ofp;

  if (argc < 3) {
    fprintf(stderr, "usage: delay <cell_file> <number_of_delays>\n");
    return EXIT_FAILURE;
  }
  strcpy(ofnam, argv[1]);
  ofp = fopen(ofnam, "w");
  sscanf(argv[2], "%d", &num);
  fprintf(ofp, "cell_interface");
  fprintf(ofp, "  $out  $dummy  $ctr $in\n\n");
  fprintf(ofp, "declare (digital)  $ctr\n\n");
  fprintf(ofp, "node $d1   delay $in\n");
  for (i = 1; i < num; i++) {
    fprintf(ofp, "node $d%-3d delay $d%d\n", i+1, i);
  }
  fprintf(ofp, "\nnode $out  mux   $ctr  $in");
  for (i = 1; i <= num; i++) {
    fprintf(ofp, " $d%d", i);
  }
  fprintf(ofp, "\n\n");
  fclose(ofp);

  return EXIT_SUCCESS;
}
```

This generator has to be compiled, and its executable placed in the generator library: NAPA only runs a generator located in that library. Of course, it is recommended to add code to verify the integrity of the inputs and the opening and closing of the file. Dig in the generic generator library of ***NAPA*** for complete examples.

File “delay” (Linux, Unix)
File “delay.exe” (Windows)

```
..(Executable built from file "delay.c")..
..
```

Although this is not the preferred way, the generator can be used as a stand-alone program. The result is a ***NAPA*** cell which could be used as a regular cell.

```
%    command line:      delay  "delay4.net"  4
```

It is recommended to use the pseudo-node “*generator*” in the *NAPA* netlist to call directly the execution and the generation of the cell on the fly. The syntax is very similar to the syntax of a cell:

File "ex07.nap"

```
title "programmable delays"

header <napa.hdr>

fs 1.0e6

node  in  clock "111 . 01 0011 000111 00001111"
node  nd  dc (digital) 3

node  out generator  delay <delay> 4 nd in

output "stdout" nd in out
terminate LOOP_INDEX > 50LL
```

NAPA runs the generator on the fly, directly and without a shell (delay "delay_0.gen" 4 nd in), which writes the cell "delay_0.gen"; ***NAPA*** then parses this new cell at once, as if the netlist contained:

```
..
node  out cell  delay "delay_0.gen"  4  nd in
..
```

Please note that the parameters of the pseudo-node "*generator*" can be used for the generation of the cell and as arguments of the cell instantiation. Some parameters can be used for the generator only (here the number '4'). This is why the unused symbol "$dummy" has been inserted in the "*cell_interface*". Other parameters are used in the instantiation only (here the identifiers 'nd' and 'in').

File "delay_0.gen"

```
cell_interface  $out  $dummy  $ctr $in

declare (digital)  $ctr

node $d1   delay $in
node $d2   delay $d1
node $d3   delay $d2
node $d4   delay $d3

node $out  mux   $ctr  $in $d1 $d2 $d3 $d4
```

It is interesting to expand again the netlist to understand the way *NAPA* parses and flattens the netlist in the presence of a generator:

```
%   NAPA ex07.nap -e > ex07a.nap
```

File "ex07a.nap"

```
#* *****  EXPANSION OF FILE  ex07.nap  ***** START ***** *#


title "programmable delays"
header <napa.hdr>
fs 1.0e6
node  in  clock "111 . 01 0011 000111 00001111"
node  nd  dc (digital) 3

#* >>  node  out generator  delay <delay> 4 nd in
#*      changed in ... cell  delay "./delay_0.gen" 4 nd in
```

```
declare (digital)  nd
node delay__d1   delay in
node delay__d2   delay delay__d1
node delay__d3   delay delay__d2
node delay__d4   delay delay__d3
node out  mux   nd  in delay__d1 delay__d2 delay__d3 delay__d4

#* <<

output "stdout" nd in out
terminate LOOP_INDEX > 50LL


#* *****  EXPANSION OF FILE  ex07.nap  ****** END ****** *#
```

HOW IT WORKS

The pseudo-node "*generator*" transfers its parameters to the generator executable and to the generated cell for parsing:

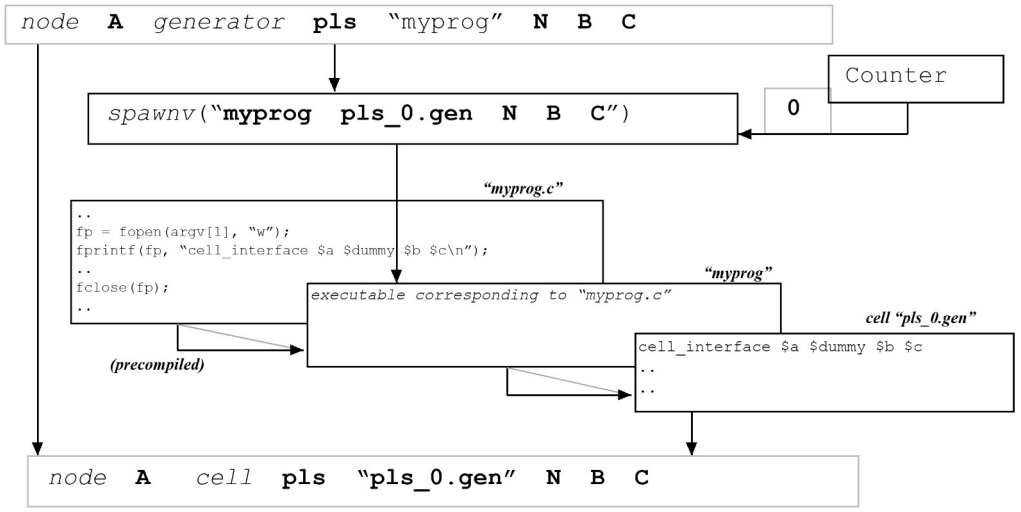


Figure 16 - The parameters of a generator

ADVANCED USERS

The remark on the cell preprocessing remains valid. It is possible to build surprising applications when you know that the parser could preprocess every token of the generated cell.

There is no limit: a generator could produce a cell containing a cell or... another generator if it makes sense to do it!

Think of reusability. Use a standard language such as ANSI-C to make your generator reusable on other platforms.

User-defined C macros

It is possible to introduce macro constants and macro functions using the instruction “directive”.

WHEN?

As soon as you need a function in a ***C*** expression and you prefer an inline definition in the ***NAPA*** source.

A FIRST EXAMPLE

Here are the definitions of a few macros:

File “myfile1.nap”

```
header <napa.hdr>
...
directive VDD      5.0
directive POW2(x)  (pow(2.0,(double)(x)))
directive TEMP
...
```

The ***C*** code produced by ***NAPA*** will include the following macros:

```
#define  VDD  5.0
#define  POW2(x)  (pow(2.0,(double)(x)))
#define  TEMP
#define  TEMP_IS_EMPTY
...
```

A SECOND EXAMPLE

A directive may *not* refer to a node or to a variable of the netlist. A directive is a pure C preprocessor macro: it is expanded before the simulator is compiled, at a time when the NAPA variables do not exist yet.

File “myfile2.nap”

```
header <napa.hdr>
...
dvar zeroK -273.15
...
directive C2K(t) (t - zeroK)
...
```

This netlist is rejected by the NAPA transpiler:

```
...
NAPA Error: (directive)
<zeroK>, a variable cannot be used in a directive
...
```

Use a macro constant instead. Take care not to begin the value with a parenthesis, as it would be read as the parameter list of a macro function:

```
...
directive ZEROK -273.15

directive C2K(t) ((t) - (ZEROK))
...
```

The C code produced by NAPA will then include:

```
...
#define ZEROK -273.15
#define C2K(t) ((t)-(ZEROK))
...
```

HOW TO DEFINE A DEFAULT VALUE?

In a header, how to define a macro with a default value?

```
...
#if defined(FOO_IS_EMPTY) || !defined(FOO)
 #undef  FOO
 #define FOO     the_default_value_of_FOO
#endif
...
```

The macro FOO, if it is not called by a directive in the *NAPA* source, will be created and a default value will be assigned. If the macro FOO is defined without a value, the default value is assigned. This is an elegant way to avoid the crash of the C compiler when a macro has no argument and is therefore translated as empty.

```
...
#if defined(TOTO_IS_EMPTY)
 #undef  TOTO
 #define TOTO    the_default_value_of_TOTO
#endif
...
```

The macro TOTO, if it is not called by a directive in the *NAPA* source, will *not* be created. A default value will be assigned if the macro has been defined without a value. This is an elegant way to avoid the crash of the C compiler when a macro has no argument and is therefore translated as empty.

User-defined C functions

NAPA accepts ***C*** expressions on several occasions. By default, the user has access to the standard mathematical library of the *ANSI-C*. Several other functions, located in the "napa.hdr" header file, are also available.

The ***NAPA*** transpiler can detect the explicit use of a ***C*** function in the main netlist or a cell or a data cell file. For each function, the transpiler adds a macro preprocessor constant in the ***C*** output program:

#define COMPILE_the_function_identifier

As we will see in the second example, it makes it possible to trigger automatically the compilation of the file containing the code of the function.

WHEN?

As soon as you need a function in a ***C*** expression.

A FIRST EXAMPLE

Here is the definition of a simple function:

File "tau.hdr"

```
#ifndef __TAU_HDR__
#define __TAU_HDR__

double tau_rc(double, double);

double tau_rc(double r, double c) {
  return r * c;
}

#endif                                                  /* __TAU_HDR__ */
```

To have this function compiled with the simulator, use the instruction "*header*":

File "ex08.nap"

```
header <napa.hdr>
header "tau.hdr"

fs    1.0e6

dvar  r1  1.200e3
dvar  c2  1.0e-6

dvar  rc  tau_rc(r1, c2)

node  a dalgebra  1.0 * exp(-TIME / rc)

output "stdout" a
terminate TIME > (10.0 * rc)
```

A SECOND EXAMPLE

When the ***C*** function is used explicitly in the netlist (and not only inside a header), ***NAPA*** detects it and can trigger the compilation of the proper code.

The ***C*** code contains the following directive: #define COMPILE_tau_rc

In the generic header library provided with the ***NAPA*** package, the header "toolbox.hdr" includes one index file per category: "Function/_function.lib" for functions, "Tool/_tool.lib" for tools, "Post/_post.lib" for postprocessing functions, "User/_user.lib" for user-defined functions and "Random_Generator/_random_generator.lib" for random generators. These index files contain the addresses of the files to be compiled. If you add your function to the index of functions, the file "tau.hdr" being placed in the same directory:

File "Function/_function.lib"

```
..
#ifdef COMPILE_tau_rc
#  include "tau.hdr"
#endif
..
```

If you take care to include the header "toolbox.hdr" in your netlist, the transpiler will automatically select your file. There is no longer any need to remember the location of the functions.

File "ex09.nap"

```
header <napa.hdr>
header <toolbox.hdr>

fs  1.0e6

dvar  r1  1.200e3
dvar  c2  1.0e-6

dvar  rc  tau_rc(r1, c2)

node  a dalgebra  1.0 * exp(-TIME / rc)

output "stdout" a
terminate TIME > (10.0 * rc)
```

METHODOLOGY

The transpiler maintains a long list of macro preprocessor constants and variables (see the ***NAPA*** Reference for an exhaustive list) and several important global variables. The ***C*** code is transpiled in one pass without any kind of precompilation. Every header is included in the main program. This is a powerful feature as every piece of code can be customized during the macro preprocessing before the ***C*** compilation.

For instance, the macro "FSL" is pointing to the sampling frequency local to the segment containing the function call: the function can be made aware of the local sampling frequency.

There is a disadvantage: no variable is local. The simulator is a single ***C*** file and does not use some ***C*** features such as separate compilation and variables local to a file. It means that the user should take care not to duplicate the declaration of an existing identifier.

HOW IT WORKS

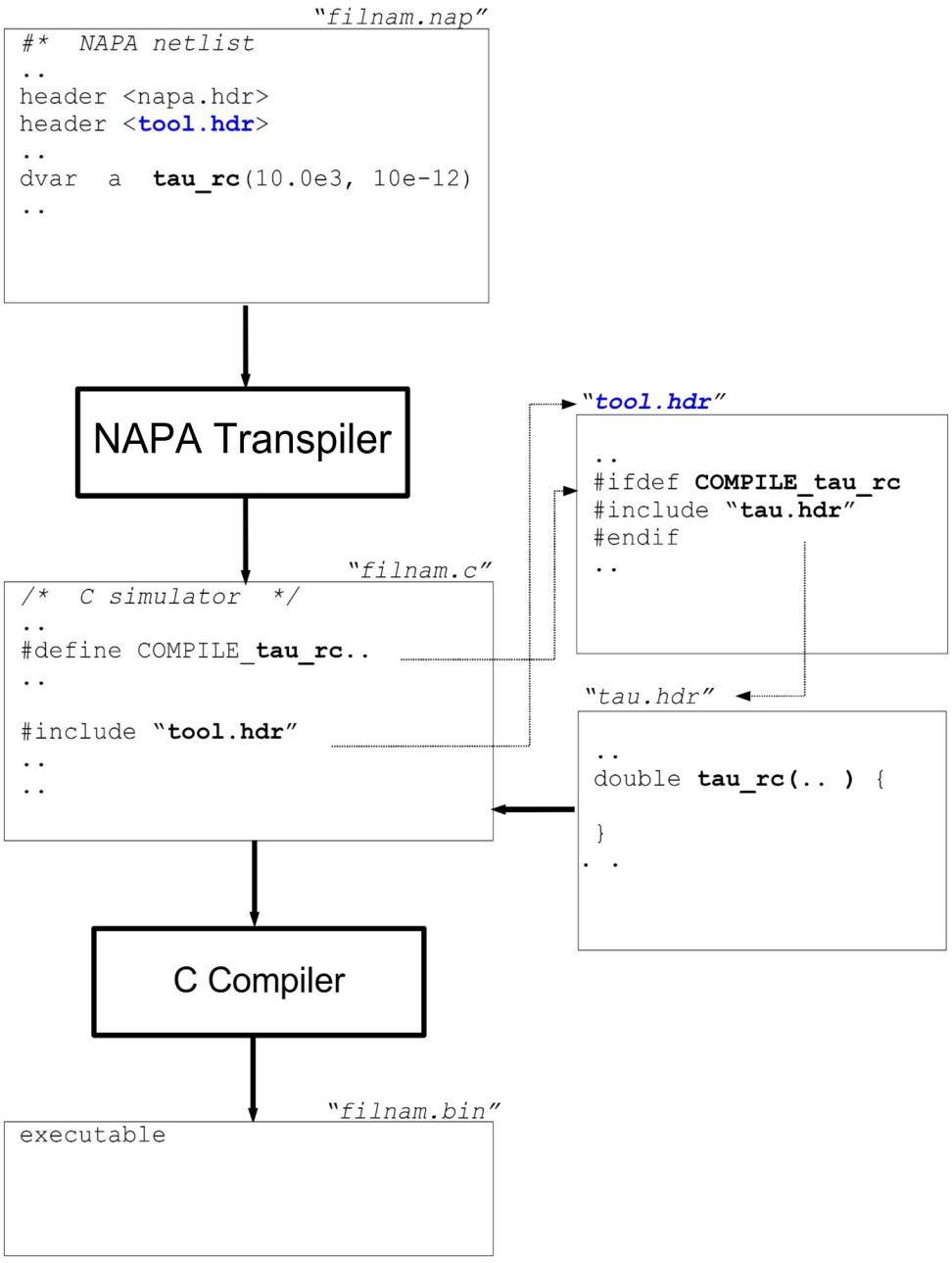


Figure 17 - The C macro preprocessor and the functions

“algebra”, “dalgebra”, “ialgebra”

The nodes “*algebra*”, “*dalgebra*” and “*ialgebra*” accept ***C*** expressions. This is the simplest way to customize ***NAPA***, making it possible to describe a primitive that does not exist in the package.

WHEN?

If it is possible to describe the primitive in a one-line ***C*** expression, this is the preferred way. Define a user function (see the previous chapter) to build a more sophisticated primitive and insert it in the expression. But if you have to write a procedure with internal memory states, if this procedure must be initialized or closed, you need more powerful objects (see the next chapter for the nodes “*duser*” and “*iuser*”).

AN EXAMPLE

In file “utility.hdr”, we have defined a recursive GCD function, “my_gcd( )” (the name “GCD( )” is already used by the NAPA package). This function has no internal memory states. It does not require any initialization and has nothing to process at the exit of the simulator process. This is the perfect candidate for an “*ialgebra*” node.

File “utility.hdr”

```
#ifndef __UTILITY_HDR__
#define __UTILITY_HDR__

long my_gcd(long, long);

long my_gcd(long n1, long n2) {                /* Greatest Common Divisor */
  if (n1 < n2) {
    return my_gcd(n2, n1);
  }
  if (n1 % n2 == 0) {
    return n2;
  }
  return my_gcd(n2, n1 % n2);
}

#endif                                                  /* __UTILITY_HDR__ */
```

This is a generic function. We will make it reusable by introducing its address in the index of functions “Function/_function.lib”:

File “Function/_function.lib”

```
..
#ifdef COMPILE_my_gcd
#  include "utility.hdr"
#endif
..
```

The use of this function is now straightforward:

File “ex10.nap”

```
title  "GCD vs. number #v for the #e first integers"

header <napa.hdr>
header <toolbox.hdr>

fs  1.0

ivar v  60
ivar e 200

node a sum   b One
node b delay a

init b   1                    // initialized to 1 to avoid a division by 0!

node c ialgebra  my_gcd(b, v)

output "stdout" b c
terminate LOOP_INDEX > e
```

Of course, this function could be used in other locations where a ***C*** expression is allowed.

The simulation output, just for fun:

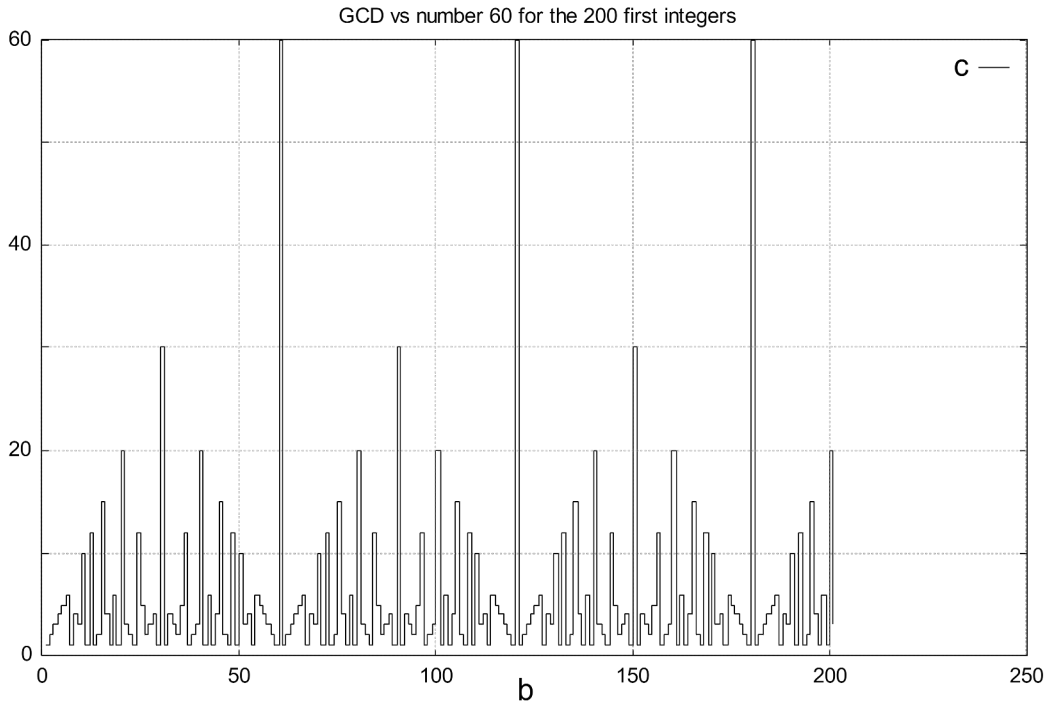


Figure 18 - Output of the GCD trials

HOW IT WORKS

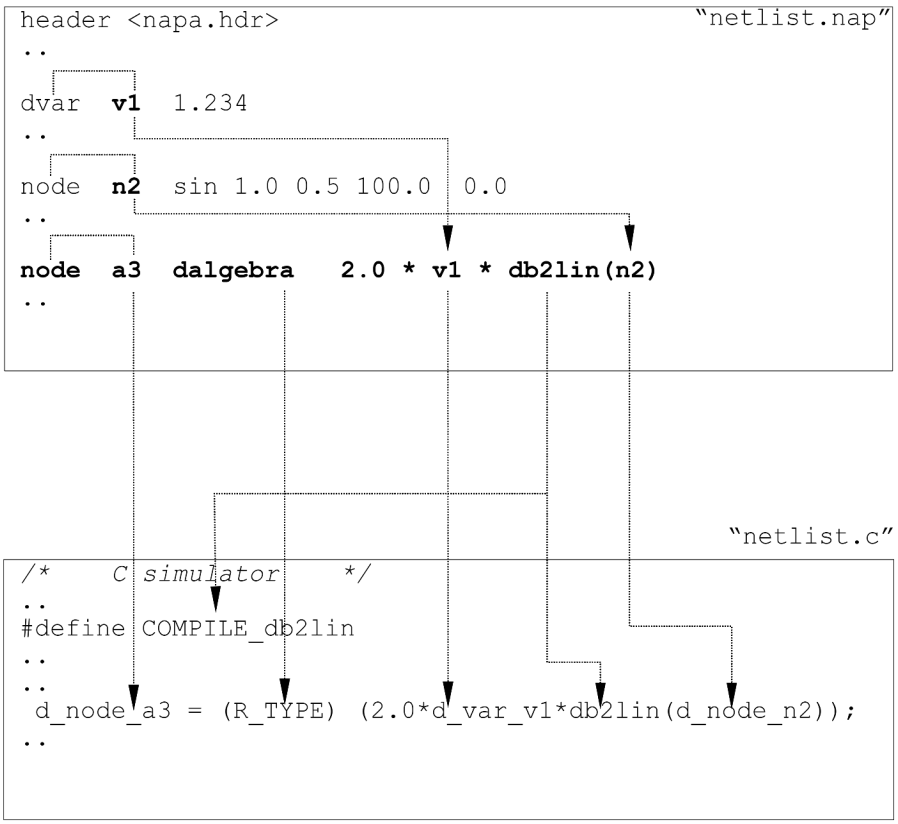


Figure 19 - The translation of a “dalgebra” node

“napa_init( )”, “napa_close( )”

Two functions “napa_init( )” and “napa_close( )” are predefined in the header file “napa.hdr”. These functions can be completed by the user. The first function is called at the initialization before any other one. The second function is called at the very end of the simulation after any other one.

WHEN?

Each time you have to customize the ***NAPA*** simulator to your operating system, to your set-up or for your convenience.

AN EXAMPLE

Here is a method to compute and display on the screen the effective duration of the simulation. The code specific to this application appears in bold in the text:

File “napa.hdr”

```
#ifndef __NAPA_HDR__
#define __NAPA_HDR__
..
time_t napa_time_start;
time_t napa_time_stop;
..
void napa_init(void) {
  ..
  (void) time(&napa_time_start);
  ..
  return;
}
..
void napa_close(void) {
  ..
  long t, h, m, s;
  ..
  (void) time(&napa_time_stop);
  t = (long) (napa_time_stop - napa_time_start);
  s = (t % 60);
  m = (t / 60) % 60;
  h = (t / 3600);
  fprintf(stderr, "\n*** STOPWATCH: %02ld:%02ld:%02ld ***\n", h, m, s);
  ..
  return;
}
..
#endif                                                  /* __NAPA_HDR__ */
```

On the screen, at the end of the simulation:

[SCREEN]

```
..

*** STOPWATCH: 00:00:19 ***
%
```

HOW IT WORKS

The ***NAPA*** simulator calls the first one from “napa_control_init( )”, at the top of the ***C*** program, and registers the second one with “atexit( )” so that it runs after the main program returns. It guarantees that the code is executed at the very beginning and the very end of the simulation:

File “simulator.c”

```
..
void napa_init(void);
void napa_close(void);
..
#include  "C:/Simulate/NapaDos/Hdr/napa.hdr"
..

int main(void) {

  napa_control_init();        /* calls napa_init() */
  ..
  napa_reset_variables();
  napa_reset_nodes();
  ..

  do {
    napa_abs_time = ..;

    ..   /* code corresponding to the netlist      */

    napa_abs_loop++;
  } while ( .. );

  napa_exit(EXIT_SUCCESS);  /* napa_close() runs here, through atexit() */

}

..
```

“napa_loop_option(*long*)”

AVAILABLE ONLY WHEN NAPA TRANSPILER IS COMPILED WITH OPTION “COMPILE_LOOP_OPTION = YES”,

***DEFAULT IS* NO.**

If a process needs to be run just before or just after the main loop, or each time in the beginning or at the end of the main loop, define the function called “napa_loop_option(long opt)” as described below.

WHEN?

Reserved to advanced users to customize ***NAPA***. **Use only when other solutions fail!**

AN EXAMPLE

File “myfile.nap”

```
header <napa.hdr>
header "./myoption.hdr"     // Header file containing the process options
..

terminate...
```

Where the file “myoption.hdr” contains the processes to be run and the macro definition needed to activate the function calls. Here is a template of such a file:

File “myoption.hdr”

```
#ifndef __MYOPTION_HDR__
#define __MYOPTION_HDR__

#define NAPA_LOOP_OPTION      /* to trigger the call of the option      */

void napa_loop_option(long opt) {
  ..
  switch (opt) {
  case 0:                     /* called just before the main loop       */
    ..
    break;
  case 1:                     /* called at the start of the main loop   */
    ..
    break;
  case 2:                     /* called at the end of the main loop     */
    ..
    break;
  case 3:                     /* called just after the main loop        */
    ..
    break;  }
  return;
}

#endif                                          /* __MYOPTION_HDR__ */
```

HOW IT WORKS

The ***C*** file generated by the ***NAPA*** transpiler includes a prototype of the function “napa_loop_option( )” and 4 function calls controlled by the macro preprocessor flag “NAPA_LOOP_OPTION”. The definition of the macro constant “NAPA_LOOP_OPTION” in a header file (here “myoption.hdr”) activates these function calls.

This is the main difference with “napa_init( )” and “napa_close( )” which are unconditionally called. The line 'napa_segment = 0L;' that precedes each call is emitted only when the simulation is periodic and uses more than one segment.

File “simulator.c”

```
...
void napa_loop_option(long);              /*    Prototype Declaration   */
...

int main(void) {                          /*      Main Program          */
  ...
  napa_control_init();
  ...

#if defined(NAPA_LOOP_OPTION)
  napa_segment = 0L;
  napa_loop_option(0);                    /*  <<<<< JUST BEFORE LOOP    */
#endif

  do {                                    /*      Main Loop             */
    napa_abs_time = ... ;

#if defined(NAPA_LOOP_OPTION)             /*  <<<<< BEGIN OF LOOP       */
    napa_segment = 0L;
    napa_loop_option(1);
#endif

    ...                                   /*  <<<<< LOOP CORE           */


#if defined(NAPA_LOOP_OPTION)
    napa_segment = 0L;
    napa_loop_option(2);                  /*  <<<<< END OF LOOP         */
#endif

    napa_abs_loop++;
  } while ( !TERMINATE );                 /*      End of Main Loop      */

#if defined(NAPA_LOOP_OPTION)
  napa_segment = 0L;
  napa_loop_option(3);                    /*  <<<< JUST AFTER LOOP      */
#endif

  ...
  napa_waypoint = 6;
  napa_exit(EXIT_SUCCESS);
}                                         /*     End of Main Program    */
```

“call”

If a process needs to be initialized at the beginning of the simulation and run periodically at each sampling period, it is possible to use the instructions “*init*” and “*call*”.

WHEN?

As it is superseded by the user-defined function of the nodes “*duser*” and “*iuser*”, this method is generally not recommended. This is nevertheless the solution to use a function that was not specifically written for ***NAPA*** and for some (good) reason you do not want to modify it to use it in the ***NAPA*** environment.

AN EXAMPLE

Here is an example of the method. This is only an example. It is much better to use node “*iuser*”.

File “junk.hdr”

```
#ifndef __JUNK_HDR__
#define __JUNK_HDR__

long junk_array[128];          /* global variables for the simulator!   */
long my_random_number;

void my_function1(void);
void my_function2(long);

void my_function1(void) {
  time_t t;
  int    i;
  (void) time(&t);
  srand((unsigned long) t);
  for(i = 0; i < 128; i++) {
    junk_array[i] = (long) rand();
  }
  my_random_number = 0;
  return;
}

void my_function2(long n) {
  static int i = 0;
  i++;
  if (i > 127) {
    i = 0;
  }
  my_random_number = junk_array[i] * n;
  return;
}

#endif                                                  /* __JUNK_HDR__ */
```

Here is an example of netlist using these functions:

File “junk.nap”

```
header <napa.hdr>
header "junk.hdr"

fs  1.0e6

init  void my_function1( )

node  a ialgebra  my_random_number % 100
call  void my_function2(10)

output "stdout" a
terminate LOOP_INDEX > 250LL
```

HOW IT WORKS

The transpiler places the call of the “init” function in the function napa_reset_variables( ), called after napa_init( ), and the call of the “call” function in the simulation loop, with the updates of the variables:

File “junk.c”

```
void napa_init(void);
void napa_close(void);
void napa_reset_variables(void);
void napa_reset_nodes(void);

#define COMPILE_my_function1
#define COMPILE_my_function2

#include  "C:/Simulate/NapaDos/Hdr/napa.hdr"
#include  "junk.hdr"
..

int main(void) {

  napa_control_init();
  napa_reset_variables();
  napa_reset_nodes();

  do {

    napa_abs_time = napa_abs_loop ..;

    /* Block 1 (variables) */
    my_function2(10);
    ..

    /* Block 2 (nodes)     */
    inode_a = (I_TYPE) (my_random_number % 100);

    /* Block 3 (output)    */
    fprintf(.., "..%lld..\n", inode_a );

    napa_abs_loop++;
  } while ( .. );
```

```
  napa_exit(EXIT_SUCCESS);  /* napa_close() runs here, through atexit() */
}

..

void napa_reset_variables(void) {
  my_function1( );
  ..
  return;
}
```

It is, of course, possible to place the pathname of the file containing the code in one of the index files included by “toolbox.hdr” but it does not make sense to reuse such a poor practice.

“duser”, “iuser”

Nodes “*algebra*”, “*dalgebra*” and “*ialgebr*a” accept only one-line expressions. There is simply no provision to initialize the function if it is necessary. The pair of instructions “*init*” and “*call*” makes it possible to initialize the process. But the mechanism is too crude and the user must manipulate two correlated functions and their parameters. The nodes “*duser*” and “*iuser*” bring user-friendliness and data encapsulation with a lot of other interesting features.

Following the instantiation, for example, of a “*duser*” node called “foo” with 3 arguments, the transpiler takes several actions.

The first action is to place a macro definition with the number of instantiations of the function.

‘#define COMPILE_duser_foo 1’

The second action is to place 4 function calls in the ***C*** code of the simulator:

in the initialization part:	‘check_duser_foo_03(arg1, arg2, arg3, 0)’
	‘init_duser_foo_03(arg1, arg2, arg3, 0)’
in the main loop	‘duser_foo_03(arg1, arg2, arg3, 0)’
in the closedown part	‘close_duser_foo_03(arg1, arg2, arg3, 0)’

In case of a restart, the transpiler will place a 5th function:

‘reset_duser_foo_03(arg1, arg2, arg3, 0)’

The last argument of these functions (here the number ‘0’) corresponds to the value of a counter of instances of the “duser” “foo”. These functions must have been defined in a header file. They must exist but could be empty depending on the process the function is written for.

WHEN?

When you have to build a function needing an initialization, with internal memory states particular for each instantiation, or with a variable number of parameters, this is the best choice. Reusability and ease of use are optimal. If your function is used to analyze a result and not to produce a signal, use the nodes “*itool*” and “*dtool*” (see the next chapter).

A FIRST EXAMPLE

File “rsin.hdr”

```
#ifndef __RSIN_HDR__
#define __RSIN_HDR__

/* **** USAGE ********************************************************* */
/* node <out> duser  rsin  <amplitude> <fmin> <fmax>                    */
/*                                                                      */
/* sinewave with random frequency and phase                             */
/*                                                                      */
/* where <out>        output node                                       */
/*       <amplitude>  amplitude                                         */
/*       <fmin>       min of random frequency                           */
/*       <fmax>       max of random frequency                           */

/* **** function prototypes ******************************************* */
/*                      amplitude  fmin    fmax   id                   */
```

```
void  check_duser_rsin_03(double, double, double, int);
void   init_duser_rsin_03(double, double, double, int);
void  reset_duser_rsin_03(double, double, double, int);
void  close_duser_rsin_03(double, double, double, int);

double      duser_rsin_03(double, double, double, int);

/* **** global variables ****************************************** */
double rsin_fr[32];
double rsin_ph[32];

/* **** resources ************************************************* */
#include "Function/random.hdr"       /* containing ‘rand_uniform()’ */

/* **** function definitions ************************************** */
void check_duser_rsin_03(double x, double fmin, double fmax, int id) {
  (void) x;
  if (id >= 32) {
    fprintf(stderr, "NAPA Run Time Error (rsin[%d])", id);
    fprintf(stderr, " array overflow\n");
    napa_exit(EXIT_FAILURE);
}

  if ((fmin < 0.0) || (fmax <= 0.0) || (fmin > fmax)) {
    fprintf(stderr, "NAPA Run Time Error (rsin[%d])", id);
    fprintf(stderr, " inconsistent frequencies [%g, %g]\n", fmin, fmax);
    napa_exit(EXIT_FAILURE);
  }
  return;
}

void init_duser_rsin_03(double x, double fmin, double fmax, int id) {
  reset_duser_rsin_03(x, fmin, fmax, id);
  return;
}

void reset_duser_rsin_03(double x, double fmin, double fmax, int id) {
  (void) x;
  rsin_fr[id] = rand_uniform(fmin, fmax);
  rsin_ph[id] = rand_uniform(0.0, _2PI_);
  return;
}

void close_duser_rsin_03(double x, double fmin, double fmax, int id) {
  (void) x; (void) fmin; (void) fmax; (void) id;
  return;
}

double duser_rsin_03(double ampl, double fmin, double fmax, int id) {
  (void) fmin; (void) fmax;
  return ampl * SIN((_2PI_ * rsin_fr[id]* TIME) + rsin_ph[id]);
}

#endif                                                /* __RSIN_HDR__ */
```

Here is a netlist using this function

File “ex11.nap”

```
header <napa.hdr>
header "rsin.hdr"

fs 1.0e6

node a  duser  rsin  1.0  100.0  1000.0
node b  duser  rsin  2.0 1000.0 10000.0

output "stdout"  a  b

terminate LOOP_INDEX > 1000LL
```

A SECOND EXAMPLE

Some of the parameters could accept default values. Other parameters are used only during the initialization phase. Using the ***C*** macro preprocessor, it is possible to build a more efficient code and increase the flexibility of the user-defined functions. As the header file could contain many complex user-defined functions, we add a macro preprocessor directive for conditional compilation. We rework a little bit the previous file:

File “rsina.hdr”

```
#ifndef __RSIN_HDR__
#define __RSIN_HDR__

/* **** USAGE ********************************************** */
/* node <out> duser  rsin  <amplitude> <fmin> <fmax>          */
/* node <out> duser  rsin  <amplitude> <bwdth>                */
/* node <out> duser  rsin  <amplitude>                        */
/*                                                            */
/* sinewave with random frequency and phase                   */
/*                                                            */
/* where <out>        output node                             */
/*       <amplitude>  amplitude                               */
/*       <fmin>       min of random frequency    [default 0.0]*/
/*       <fmax>       max of random frequency                 */
/*       <bwdth>      max of random frequency    [default FSL/2.0] */

/* **** function head trimming ***************************** */
#define check_duser_rsin_03(a,b,c,d) check_duser_rsin(  b,        c,d)
#define  init_duser_rsin_03(a,b,c,d)  init_duser_rsin(  b,        c,d)
#define reset_duser_rsin_03(a,b,c,d) reset_duser_rsin(  b,        c,d)
#define close_duser_rsin_03(a,b,c,d) close_duser_rsin(              )
#define       duser_rsin_03(a,b,c,d)       duser_rsin(a,            d)

#define check_duser_rsin_02(a,b,c)   check_duser_rsin(  0.0,b,      c)
#define  init_duser_rsin_02(a,b,c)    init_duser_rsin(  0.0,b,      c)
#define reset_duser_rsin_02(a,b,c)   reset_duser_rsin(  0.0,b,      c)
#define close_duser_rsin_02(a,b,c)   close_duser_rsin(               )
#define       duser_rsin_02(a,b,c)         duser_rsin(a,            c)

#define check_duser_rsin_01(a,b)     check_duser_rsin(  0.0,FSL/2.0,b)
#define  init_duser_rsin_01(a,b)      init_duser_rsin(  0.0,FSL/2.0,b)
#define reset_duser_rsin_01(a,b)     reset_duser_rsin(  0.0,FSL/2.0,b)
#define close_duser_rsin_01(a,b)     close_duser_rsin(               )
#define       duser_rsin_01(a,b)           duser_rsin(a,            b)

/* **** function prototypes ******************************** */
/*                  amplitude  fmin    fmax   id              */
```

```
void check_duser_rsin(        double, double, int);
void  init_duser_rsin(        double, double, int);
void reset_duser_rsin(        double, double, int);
void close_duser_rsin(         void              );
double     duser_rsin(double,                int);


#ifdef COMPILE_duser_rsin   /* ---- CONDITIONAL COMPILATION -START- */

#undef  RSIN_MAX
#define RSIN_MAX   COMPILE_duser_rsin   /* number of rsin functions */

/* **** global variables ************************************** */
double rsin_fr[RSIN_MAX];

double rsin_ph[RSIN_MAX];

/* **** resources ********************************************* */
#include "Function/random.hdr"       /* containing ‘rand_uniform()’ */

/* **** function definitions ********************************** */
void check_duser_rsin(double fmin, double fmax, int id) {
  if ((fmin < 0.0) || (fmax <= 0.0) || (fmin > fmax)) {
    fprintf(stderr, "NAPA Run Time Error (rsin[%d])", id);
    fprintf(stderr, " inconsistent frequencies [%g, %g]\n", fmin, fmax);
    napa_exit(EXIT_FAILURE);
  }
  return;
}

void init_duser_rsin(double fmin, double fmax, int id) {
  reset_duser_rsin(fmin, fmax, id);
  return;
}

void reset_duser_rsin(double fmin, double fmax, int id) {
  rsin_fr[id] = rand_uniform(fmin, fmax);
  rsin_ph[id] = rand_uniform(0.0, _2PI_);
  return;
}

void close_duser_rsin(void) {
  return;
}

double duser_rsin(double ampl, int id) {
  return ampl * SIN((_2PI_ * rsin_fr[id]* TIME) + rsin_ph[id]);
}

#endif   /* ----------------------- CONDITIONAL COMPILATION -END-- */

#endif                                              /* __RSIN_HDR__ */
```

Here is a netlist using this enhanced user-defined function, with the three possible syntaxes:

File “ex12.nap”

```
header <napa.hdr>
header "rsina.hdr"

fs 1.0e6

node a  duser  rsin  1.0   100.0  1000.0    // frange = [100.0, 1000.0]
node b  duser  rsin  2.0 10000.0            // frange = [0.0,  10000.0]
node c  duser  rsin  3.0                     // frange = [0.0, 500000.0]

output "stdout"  a  b  c

terminate LOOP_INDEX > 1000LL
```

Of course, the pathname of the file “rsina.hdr” could be added to the index of user-defined functions “User/_user.lib” as described earlier. The five functions must exist but could be empty. This mechanism provides remarkable user-friendliness, as every possible scheme is realizable by sharing the process between the initialization phase, the main simulation loop and the close-down phase.

Compute in the reset function the data that must not change during the simulation. Minimize the computations in the function itself by precomputing what you can in the initialization and the reset functions.

The simulator written by the transpiler has an **I/O manager** you can use to open and close files. In the ***NAPA*** package, you will find other resources (like a dynamic memory manager) to help you build the simplest and most powerful functions possible. These resources are located in a sub-directory of the generic header library and are built to help the user.

HOW IT WORKS

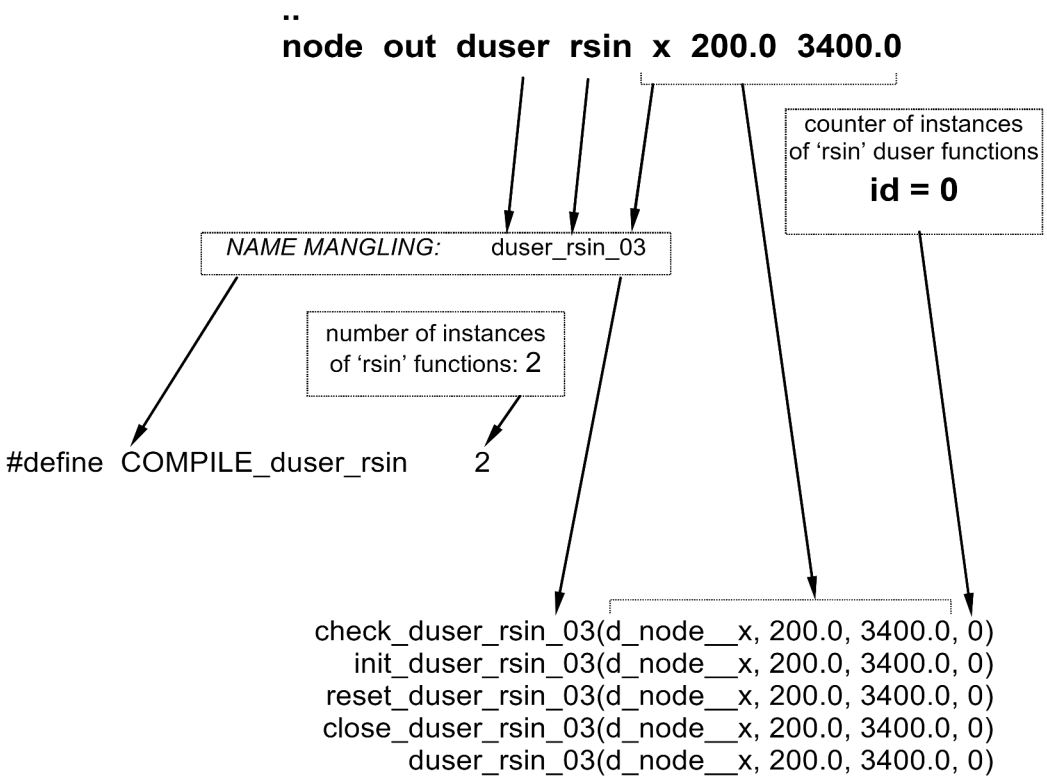


Figure 20 - Name mangling for a user-defined function “duser”

And here is the way the ***NAPA*** transpiler places these lines of code:

File “simulator.c”

```
..
void napa_init(void);
void napa_close(void);
..

#define COMPILE_duser_rsin   2
..

#include  "C:/Simulate/NapaDos/Hdr/napa.hdr"
#include  "rsina.hdr"

..

int main(void) {

  napa_control_init();
  napa_reset_variables();
  napa_reset_nodes();

  check_duser_rsin_03(dnode_x, 200.0, 3400.0, 0);
  check_duser_rsin_01(dnode_y, 1);
  init_duser_rsin_03(dnode_x, 200.0, 3400.0, 0);
  init_duser_rsin_01(dnode_y, 1);
```

```
  ..

  do {
    napa_abs_time = ..;
    ..
    dnode_out = duser_rsin_03(dnode_x, 200.0, 3400.0, 0);
    dnode_s01 = duser_rsin_01(dnode_y, 1);
    ..
    napa_abs_loop++;
  } while ( .. );

  close_duser_rsin_03(dnode_x, 200.0, 3400.0, 0);
  close_duser_rsin_01(dnode_y, 1);
  napa_exit(EXIT_SUCCESS);  /* napa_close() runs here, through atexit() */

}
..
```

Often, the user-defined function includes macros to configure on the fly the behavior of the function. The NAPA transpiler verifies that directives in the NAPA netlist are actually used in the included headers. A warning is issued if the macro is not registered by the user. When such a macro is expected to be used as a directive in the NAPA netlist, it is, therefore, useful to register the macro as illustrated below:

File “mysim.nap”

```
header <napa.hdr>
header "somecode.hdr"

fs 1.0e6

directive DOTHIS     TRUE

...

terminate LOOP_INDEX > 1000LL
```

File “somecode.hdr”

```
#ifndef __SOMECODE_HDR__
#define __SOMECODE_HDR__

...
#define DOTHIS_IS_REGISTERED    /* to register directive DOTHIS */
...

#endif
```

ADVANCED USERS: OPTIONS

The way the macro definition corresponding to the instruction “*directive*” is placed shows how ***NAPA*** is capable of sending messages to the macro preprocessor to configure the ***C*** code for compilation. It is, therefore, possible to implement several variants of a user-defined function using proper macro definitions and macro preprocessor switches. A default value should always be provided; the user can configure the function through the use of instructions “*directive*”.

In some cases, we would like to define some options as text between parentheses:

```
header <napa.hdr>
header <User/sequence.hdr>

fs 1.0e6
node out  iuser  sequence  -10 10 (shuffle)  // user function with option
...
terminate LOOP_INDEX > 1000LL
```

This mechanism will be explained in a specific paragraph a few pages below.

“dtool”, “itool” or “tool”

The tool is the most powerful primitive of ***NAPA.*** These nodes are intended to be used to implement the analysis tools necessary for the simulation and to take control of the simulation. The mechanism is based on the mechanism of the “*duser*” and “*iuser*” nodes with several additional features.

Please refer to the paragraph concerning the “duser” and “iuser” as these user tools extend the same concept.

The most significant characteristic of the tool is its ability to synchronize automatically with the other tools of the simulation.

Please note that the instruction “tool” is an expansion and is equivalent to a node “itool” returning “void”.

WHEN?

If your function is used to analyze a result and not to produce a signal, use the nodes “*itool*” and “*dtool*”.

AN EXAMPLE

Here is an example of an “*itool*” printing the mean value of “num” samples of a signal.

File “mean.hdr”

```
#ifndef __MEAN_HDR__
#define __MEAN_HDR__

/* ********************************************************************** */


/*   USAGE:   node  <no>  itool  mean  <"filnam">  <ni> <norm>  <num>      */

/*   Where    <"filnam"> is the pathname of the output file                */
/*            <ni>       is the identifier of the node to be analyzed      */
/*            <norm>     is the reference level for the analysis of <ni>   */
/*            <num>      is the number of points to analyze                */
/*            <no>       is the number of single tasks already done        */

/*   THIS TOOL IS SYNCHRONIZABLE.                                          */

/* ** FUNCTION HEAD TRIMMING ************************************************ */

#define check_itool_mean_04(a,b,c,d,e) check_itool_mean(         d,e)
#define reset_itool_mean_04(a,b,c,d,e) reset_itool_mean(           e)
#define  init_itool_mean_04(a,b,c,d,e)  init_itool_mean(a,  c,#b,d,e)
#define close_itool_mean_04(a,b,c,d,e) close_itool_mean(a,         e)
#define       itool_mean_04(a,b,c,d,e)       itool_mean(  b,c,     e)

/* ** PROTOTYPES ************************************************************ */

void check_itool_mean(                          long, int);
void reset_itool_mean(                                int);
void  init_itool_mean(char*,        double, char*, long, int);
void close_itool_mean(char*,                          int);
I_TYPE     itool_mean(      double, double,               int);

void mean_compute_and_print(long, int);
double mean_function(int);

/* ** INCLUDE RESOURCES ***************************************************** */

#include "../Resource/dm.hdr"            /* dynamic_memory_manager(..)    */
```

```
#include "../Resource/format.hdr"         /* remove_prefix(..)                  */

/* ** MACRO CONSTANTS ************************************************** */

#ifdef COMPILE_itool_mean                 /* compilation control directive      */

#define MEAN_NUM_INITIAL_IS_REGISTERED

#if defined(MEAN_NUM_INITIAL_IS_EMPTY) || !defined(MEAN_NUM_INITIAL)
 #undef  MEAN_NUM_INITIAL
 #define MEAN_NUM_INITIAL     0LL
#endif

#define MEAN_MAX_IS_REGISTERED

#undef  MEAN_MAX
#define MEAN_MAX    COMPILE_itool_mean    /* number of tools “mean”             */

#define MEAN_RMS_IS_REGISTERED

#ifndef MEAN_RMS
#define MEAN_RMS   YES                    /*RMS or LINEAR MEAN                  */
#endif

/* ** GLOBAL VARIABLES ************************************************* */

double *mean_in_array[MEAN_MAX];

long    mean_num[MEAN_MAX];               /* store counter                      */
long    mean_np[MEAN_MAX];                /* number of points                   */
FILE   *mean_fp[MEAN_MAX];                /* output file pointer                */

/* ** TOOL DEFINITION ************************************************** */

void check_itool_mean(long acc, int id) {
  if (acc < 2) {
    fprintf(stderr, "NAPA Run Time Error: (mean[%d])\n", id);
    fprintf(stderr, " A minimum of 2 points must be accumulated\n");
    napa_exit(EXIT_FAILURE);
  }
  return;
}

void reset_itool_mean(int id) {
  long i;
  for (i = 0; i < mean_np[id]; i++) {
    mean_in_array[id][i] = 0.0;
  }
  return;
}

void init_itool_mean(char *fnam, double norm, char *nm, long acc, int id) {
  mean_np[id] = acc;               /* FIXED as memory is allocated accordingly */
  (void) DA_MANAGER(ALLOCATE,  &(mean_in_array[id]),   acc,  "mean");
  (void) IO_MANAGER(OPENWRITE, &(mean_fp[id]), fnam, ".out", "mean");
  fprintf(mean_fp[id], "# %s\n", TITLE);
#if  MEAN_RMS == YES
  fprintf(mean_fp[id], "# (tool              ) RMS mean\n");
#else
  fprintf(mean_fp[id], "# (tool              ) mean\n");
#endif
  fprintf(mean_fp[id], "# (transpiler version  ) %s\n",    NAPA_VERSION);
  fprintf(mean_fp[id], "# (source file       ) %s\n",    SOURCE);
  fprintf(mean_fp[id], "# (random seed       ) %lld\n",  RANDOM_SEED);
  fprintf(mean_fp[id], "# (normalisation     ) %s/%g\n", remove_prefix(nm), norm);
  fprintf(mean_fp[id], "# (number of samples ) %ld\n",   acc);
  fprintf(mean_fp[id], "# (sampling frequency) %g Hz\n", FSL);
  fprintf(mean_fp[id], "# (sampling period   ) %g s\n",  1.0 / FSL);
```

```
  fprintf(mean_fp[id], "# \n");
  fprintf(mean_fp[id], "# \n");          /* lines beginning with "# " are       */
  fprintf(mean_fp[id], "# \n");          /* ignored by software such as GNUPLOT */
  fprintf(mean_fp[id], "# \n");
  fprintf(mean_fp[id], "# %s\n", CREATED);
  fprintf(mean_fp[id], "# packet  mean_value");
#ifdef EXPORT
  fprintf(mean_fp[id], "%s", E_HEAD);
#endif
  fprintf(mean_fp[id], "\n");
  napa_msg->n = TOOL_WAIT;               /* initialize the tool state machine   */
  return;
}

void close_itool_mean(char *fnam, int id) {
  (void) DA_MANAGER(FREE,  &(mean_in_array[id]), mean_np[id],  "mean");
  (void) IO_MANAGER(CLOSE, &(mean_fp[id]),       fnam, ".out", "mean");
  return;
}

I_TYPE itool_mean(double x, double norm, int id) {
  static long packet[MEAN_MAX];
  long   *num;
  long   *pak;
  double xr;

  num = &(mean_num[id]);
  pak = &(packet[id]);

  switch (napa_msg->n) {                 /* status of the state machine         */

  case TOOL_WAIT:
    if (STOP == napa_msg->i) {           /* wait for start signal               */
      break;
    }
    reset_itool_mean(id);
    *num = -((long) MEAN_NUM_INITIAL);   /* countdown is starting               */
    napa_msg->n = TOOL_COUNTDOWN;
    [[fallthrough]];
  case TOOL_COUNTDOWN:
    if (*num < 0) {
      (*num)++;                          /* continue the countdown              */
      break;
    }

    napa_msg->n = TOOL_ACCUMUL;          /* data accumulation is starting       */
    [[fallthrough]];
  case TOOL_ACCUMUL:
    xr = x / norm;
    mean_in_array[id][*num] = xr;
    (*num)++;
    if (*num < mean_np[id]) {            /* continue to accumulate data         */
      break;
    }

    napa_msg->n = TOOL_COMPUTE;          /* analysis of data is starting        */
    [[fallthrough]];
  case TOOL_COMPUTE:
    mean_compute_and_print(*pak, id);
    (*pak)++;
    napa_msg->i = STOP;
    napa_msg->n = TOOL_WAIT;             /* tool waiting for start signal       */
    break;
  }

  napa_msg->o = *pak;
  return napa_msg->o;
}
```

```
void mean_compute_and_print(long packet, int id) {
  double m;
  m = mean_function(id);
  fprintf(mean_fp[id], " %3ld % 14e ", packet, m);
#ifdef EXPORT
  fprintf(mean_fp[id], E_FORMAT, E_LIST);
#endif
  fprintf(mean_fp[id], " \n");
  return;
}

double mean_function(int id) {
  double s, mval;
  long   i;
  s = 0.0;
#if  MEAN_RMS == YES
  for (i = 0; i < mean_np[id]; i++) {
    s += mean_in_array[id][i] * mean_in_array[id][i];
  }
  mval = sqrt(s / mean_np[id]);
#else
  for (i = 0; i < mean_np[id]; i++) {
    s += mean_in_array[id][i];
  }
  mval = s / mean_np[id];
#endif
  return mval;
}

#endif                                                  /* COMPILE_itool_mean */

/* ************************************************************************** */

#endif                                                       /* __MEAN_HDR__ */
```

Here is an example of a netlist using this user-defined primitive, supposing that the pathname of the header file is properly placed in the index of tools “Tool/_tool.lib”:

File “ex13.nap”

```
header <napa.hdr>
header <toolbox.hdr>

fs 1.0e6

dvar  ampl  logsweep(TOOL_INDEX, 0.1, 10, 11)   &update

node  a     triangle 0.0 ampl 1234.0 0.0
node  void  itool  mean  "stdout"  a 1.0  10000

terminate   TOOL_INDEX > 10
```

or simply

```
header <napa.hdr>
header <toolbox.hdr>

fs 1.0e6

dvar  ampl  logsweep(TOOL_INDEX, 0.1, 10, 11)   &update

node  a     triangle 0.0 ampl 1234.0 0.0
tool        mean  "stdout"  a 1.0  10000
```

```
terminate   TOOL_INDEX > 10
```

HOW IT WORKS

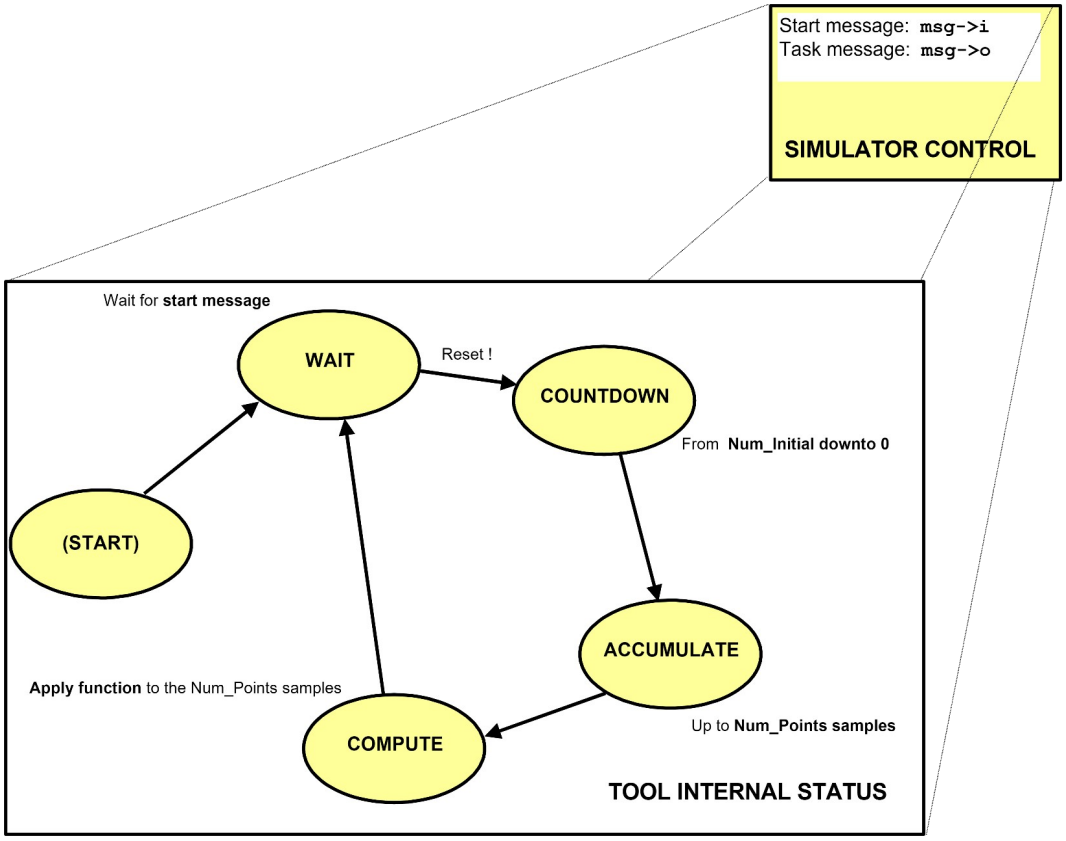


Figure 21 - The synchronization mechanism of a tool

- During the wait state
  The tool is waiting for authorization to start. It is doing nothing. Its answer through the task message is the number identifying the completion of the previous task. The tool quits this state as soon as the start message is set to ‘true’. The simulator is allowed to set this start message to ‘true’ but has no way to reset it. When quitting this state, this is the right moment to reset the internal array corresponding to the tool instantiation.
- During the countdown state, the tool starts the sampling counter with a negative value (-MEAN_NUM_INITIAL), the number of samples to skip, given by an optional directive. This is to allow the system to be in a steady-state mode. During this state, the tool increments the sampling counter. Its answer through the task message is the number identifying the completion of the previous task. The tool quits this state as soon as the sampling counter equals 0.
- During the accumulate state,
  the tool stores the current value of the input signal inside an internal array particular for each instantiation of the tool. It increments the sampling counter accordingly. Its answer through the task message is the number identifying the completion of the previous task. The tool quits this state as soon as the number of stored samples reaches the number of samples requested by the simulator.
- During the compute state,
  the tool performs the analysis on the array of samples accumulated during the previous state. The results are output in the output file with the proper format. The answer of the tool through the task message is the number identifying the completion of the current task. The tool quits this state immediately by resetting the start message.

Corresponding to the answer by the task message, the tool returns a value to the node which is equal to the number corresponding to the last task which is completed.

The simulator ensures the synchronization. The simulator manages a **mailbox** with a pointer ‘napa_msg’ for each instantiation of a tool. In this mailbox, there are two messages: a start message `'napa_msg->i'` and a task message `'napa_msg->o'` and other useful information (node name ‘napa_msg->u’, tool name `'napa_msg->t'`, segment `'napa_msg->s'`, mailbox id `'napa_msg->m'` and user-defined integer `'napa_msg->n'` and a few pointers left to users).

The mailbox is a structure defined as:

```
..
typedef struct {
  long      *store_i;                       /* left to user */
  I_TYPE    *store_l;                       /* left to user */
  R_TYPE    *store_d;                       /* left to user */
  C_TYPE   **store_s;                       /* left to user */
  I_TYPE     n;                             /* left to user */
  I_TYPE     i;                             /* message in   */
  I_TYPE     o;                             /* message out  */
  long       m;                             /* mailbox ID   */
  long       s;                             /* segment ID   */
  char       t[64];                         /* tool name    */
  char       u[2047];                       /* node name    */
} MAILBOX_RECORD;

..
MAILBOX_RECORD *napa_msg;
..
```

At each main loop period, the simulator checks the answer from the various tools. If at least one tool does not return the answer corresponding to the task to be completed, the simulator lets the tools run. As soon as all the tools return an answer corresponding to the task to be completed, the simulator increments its task counter “TOOL_INDEX” and sets all the start messages in the mailbox to ‘TRUE’. All the tools are therefore requested to start synchronously.

There are therefore two kinds of output from a tool:

- **an explicit output corresponding to the output value of the node**
- **an implicit output through the mailbox (napa_msg->o)**

We have seen that the answer through the mailbox controls the value of ‘TOOL_INDEX’. This index is the preferred way to know the status of the entire set of tools. This is the recommended index to test for the termination of the simulation. In some cases, we could be interested in the status of a particular tool; in this case, the output of the node provides the requested information.

It is interesting to note that this synchronization mechanism is not mandatory. It is possible to write tools like any other user-defined functions “*duser*” or “*iuser*”. Of course, this kind of tool will ignore the synchronization messages and will run at its own pace. Not to disturb the other tools, a small trick is used: the simulator writes in the task message of the tool mailbox the answer corresponding to the completion of the current task just before the function call. A tool with a synchronization mechanism will immediately reset the message to the previous task number if its task is not completed, warning the simulator that the task is not completed. The tool without the synchronization mechanism will fail to answer, leaving a task message meaning ‘*my task is completed*’. Therefore this tool will not disturb the functionality of the simulator nor the synchronization of the other tools!

The tool described here is a very general type. Other tools may not accumulate data, or may compute at each sample period. The user could take the previous description as an example or could dig in the generic header library for other examples.

The mailbox is built and managed automatically. Among its fields are the input message (‘false’ or ‘true’), the output message (the number of the last packet of data completely processed), the mailbox number and the number of the segment containing the tool; the last two are useful to write complex tools.

ADVANCED USERS

Have a look at the code of the example. To simplify the code, several resource managers are called. You are encouraged to use them. It is particularly important to detect generic tasks and write the corresponding resource manager.

One resource manager is built into the ANSI-C code produced by the ***NAPA*** transpiler. This is the resource manager of the I/O streams: **‘IO_MANAGER(...)’**. Another one is built only when arrays of pointers are defined in the ***NAPA*** user's netlist.

Other resource managers are located in the generic header library to help the writing of the user's header files. It is highly recommended to use them. In some situations these resource managers can speed up the simulation: the “SC_MANAGER(...)” is capable of initializing the sine and cosine tables of the FFT and window functions, using the shortest way for the computations and offering an already built table to tools when it is possible.

“post”

This instruction extends the capability of the tools, allowing postprocessing the output files of tools (and time-domain output from the instruction “output”).

In some cases, we would like to define some options as a string. We must avoid collision of identifiers between these options and nodes or variables. The same mechanism as described in the “duser” paragraph is available.

WHEN?

When an output file must be postprocessed.

AN EXAMPLE

File “analyze.hdr”

```
#ifndef __ANALYZE_HDR__
#define __ANALYZE_HDR__

/* ********************************************************************** */

/*   USAGE:   post analyze  <filnam>  <col>                               */

/*   Where    <"filnam"> is the pathname of the output file               */
/*            <col>      is the column number to postprocess              */

/* ** FUNCTION HEAD TRIMMING ************************************************ */

#define prepare_post_analyze_02(a,b,c,d,e)  prepare_post_analyze(a,    d,e)
#define execute_post_analyze_02(a,b,c,d,e)  execute_post_analyze(  b,c,d,e)

/* ** PROTOTYPES ************************************************************ */

void prepare_post_analyze(char*,               int, int);
void execute_post_analyze(       char*, char*, int, int);

void analyze_read_file(char*, double**, int, int);
void analyze_compute(double*, double*, double*, double*, int);

void analyze_print(char*, double, double, double, int);

/* ** MACRO CONSTANTS ******************************************************* */

#ifdef COMPILE_post_analyze              /* compilation control directive */

#undef  ANALYZE_MAX
#define ANALYZE_MAX   COMPILE_post_analyze  /* number of analyze tools     */

/* ** GLOBAL VARIABLES ****************************************************** */

/* ** TOOL DEFINITION ******************************************************* */

void prepare_post_analyze(char *tool, int column, int id) {
  if (column < 1) {
    fprintf(stderr, "NAPA Run Time Error: (analyze[%d] of %s)\n", id, tool);
    fprintf(stderr, " column number must be larger than 0\n");
    napa_exit(EXIT_FAILURE);
  }
  return;
}
```

```
void execute_post_analyze(char *filin, char *filout, int column, int id) {
  double *array = NULL;
  double number, mean, sigma;
  analyze_read_file(filin, &array, column, id);
  analyze_compute(array, &number, &mean, &sigma, id);
  analyze_print(filout, number, mean, sigma, id);
  return;
}

void analyze_read_file(char *fi, double **a, int c, int id) {
  ...
  return;
}

void analyze_compute(double *a, double *n, double *m, double *s, int id) {
  ...
  return;
}

void analyze_print(char *fo, double n, double m, double s, int id) {
  ...
  return;
}

#endif                                              /* COMPILE_post_analyze */

/* ********************************************************************** */

#endif                                                  /* __ANALYZE_HDR__ */
```

In the simulator, the first function “prepare_post_analyze_02” is placed after the initialization of the tools. The second function “execute_post_analyze_02” is placed after the close of the tools.

Please note that ***NAPA*** automatically places the parameters corresponding to the file to process and the ID number.

Here is an example of a netlist using this postprocessing function, supposing that the pathnames of the header files are properly placed in the index of postprocessing functions “Post/_post.lib”:

File “ex29.nap”

```
header <napa.hdr>
header <toolbox.hdr>

fs 1.0e6

node  a  noise 0.0 0.1
node  b  sine   0.0 1.0 1234.0 0.0
node  s  sum   a  b

tool           tsnr     "tsnr.out"  s 1.0 5000.0 10000
post           analyze  "stat0.out"  3

output "signal.out"  s a b
post           analyze  "stat1.out"  1
post           analyze  "stat2.out"  2
post           analyze  "stat3.out"  3

terminate   TOOL_INDEX > 100
```

HOW IT WORKS

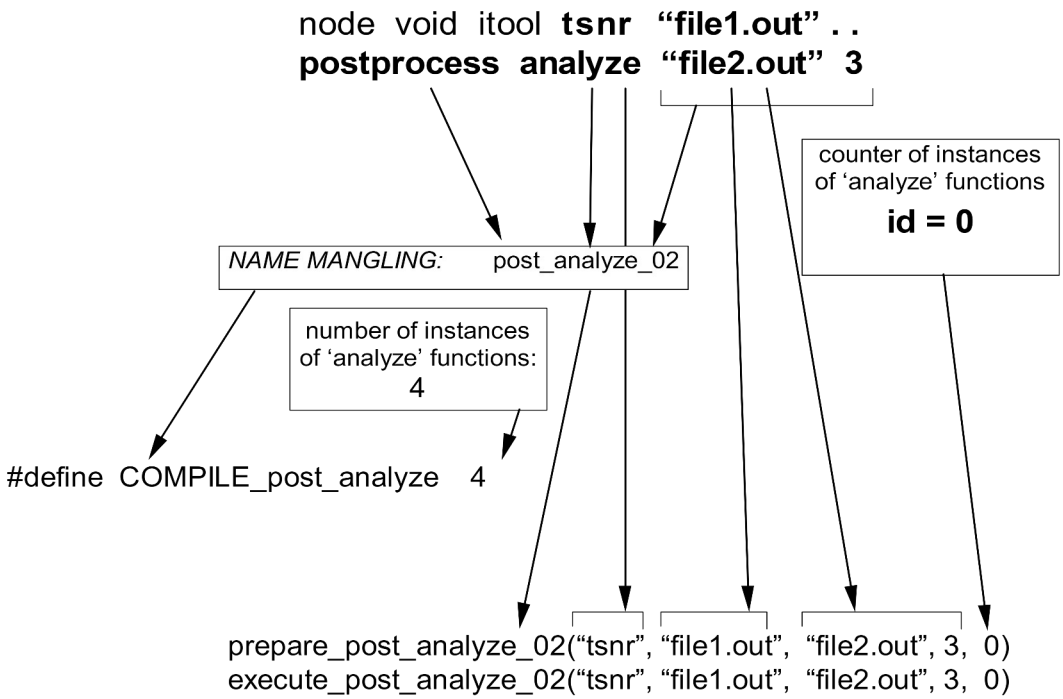


Figure 22 - Name mangling for a postprocessing function

ADVANCED USERS

Postprocessors can be used alone to process the output files of a previous ***NAPA*** simulation. The instruction “void” sets the file to process.

```
header <napa.hdr>
header <toolbox.hdr>

void   "file.out"
post    analyze  "stat.out"  3   // analyze the file "file.out"

terminate
```

Options

(<option>)

WHEN?

In a user-defined function, a user-defined tool or a postprocessing function, as an optional parameter.

AN EXAMPLE

It is possible to define options for the nodes “duser”, “iuser”, “dtool” and “itool” and for the instruction “*post*”. The ***NAPA*** transpiler automatically builds appropriate macro functions which contain all the mechanisms to query the existence of an option for an instantiation of such functions. The writer of a user function may call the macro functions “ISOPTION( )” and/or “ISNOTOPTION( )”, as in this example where the itool “wonder” has two options: (do) and (do_not):

File “myfile.nap”

```
...
tool  wonder "fila.out"  s1 1.0 2500.0   (do)
...
tool  wonder "filb.out"  s4 1.0 5000.0
...
```

The user-defined tool:

File “wonder.hdr”

```
int itool_wonder_opt[WONDERMAX];      /* options of itool ‘wonder’ */

...
void check_itool_wonder(... , int id) {
  ...
  if        (ISOPTION("itool_wonder",id, "do"     )) {
    itool_wonder_opt[id] = 1;
  } else if (ISOPTION("itool_wonder",id, "do_not"  )) {
    itool_wonder_opt[id] = 0;
  } else {
    itool_wonder_opt[id] = 0;         /* default is (do_not)       */
  }
  if        (ISNOTOPTION("itool_wonder",id)) {
    fprintf(stderr, "\nNAPA Run Time Error: (wonder[%d])\n", id);
    fprintf(stderr, " Option is not valid\n");
    fprintf(stderr, " Valid keywords are: (do), (do_not)\n\n");
    napa_exit(EXIT_FAILURE);
  }
  ...
  return;
}
...
```

HOW IT WORKS

The ***NAPA*** transpiler writes an “ad hoc” function “napa_check_for_option()” including options, functions and instances from the ***NAPA*** netlist, with the corresponding macro function 'ISOPTION()':

```
...
#define ISOPTION(f,i,o) (0.0<(*napa_check_for_option(f,i,o,1)))
#define ISNOTOPTION(f,i) (ISOPTION(f,i,"_another_"))
#define ISPARAMETER(f,i,o) \
        (((double*)NULL)!=napa_check_for_option(f,i,o,0))
#define GETPARMADDRESS(f,i,o) napa_check_for_option(f,i,o,0)
#define ISDELAYED(f,i) (ISOPTION(f,i,"delayed_node"))
...
double *napa_check_for_option(const char *strfun, long id,
                              const char *stropt, int ctr) {
  long n;
  long o = -1L;
  long f = -1L;
  double *r0;
  static double zero = 0.0;
  static double one = 1.0;
  const char *list_fun[2] = { "itool_wonder", "duser_pwl" };
  const char *list_opt[2] = { "do", "periodic" };
  double *list_val[2] = { NULL, NULL };
  long nfun = 2L;
  long nid = 2L;
  long nopt = 2L;
  static int table_opt[2][2][2] = {
    { { 1, 0 },
      { 0, 0 } },
    { { 0, 1 },
      { 0, 0 } }
  };
  r0 = ( 0 == ctr ) ? (double*) NULL : &zero;
  for (n = 0L; n < nfun; n++) {
    if ( 0 == strncmp(list_fun[n], strfun, 63) ) {
      f = n;
      break;
    }
  }
  if ( (-1L == f) || (0L > id) || (nid <= id) ) {
    return r0;
  }
  if ( 0 == strncmp("_another_", stropt, 63) ) {
    if ( 0 != ctr ) {
      for (n = 0L; n < nopt; n++) {
        if ( 0 < table_opt[f][id][n] ) {
          return &one;
        }
      }
      return &zero;
    } else {
      return (double*) NULL;
    }
  }
  for (n = 0L; n < nopt; n++) {
    if ( 0 == strncmp(list_opt[n], stropt, 63) ) {
      o = n;
```

```
      break;
    }
  }
  if ( -1L == o ) {
    return r0;
  }
  if ( 0 != ctr ) {
    if ( 0 < table_opt[f][id][o] ) {
      table_opt[f][id][o] = -table_opt[f][id][o];
      return &one;
    }
  } else {
    return list_val[ABS(table_opt[f][id][o])];
  }
  return r0;
}
...
```

This function is called by the user's function or tool or the postprocess function like in this example:

option = ISOPTION("itool_wonder", 0, "do");

will return TRUE at the first call. It returns FALSE afterwards. At the end of the option identification, it is recommended to call

error = ISNOTOPTION("itool_wonder", 0);

to check for any spurious option.

The function "napa_check_for_option()" will be simplified by the **NAPA** transpiler if it makes sense, and could be as simple as this, when no user function, tool or postprocessing function of the netlist has an option:

```
...
double *napa_check_for_option(const char *strfun, long id,
                              const char *stropt, int ctr) {
  (void) strfun; (void) id; (void) stropt;
  static double zero = 0.0;
  if ( 0 == ctr ) {
    return (double*) NULL;
  } else {
    return &zero;
  }
}
...
```

Multiple Output User's Functions

WHEN?

Sometimes, a user function ("duser", "iuser", "dtool" or "itool") needs multiple outputs. The ***NAPA*** node has by construction a single output. It is nevertheless quite simple to write a user function with multiple outputs.

A first function call triggers the computation of all outputs. This function stores all the results in global variables but outputs its id value as a tag. This tag is then used as an entry for the other function calls corresponding to the specified output.

AN EXAMPLE

File "sun.nap"

```
...
header  "./sun.hdr"
...
#*   geolocation for the town of Nice
dvar latitude       43.70
dvar longitude    7.25
dvar timezone        1

#*    computation of the position of the sun
node nice     duser sun   latitude longitude  2009 APRIL 1  13.0 timezone
node az       duser sun   nice  (azimuth)
node alt      duser sun   nice  (altitude)
node sunset duser sun     nice  (sunset)
...
```

File "sun.hdr"

```
#ifndef __SUN_HDR__

/* ************************************************************** */

/** USAGE computes the position of the sun as a function of etc ..  */

/* ************************************************************** */

#define check_duser_sun_08(a,b,c,d,e,f,g,h) \
        check_duser_sun(a,b,c,d,e,f,g,h)

#define init_duser_sun_08(a,b,c,d,e,f,g,h) \
        init_duser_sun(a,b,c,d,e,f,g,h)

#define reset_duser_sun_08(a,b,c,d,e,f,g,h) \
        reset_duser_sun(a,b,c,d,e,f,g,h)

#define duser_sun_08(a,b,c,d,e,f,g,h) \
        duser_sun(a,b,c,d,e,f,g,h)

#define close_duser_sun_08(a,b,c,d,e,f,g,h) \
```

```
        close_duser_sun(a,b,c,d,e,f,g,h)

#define check_duser_sun_01(a,b)  check_duser_extract_sun(a,b)

#define init_duser_sun_01(a,b)   init_duser_extract_sun(a,b)
#define reset_duser_sun_01(a,b)  reset_duser_extract_sun(a,b)

#define duser_sun_01(a,b)        duser_extract_sun(a,b)
#define close_duser_sun_01(a,b)  close_duser_extract_sun(a,b)

/* *** PROTOTYPES ********************************************** */

void   check_duser_sun(double,double,long,long,long,double,long,int);
void   init_duser_sun(double,double,long,long,long,double,long,int);
void   reset_duser_sun(double,double,long,long,long,double,long,int);
double duser_sun(double,double,long,long,long,double,long,int);
void   close_duser_sun(double,double,long,long,long,double,long,int);

void    check_duser_extract_sun(double,int);
void    init_duser_extract_sun(double,int);
void    reset_duser_extract_sun(double,int);
double duser_extract_sun(double,int);
void    close_duser_extract_sun(double,int);

/* ************************************************************** */

#ifdef  COMPILE_duser_sun

#undef  SUN_MAX
#define SUN_MAX    COMPILE_duser_sun

double sun_azimuth[SUN_MAX];
double sun_altitude[SUN_MAX];
double sun_sunrise[SUN_MAX];
double sun_sunset[SUN_MAX];
double sun_day_length[SUN_MAX];
int    sun_opt[SUN_MAX];

/* ************************************************************** */

void check_duser_sun(..., long timezone, int id) {
  ...
  if (ISNOTOPTION("duser_sun",id)) {
    fprintf(stderr, "\nNAPA Run Time Error: (sun[%d])\n", id);
    fprintf(stderr, " Option is not valid\n\n");
    napa_exit(EXIT_FAILURE);
  }
  ...
  return;
}

void init_duser_sun(..., long timezone, int id) {
  ...
  return;
}
void reset_duser_sun(..., long timezone, int id) {
  ...
  return;
```

```
}

double duser_sun(..., long timezone, int id) {
  ...
  sun_azimuth[id]    = ...;   /* compute and load global variables  */
  sun_altitude[id]    = ...;
  sun_sunset[id]     =  ...;
  sun_sunrise[id]    =  ...;
  sun_day_length[id] =  ...;
  return (double) id; /* returns id as tag for other function calls */
}

void close_duser_sun(..., long timezone, int id) {
  return;
}

/* ************************************************************** */

void check_duser_extract_sun(double tag, int id) {
  if        (ISOPTION("duser_sun", id, "azimuth" )) {
    sun_opt[id] = 0;
  } else if (ISOPTION("duser_sun", id, "altitude")) {
    sun_opt[id] = 1;
  } else if (ISOPTION("duser_sun", id, "sunset")) {
    sun_opt[id] = 2;
  } else if (ISOPTION("duser_sun", id, "sunrise")) {
    sun_opt[id] = 3;
  } else if (ISOPTION("duser_sun", id, "day_length")) {
    sun_opt[id] = 4;
  } else {
    sun_opt[id] = 4;          /* default output                     */
  }
  if        (ISNOTOPTION("duser_sun", id)) {
    fprintf(stderr, "\nNAPA Run Time Error: (sun[%d])\n", id);
    fprintf(stderr, " Option is not valid\n\n");
    napa_exit(EXIT_FAILURE);
  }
  ...
  return;
}

void init_duser_extract_sun(double tag, int id) {
  return;
}

void reset_duser_extract_sun(double tag, int id) {
  return;
}

double duser_extract_sun(double tag, int id) {
  double out;
  long   itag;
  itag = (long) tag;
  if ((itag < 0) || (itag >= SUN_MAX)) {
    fprintf(stderr, "\nNAPA Run Time Error: (sun[%d])\n", id);
    fprintf(stderr, " tag <%ld> overflow\n\n", itag);
    napa_exit(EXIT_FAILURE);
  }
```

```
  switch (sun_opt[itag]) {
  case 0:
    out = sun_azimuth[itag];
    break;
  case 1:
    out = sun_altitude[itag];
    break;
  case 2:
    out = sun_sunset[itag];
    break;
  case 3:
    out = sun_sunrise[itag];
    break;
  case 4:
    out = sun_day_length[itag];
    break;
  }
  return out;
}

void close_duser_extract_sun(double tag, int id) {
  return;
}

#endif


/*****************************************************************  */
#endif                                                  /* __SUN_HDR__ */
```

HOW IT WORKS

The output of the first function call ("tag" in the example below) has a double effect. First, being used in further function calls as input, it forces the ***NAPA*** transpiler to place these function calls after the first one. Second, it indicates the address of the storage area. In this example, this is the ID of the first function call.

NAPA source:

```
...
node  tag   duser sun  lat lon  2009 APRIL 1  13.0  tz
node  az    duser sun  tag  (azimuth)    // refers to 1st function call
node  alt   duser sun  tag  (altitude)   // idem

...
```

Corresponding ***C*** code:

```
...
dnode_tag = duser_sun_08(dvar_lat,dvar_lon,2009,APRIL,1,13.0,dvar_tz, 3);
dnode_az  = duser_sun_01(dnode_tag, 4);
dnode_alt = duser_sun_01(dnode_tag, 5);
...
```

According to the code of file "sun.hdr" above, the value of "dnode_tag" is the ID of the first function call, i.e. 3. The second and third function calls get the data stored by the first call through this ID.

Passing Parameters by Addresses

Virtually all parameters of the ***NAPA*** netlist are passed by values to the internal ***C*** functions. In some cases, when you have to pass numerous parameters, or if you want to pass a variable number of these parameters, it is useful to pass them by address.

NAPA provides a way to do it, but only for sets of ***NAPA*** variables ganged in an array of pointers. See the instruction "ganging" in the ***NAPA*** Reference.

WHEN?

In a user-defined function, a user-defined tool or a postprocessing function.

AN EXAMPLE

File "example.nap"

```
header <napa.hdr>
header <toolbox.hdr>
header <lpfilter.hdr>

fs     1.0e6

dvar   r1   rand_normal(1.0e3, 12.3)
dvar   r2   rand_normal(0.5e3,  4.7)
dvar   c0   110.0e-9

string nam1   "proto1"
string nam2   "proto2"

event  newjob  (new) TOOL_INDEX

update r1   when newjob
update r2   when newjob

ganging    tau1[3]  nam1 r1 c0                        //  gang 3 variables
ganging    tau2[5]  nam2 r1 c0 r2 c0                  //  gang 5 variables

node in    noise  0.0 1.0
node out1  duser  lpfilter  tau1  in                  //  passing array tau1[]
node out2  duser  lpfilter  tau2  in                  //  passing array tau2[]

tool  tf  "tf1.out"  in 1.0  out1 1.0  100000
tool  tf  "tf2.out"  in 1.0  out2 1.0  100000

directive NTF  25

terminate TOOL_INDEX >= 10
```

The user-defined function of this example should be defined to receive these arrays of pointers (see below the description of the type DATA_RECORD).

Using the ***C*** operator '**&**', the user-defined function has access to the parameters ganged in the arrays of pointers (see the instruction "ganging"). As these parameters are passed by addresses, it is possible to access the updated values of the ***NAPA*** variables.

There is a severe risk if the user-defined function reassigns values to these parameters. It could cause dramatic errors in the simulation! Use these reassignments with extreme care.

HOW IT WORKS

The ***NAPA*** transpiler defines a generic structure 'DATA_RECORD' in the ***C*** simulator:

```
..
typedef struct {
  char *name;               /*  containing the name of the array    */
  int   length;             /*  containing the size of the vectors  */
  int   nrow;               /*  number of rows                      */
  int   ncol;               /*  number of columns                   */
  C_TYPE **s_ptr;           /*  pointing to array of names of parms */
  I_TYPE **i_ptr;           /*  pointing to NULL by default         */
  R_TYPE **d_ptr;           /*  pointing to NULL by default         */
  C_TYPE **c_ptr;           /*  pointing to NULL by default         */
  X_TYPE **x_ptr;           /*  pointing to NULL by default         */
} DATA_RECORD;
..
```

and declares structures corresponding to the array of pointers:

```
..
DATA_RECORD  record_tau1;
DATA_RECORD  record_tau2;
..
```

In the ***C*** simulator, there is an appropriate ad-hoc dynamic allocation of the tables, followed by the definition of a few parameters and the setting of pointers:

```
static char tau1[3][64] = {"nam1", "r1", "c0"};
static char tau2[5][64] = {"nam2", "r1", "c0", "r2", "c0"};
..
record_tau1.length = 3;
record_tau1.nrow = 1;
record_tau1.ncol = 3;
strcpy(record_tau1.name, "tau1");
record_tau1.s_ptr[0] =  tau1[0];
record_tau1.c_ptr[0] =  svar_nam1;
record_tau1.s_ptr[1] =  tau1[1];
record_tau1.d_ptr[1] = &dvar_r1;
record_tau1.s_ptr[2] =  tau1[2];
record_tau1.d_ptr[2] = &dvar_c0;

record_tau2.length = 5;
record_tau2.nrow = 1;
record_tau2.ncol = 5;
strcpy(record_tau2.name, "tau2");
record_tau2.s_ptr[0] =  tau2[0];
record_tau2.c_ptr[0] =  svar_nam2;
record_tau2.s_ptr[1] =  tau2[1];
record_tau2.d_ptr[1] = &dvar_r1;
record_tau2.s_ptr[2] =  tau2[2];
record_tau2.d_ptr[2] = &dvar_c0;
record_tau2.s_ptr[3] =  tau2[3];
record_tau2.d_ptr[3] = &dvar_r2;
record_tau2.s_ptr[4] =  tau2[4];
record_tau2.d_ptr[4] = &dvar_c0;
..
```

As the pointers are set to NULL by default in the ***C*** simulator, the pointers not assigned still point to NULL. This provides a simple and reliable way to test the type of variables passed through the array of pointers.

Here, the structure corresponding to "tau1[]" is defined as:

ganging

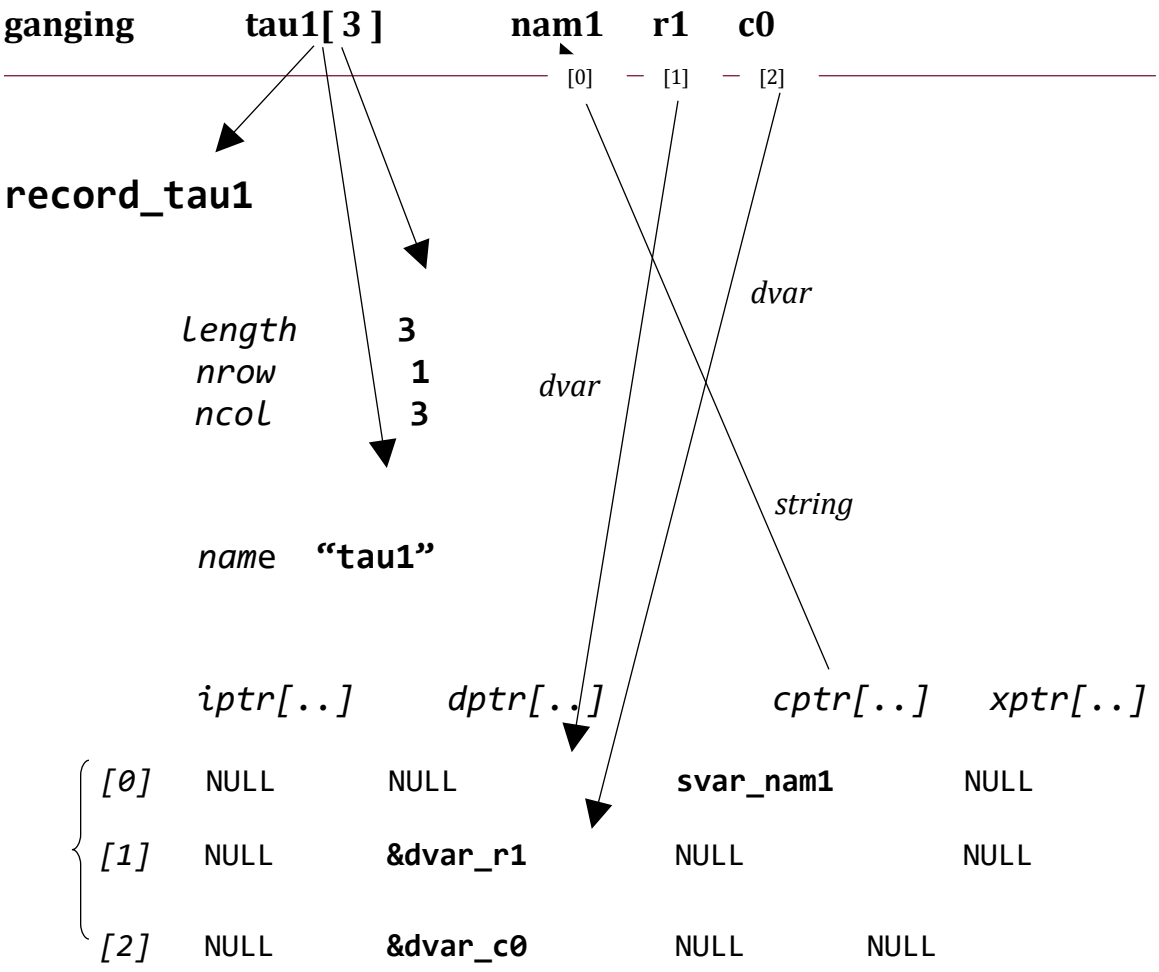


Figure 23 - How *NAPA* variables are passed by addresses

Conclusion

NAPA is more than a transpiler or a simulator. This is an evolving methodology to simulate more and more complex mixed-signal netlists. Each IC we are designing brings new simulation challenges and new solutions. Each time ***NAPA*** integrates the new ideas we have developed for the benefit of its users.

We welcome your feedback. E-mail yves.leduc.be@gmail.com