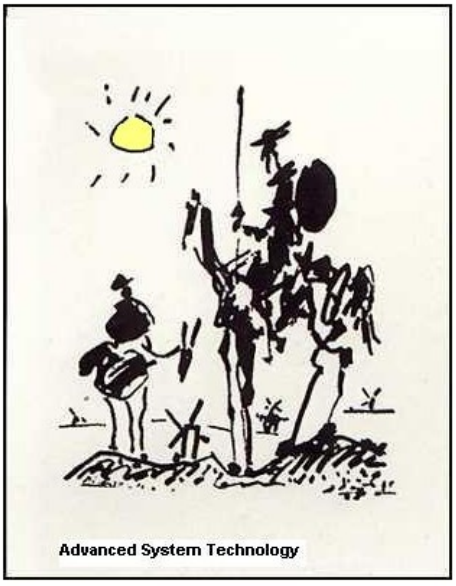
NAPA User's Guide

Version **5.03**, November 4, 2026



Yves Leduc
Greg C. Warwar
Richard K. Hester

Table of Contents

INTRODUCTION **8**

The *NAPA* Transpiler 8
NAPA Identifiers 9
NAPA Iterative Identifiers 9
NAPA Instructions 9
Comments 10
Nodes or Variables? 11

THE CONCEPT OF *NAPA* **12**

The *NAPA* Node 12
The *NAPA* Netlist 13
The Sampling Frequency 13
The Netlist Processing 15
Handling the Delays 17
Node Determination and Sorting 17
The Netlist to *C* Translation 18
The Simulation Control 19
NAPA in a Computer Network 19
NAPA Transpiler Command Line 19
NAPA Portability 21
C Limitations: Important Warnings 21
C Casting: Important Warnings 22
NAPA Process Flow 23
The *NAPA* File System 24
1. Absolute Reference *24*
2. Generic Library Reference *24*
3. Root Directory Reference *24*
4. Main Directory Reference *24*
5. Current Cell Directory Reference *24*
Directory Definitions and Processing 25
Best Practices 25

***NAPA* INSTRUCTIONS** **26**

***NAPA* NODES** **29**

Usage 29
Chameleonic nodes 29
Nodesets 30
Pseudo-Nodes 31
Unnamed and Unused Signals 31
Automatic Nodes 31
Bit Field Extractor 32
Node Kind List 32

***NAPA* USER'S VARIABLES** **35**

Usage 35
Variable Type 35

REGISTER ARITHMETIC **36**

Width limited nodes and variables 36

LIST OF *NAPA* INSTRUCTIONS **39**

ALIAS....39
ARRAY....39
ASSERT....41
CALL....42
COMMAND_LINE....43
COMMENT....44
DATA....44
DEBUG....46
DECIMATE....47
DECLARE....47
DIRECTIVE....49
DROP....51
DUMP....51
DVAR, IVAR....52
ERROR....53
EVENT....53
EXPORT....54
FORMAT....55
FS....56
GANGING....57
GATEWAY....58
HEADER....59
INIT....60
INJECT....61
INPUT....62
CELL_INTERFACE, DATA_INTERFACE....63
INTERLUDE....63
INTERPOLATE....64
IVAR....64
LOAD....65
NAPA_VERSION....65
NODE....66
NOMINAL....66
NUM_INITIAL....67
OPCODE....67
OUTPUT....68
PING....71
POST....71
RANDOM_SEED....73
RESTART....73
STRING....73
STUCK....74
SYNCHRONIZE....75
TERMINATE....75
TITLE....76
TOOL....76
TS....77
UPDATE....77
VOID....79
WARNING....79
 # (COMMENT)....80

INSTRUCTION QUALIFIERS....81

AFTER, BEFORE....81
WHEN....82

WITH....82
(EXPAND) | (NOEXPAND)....83
(NEGATIVE) | (POSITIVE) | (DUAL) | (BOTH)....83
(NO) | (YES)....83
(NOCHECK)....84
(HEX)....84
(DIGITAL)....84
(ANALOG)....84
(STRING)....84
(CONSTANT)....84
(TRUE)....84
(ARITHMETIC) | (GEOMETRIC) | (HARMONIC) | (RMS)....85
(POINTER)....85
(NEW)....85
(<OPTION>[::<PARM>])....85
(<REAL_TYPE_OUTPUT_SCALING>)....86
(<INTEGER_TYPE_OUTPUT_CONFIGURATION>)....86

SHORT FORMS....87

&UPDATE....87
&CONSTANT....87
&EXPORT....88
&DELAYED....88

SPECIAL SYMBOLS....90

$ 90
./ 90
~/ 90
< ... >....90
(...)....90
....91
... // 91
. (ONE DOT)....91
.. (TWO DOTS)....91
... (THREE DOTS)....91
:....91
::....91

NODE KINDS....92

ADC: N-LEVEL SIGNED A/D CONVERTER....92
ALGEBRA: CHAMELEONIC *C* EXPRESSION....93
ALU: USER-DEFINED ALU....94
AND: N-INPUT AND ELEMENT....95
AVERAGE: AVERAGE OF N INPUTS....95
BSHIFT: BARREL SHIFTER....95
BTOI: N-BIT CONVERSION TO UNSIGNED INTEGER....96
BUFFER: NON-INVERTING BUFFER....96
BWAND: N-INPUT BITWISE AND....96
BWBUFFER: BITWISE NON-INVERTING BUFFER....97
BWINV: BITWISE INVERTER....97
BWNAND: N-INPUT BITWISE NAND....97
BWNOR: N-INPUT BITWISE NOR....97
BWNOT: BITWISE NOT....98
BWOR: N-INPUT BITWISE OR....98
BWXNOR: N-INPUT BITWISE XNOR....98

BWXOR: N-INPUT BITWISE XOR....99
CELL: SUBCIRCUIT INSTANTIATION FROM A FILE....99
CHANGE: WATCHDOG....100
CLIP: CLIP ELEMENT....100
CLOCK: DIGITAL CLOCK GENERATOR....101
COMP: COMPARATOR....101
CONST: CONSTANT....102
COPY: SIGNED COPY....103
COSINE: COSINE WAVE GENERATOR....103
DAC: N-LEVEL SIGNED D/A CONVERTER....103
DALGEBRA: *C* EXPRESSION CAST TO REAL TYPE....104
DC: DC SOURCE....104
DELAY: SINGLE OR MULTIPLE DELAY....105
DIFFERENTIATOR: NON-INVERTING DIFFERENTIATOR....106
DIV: DIVIDER ELEMENT....107
DTOI: ANALOG-TO-DIGITAL TYPE CONVERSION....107
DTOOL: USER-DEFINED TOOL....107
DUSER: USER-DEFINED FUNCTION....108
EQUAL: EQUALITY....109
FZAND: N-INPUT AND ELEMENT (FUZZY LOGIC)....110
FZBUFFER: NON-INVERTING BUFFER (FUZZY LOGIC)....110
FZINV: NEGATION ELEMENT (FUZZY LOGIC)....110
FZNAND: N-INPUT NAND ELEMENT (FUZZY LOGIC)....110
FZNOR: N-INPUT NOR ELEMENT (FUZZY LOGIC)....111
FZNOT: NEGATION ELEMENT (FUZZY LOGIC)....111
FZOR: N-INPUT OR ELEMENT (FUZZY LOGIC)....111
FZXNOR: 2 INPUTS XNOR ELEMENT (FUZZY LOGIC)....111
FZXOR: 2 INPUTS XOR ELEMENT (FUZZY LOGIC)....112
GAIN: GAIN ELEMENT....112
GENERATOR: SUBCIRCUIT GENERATION FROM A FILE....112
HOLD: HOLD AND TRACK ELEMENT....113
IALGEBRA: *C* EXPRESSION CAST TO INTEGER TYPE....114
INTEGRATOR: NON-INVERTING INTEGRATOR....114
INV: NEGATION ELEMENT (BOOLEAN LOGIC)....114
ITOB: BIT EXTRACTOR FROM DIGITAL NODE....115
ITOD: DIGITAL-TO-ANALOG TYPE CONVERSION....115
ITOOL: USER-DEFINED TOOL....115
IUSER: USER-DEFINED FUNCTION....116
LATCH: SR LATCH....118
LSHIFT: LEFT SHIFT ELEMENT....118
MAX: MAXIMUM OF N INPUTS....118
MERGE: N-INPUT MULTIPLEXER FROM EXCLUSIVE LOOP SEGMENTS....119
MIN: MINIMUM OF N INPUTS....119
MOD: MODULO DIVIDER ELEMENT....120
MULLER: C MULLER ELEMENT, N INPUTS (BOOLEAN LOGIC)....120
MUX: N-INPUT MULTIPLEXER CONTROLLED BY INTEGER LEVELS....120
NAND: N-INPUT NAND ELEMENT (BOOLEAN LOGIC)....121
NOISE: SOURCE OF NOISE....121
NOR: N-INPUT NOR ELEMENT (BOOLEAN LOGIC)....121
NOT: NEGATION ELEMENT (BOOLEAN LOGIC)....121
OFFSET: DC LEVEL SHIFTER ELEMENT....122
OR: N-INPUT OR ELEMENT (BOOLEAN LOGIC)....122
OSC: OSCILLATOR....122
POLY: POLYNOMIAL OF ORDER N....123
PROD: N-INPUT MULTIPLIER ELEMENT....123

QUANT: QUANTIZER....123
RAM: RANDOM ACCESS MEMORY....124
RAM2: DUAL PORT RANDOM ACCESS MEMORY....124
RECT: RECTIFIER ELEMENT....125
REGISTER: DATA REGISTER....125
RELAY: ONE INPUT RELAY, NORMALLY CLOSED....125
RIP: BITWISE RIP BUS....126
ROM: READ ONLY MEMORY....127
ROM2: DUAL PORT READ ONLY MEMORY....127
RSHIFT: RIGHT SHIFT ELEMENT WITHOUT ROUNDING....128
RSHIFT1: RIGHT SHIFT ELEMENT, ROUNDING HALF AWAY FROM ZERO....128
RSHIFT2: RIGHT SHIFT ELEMENT, ROUNDING HALF TO EVEN....129
SIGN: SIGN OF SIGNAL....129
SINE: SINE WAVE GENERATOR....129
SQUARE: SQUARE WAVE SOURCE....130
STEP: STEP FUNCTION SOURCE....130
SUB: SUBTRACTION ELEMENT....131
SUM: N-INPUT SUMMING ELEMENT....131
TEST: C CONDITION, OUTPUT 0 OR 1....131
TOGGLE: TOGGLE FLIP-FLOP....132
TRACK: TRACK AND HOLD ELEMENT....132
TRIANGLE: TRIANGLE WAVE SOURCE....132
TRIG: TRIGGER (DUAL, POSITIVE OR NEGATIVE EDGE TRIGGER)....133
UADC: N-LEVEL UNSIGNED A/D CONVERTER....134
UDAC: N-LEVEL UNSIGNED D/A CONVERTER....134
WSUM: WEIGHTED SUM OF N INPUTS....135
XNOR: N-INPUT XNOR ELEMENT (BOOLEAN LOGIC)....135
XOR: N-INPUT XOR ELEMENT (BOOLEAN LOGIC)....136
ZERO: INSERTION OF ZEROES....136

NAPA CONSTANTS AND TYPES....137

NAPA CONSTANTS....137
CONSTANT TYPES....139
GENERIC TYPES AND OUTPUT FORMATS....140

GLOBAL VARIABLES....141

NAPA C FUNCTIONS AND MACRO FUNCTIONS....143

AVAILABLE C MACRO FUNCTIONS....143
AVAILABLE C FUNCTIONS....145
USER C FUNCTIONS....146

USER-DEFINED FUNCTIONS AND TOOLS....147

THE CONCEPT....147
TOOL SYNCHRONIZATION....149
AN EXAMPLE OF TOOL....149
RESOURCE MANAGERS....153

APPENDIX A....155

NAPA SIMULATION FLOW. ORDER OF EXECUTION....155
Initialization....155
Main Loop....155
Termination....156

APPENDIX B....157

NAPA Reserved Identifiers....157
APPENDIX C....166
NAPA File Naming Recommendation....166
APPENDIX D....167
NAPA Netlist Example....167
APPENDIX E....172
Quick Reference: *NAPA* instructions....172
APPENDIX F....175
Quick Reference: Node Syntax....175
APPENDIX G....181
Quick Reference: The *NAPA* File System....181

Introduction

NAPA *offers IC designers a comprehensive framework for the high-level simulation of complex mixed-signal networks.*

The *NAPA* Transpiler

NAPA is a netlist-to-***C*** transpiler. ***NAPA*** generates an optimized cycle-based simulator, which is compiled and executed. The goal is to offer a framework to describe a mixed-signal sampled-data network quickly and safely with minimal constraints. The strategy used by ***NAPA*** is to help the designer produce the fastest possible ***C*** program to simulate a netlist without worrying about the complexities of actual ***C*** programming.

Just as an electrical circuit can be modeled as a set of differential equations, a discrete-time network, like an analog modulator or a digital filter, can be modeled as a set of difference equations. The process of converting a netlist into a set of difference equations is straightforward and the perfect type of job for a computer.

NAPA reads in a netlist and writes out an optimized ***ANSI-C*** program. The output ***C*** program is combined with pre-written ***C*** program header files containing user-defined functions.

The ***C*** simulator written by ***NAPA*** uses "*long long integer*" and "d*ouble precision*" respectively as internal representations of digital and analog nodes. Register arithmetic emulation allows simulation of the exact implementation of digital circuits if needed. Node types are determined automatically by netlist analysis and object determination. Type consistency is checked while the simulator is built.

NAPA is designed to be easily extended by users. Users can build high-level models and introduce them as header files, cells, or cell generators. Signal analysis is performed using synchronized "smart tools" that can accumulate the data necessary for analysis, open and close output files, while controlling the simulation flow. Tools are written in ***C*** and are designed to be modified, extended, or rewritten by users if necessary.

NAPA takes advantage of the type checking of ***ANSI-C*** to catch errors. It also has an extensive set of verifications capable of detecting errors and unwanted casting in the netlist description. ***NAPA*** can be used advantageously with a macro preprocessor such as ***MAC***. This preprocessor prepares the netlist, adding user-friendliness, documentation, and flexibility to the description.

NAPA runs under 64-bit Windows with GCC (MinGW-w64, shipped with ***NAPA***). The ***C*** code written by ***NAPA*** needs GCC or a compatible compiler. Since version 5.02, the sources of the transpiler and of the simulators are also prepared for Linux and Unix (options -D__LINUX__ and -D__UNIX__); the scripts remain Windows-only.

NAPA Identifiers

NAPA, like ***C***, is case sensitive. ***NAPA*** identifiers must begin with a letter followed by zero or more alphanumeric characters (including the underscore character "_"). *Identifiers starting with the prefixes "**napa_**", "**DEBUG_**" or "**COMPILE_**"* are reserved keywords (these three prefixes are case sensitive). Characters "$" and "#" have special meaning and should be reserved for this purpose only. A complete list of ***NAPA*** identifiers and reserved keywords is given in appendix B. Reserved keywords cannot be used as ***NAPA*** node or variable identifiers.

Examples of valid identifiers:

```
s1
```

```
id
Sig12a
name_3B
```

NAPA Iterative Identifiers

An iterative identifier is an identifier ending with "**N..M**" where N and M are non-negative integers. It replaces a sequence of identifiers with the same root followed by an integer between N and M. This helps keep the ***NAPA*** netlist as compact and readable as possible.

Example of valid iterative identifiers:

```
abc3..7
file2..0
```

These iterative identifiers (if accepted by the context) are equivalent to:

```
abc3   abc4   abc5   abc6   abc7
file2   file1   file0
```

There is another way to iterate nodes. See "Nodesets" below.

NAPA Instructions

A ***NAPA*** netlist is a sequence of one-line instructions. It is nevertheless possible to extend an instruction across several lines using a continuation character (see below). An instruction is identified by a keyword which must be the first token of the line. Depending on the instruction type, zero or more parameters follow the instruction identifier. The parameters of an instruction are identified by their positions. An instruction is limited to about 14,000 characters. A word, a string, the definition of a variable and the definition of a node (except the ***C*** expressions of dc, const, algebra, dalgebra, ialgebra and test) are limited to 2046 characters: the transpiler stops with a message when a limit is exceeded.

<instruction_keyword> [<parameter...>]

Instructions belong to five classes: declaration, action, control, input/output and format, corresponding to the usual features available in programming languages. Some ***NAPA*** instructions are location dependent. A list of instructions is given in a chapter below.

An instruction can be extended across several lines using the continuation mark "..." at the end of each line to be continued:

<beginning_of_an_instruction> ...

<continuation_of_the_instruction>

A continuation character is not active inside a string (delimited by double quotes).
An example of an instruction spread across several lines is shown below (see also the paragraph about comments below):

```
node out dalgebra  1.0           ...
                 + 2.0*in        ...
                 + 3.0*in*in
```

This is identical to:

```
node out dalgebra  1.0 + 2.0*in + 3.0*in*in
```

Comments

NAPA supports two kinds of comments: whole-line comments and right-hand comments.
As the standard ***NAPA*** flow always runs the macro preprocessor ***MAC*** first, begin a whole-line comment with "#*" or "##": ***MAC*** issues a warning for a line beginning with "#" alone.
C comments (/* ... */) are not supported.

```
##     this is a whole line of comment
```

#* this is a whole line of comment

<some relevant *NAPA* netlist line> // this is a right-hand comment

```
#* This is an example of a whole line of comment

node  w8 dc (digital) 255                    // comment here if you want
```

Of course, a right-hand comment is not active inside a string (delimited by double quotes).
Comments may follow continuation marks:

```
node out dalgebra  1.0           ...        // offset term
                 + 2.0*in        ...        // linear gain
                 + 3.0*in*in                // quadratic term
```

Nodes or Variables?

To describe a system, you will mainly use the ***NAPA*** "node". The nodes represent the structure of the modules, both the signals and the blocks that produce them. Parameters are introduced with two other basic instructions, "ivar" and "dvar". These parameters are the preferred way to describe the external world. Parameters are defined by default as constants but may be updated explicitly using dedicated instructions.

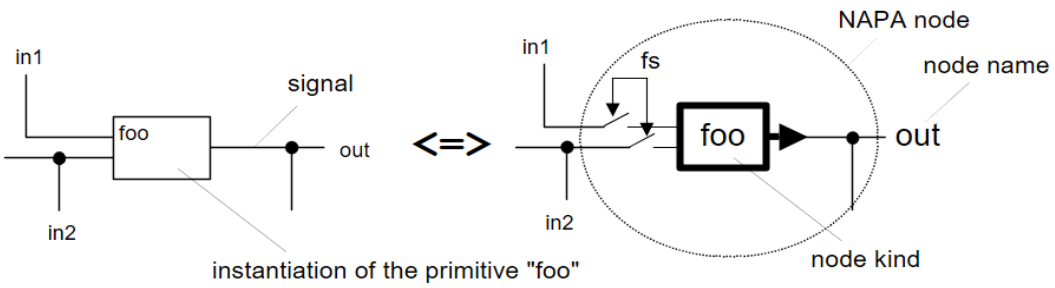
Example: the gain of an amplifier is a parameter, determined by the (external) specifications of the module. It is preferable to use an "*ivar*" or a "*dvar*" to introduce this parameter. But if the gain of a gain-controlled amplifier depends on the states of the system, it is defined as a node, since it is a signal of the system.

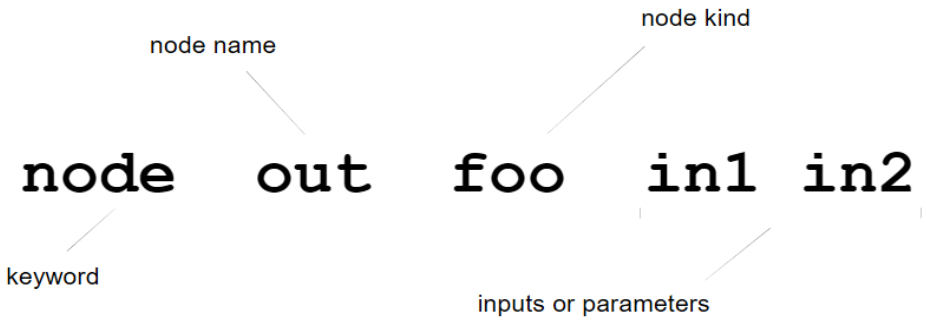
Note: On some occasions, you will have difficulty choosing. Very often, one of the two gives a cleaner description.

The Concept of *NAPA*

*The goal of this chapter is to give users an in-depth view of the **NAPA** transpiler. A good way to understand the concept is to compare the **NAPA** input netlist with the **ANSI-C** output code produced by the transpiler.*

The *NAPA* Node



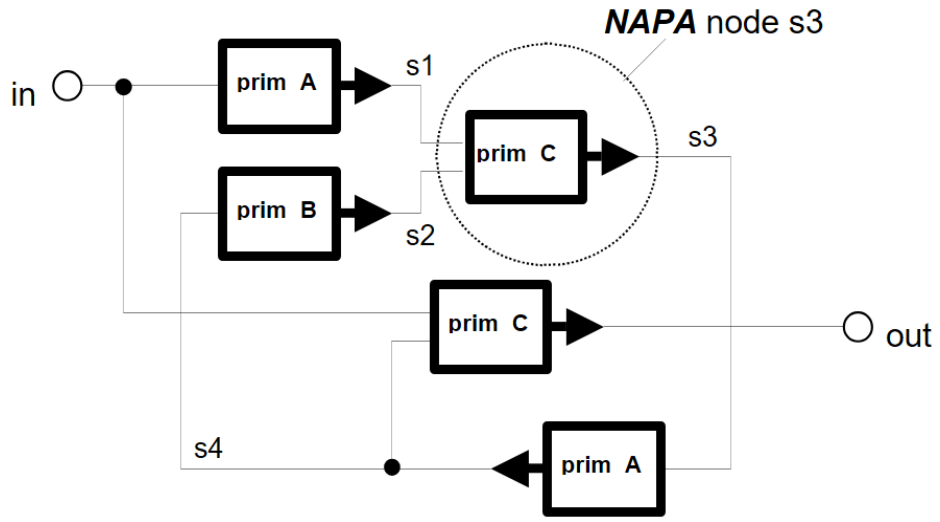


The network is built from *unidirectional* primitives having one output and one or more inputs. Inputs are *sampled* at a defined sampling frequency. The association of an output node with an instance of a primitive is called a ***NAPA*** node. Therefore, a ***NAPA*** node cannot be the output of two primitives and for this reason cannot be defined twice in a netlist.

An important point about nodes is that they are represented internally as either a double precision value (analog nodes) or a long long integer value (digital nodes). Some nodes are always analog or always digital type while others can be either, depending on how they are used or connected together.

The *NAPA* Netlist

The ***NAPA*** netlist is the description of the circuit to be simulated. Consider, for example, the primitives called “prim_A”, “prim_B” and “prim_*C*” in the following circuit:



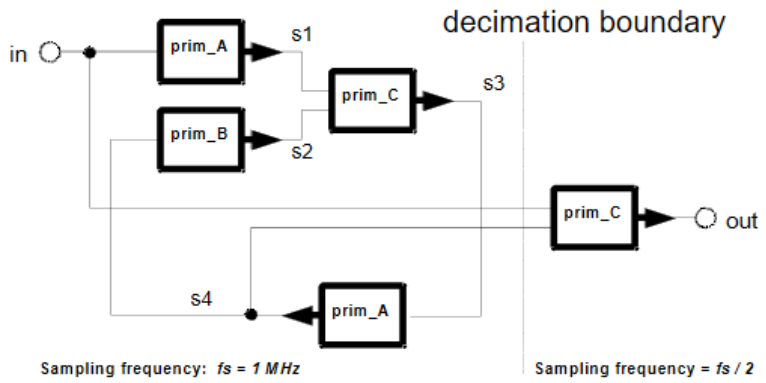
The network can be translated into a ***NAPA*** netlist as a set of nodes:

```
...
node s3  prim_C  s1  s2
node s1  prim_A  in
node s2  prim_B  s4
node s4  prim_A  s3
node out prim_C  in  s4
...
```

The netlist is evaluated at a regular rate chosen by the user. Data are sampled by default at 1.0 Hz. The user sets it with the instruction “fs”.

The Sampling Frequency

The ***NAPA*** transpiler is designed to generate simulators for mixed-signal sampled-data networks. A main sampling frequency determines the pace of the simulation. Local sampling frequency can be decreased (“downsampling” or “decimation”) or increased (“upsampling” or “interpolation”), depending on specific ***NAPA*** instructions. This feature is especially interesting in the description of digital filters. Considering the previous example, we can imagine that part of the network is running at half the main sampling frequency:



The instruction "fs" defines the main sampling frequency. The decimation boundary divides the network into two segments. This separation is translated in the netlist by the instruction "decimate". This instruction decreases the sampling frequency of the second segment by the decimation factor (here a decimation factor of 2):

```
...
fs       1.0e6                  // main sampling frequency

node s3  prim_C  s1  s2         // part running at 1 MHz
node s1  prim_A  in
node s2  prim_B  s4
node s4  prim_A  s3

decimate  2

node out prim_C  in  s4         // part running at 0.5 MHz
...
```

Interpolation follows the same principle. The interpolated rate is higher than the nominal sampling frequency, in the following example by a factor of 16 (instruction "*interpolate*"):

```
...
fs       1.0e6                  // main sampling frequency

node s3  prim_C  s1  s2         // part running at 1 MHz
node s1  prim_A  in
node s2  prim_B  s4
node s4  prim_A  s3

interpolate  16

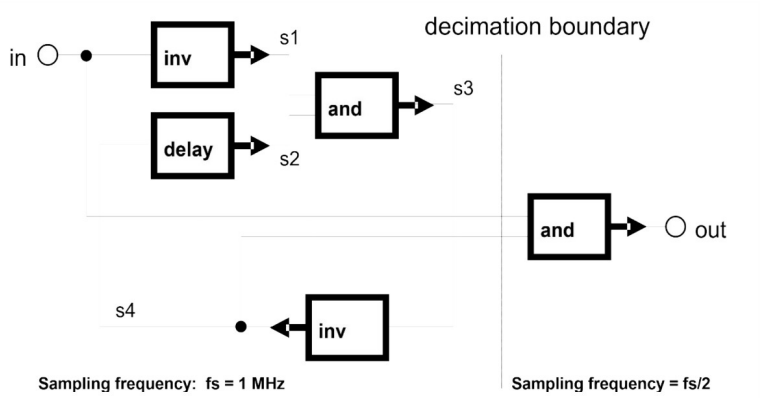
node out prim_C  in  s4         // part running at 16 MHz
...
```

The description has to be completed by simulation controls and by other parameters controlling the way data are analyzed and output.

The Netlist Processing

The ***NAPA*** transpiler has the task of organizing and verifying the data flow of the sampled network, introducing delays where the user requests, without introducing any other delay. While processing the netlist, ***NAPA*** determines and verifies the node type (analog or digital) by inspection and sorting. ***NAPA*** also verifies that internal loops contain at least one delay (see the node determination).

We will consider the following example:



The ***NAPA*** netlist corresponding to this network is:

```
...
fs       1.0e6                      // main sampling frequency

node s3  and   s1 s2                // and gate
node s1  inv   in                   // inverting gate
node s2  delay s4                   // delay element
node s4  inv   s3                   // inverting gate
node in  (to be completed)          // input signal

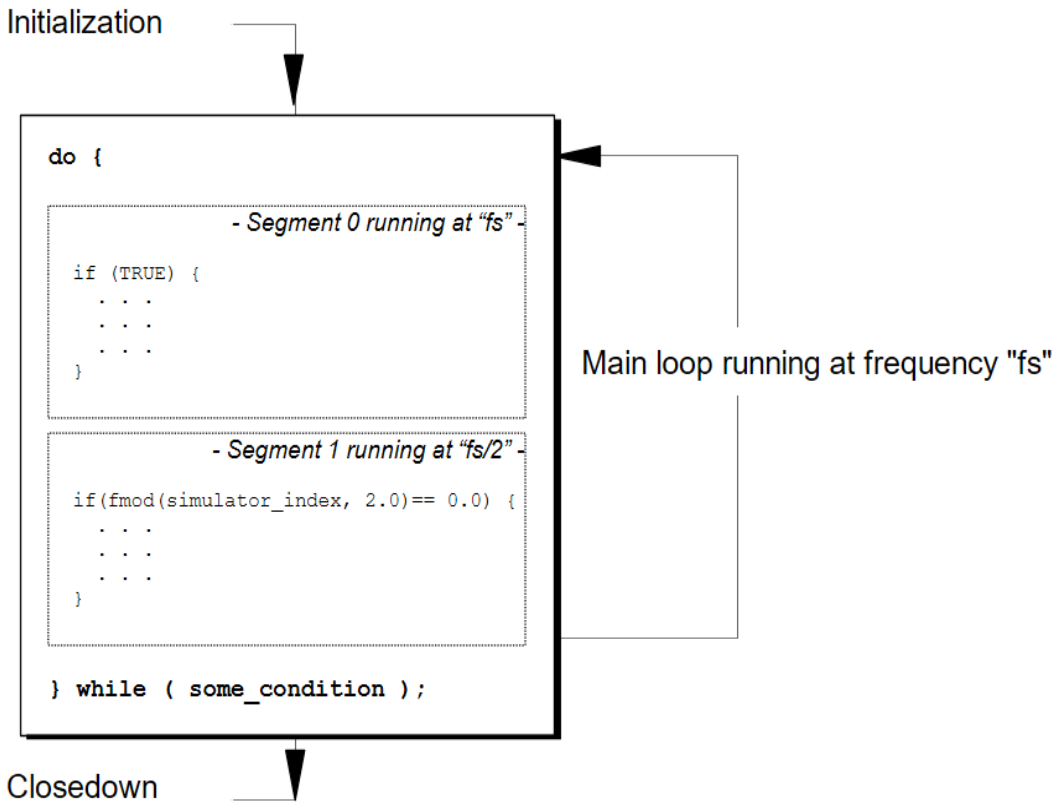
decimate  2                         // decimation by 2

node out and   in s4                // and gate
...
```

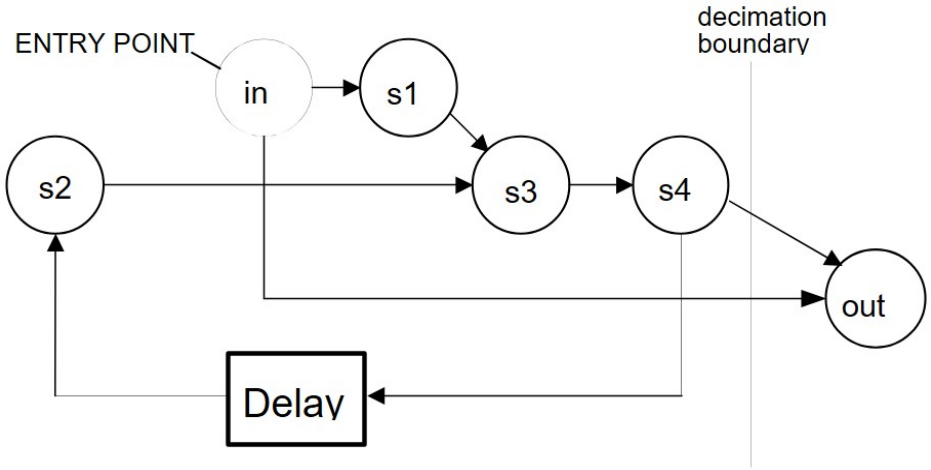
NAPA organizes the simulation around a main loop describing the data flow occurring inside one sampling cycle "1 / fs". The netlist is divided by decimation and interpolation instructions into segments, with the main segment being segment 0. Decimated and interpolated segments are introduced in the main loop as ***C*** "if" statements.

The transpiler sorts the nodes of the entire netlist. In the example, segment 0 will contain the set of nodes running at the nominal sampling frequency "*fs*", and segment 1 will contain the nodes running at the decimated speed, here "fs / 2". Segment 0 contains nodes "s3", "s1", "s2", "s4" and "in". Segment 1 contains the node "out".

Some nodes (output of signal generators) are clearly not dependent on any other nodes. They are therefore unconditionally determined and are the entry points of the simulation. Delays represent a special class of nodes, as they depend on the value of their input one loop earlier, not on its current value.



The node tree corresponding to the netlist of the example is:



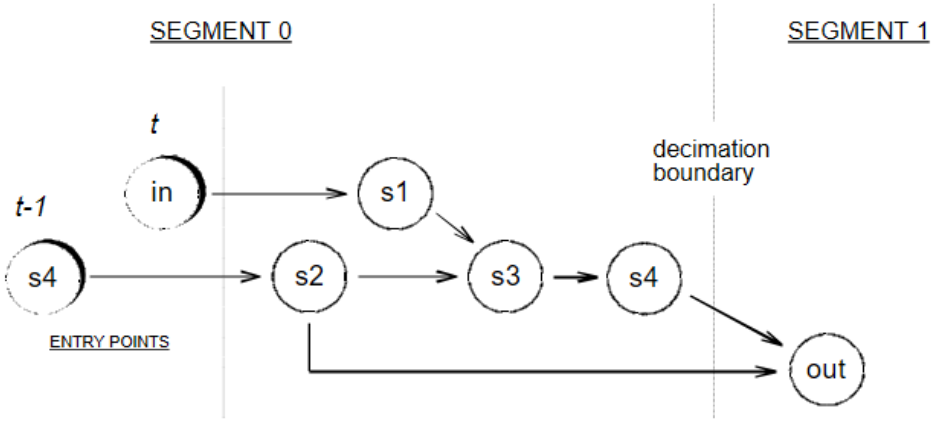
This tree corresponds to the following table:

SEGMENT	ACTION CLASS	DEPENDENCY	DETERMINED?
[0]	immediate action	s1, s2	?
	immediate action	in	?
	delayed action	s4	?
	immediate action	s3	?
	immediate action	(*none*)	yes
[1]	immediate action	in, s4	?

Handling the Delays

Delayed action nodes are processed first. New entry points are inserted for each delayed node at the beginning of corresponding segments. The entry points do not belong to the segment but are the entry points of the whole set of nodes of this segment. They represent the bridges between two consecutive simulation loops. A delay brings the value taken by a node from one loop into the next one.

After insertion of new entry points corresponding to delays (here for node “s4”), the tree becomes:



Node Determination and Sorting

Nodes are sorted and their determination is checked by automatic netlist inspection. Entry points corresponding to delays are sorted first. All entry points are considered as unconditionally determined.

Other nodes are determined as soon as their inputs are determined. In the example, node “s1” is determined, as its input node “in” is determined.

At this point of the process, residual internal loops of nodes cannot be determined in any way. Therefore they are flagged as errors by the ***NAPA*** transpiler. They correspond in the original ***NAPA*** netlist to loops without delay. This is the typical cause of an undetermined node.

Node types (analog or digital) are obtained from node kinds and/or primitive input nodes. In the example, node “in” is digital by construction; node “s2” is digital as it is the delayed image of node “s4” which is digital type by construction (output of an inverting gate).

SEGMENT	NODE	ACTION CLASS	DETERMINED?	TYPE
[0]	s2	delayed action	yes	Digital type
	in	immediate action	yes	Digital type
	s1	immediate action	yes	Digital type
	s3	immediate action	yes	Digital type
	s4	immediate action	yes	Digital type
[1]	out	immediate action	yes	Digital type

Type consistency is checked during node determination.

The Netlist to *C* Translation

From the processed list of nodes, the ***NAPA*** transpiler translates the netlist into ***C*** code. The resulting code is a flattened description of the netlist without function calls.

```
...
double    napa_abs_time;
long long inode_s2;
long long inode_in;
long long inode_s1;
long long inode_s3;
long long inode_s4;
long long inode_out;
...
int main(void) {
  ...
  napa_reset_nodes();                          /* initialization    */
  napa_abs_loop = 0LL;                         /* loop counter      */
  do {                                         /* main loop         */
    napa_abs_time = ...;
    /* always */ {
      napa_segment = 0;
      inode_s2 = inode_s4;                     /* delay element     */
      inode_in = ...;                          /* clock generation  */
      inode_s1 = !inode_in;                    /* inverting gate    */
      inode_s3 = inode_s1 && inode_s2;         /* and gate          */
      inode_s4 = !inode_s3;                    /* inverting gate    */
    }
    if (napa_abs_loop % 2LL == 0LL) {          /* decimate by 2     */
      napa_segment = 1;
      inode_out = inode_in && inode_s4;        /* and gate          */
    }
    napa_abs_loop++;
  } while ( some_condition );                  /* end of main loop  */
  ...
  return EXIT_SUCCESS;
```

```
}
```

Note that the position of the code generating the signal “inode_s2” naturally introduces the specified delay, with other nodes having an immediate action. Code producing the signal “inode_out” is placed in a separate segment and is effectively running at half the speed of the first part.

The Simulation Control

The simulation is controlled with the help of variables. Variables are used to control sources (e.g. “dc”, “sine”), to parametrize nodes (“gain”, “clip”) and to control the simulation or the data flow. ***NAPA*** user-defined analysis tools (smart tools) are especially built to take control of the simulation. The output node of a smart tool is a flag indicating the status of the ongoing process. Built-in mechanisms keep concurrent analysis tools synchronized. The simulation loop is controlled by a termination condition. For a complete example, see appendix D, page 167.

NAPA in a Computer Network

NAPA output is generally directed to files. If a large volume of output is expected, keep the output files on a local disk, especially if the network is slow.

The output stream could severely limit the speed of execution of the simulation. Avoid writing large amounts of data (especially to the screen); use an event condition to limit the amount of output, etc.

NAPA Transpiler Command Line

The ***NAPA*** command line takes one parameter and options:

```
% NAPA_transpiler_pathname <“input_source_file”> [ options ]
% NAPA_transpiler_pathname < -help >
```

Where ‘options’ are:

-h[dr]	<header_directory_pathname>
-n[et]	<net_directory_pathname>
-g[en]	<generator_directory_pathname>
-d[ir]	<generic_directory_pathname>
-u[ser]	<“user_name_in_one_single_word”>

The input source file is a ***NAPA*** file containing the user's ***NAPA*** source and the library pathnames are respectively the pathnames of the directories “Hdr” (“-h” option), “Net” (“-n” option) and “Gen” (“-g” option). These directories contain the reusable headers, cells and cell generators. Their defaults are “Hdr”, “Net” and “Gen”, relative to the generic directory given by the “-d” option, or else to the current working directory.

The user name (with option “-u”) will be added automatically to output files to complement the documentation. Any underscore in the user name will be replaced by a space.

Scripts are available to launch the simulation in several environments (currently Windows). They will not be described here as they can be heavily customized.

A second set of options provides information about the transpiler:

-a[uthor]
-b[uilt]
-help, -?

Three other options are available:

-l[ist]
-e[xpand]
-v[erbose]

The first one lists the nodes, variables and variable updates on the standard output. The location of the definition of each identifier is also output. No ***C*** file is generated. This option cannot be used in conjunction with "-e".

The second one expands the source net files (main, cells and data cells) into a single expanded netlist on the standard output. The names of the identifiers are exactly the names produced during a regular transpilation of ***NAPA***. It shows how cells are expanded and how the final identifiers are built. No ***C*** file is generated. This option cannot be used in conjunction with "-l".

The last option switches the transpiler to verbose mode. It gives some detail about the transpilation process and is of limited interest to a regular user.

Several directory organizations are possible; here is an example:

/home/*NAPA*	→ /Simulate/NapaDos/Hdr	contains generic header files	*(*.hdr)*
	→ /Simulate/NapaDos/Net	contains generic cell files	*(*.net)*
	→ /Simulate/NapaDos/Gen	contains generic generator files	*(*.exe)*
	→ /Simulate/NapaDos/Dos	contains the ***NAPA*** transpiler and scripts	
/home/jdoe	→ /jdoe/*NAPA*/project_1	contains project's source files	*(*.nap)*
			(.rom)*
			...
		contains project's header files	*(*.hdr)*
		contains project's cell files	*(*.net)*
			(.dat)*
		contains project's generator files	*(*.c)*
			(.exe)*
		contains simulation output	*(*.out)*
	→ /jdoe/*NAPA*/Hdr	contains user's header files	*(*.hdr)*
	→ /jdoe/*NAPA*/Net	contains user's cell files	*(*.net)*
			(.dat)*
	→ /jdoe/*NAPA*/Gen	contains user's generator files	*(*.exe)*
			(.c)*

To run ***NAPA***, set the working directory to '*/jdoe/NAPA/project_1'*. For example, type:

```
%  /Simulate/NapaDos/Dos/NapaDos.exe myfile.nap -u "JDoe" -d /Simulate/NapaDos
      > myfile.c ↵
```

```
%  gcc myfile.c -o myfile.exe ↵
%  myfile.exe > myfile.out ↵
```

Where “NapaDos.exe” is the name of the ***NAPA*** transpiler, “gcc” is the ***C*** compiler and “myfile.nap” is the netlist to be simulated. Standard output of the simulator is redirected to “myfile.out”. On other platforms, the command could be slightly different. As the ***C*** compiler has many options, choose the optimization level carefully. Optimizations reduce the execution time but may slow down the compilation significantly. Avoid hazardous optimizations, of course. Some tuning could be necessary.

With the macro preprocessor ***MAC*** (optional):

```
% /Simulate/MacDos/Dos/MacDos.exe    myfile.nap -c -l  > myfile.tmp ↵
% /Simulate/NapaDos/Dos/NapaDos.exe  myfile.tmp -u "Jdoe" -d /Simulate/NapaDos >
 myfile.c ↵
%  gcc myfile.c -o myfile.exe ↵
%  myfile.exe > myfile.out ↵
```

Using the appropriate script, the command could be as simple as:

```
%  NAPA  myfile ↵
```

NAPA Portability

NAPA is built to be portable. It is the user's responsibility to write new headers with ***ANSI-C*** syntax. It is also their responsibility to use file names compatible with Windows and UNIX in order to ensure perfect compatibility of the netlists if needed. Some compatibility problems are expected on operating systems whose file naming conventions differ from UNIX's.

Internally, ***NAPA*** follows the UNIX pathname conventions. Under Windows it is compiled with GCC (MinGW-w64, shipped with ***NAPA***).

C Limitations: *Important Warnings*

In principle, ***NAPA*** gives access to ***C*** in several ways. Dedicated nodes (“const”, “dc”, “algebra”, “ialgebra”, “dalgebra” and “test”), variables (“dvar”, “ivar”, “update” and “event”), several instructions (“terminate”, “drop”, “assert”...) and conditional expressions (“...when”) accept regular ***C*** expressions. Other nodes (“duser”, “iuser”, “dtool”, “itool”, “init”, “call”...) trigger calls to ***C*** functions.

Some restrictions prevent side effects:

The ***NAPA*** parser will reject the unary operators “++” and “--” inside a ***C*** expression placed in a ***NAPA*** netlist.

A ***C*** function placed in an expression of a ***NAPA*** netlist will be misinterpreted if it has the same name as a node or a variable. There are several specific checks but a name collision remains possible, resulting in a ***C*** compiler error.

C Casting: *Important Warnings*

Although ***ANSI-C*** is safer than old ***K&R C***, some potential problems remain. For instance, casting remains a potentially dangerous operation in ***C*** expressions:

Casting, first way to get unexpected results:

```
node a  dc  (digital) 5
node b  dc  (digital) 3
node c  dalgebra  a/b                  // cast to double by "dalgebra"
```

The resulting value of 'c' is 1.0, not 1.6667 as the integer division is applied before the casting. Whenever possible, ***NAPA*** refuses an integer where a real number is expected. But the node *"dalgebra"* accepts a ***C*** expression. Nothing can be done to detect this potential problem.

Here you should prefer:

```
node a  dc  (digital) 5                                  // cast to digital
node b  dc  (digital) 3                                  // cast to digital
node c  div  a  b
```

The resulting value of 'c' is now 1 as node *"div"* divides two digital nodes 'a' and 'b'. Clean and safe.

```
node a  dc   5.0                                         // cast to analog
node b  dc   3.0                                         // cast to analog
node c  div  a  b
```

The resulting value of 'c' is now 1.6667 as node *"div"* divides two analog nodes 'a' and 'b'. Clean and safe.

Casting, a second way to get unexpected results:

```
node s1  dalgebra rand_uniform(-100.0, 100.0)            // random noise
node s2  ialgebra s1                  // cast to long long int by 'ialgebra'
```

Histograms of "s2" show twice as many occurrences of 0 as of any other number: node "ialgebra" converts to long long by truncation, and all numbers between -0.9999 and 0.9999 give 0. This is not what was intended.

Using dedicated ***NAPA*** primitives removes the problem:

```
node s1  dalgebra rand_uniform(-100.0, 100.0)            // random noise
node s2  dtoi s1
```

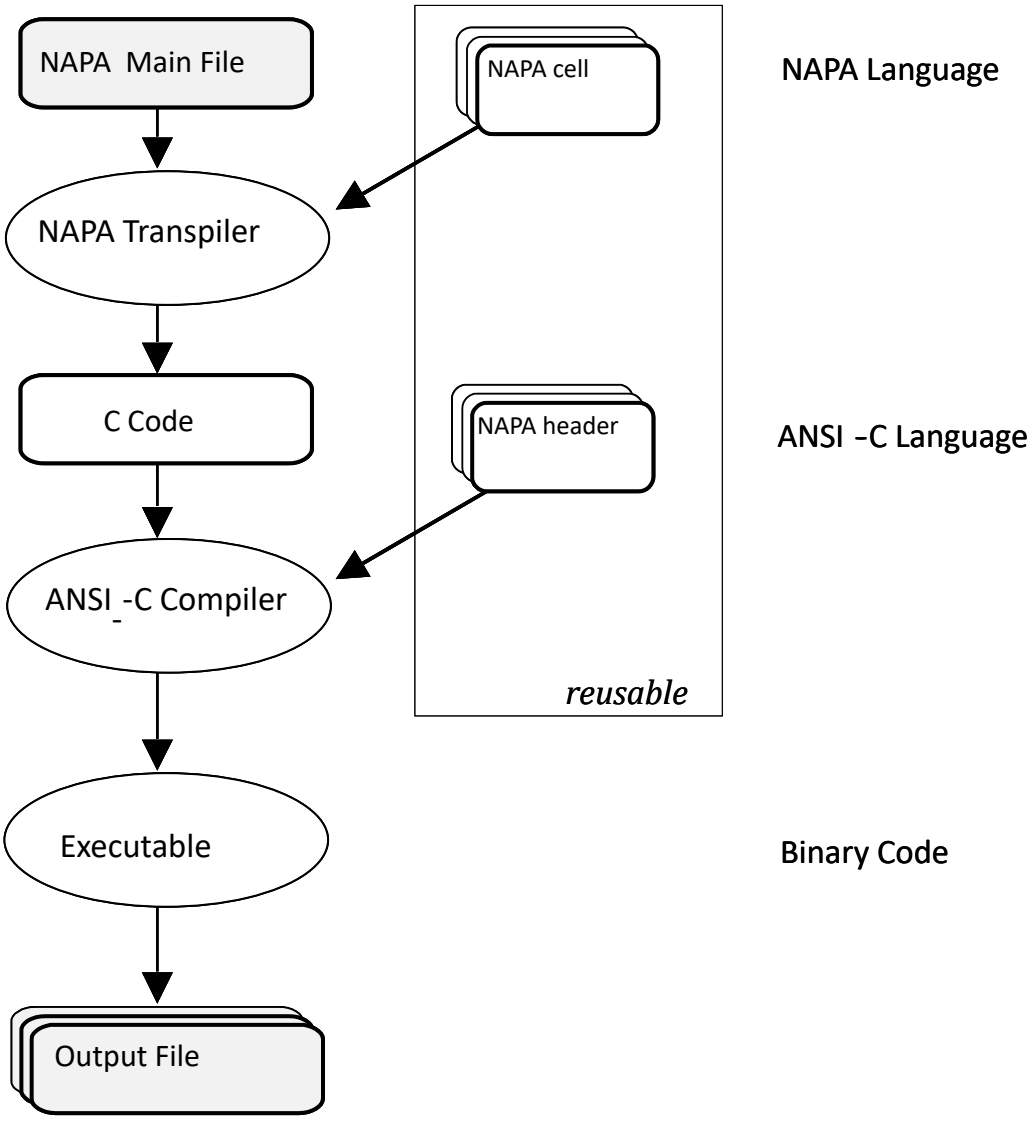
The node *"dtoi"* rounds to the nearest integer (halves away from zero). Only numbers between -0.4999.. and 0.4999.. produce 0.

NAPA Process Flow

The simulator executable is generated through a two-step process:

1. **NAPA Transpilation**: The **NAPA** transpiler processes the netlist files and converts them into **C** source code.
2. **C Compilation**: The **ANSI-C** compiler processes the generated **C** code to produce the final executable simulator.

This two-step process allows **NAPA** to leverage the optimization capabilities of standard **C** compilers while providing its own high-level simulation language.



The *NAPA* File System

A ***NAPA*** netlist can be highly hierarchical. The transpiler supports several pathname formats to locate files:

1. Absolute Reference

Syntax: **"/absolute/path"**
Example: "/home/jdoe/examples/anycell.net"

2. Generic Library Reference

Syntax: **<filename>**
Examples:

<wondertool.hdr> for headers in a header call
<wondercell.net> for cells in a cell call
<wondercell.gen> for generators in a generator call

3. Root Directory Reference

Syntax: **"filename"**
Example: "mydata.dat"

4. Main Directory Reference

Syntax: **"~/filename"**
Example: "~/mycell.net"

5. Current Cell Directory Reference

Syntax: **"./filename"**
Example: "./mytool.hdr"

Directory Definitions and Processing

- **Generic Library**: Three specialized libraries containing reusable resources: the header library, the cell library, and the generator library. The pathnames of these directories are passed to the ***NAPA transpiler*** through the command line.
  Generic library references are only valid when calling headers, cells, data cells, or generators.
- **Root Directory**: The working directory from which the ***NAPA*** transpiler was called.
- **Main Directory**: The directory containing the ***NAPA*** main netlist.
- **Current Cell Directory**: The directory containing the cell currently being processed.

The ***NAPA transpiler*** processes pathname strings immediately and translates them into pathnames compatible with the operating system. To help the user, error messages will reference the operating system pathnames rather than the ***NAPA*** file system syntax.

Best Practices

Understanding the ***NAPA*** file system is essential when building netlists based on a set of files or directories. Choosing the right kind of reference lets you copy or move a set of files without breaking the links between them.

NAPA Instructions

*The **NAPA** instructions are divided into five classes: declaration, action, control, input/output and format.*

Each ***NAPA*** instruction is a single statement, written on one line (or continued with "..."). Its keyword is the first word of the line (except for the width prefix of register arithmetic, e.g. "(8) node ..."). Some of them are used in specific situations and are not of general interest (in *italic* below).

alias	Declaration
array	Declaration
assert	Action
call	*Action*
cell_interface	Declaration
command_line	I/O
comment	*Declaration*
data	Declaration
data_interface	Declaration
debug	Declaration
decimate	Control of simulation flow
declare	Declaration
directive	Declaration
drop	Control of simulation flow
dump	*Action*
dvar	Declaration
error	*Declaration*
event	Declaration
export	I/O

format	*Output format*
fs	Declaration
ganging	Declaration
gateway	*Control of simulation flow*
header	Declaration
init	Action
inject	Action
input	*I/O*
interlude	Declaration
interpolate	Control of simulation flow
ivar	Declaration
load	*Action*
napa_version	*Declaration*
node	Declaration
nominal	Control of simulation flow
num_initial	Declaration (DEPRECATED)
opcode	Declaration
output	I/O
ping	*Declaration*
post	Action
random_seed	Declaration
restart	*Action*
string	Declaration
stuck	*Declaration*
synchronize	*Declaration*

terminate	Control of simulation flow
title	Declaration
tool	Declaration
ts	Declaration
update	Action
void	*Declaration*
warning	*Declaration*

NAPA Nodes

*The **NAPA** node is the key element of the netlist.*

Usage

A node is a ***NAPA*** object describing part of the netlist. Nodes are always defined as the output of an object (see chapter presenting the ***NAPA*** concept). The node represents the object and its output signal together. It is generally not necessary to predeclare nodes. Internal loops of nodes in the netlist must include at least one delay. Apart from "delay" nodes, no node contains an implicit delay (a user-defined node may contain delays, depending on its ***C*** code). ***NAPA*** sorts the nodes and determines the node types (analog or digital) after analysis of the netlist. The transpiler is able to detect undetermined nodes, unauthorized loops, self-referencing nodes and wrong combinations of node types.

Nodes are initialized by default, although users may modify their initialization (see instruction "*init*"). As initialization is an important issue, read the documentation of the instruction "init" carefully.

node <nod_nam> <kind> [<parm...>] [<nod_nam...>]

In the generated ***C*** code, ***NAPA*** nodes are declared, defined and used with the prefix "inode_" or "dnode_" depending on the type: digital (long long integer) or analog (double precision). The ***NAPA*** nodes are translated as in-line ***C*** code to obtain better efficiency. This translation is context-dependent and is handled by the transpiler. It means that the transpiler handles the syntactic verification, declaration, initialization and code customization for the user.

NAPA is a strongly typed language. Nodes are represented in the simulator as

analog node	***NAPA*** analog type	(***C*** double precision)	prefix "dnode_"
digital node	***NAPA*** digital type	(***C*** long long integer)	prefix "inode_"

Chameleonic nodes

Whenever possible, a ***NAPA*** node takes the type of its inputs. Such nodes are called "**chameleonic nodes**". They can be used in both analog and digital contexts.

Nodesets

A set of nodes may be defined in a single instruction. As it is less readable, this form is mainly meant for generators.

```
node[4] out[2..5] sum  in[4..1] in0
```

This is equivalent to the set of lines:

```
node    out2     sum  in4       in0
node    out3     sum  in3       in0
node    out4     sum  in2       in0
node    out5     sum  in1       in0
```

Nodesets involving a cell or a generator are not allowed (severe side effects). The declaration of width must match the iterators when they are used in the definition of the nodeset.

In the example above, the width is declared as 4 and the iterators indeed span 4 values.

Other examples of application of the nodesets:

```
node[2] (out[1..2]) sum  a1..4  b[4..5]    // example with parentheses
```

This is equivalent to the set of lines:

```
node (out1) sum  a1 a2 a3 a4  b4
node (out2) sum  a1 a2 a3 a4  b5
```

This example applies the concept to file names and variables:

```
ivar npts1   1000
ivar npts2  10000
node[2] void  itool fft "file[1..2].out"  in[4..5] ref  npts[1..2]
```

This is equivalent to the set of lines:

```
ivar npts1   1000
ivar npts2  10000

node void  itool fft "file1.out"  in4 ref  npts1
node void  itool fft "file2.out"  in5 ref  npts2
```

Pseudo-Nodes

Some nodes ("*cell*" and "*generator*") do not behave as true ***NAPA*** nodes. They trigger the instantiation of a ***NAPA*** netlist. The syntax depends directly on the way the "*cell*" (or the "*generator*") interface is written (see cell node). Although it is better to follow the syntax of a regular node, with output on the left side and parameters and inputs on the right side, the user can define a cell with several outputs. As a regular node has only one output, the syntax must be changed: output nodes also appear on the right-hand side:

```
node out1 cell mycel1 "cellfil1.net" out2 in1 in2 parm1
```

It is strongly recommended to write the cell interface so that all the outputs appear on the right:

```
node void cell mycel2 "cellfil2.net" out1 out2 parm1 in1 in2
```

The node "void" now corresponds to a dummy node in the cell netlist. See also paragraph below.

Unnamed and Unused Signals

NAPA flags unused nodes or variables. To suppress the corresponding warning messages, the special identifier "void" can be used as node or variable name. The transpiler automatically gives the corresponding ***C*** variable a unique internal name.

Another way to suppress the warning is to place the node or variable identifier in the definition between parentheses. The node or variable can still be used. This is to be used with care as the absence of warning could hide a true error.

```
dvar (opt)  1.0 + b
node (err)  sub  out ref

node void   itool fft "stdout"  x 1.0  100000
```

Identifier "void" is used in the definition of node "*mux*" to define an empty input slot.

"void" also has a specific meaning in the instruction "post", and is an instruction of its own.

Automatic Nodes

To simplify netlists, the transpiler provides three "automatic nodes", "One", "Zero" and "Ground", generated when they are used.

One	DIGITAL	constant value 1
Zero	DIGITAL	constant value 0
Ground	ANALOG	constant value 0.0

The ***NAPA*** transpiler processes these signals exactly as if they were issued from "dc" nodes, but they cannot BE DEFINED OR REDEFINED BY THE USER:

```
node One    dc (digital)  1
node Zero   dc (digital)  0
node Ground dc (analog)   0.0
```

Bit Field Extractor

A single bit from any digital node can be accessed through the "bit field extractor". The corresponding nodes are created automatically. The notation is:

...<nod_nam>:<integer>...

The internal representation of a ***NAPA*** node or variable is based on the ***C*** "long long integer", i.e. 64 bits with 2's complement coding. Bit 0 is the least significant bit.

An example of bit field use:

```
...
node d4 sum     d1 d2 d3
node d5 nand    d2:4  d2:7          // bits 4 and 7 of d2
output "stdout" d4 d2:4 d2:5 d5
...
```

Node Kind List

TYPE: inputs → output; R analog, I digital, X either; Chameleonic: the output type follows the inputs; Declared: type of the memory array; Pseudo Node: cell or generator.

NODE KIND	BRIEF DESCRIPTION	TYPE
1. adc	N-level signed analog-to-digital converter	*R → I*
2. algebra	generalized mathematical ***C*** expression	*Chameleonic*
3. alu	user-defined ALU	*Chameleonic*
4. and	and of N input nodes (Boolean logic)	*I → I*
5. average	average of N input nodes	*R → R*
6. bshift	barrel shifter	*I → I*
7. btoi	N-bit to integer conversion	*I → I*
8. buffer	non-inverting buffer (Boolean logic)	*I → I*
9. bwand	bitwise and (Boolean logic)	*I → I*
10. bwbuffer	bitwise non-inverting buffer (Boolean logic)	*I → I*
11. bwinv	bitwise inverter (Boolean logic)	*I → I*
12. bwnand	bitwise nand (Boolean logic)	*I → I*
13. bwnor	bitwise nor (Boolean logic)	*I → I*
14. bwnot	strictly equivalent to "bwinv" (Boolean logic)	*I → I*
15. bwor	bitwise or (Boolean logic)	*I → I*

16.	**bwxnor**	bitwise xnor (Boolean logic)	*I → I*
17.	**bwxor**	bitwise xor (Boolean logic)	*I → I*
18.	**cell**	subcircuit instantiation	*Pseudo Node*
19.	**change**	watchdog	*X → I*
20.	**clip**	clipping	*Chameleonic*
21.	**clock**	clock generation (sequence of 0 and/or 1)	*→ I*
22.	**comp**	comparator	*X → I*
23.	**const**	constant with optional casting	*→ I or R*
24.	**copy**	signed copy	*Chameleonic*
25.	**cosine**	cosine wave generator	*R → R*
26.	**dac**	N-level signed digital-to-analog converter	*I → R*
27.	**dalgebra**	generalized ***C*** expression - casting to real	*X → R*
28.	**dc**	DC source with optional casting	*→ R or I*
29.	**delay**	simple or multiple delay	*Chameleonic*
30.	**differentiator**	non-delayed non-inverting differentiator	*Chameleonic*
31.	**div**	division of 2 input nodes	*Chameleonic*
32.	**dtoi**	real to integer conversion	*R → I*
33.	**dtool**	equivalent to duser but not reset at restart	*X → R*
34.	**duser**	user-defined analog type function	*X → R*
35.	**equal**	equality of 2 input nodes	*X → I*
36.	**fzand**	and of N input nodes (Fuzzy logic)[1]	*R → R*
37.	**fzbuffer**	non-inverting buffer (Fuzzy logic)	*R → R*
38.	**fzinv**	strictly equivalent to “fznot” (Fuzzy logic)	*R → R*
39.	**fznand**	nand of N input nodes (Fuzzy logic)	*R → R*
40.	**fznor**	nor of N input nodes (Fuzzy logic)	*R → R*
41.	**fznot**	negation of input node (Fuzzy logic)	*R → R*
42.	**fzor**	or of N input nodes (Fuzzy logic)	*R → R*
43.	**fzxnor**	xnor of 2 input nodes (Fuzzy logic)	*R → R*
44.	**fzxor**	xor of 2 input nodes (Fuzzy logic)	*R → R*
45.	**gain**	gain	*Chameleonic*
46.	**generator**	subcircuit generation and instantiation	*Pseudo Node*
47.	**hold**	hold (and track)	*Chameleonic*
48.	**ialgebra**	generalized ***C*** expression - casting to integer	*X → I*
49.	**integrator**	non-delayed non-inverting integrator	*Chameleonic*
50.	**inv**	strictly equivalent to “not” (Boolean logic)	*I → I*
51.	**itob**	bit extractor from integer	*I → I*
52.	**itod**	digital-to-analog type conversion	*I → R*
53.	**itool**	equivalent to iuser but not reset at restart	*X → I*
54.	**iuser**	user-defined digital type function	*X → I*
55.	**latch**	SR latch (Boolean logic)	*I → I*
56.	**lshift**	left shift element	*I → I*
57.	**max**	maximum of N input nodes	*Chameleonic*
58.	**merge**	merge N input nodes from exclusive segments	*Chameleonic*

1 ***NAPA*** provides the fuzzy gates (Zadeh operators) and nothing else of fuzzy logic.

59.	**min**	minimum of N input nodes	*Chameleonic*
60.	**mod**	modulo division of 2 input nodes	*Chameleonic*
61.	**muller**	C Muller, N input nodes (Boolean logic)	*I → I*
62.	**mux**	multiplexer (N inputs)	*Chameleonic*
63.	**nand**	nand of N input nodes (Boolean logic)	*I → I*
64.	**noise**	source of noise (normal)	*→ R*
65.	**nor**	nor of N input nodes (Boolean logic)	*I → I*
66.	**not**	negation of input node (Boolean logic)	*I → I*
67.	**offset**	DC level shifter	*Chameleonic*
68.	**or**	or of N input nodes (Boolean logic)	*I → I*
69.	**osc**	oscillator	*R → R*
70.	**poly**	polynomial of order N	*Chameleonic*
71.	**prod**	multiplier, product of N input nodes	*Chameleonic*
72.	**quant**	quantizer	*Chameleonic*
73.	**ram**	random access memory	*Declared*
74.	**ram2**	dual port random access memory	*Declared*
75.	**rect**	rectifier	*Chameleonic*
76.	**register**	data register	*Chameleonic*
77.	**relay**	relay, normally closed	*Chameleonic*
78.	**rip**	bitwise rip bus	*I → I*
79.	**rom**	read only memory	*Declared*
80.	**rom2**	dual port read only memory	*Declared*
81.	**rshift**	right shift element without rounding	*I → I*
82.	**rshift1**	right shift, rounding half away from zero	*I → I*
83.	**rshift2**	right shift, rounding half to even	*I → I*
84.	**sign**	sign of input	*X → I*
85.	**sine**	sine wave generator	*R → R*
86.	**square**	square wave source	*→ R*
87.	**step**	step function source	*→ R*
88.	**sub**	subtraction of 2 input nodes	*Chameleonic*
89.	**sum**	sum of N input nodes	*Chameleonic*
90.	**test**	***C*** condition, output 0 or 1	*X → I*
91.	**toggle**	toggle flip-flop (Boolean logic)	*I → I*
92.	**track**	track (and hold)	*Chameleonic*
93.	**triangle**	triangle wave source	*→ R*
94.	**trig**	trigger	*X → I*
95.	**uadc**	N-level unsigned analog-to-digital converter	*R → I*
96.	**udac**	N-level unsigned digital-to-analog converter	*I → R*
97.	**wsum**	weighted sum of N input nodes	*Chameleonic*
98.	**xnor**	exclusive nor of N input nodes (Boolean logic)	*I → I*
99.	**xor**	exclusive or of N input nodes (Boolean logic)	*I → I*
100.	**zero**	insertion of zeroes	*Chameleonic*

NAPA User's Variables

*A simulator is controlled by parameters. These parameters are not signals to process. In **NAPA**, they are called "variables".*

Usage

Besides nodes, a netlist declares user variables. ***NAPA*** user variables behave just like their ***C*** counterparts. Variables are not part of the node netlist description. They serve as parameters of the nodes or to control the simulation. Their declaration is mandatory. By default, the variables are not updated, but update is possible. The variable definition is a ***C*** expression possibly containing other variables, nodes or global variables. ***NAPA*** global variables are predefined in the generated ***C*** code and available to the user. Self-referencing is not authorized in the definition of a variable but perfectly legal for an update:

```
dvar   v1    v1 + v2                // illegal!
ivar   v3    0
update v3    v3 + 1                 // perfectly OK
```

Variable Type

Three types of user's variables are available: digital, analog and string variables. Variables are represented internally in the ***C*** simulator as:

dvar	***NAPA*** analog type	(***C*** double precision)	prefix "dvar_"
ivar	***NAPA*** digital type	(***C*** long long integer)	prefix "ivar_"
string	***NAPA*** string type	(***C*** char array)	prefix "svar_"

Register Arithmetic

The simulation of digital filters requires performing arithmetic on digital numbers coded with a limited number of bits.

Width limited nodes and variables

Digital nodes and variables can be coded on a limited number of bits. Without any specific declaration, the ***NAPA*** transpiler declares the digital nodes and variables as ***ANSI-C*** long long integer. Following ***ANSI-C*** standard, the "long long integer" is coded with a minimum of 64 bits. The ***NAPA*** transpiler relies on the operators of the ***C*** compiler to perform the arithmetic and logic operations. Registers wider than 64 bits therefore cannot be emulated directly.

Nevertheless, it is possible to emulate the behavior of limited register width arithmetic by declaring the digital nodes or variables as width limited. Of course, it is not allowed to limit the coding of analog nodes and variables or to extend the coding of digital values to a number of bits beyond the "long long integer" size.

To limit the number of bits, it is necessary to declare the width of the nodes and the variables:

'('width')' "node" <nod_nam> <node_kind_returning_a_digital>...

'('width')' "ivar" <var_nam> ...

'<'width'>' "node" <nod_nam> <node_kind_returning_a_digital>...

'<'width'>' "ivar" <var_nam> ...

Parentheses denote a signed (2's complement) coding, angle brackets an unsigned coding. The width is a constant integer: 1 to 64 for a signed coding (64 is the coding of the C long long integer: no limitation), 1 to 63 for an unsigned coding.

The width applies to the value of the node, i.e. to the output of its node kind, once the operation is done. The width of an input node does not limit the result of the node it feeds: for example, an unsigned 8-bit node equal to 222 shifted by 3 to the left gives 1776, unless the output node is itself declared on 8 bits.

```
(16)  node r1  gain   m9  a              // 2's complement 16 bits
<16>  node r2  offset off b              // unsigned 16 bits

(16)  ivar m9  -9                        // 2's complement 16 bits
<16>  ivar off  127                      // unsigned 16 bits
```

An overflow wraps around according to the coding, for example, when incrementing value 0 coded with 3 bits:

3 bits	2's complement	3 bits	unsigned
000_2	0_{10}	000_2	0_{10}
001_2	1_{10}	001_2	1_{10}
010_2	2_{10}	010_2	2_{10}
011_2	3_{10}	011_2	3_{10}
100_2	-4_{10}	100_2	4_{10}
101_2	-3_{10}	101_2	5_{10}
110_2	-2_{10}	110_2	6_{10}
111_2	-1_{10}	111_2	7_{10}
000_2	0_{10}	000_2	0_{10}
001_2	1_{10}	001_2	1_{10}
...	...	...	...

NAPA works on the value of the number, i.e. the decimal representation coded as a "long long integer". The result of a high level operation like "*min*" or "*max*" does not have the same meaning for a 2's complement and for an unsigned number. It means that the user must take care to choose the proper coding.

Arithmetic or logic operations are performed with ***ANSI-C*** 64-bit operators on 64-bit code. Appropriate sizing (4 bits in the example below) is then applied. Results are coded in a 64-bit 2's complement format. Decimal representation corresponding to this 64-bit format is then equal to the value obtained using a smaller register width.

Another example:

```
...
(4)  node  sevena  dc (digital) 7
(4)  node  fivea   dc (digital) 5
<4>  node  sevenb  dc (digital) 7
<4>  node  fiveb   dc (digital) 5
...
(4)  node  a12  sum  sevena fivea         //   signed integer wrap-around
<4>  node  b12  sum  sevenb fiveb         // unsigned integer wrap-around
...
output  "stdout"  sevena fivea sevenb fiveb  a12 b12
...
```

Using 4-bit-wide registers and ALU, the operations are:

register "sevena" 0111_2
register "fivea" 0101_2

register "sevenb" 0111_2
register "fiveb" 0101_2

register "a12" 1100_2 **OVERFLOW!** decimal correspondence: -4_{10}
register "b12" 1100_2 decimal correspondence: 12_{10}

Using the 64-bit-wide registers and ALU of the C compiler, the operations are EMULATED as follows (only the 32 lower bits are shown, the upper bits repeat the leftmost bit shown):

register "`sevena`"	0000 0000 0000 0000 0000 0000 0000 0111_2	
emulating 4 bits	0000 0000 0000 0000 0000 0000 0000 $\mathbf{0111}_2$	
register "`fivea`"	0000 0000 0000 0000 0000 0000 0000 0101_2	
emulating 4 bits	0000 0000 0000 0000 0000 0000 0000 $\mathbf{0101}_2$	
register "`sevenb`"	0000 0000 0000 0000 0000 0000 0000 0111_2	
emulating 4 bits	0000 0000 0000 0000 0000 0000 0000 $\mathbf{0111}_2$	
register "`fiveb`"	0000 0000 0000 0000 0000 0000 0000 0101_2	
emulating 4 bits	0000 0000 0000 0000 0000 0000 0000 $\mathbf{0101}_2$	
register "`a12`"	0000 0000 0000 0000 0000 0000 0000 1100_2	
emulating 4 bits	1111 1111 1111 1111 1111 1111 1111 $\mathbf{1100}_2$	equivalent to -4_{10}
register "`b12`"	0000 0000 0000 0000 0000 0000 0000 1100_2	
emulating 4 bits	0000 0000 0000 0000 0000 0000 0000 $\mathbf{1100}_2$	equivalent to 12_{10}

The output of the ***NAPA*** simulation is:

```
...
( output   )
...
...  sevena  fivea  sevenb  fiveb   a12   b12
...       7      5       7      5    -4    12
```

The 4-bit registers are correctly emulated. But emulation is not strictly equivalent to the hardware, as the user must declare which coding is emulated.

To be safe, bitwise operations should use unsigned widths.

List of *NAPA* Instructions

*A **NAPA** netlist is a sequence of one-line instructions describing the network to be simulated, its activation and the analysis tools if any.*

Each instruction is described by one line that begins with a ***NAPA*** keyword (after the optional width prefix of register arithmetic), followed by the identifiers or parameters required by the instruction. A continuation character ("...") can extend the instruction to the next line. Blank lines or comment lines may separate instructions. Nodes and variables may be considered as instructions.

alias

Gives another name to a node or a variable, for readability.

alias <name> <target_node>

alias <name> <target_var>

The new name of the aliased node or variable may be used in the ***NAPA*** netlist instead of the target.

It is useful to replace a complex name generated for a local variable of a cell: the local name is 'promoted' to global. Please note that it is impossible to alias a node or a variable twice.

File "myfile.nap"

```
...
alias   gain     modulator__integrator__gain
...
```

array

The "*array*" statement must be used to declare the type and the size of the "*ram*" and "*rom*" nodes.

array (analog) <name> ‘[’<size>‘]’ [“filpathnam”]

array (digital) <name> ‘[’<size>‘]’ [“filpathnam”]

array (hex) <name> ‘[’<size>‘]’ [“filpathnam”]

It is also used to gang parameters to be transferred by address to a user function (same as the instruction “ganging”, which is preferred):

array (pointer) <name> ‘[’<size>‘]’ <nod_nam | var_nam | arr_nam...>

array (pointer) <name> ‘[ ]’ <nod_nam | var_nam | arr_nam...>

Qualifier *“(hex)”* is equivalent to “*(digital)*” but indicates to the ***NAPA*** transpiler that the initialization file contains hexadecimal data (the addresses remain decimal). Hexadecimal data should follow ***C*** syntax (number beginning with ‘0x’).

The size must be a constant digital type. Except for arrays of pointers, the size of arrays may be defined by an “*ivar*” but this variable cannot be updated.

The file contains the data. This file is mandatory for a ROM. For a RAM, the file corresponds to an initialization and is optional.

```
array (digital) rom1[2048]  "file.rom"
array (analog)  ram2[64]

ivar   sz       100
array (digital) ram3[sz]
```

The input file must contain a number of data matching the declared array size; lines after these data are ignored, with a warning. This file may contain comment lines (beginning with ‘#’, possibly after blanks) and blank lines. A line must contain two numbers: the address and the corresponding data, in C syntax (a leading 0 means octal: 08 or 09 is refused). These numbers may be followed by right-hand comment beginning with ‘//’. Addresses must be ordered. Addresses range from 0 to size-1.

The ***NAPA*** transpiler will not read the content of the file itself but will instruct the simulator to do it at run time.

For example, an 8-word ROM:

```
array (digital) myrom[8]  "prog.rom"
```

with the file “prog.rom”:

```
# ROM program - revision 1.20 -
#      address   data
       0         14
       1         132345
       2         23
       3         54365      // corrected by JDoe (10/2/1994)
       4         28991
       5         0
       6         0
       7         0
#
```

Arrays of pointers allow ganging parameters to transfer them by address to a function, a user-defined function or a tool. The declaration of the size is optional. It corresponds to the exact number of items. Ganging of different types of nodes, variables and other arrays of pointers is allowed.

```
...
dvar    c1   1.1
dvar    r2   2.2

dvar    freq 1.2345e3

string  nam  "rc"
string  tag  "typic"

dvar    wt   _2PI_ * freq * TIME
update  wt
...
array (pointer)  Coef[3]   nam  c1 r2
array (pointer)  Num[2]    n1 n2
array (pointer)  Den[5]    d1..5

array (pointer)  Equ[4]    wt tag Num Den
array (pointer)  All[]     Equ tag s1
...
node s1  sum     a  b
node out duser   myfun  in   Coef  All  54321.0
...
```

Parameters ganged in an array of pointers may be referenced by their numbers. An example is shown below:

```
...
array (pointer)  Num[2]    a b
array (pointer)  Den[5]    d1..5
array (pointer)  Equ[2]    Num Den
...
node s1  sum   b  d5  Num.2   // second parameter of array Num, i.e. b
output   "stdout"  Den.3  a   // third parameter of array Den,  i.e. d3
...
```

NOTE:

If a file is declared for a RAM, data will be read at initialization.

LOCATION INDEPENDENT INSTRUCTION

This instruction is a declaration which can be located anywhere in the netlist. For instance, it is not necessary that the corresponding *"ram"* or *"rom"* nodes appear before the array instruction.

assert

The "*assert*" statement is used as a watchdog to detect abnormal conditions at run time for debugging purposes.

assert <“message”> <*C*_Boolean_expression>

assert <“message”> <*C*_Boolean_expression> when <event_condition>

If the ***C*** Boolean expression is false, the simulation stops, the message (a string constant) is sent to standard error output “stderr” and normal termination occurs. Any expression returning an integer is accepted, a function call included.

```
assert "Too long!" LOOP_INDEX < 1000000LL

assert "Wrong!" OK(sigval)
```

“*assert*” can be inactivated by defining the ***C*** preprocessor directive “*NO_ASSERT*” in user's profile header (see “header”) or using the instruction “*directive*”:

```
directive  NO_ASSERT
```

“*assert*” can be advantageously used in combination with statement “*gateway*” to control the exit point of the simulation. Combination of “*assert*” and “*dump*” instructions makes debugging easier.

NOTE:
The message is a constant string where it is *not* possible to use variable indirection (#...).

LOCATION DEPENDENT INSTRUCTION

This instruction depends on its position inside the netlist. It is segment dependent. “assert” instructions are executed in the section of variable updates, before the node updates and the time-domain output. During the first loop of the simulation, the condition is ignored to avoid unwanted triggering.

call

You can call a user-defined function in the same way that you update a variable, using instruction “*call*”. This instruction is not used very often as ***NAPA*** offers more powerful features (see “*duser*”, *“iuser”,* “*itool*” and “*dtool*”) and its usage is *NOT* recommended. User-defined-function code is located inside a header file which has to be declared inside a “*header*” instruction. The transpiler processes calls in a way similar to variable updates.

call <return_variable> <*C*_function>

call void <*C*_function_returning_void>

For example:

```
header <napa.hdr>
header "hello.hdr"     // file containing code of function "greetings"
...
call void greetings(1)
...
```

Function “greetings( )” being a function returning void, defined in the user's ***ANSI-C*** header file:

File “hello.hdr”

```
#ifndef  __HELLO_HDR__
#define  __HELLO_HDR__

/* *** FUNCTION PROTOTYPE ******************************** */
void greetings(int);

/* *** FUNCTION DEFINITIONS ****************************** */
void greetings(int choice){
  switch (choice) {
  case  1: fprintf(stderr, "Hello John\n");  break;
  case  2: fprintf(stderr, "Good bye\n");    break;
  default: fprintf(stderr, "Error\n");       break;
  }
  return;
}
/* ******************************************************** */
#endif                                    /* __HELLO_HDR__  */
```

The calls are sorted with the updates, by their dependencies; independent ones keep the order of the netlist. They are executed in the section of variable updates. Functions called by instruction “*call*” are not initialized automatically. These functions are not usually returning any value although this is perfectly possible. It is important to note that more powerful primitives supersede this instruction. Therefore the usage of “*call*” should be limited to very specific situations.

LOCATION DEPENDENT INSTRUCTION

This instruction depends on its position inside the netlist. It is segment dependent. “call” and “update” instructions are sorted together by their dependencies (“assert” first); independent ones keep the order of the netlist.

command_line

command_line <var_nam...> | fs | ts | void

This instruction allows the simulator produced by NAPA to take parameters from the command line. This is useful when the executable is given to a third party. The variables must be declared using “*dvar*”, “*ivar*” or “*string*” instructions. These declarations give no value, only an optional comment string. Sampling frequency *“fs”* may be input from the command line. In this case no value should be assigned to sampling frequency in the ***NAPA*** netlist.

Use “*void*” identifier to indicate that the stand-alone simulator does not take any input.

For example, if the ***NAPA*** file “my_module.nap” contains the following instructions:

File “my_module.nap”

```
...
fs
...
command_line  fs infile  value1
...
command_line  np
...
```

```
string infile  "some optional string comment"              //  input file
ivar   np      "another optional string comment"           //  parameter
dvar   value1  "and a third optional string comment"       //  value
...
```

The executable “my_module.exe” produced from the ***C*** source will be called as:

```
%  my_module.exe  1.0e6  "foo.dat"  1.2345  18 ↵
%
```

The value 1.0e6 will be assigned to sampling frequency, “foo.dat” will be assigned to the string variable <*infile*>, the value 1.2345 to the analog variable <*value1*> and the value 18 to the digital variable <*np*>.

See also instruction “*random_seed*” to configure the random number generator properly.

LOCATION INDEPENDENT INSTRUCTION

comment

This instruction publishes a comment during the parsing of the netlist and writes a message on the “stderr” output. Comments in a job are concatenated. Use “\n” to break lines.

comment “<some_message>”

This instruction also documents the usage of the stand-alone simulator created when the instruction “command_line” is used.

A ***C*** macro “COMMENT” is defined accordingly in the ***C*** produced by NAPA.

File “opamp.dat”

```
...
comment  "here is an example of comment,\n"
comment  "a second one\nand a third one\n"
...
```

data

data <“file_pathname”> parameters ...

data <“file_pathname”> names::parameters ...

The first syntax corresponds to the transfer of parameters by position. The second syntax corresponds to the transfer of parameters by name.

This instruction is intended to instantiate sets of variables. This instruction is related to pseudo-node “*cell”*. Only variable definitions and related instructions are allowed inside the data file. In particular, nodes and control instructions (“drop”, “decimate”...) are forbidden. The first line of the data file is the data cell interface and must begin with the keyword “data_interface” followed by formal parameters.

For more information concerning this instruction, see the pseudo-node "*cell*" although the pseudo-node "*cell*" does not support the transfer of parameters by name.

In the following examples, the instruction "*data*" is used to call the data related to an opamp, the data depending on a bias current.

File "opamp.dat"

```
data_interface $gm $gds $ibias

dvar $isqr    $ibias * $ibias
dvar $isqrt   pow($ibias, 0.5)
dvar $gm      -296.2e-6 + 233.3e-6 * $isqrt - 1.220e-6 * $ibias
dvar $gds     -13.16e-9 + 1.187e-9 * $ibias + 2.322 * $isqr
```

Here is the instantiation of this data cell with a transfer of parameters by position, the parameters must be ordered in the exact order of the data cell parameters:

```
...
dvar  ib  10e-6
data  "opamp.dat"  gm1 gds1 ib
...
```

and the instantiation of this data cell with a transfer of parameters by name. The names must correspond to parameters in the definition of the data cells:

```
...
dvar  ib  10e-6
data  "opamp.dat"  ibias::ib  gm::gm1  gds::gds1
...
```

With a transfer by name, the data cell may have more parameters than the instantiation list. It is therefore easier for a data cell to be used in different simulation netlists!

This instruction obeys the ***NAPA*** file system. In the following examples, the first call points to a file located in the main directory, the second call points to the current cell directory, the third call points to the root directory, i.e. the working directory where the ***NAPA*** command was invoked:

```
data "~/opamp.dat" v1 v2
data "./opamp.dat" v1 v2
data "opamp.dat"   v1 v2
```

Both the instantiation parameters and the formal parameters may contain iterative identifiers:

File "file1.dat"

```
data_interface $gain $h0..3 $scale

dvar $gain  1000.0
dvar $h0    0.00*$scale
dvar $h1    1.01*$scale
dvar $h2    2.01*$scale
dvar $h3    3.01*$scale
```

```
...
data "file1.dat"  parm0..4  0.50
...
```

This is equivalent to:

File “file2.dat”

```
data_interface $gain $h0 $h1 $h2 $h3 $scale

dvar $gain  1000.0
dvar $h0    0.00*0.50
dvar $h1    1.01*0.50
dvar $h2    2.01*0.50
dvar $h3    3.01*0.50



...
data "file2.dat"  parm0 parm1 parm2 parm3 parm4 0.50
...
```

LOCATION DEPENDENT INSTRUCTION

debug

debug <debug_level_number | identifier ...>

The “*debug*” statement is adding a ***C*** preprocessor directive in the output ***C*** program for each entry, using “DEBUG_MODE_” as prefix.

This is practical when debugging a new user header. Use the instruction “*debug*” to activate the appropriate debugging level. The identifier, or the number, is used as suffix to form a macro identifier in the output ***C*** program.

```
debug  2
```

This will trigger, in the code generated by NAPA**,** the definition of the macro:

```
#define DEBUG_MODE_2
```

```
debug  SAMPLING
debug  TOOL


debug  SAMPLING TOOL
```

The two lines “debug SAMPLING” and “debug TOOL”, or the single line “debug SAMPLING TOOL”, trigger in the code generated by ***NAPA*** the definition of two macros:

```
#define DEBUG_MODE_SAMPLING
#define DEBUG_MODE_TOOL
```

The user should consult the documentation of the user-defined functions or tools contained in the header files to obtain the list of the debug parameters corresponding to these functions.

LOCATION INDEPENDENT INSTRUCTION

This instruction is a declaration which can be located anywhere in a netlist but not in a data cell.

decimate

Keyword "*decimate*" introduces a new segment in the netlist. Nodes and variables are processed by segment (see chapter describing the concept of ***NAPA***, page 12). A decimation is specified by the keyword "decimate", the decimation factor and, if needed, an initial value:

decimate [fs] <decimation_factor> [<decimation_initial_value>]

Decimation factor must be a strictly positive integer CONSTANT. No variable is allowed. The decimation initial value is optional (default = 0) but must also be a non-negative integer constant. It allows the initialization of the corresponding decimation counter to a value other than 0 in the generated ***C*** code.

NAPA computes the local sampling frequency determined by the decimate factor. Anything that follows the "*decimate*" line will only be evaluated at the decimated rate. See also appendix A, page 155, for more information. Decimation factor is relative to the previous segment rate, except when the keyword "fs" is added: it then refers to the absolute sampling frequency defined by the instruction "fs".

For example:

```
#* segment running at sampling frequency fs
...
decimate  16
...
#* segment running at sampling frequency fs/16

#* segment running at sampling frequency 4MHz
...
fs 4.0e6
...
interpolate  3
...
#* segment running at sampling frequency 12MHz
...
decimate fs  4
...
#* segment running at sampling frequency 1MHz
```

LOCATION DEPENDENT INSTRUCTION

This instruction deeply modifies the behavior of the simulation. Be careful to place this instruction at the appropriate location in the netlist.

declare

It is generally not necessary to declare node types in ***NAPA.*** This instruction is used mainly to document complex or highly hierarchical user-defined cells, preventing ***NAPA*** from issuing cryptic error messages by checking the type of variables or nodes at the highest possible levels.

declare (analog) <identifier_list>

declare (digital) <identifier_list>

declare (string) <identifier_list>

declare ( ) <identifier_list>

declare (constant) <identifier_list>

declare (true) <a_function_returning_a_boolean>

Types of nodes or variables can be declared many times in a netlist. Of course, type declarations must agree. After the automatic node determination by netlist inspection, the transpiler will check that the node type corresponds to the type given in the declaration. Variable definitions are checked directly against their declarations. Four kinds of type declarations are possible: (analog), (digital), (string) or ( ); (constant) and (true) are checks, described below.

The fourth kind means that the type of nodes or variables is not known at the time the netlist was written but that all the nodes or variables inside the list must have the same type. This is useful when writing cells with chameleonic nodes.

The declaration '*declare (constant)...*' checks if the elements are defined as constants, i.e. that a variable is not updated or that a node is a "dc" or a "const" node.

'*declare (true)...*' is the only declaration which is executed at run time, at the very beginning of the initialization. It is a good way to identify a faulty value before it does any harm.

A first example of type declaration in a cell:

File "mycell1.net"

```
cell_interface  $nodout $nodin $parm1 $parm2

# this cell accepts either both digital or both analog inputs
# following declaration will check homogeneity of inputs as soon as
# this cell will be instantiated: listed nodes and parameters in
# following declaration must be the same type.

declare () $nodin    $parm1 $parm2

node $s1      gain   $parm1  $nodin
node $nodout  offset $parm2  $s1
```

Another example of type declaration:

File "mycell2.net"

```
cell_interface    $h0..6

declare (analog)  $h0..2  $h4
declare (digital) $h3
declare ( )       $h5..6
...
```

Type declaration is used also when ***NAPA*** is unable to determine the type of nodes. It occurs in very few cases, when a loop of chameleonic nodes gives no hint about the type of the signals.

An example of declarations to check the value of an expression:

File “mycell3.net”

```
cell_interface    $in  $gain $ndel

declare (analog)    $gain
declare (digital)   $ndel
declare (constant)  $gain $ndel

declare (true)      $gain > 0.0
declare (true)     ($ndel > 0) && ISEVEN($ndel)
...
```

A variable can also be declared constant directly:

```
dvar  foo 2.0    &constant
...
```

LOCATION INDEPENDENT INSTRUCTION when declaring types or constant
LOCATION DEPENDENT INSTRUCTION when using ‘*declare (true)...*’.

directive

The compilation of user-defined functions is often configured by ***C*** macro preprocessor definitions. Using instruction *“directive”*, the user can introduce a macro definition inside the produced ***C*** code (Another way is to use a user’s profile contained in a header file).

directive <macro_identifier> [(<parameter>)] [<value>]

directive (<macro_identifier>) [<value>]

Directives with values directly produce the corresponding macro definition. Directives without value produce an additional definition with suffix “_IS_EMPTY” to ease the programming of user functions.

NAPA checks that each directive is used (registered) by a header; parentheses around the directive identifier disable this check.

For example:

```
...
directive  WINDOW       BLACKMAN_HARRIS
directive  (FOO)        0.0
directive  MYFILE       "./myfile.dat"
directive  REPEAT       10
directive  NO_BANNER
directive  MYFUN1(x,y,z) x*pow(y,z)
directive  MYFUN2(x,y)
...
```

This source will be translated in the produced ***C*** code as:

```
...
#define FOO           0.0
#define MYFILE        "./myfile.dat"
#define MYFUN1(x,y,z) x*pow(y,z)
#define MYFUN2(x,y)
#define MYFUN2_IS_EMPTY
#define NO_BANNER
#define NO_BANNER_IS_EMPTY
#define REPEAT        10
#define WINDOW        BLACKMAN_HARRIS
...
```

This is equivalent to:

```
...
header "my_profile.hdr"
...
```

Where “my_profile.hdr” is a file containing explicitly the ***C*** preprocessor macro definitions.

File “my_profile.hdr”

```
/* *** C header file containing C preprocessor directives ********** */


#ifndef  __MY_PROFILE__
#define  __MY_PROFILE__
...
#define  WINDOW         BLACKMAN_HARRIS
#define  FOO            0.0
#define  MYFILE         "./myfile.dat"
#define  REPEAT         10
#define  NO_BANNER_IS_EMPTY
#define  NO_BANNER
#define  MYFUN1(x,y,z)  x*pow(y,z)
#define  MYFUN2_IS_EMPTY
#define  MYFUN2(x,y)
...
#endif                                              /* __MY_PROFILE__ */
```

See also “*header*” below.

Please note that it is forbidden to use a ***NAPA*** keyword as a ***NAPA*** directive identifier or to duplicate the declaration or the definition of a directive. It is not allowed to define a directive referring to a node or a variable.

The user should consult the documentation of the user-defined functions or tools contained in the header files to obtain the list of the directives corresponding to these functions.

LOCATION INDEPENDENT INSTRUCTION

This instruction is a declaration which can be located anywhere in the netlist. Directives are sorted alphabetically in the C code produced by ***NAPA***.

drop

Keyword "*drop*" introduces a new segment in the netlist. Nodes and variables are processed by segment (see chapter describing the concept of ***NAPA***, page 12). The "*drop*" statement is somewhat similar to the "*decimate*" or "*interpolate*" statement.

drop [fs] <*C*_Boolean_expression>

If the expression is true, everything below the "drop" statement in the local segment is executed; otherwise the segment is skipped. Instruction has no effect on other segments below. "drop" is relative to the previous segment rate, except when the keyword "fs" is added: in this case it refers to the absolute sampling frequency defined by instruction "fs". It does not refer to any other condition in previous "drop" segments.

It is important to note that the "drop" condition is computed at the beginning of the loop before the update of nodes or variables. Therefore all references to nodes or variables in this Boolean expression refer to the values of the *previous* loop.

CAUTION:
NAPA does not take into account the change of sampling rate caused by "drop": in a "drop" segment, FSL is the rate of the previous segment (of the main loop with "drop fs") and PERIODIC is false.

LOCATION DEPENDENT INSTRUCTION

This instruction modifies deeply the behavior of the simulation. Be careful to place this instruction at the appropriate location in the netlist.

dump

The instruction "dump" writes the state of the simulation to a file. Dump is conditional. Only one "*dump*" is authorized inside a ***NAPA*** netlist. The "*dump*" condition can be fired several times during the simulation. This instruction should be reserved for debugging purposes.

dump <"file_name"> true

dump <"file_name"> false

dump <"file_name"> when <event_condition>

<event_condition> is a Boolean combination of events defined by instruction "*event*".

If the Boolean expression returns TRUE, then the dump will be issued at the end of the main loop.

"*dump*" produces a file containing ***C*** instructions and ***C*** preprocessor directives. The values of all the nodes and variables are dumped. Decimation variables, current time and loop index are also output. This file is compatible with the instruction "*load*".

```
...
event  ev1   ((LOOP_INDEX % 1000LL) == 0LL)
event  ev2   TIME > 10.0e-6
...
dump   "foo.dmp"  when (ev1 && ev2)
...
```

CAUTION:

This instruction should be used with care as a misuse can produce a tremendous amount of data... Therefore the usage of "*dump*" should be limited to very specific situations.

TIPS:

"*assert*" statement will produce a dump at the exit of the simulation if a "*dump*" is declared. To produce a dump only in this case, use a condition which is never TRUE (for instance, use the constant FALSE as Boolean expression).

LOCATION DEPENDENT INSTRUCTION

Triggering is segment dependent, but the dump is actually written at the end of the main loop.

dvar, ivar

dvar <var_nam> [<initial_value>] [&update | &constant] [&export]

ivar <var_nam> [<initial_value>] [&update | &constant] [&export]

Here "dvar" stands for analog variables (double precision) and "*ivar*" for digital variables (long long integer).

Short forms to ask for update, export or to declare "*dvar*" or "*ivar*" as a constant may be added. See the chapter "Short Forms".

Please note that there is a special syntax for variables when they are part of the instruction "*command_line*". See this instruction for this special case.

These instructions declare and initialize variables at the same time. They are sorted. The initial value can be any ***C*** expression and can involve previously declared user's variables. Self-reference is not allowed. Initial value is not mandatory: default values are ANALOG_INI or DIGITAL_INI, depending on the type of the variable. Initial value may be overwritten by instruction "*input*".

Each variable must be declared once and *only* once. The ***NAPA*** transpiler sorts variables although it is not generally recommended to scramble the definitions (to maintain human readability).

A variable and a node cannot share the same name.

Variable values can be introduced at run-time from a file or "*stdin*" using instruction "*input*". In this case, if the user has already initialized the variable in a variable definition, a warning is produced.

In some situations, a declaration of a variable is necessary but no definition is needed. In this case, use the special syntax:

```
...
dvar myvar    (void)
ivar yourparm (void)
...
```

Variable declarations may be completed by variable updates if the variables must vary during simulation.

Although variables are defined before nodes in the generated ***C*** code, it is nevertheless allowed to define variables using nodes' values. For initialization, ANALOG_INI or DIGITAL_INI will replace the nodes' values. See also paragraphs concerning "update" and "input" below.

✘ ***NAPA*** handles variables and nodes differently:

VARIABLES ARE SORTED SEGMENT BY SEGMENT.
VARIABLES ARE UPDATED BEFORE NODES IN A SIMULATION LOOP.

Compare variables to nodes, and see also appendix A, page 155, for more information.

LOCATION DEPENDENT INSTRUCTION

The ***NAPA*** transpiler sorts the variables. They are therefore not declared and initialized in the order they appear in the netlist. Thus a variable can refer to a variable not yet declared. Of course, no loop of definitions is allowed. Variables using local sampling frequency must be placed at the correct position with respect to the decimation and interpolation instructions.

✘ **CAUTION**:

A COMMON MISTAKE is to believe that a variable is updated by default. Use instruction "*update*"!

---

error

This instruction issues an error and stops the parsing of the netlist. It prevents the simulation of the netlist and writes a message on the "stderr" output.

error "<some_message>"

---

event

This is a special kind of integer variable (see "*ivar*"). This variable is automatically updated (see "*update*") at the local sampling rate. This is the only variable authorized in a condition applied to an "*update*", an "*output*" or a "*dump*". The definition of an event is expected to return an integer value (non-zero value being considered as TRUE). The ***NAPA*** transpiler sorts the events, like the other variables.

The modifier *(new)*, placed in front of the ***C*** expression, makes the event watch for a change in this expression. The event is FALSE when the expression is unchanged, TRUE when it changes. It is important to note that the ***C*** expression itself may return an integer or a real value but that the event is returning an integer value.

CAUTION: It is important to note that the events are updated in the loop before the nodes. Therefore they cannot react to a node change before the next loop.

event <event_var_name> <*C*_expression_returning_an_integer>

event <event_var_name> (new) <*C*_expression_returning_a_number>

```
...
event  ev0   ampldb == -3.0
event  ev1   (new) ampldb
event  ev2   (LOOP_INDEX % 100LL) == 0LL
...
```

```
output "stdout"  a b c d   when ev0
update  ampl               when ev1 && ev2
update  freq               when ev2
...
```

LOCATION DEPENDENT INSTRUCTION

---

export

Additional node(s) or variable(s) can be transferred to user-defined tools with:

export [<var_nam> | <nod_nam> |

LOOP_INDEX | ABS_LOOP_INDEX | REL_LOOP_INDEX |

TIME | ABS_TIME | REF_TIME | REL_TIME | WALL_CLOCK] ...

Exported nodes or variables must be digital or analog type; string variables cannot be exported.

User's functions or tools can be prepared to consider these additional nodes or variables, using the internal macro EXPORT. Please note that some tools will only use the first exported variable and ignore the other ones.

Please consult header files for information concerning a specific user's function or tool.

```
...
dvar     myvar1   log(TIME)
ivar     myvar2   2

...
update  myvar1
...
#* 3 variables are exported here: myvar1, myvar2 and WALL_CLOCK

...
export  myvar1  WALL_CLOCK
...
export  myvar2
...
```

An example: the SMART TOOL. Although this variable does not belong to the explicit list of inputs of the tool, “tsnr” will include the current value of variable “sig_freq” inside the output file “/jdoe/simu/mytsnr.out” as part of the simulation results.

File “/simu/mytsnr.nap”

```
...
dvar   sig_freq   1000.0
export sig_freq
...
node ctr itool tsnr  "/jdoe/simu/mytsnr.out"  s3  1.0  1.0e3  16384
...
update  sig_freq   (TOOL_INDEX + 1)*1000.0
...
```

The user-defined tool “tsnr” has been defined to add, in its output file, an extra column relative to the export identifier if any.

File “/simu/mytsnr.out”

```
# MYTSNR
# (tool                ) tsnr - frequency domain analysis
# (transpiler version  ) NAPA V5.03
# (source file         ) mytsnr.nap
# (random seed         ) 474285303
# (normalization       ) s3 / 1
# (bandwidth           )   1.000  Hz ..  1.000 kHz
# (samples             ) 16384
# (sampling frequency  )   1.000 MHz
# (frequency resolution)  61.035  Hz
# (window              ) 4 Sample Blackman Harris
#
#
# Sun Nov  1 20:50:07 2026 by YLEDUC
# packet        freq        sig_RMS          noise           tsnr        sig_freq
   0   9.765625e+002 -7.446675e+001 -7.053087e+001 -3.935882e+000   1.000000e+003
   1   9.765625e+002 -6.731440e+001 -6.917046e+001  1.856052e+000   2.000000e+003
   2   9.765625e+002 -6.512889e+001 -7.023648e+001  5.107589e+000   3.000000e+003
   3   9.765625e+002 -6.394195e+001 -7.122892e+001  7.286974e+000   4.000000e+003
...
# end of output file
```

It is also possible to export a variable by adding “&export” to the definition of the variable:

File “/mysimu.nap”

```
...
dvar     myvar1   log(TIME)      &export   &update
ivar     myvar2   2              &export
...
```

LOCATION INDEPENDENT INSTRUCTION

This instruction is a declaration which can be located anywhere in the netlist.

format

This instruction defines the output format for nodes, variables and strings in the file produced by the instruction “*output*”. These formats may be used in user’s functions.

format (analog) <“*C*_double_output_format”> | S | M | L

format (digital) <“*C*_long_long_output_format”> | S | M | L

format (hex) <“*C*_long_long_output_format”> | S | M | L

format (string) <“*C*_string_output_format”> | S | M | L

The formats follow the syntax of ***C*** printf. The default output format for analog type is “ **% .12e**”, a format compatible with double float numbers. For digital type it is “ **% 1111d**”, a format compatible with long long integer and “ **%12s**” for string. Predefined format ‘**S**’, ‘**M**’ and ‘**L**’ are available, the format ‘M’ corresponds to the default value, ‘S’ and ‘L’ respectively to smaller and larger formats, with the exception of hex type where the largest value is the default. The predefined hex formats write the prefix 0X before every value, 0 included.

Predefined Formats	**S**	**M**	**L**
Analog type:	" % .9e"	" % .12e"	" % .15e"
Digital type:	" % 6lld"	" % 11lld"	" % 21lld"
Hex type:	" 0X%04llX"	" 0X%08llX"	" 0X%016llX"
String type:	" %6s"	" %12s"	" %24s"

```
format (digital) " <%5llx> "
format (analog)  " %5.2f"
format (string)  " %5s"

format (analog)  L
format (hex)     S
```

NAPA verifies only that the format holds one conversion, and only one, fitting the type: e, f or g for (analog), d, i, u, o or x for (digital) and (hex), s for (string). The values of (digital) and (hex) being long long integers, a conversion without the length modifier 'll' gets it: '%6d' is used as '%6lld'. The rest of the format is not verified: an incoherent format is a classical and hazardous error in ***C*** programming. Prefer the predefined '**S**', '**M**' or '**L**'!

NOTE:
A format cannot be given to one particular node or variable.

LOCATION INDEPENDENT INSTRUCTION

This instruction is a declaration which can be located anywhere in the netlist.

---

fs

The main sampling frequency is set (optionally) by:

fs [<sampling_frequency>]

The sampling frequency must be a positive double precision CONSTANT number. This is the main sampling frequency of the simulation. See also instruction *"ts"*.

```
fs  1.0e6
```

Default value is 1.0, with a warning when neither "fs" nor "ts" is given.

```
fs
```

At most one instruction "fs" is allowed in a netlist.

CAUTION:
Sine wave generators and similar node kinds use the sampling frequency to compute the signal. A common mistake is to forget to define "*fs*". In this situation, ***NAPA*** uses the default value, the simulator risks processing sine or cosine of huge numbers, losing accuracy or simply overflowing.

LOCATION INDEPENDENT INSTRUCTION

This instruction is a declaration which can be located anywhere in the netlist.

ganging

It is used to gang parameters to transfer by address to a user function (identical to "*array (pointer)*"):

ganging <nam> '['<size>']' <nod_nam | var_nam | string | num | arr_nam...>

ganging <nam> '[ ]' <nod_nam | var_nam | string | num | arr_nam...>

The array of pointers allows ganging parameters to transfer them by address to a function, a user-defined function, or a tool. The parameters are ganged in their specified position ('ganging by position') but this instruction supports the transfer of parameters by name if the operator '::' is used ('ganging by name').

ganging <nam> '['<size>']' <id1::id2...>

ganging <nam> '[ ]' <id1::id2...>

The parameter "id2" is transferred by address but it is tagged by an identifier "id1". A user function receiving such an array may be written in such a way that it reorders the list of ganged parameters according to a template.

The declaration of the size is optional. It corresponds to the exact number of items. Ganging of different types of nodes, variables and other arrays of pointers is allowed.

```
...
dvar     c1   1.1
dvar     r2   2.2
dvar     freq 1.2345e3

string   nam  "rc"
string   tag  "typic"

dvar     wt   _2PI_ * freq * TIME
update   wt
...
ganging  Coef[3]   nam  c1 r2
ganging  Num[2]    n1 n2
ganging  Den[5]    d1..5

ganging  Equ[5]    wt tag 1.23 Num Den
ganging  All[]     Equ tag c1
...
node s1  sum   a  b
node out duser myfun  in   Coef  All  54321.0
...
```

Parameters ganged in an array of pointers may be referenced by an id. An example is shown below, where the duser function "myfun" finds the parameters of "parm" by their names (template 'Circuit, A, Offs, Cin, Rload'), not by their positions; this lookup is done inside the function.

```
...
dvar     c1   1.1
dvar     r2   2.2
dvar     gain 1.0e3
string   nam
...
node     in     osc 0.0 1.0 1234.5 0.0
...
ganging  rc[]   Cin::c1  Rload::r2
ganging  op[]   A::gain  Offs::0.0
ganging  parm[] rc op  Circuit::nam

node out duser myfun  in   parm in
...
```

Parameters ganged in an array of pointers may be referenced by their numbers. An example is shown below:

```
...
ganging  Num[2]    a b
ganging  Den[5]    d1..5
ganging  Equ[2]    Num Den
ganging  prm[5]    1.2345 ampl frq  "input signal" TRUE
...
node s1  sum   b  d5  Num.2   // second parameter of array Num, i.e. b
output   "stdout"  Den.3  a   // third parameter of array Den,  i.e. d3
...
```

LOCATION INDEPENDENT INSTRUCTION

gateway

The "*gateway*" statement is used to control the exit point of the simulation corresponding to "*assert*" statement(s). Syntax:

gateway [<count_down_number>]

Only one "*gateway*" is allowed in a simulation. This instruction allows the simulation to continue after an exit request until the segment containing the gateway is completely executed. The "*gateway*" statement can initiate a countdown to force the execution of the segment several times before exiting. It is disabled when "assert" is disabled (NO_ASSERT).

```
gateway


gateway 12
```

LOCATION DEPENDENT INSTRUCTION

header

Header files are ***C*** code that is to be included in the output program to be compiled with the ***C*** code generated by ***NAPA***. Header files are not precompiled, so that they can use the ***C*** preprocessor directives contained in user's profile header files or ***NAPA*** instruction "*directive*".

To include a header file:

header <"file_pathname"> [(noexpand)]

header <"file_pathname"> (expand)

The *first* header file that *must* be included is "napa.hdr". This header contains information needed to generate correct ***C*** code. This header file also defines a few useful functions to get you started. A copy of "napa.hdr" is currently located in the library of generic headers. The users can customize this file or a copy of this file: any header name starting with the letters <*napa*> with a suffix <.hdr> is acceptable.

Optional qualifier keywords *(expand)* and *(noexpand)* control the header expansion into generated ***C*** code. The header files are respectively copied or included (as ***C*** preprocessor directive "*#include*") in the output ***C*** program depending on the qualifier (*expand*) or (*noexpand*). Default and recommended value is (*noexpand).* Qualifier (*expand*) cannot generally be used for headers that themselves include other headers, as they could not be correctly recognized (conflict between the ***C*** pathname and the ***NAPA*** pathname). It has therefore a limited interest.

A directory is dedicated to store a library of reusable headers. The pathname of this library is indicated to ***NAPA*** through the command line. Using angle brackets "<...>" is a short form to include header files belonging to this directory. If "/NAPAlib/hdr" is the pathname of this directory, the following instructions are equivalent:

```
header <napa.hdr>

header "/NAPAlib/hdr/napa.hdr"
```

Headers are processed in the order they are specified in the ***NAPA*** netlist. (Take care to mention proper pathname!). A call of a header already processed is always ignored if spelling is identical.

A user's profile header can be added to control the ***C*** compilation and to profile user-defined functions (see the example "my_profile.hdr" in the instruction "directive"). If used, such header must be placed just after the header "napa.hdr". This header is optional. It contains typically ***C*** preprocessor directives ("*#define*") controlling the compilation of the generated ***C*** code. This is not the preferred way to introduce a user's profile: see instruction *"directive".* Several users' profile headers can be used if it is more convenient. Please follow the ***NAPA*** recommendation: use ".hdr" suffix for ***NAPA*** header files (see appendix C page 166).

A typical ***NAPA*** netlist could contain lines like:

```
...
header <napa.hdr>
header "/home/NAPA/hdr/tool/fft1.hdr"
...
```

The same header can be called several times in the netlist without causing multiple inclusions in the ***C*** file.

TIPS:

It is always a good idea to place the corresponding header instruction in the cell calling a user's function or tool. This guarantees that the necessary header file is actually included.

LOCATION INDEPENDENT INSTRUCTION

BUT headers are processed in the order they appear in the netlist. Header "napa.hdr" must be the first header of the netlist.

init

You can call a ***C*** function to initialize something in the same way that you initialize a variable, using instruction "*init*". This statement is also the preferred way to initialize nodes.

init void <*C*_function>

init <nod_nam> <*C*_expression>

These are executed only in the initialization section. Nodes can be initialized with "*init*" but must be defined by a specific "*node*" instruction. It is not allowed to initialize variables using this instruction.

For example:

```
header <napa.hdr>                          // generic header file
header "array.hdr"
...
init s4   3.0                              // node initialization
init void zero_arrays(0.0)                 // execution of function
...
```

Function "zero_arrays( )" being defined in a user's header file "array.hdr":

File "array.hdr"

```
#ifndef __ARRAY_HDR__
#define __ARRAY_HDR__

/* ** GLOBAL VARIABLES ************************************* */
double array[100];

/* ** FUNCTION PROTOTYPES ********************************** */
void zero_arrays(double);

/* ** FUNCTION DEFINITIONS ********************************* */
void zero_arrays(double valinit) {
  int i;
  for (i=0; i < 100; i++) {
    array[i] = valinit;
  }
  return;
}


#endif                                          /* __ARRAY_HDR__ */
```

If necessary, nodes are automatically initialized to “ANALOG_INI” or “DIGITAL_INI” depending on their types. Nevertheless, you should consider initializing the **output** of “*delay*” node, “*integrator*”, *“toggle”,* “*latch*”, “*hold*”, “*track*”, “*register*” and “*muller*” nodes, as they contain or could contain memory elements. This reflects the reset a designer puts in a module.

Other nodes cannot be initialized, as the simulator will immediately overwrite their values. In this case, a warning is issued as the initialization of such a node has no effect.

To verify or validate a circuit, initialization of these nodes could be especially important. Although it does not guarantee complete coverage, it is a good idea to randomly initialize nodes which will not be reset in the physical implementation.

Here is an example where the initialization of a node is mandatory: the computation of the minimum value taken by a node during a simulation.

```
#* "amin" is taking the 'historical' minimum value of node "a"

node oldmin  delay  amin          // "oldmin" is the previous minimum
node amin    min    a  oldmin     // minimum of "a" and "oldmin"
init oldmin  9.9e99               // initialization to a huge value
```

LOCATION DEPENDENT INSTRUCTION

Initialization can include segment dependent values. Initializations are sorted with the variables, by their dependencies, and executed as early as possible.

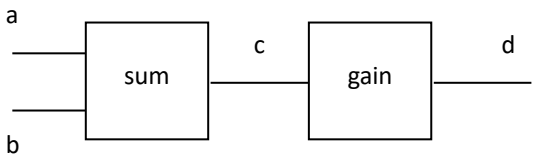
inject

It is possible to inject a signal on an ANALOG node without reworking the netlist.

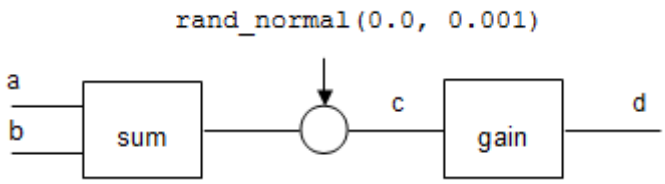
inject <nod_nam> <*C*_expression>

Any analog node is a candidate for noise injection, including “*dc*” or “*const*” nodes if they are analog type. The ***C*** expression describes the noise to be injected. The noise is added to the signal at a rate equal to the local sampling frequency of the signal receiving the noise, independently of the location of the “*inject*” instruction in the netlist. The injected signal is added wherever the node is read, including by “output” and “export”.

```
node   c  sum   a   b
node   d  gain  2.0 c
```



```
node   c  sum   a   b
...
inject c  rand_normal(0.0, 0.001)                  // noise injection
...
node   d  gain  2.0 c                              // noisy signal
```



input

The executable produced from the ***NAPA*** netlist can be prepared to accept data from a data file. Several inputs can be used in the same netlist. Only digital or analog variables can be input. It is not allowed to input nodes or string variables. Variables must be declared using the instructions "*dvar*" or "*ivar*". The variable declaration determines the type of the variable (analog or digital). The simulator, in a way similar to variable updates, processes the inputs but the input does not depend on the "decimate", "interpolate" or "drop" instructions. "ivar" may be input as regular decimal numbers, octal with prefix 0 or hexadecimal with prefix 0X.

This is not the preferred way to input data. Prefer dedicated user-defined functions, as they are more powerful and programmable than this instruction "*input*".

It is important to note that the simulation will stop if the end of the input file is reached. During the simulation, one line of values is read at each loop.

input "stdin" <var_nam...>

input <"file_pathname"> <var_nam...>

input <myvar> <var_nam...>

Here is an example of input:

```
...
dvar value1
dvar value2
ivar value3
ivar value4

string filin   "somefile.dat"

input "myfile.dat"   value1 value2 value3
input  filin         value4

node   s1  dalgebra  value1 * 10e-6
...
```

For example, the input file below gives “value1”, “value2” and “value3” in three columns; comment lines are allowed, blank lines are not:

```
#  this is a first comment
   1.0e-6          1.230        5
    2.0e-6         1.240        4         // modified by JDoe
    3.0e-6         1.250        6
#  and another comment
    4.0e-6         1.270        5
   5.0e-6          1.280        2
```

LOCATION INDEPENDENT INSTRUCTION

This instruction is *not* segment dependent.

---

cell_interface, data_interface

cell_interface < $node | $variable | $parameter...>

data_interface < $variable | $parameter...>

These instructions are particular to a cell or a data cell. These keywords must be the first instruction of the file (preferably on its first line). See the node “*cell*” and the instruction “*data*” for details.

---

interlude

You can include an interlude between the successive executions of a tool (optional):

interlude <range>

interlude <min> <max>

interlude <min> <max> <res>

The range (long) and the min and max numbers must be non-negative CONSTANT numbers (long). Default value is 0. The res number is the resolution of the values produced by the instruction.

The execution of the tools is separated by 1000 loops:

```
interlude 1000
```

The execution of the tools is separated by a random number of loops between 1000 and 1020:

```
interlude 1000  1020
```

The execution of the tools is separated by a random number of loops, multiple of 5, between 1000 and 1020:

```
interlude 1000  1020 5
```

LOCATION INDEPENDENT INSTRUCTION

This instruction is a declaration which can be located anywhere in the netlist.

interpolate

Keyword "*interpolate*" introduces a new segment in the netlist. Nodes and variables are processed by segment (see chapter describing the concept of ***NAPA***). An interpolation is specified by the keyword "interpolate" and the interpolation factor:

interpolate [fs] <interpolation_factor>

Interpolation factor must be a strictly positive integer CONSTANT. No variable is allowed.

NAPA computes the local sampling frequency determined by the interpolate factor. Anything that follows the "*interpolate*" line will be evaluated at the interpolated rate. See also appendix A, page 155, for more information. Interpolation factor is relative to the previous segment rate, except when the keyword "fs" is added: it then refers to the absolute sampling frequency defined by the instruction "fs".

For example, interpolation by 8:

```
#* segment running at sampling frequency fs
...
interpolate  8
...
#* segment running at sampling frequency 8*fs
```

Another example:

```
fs      3.0e6
...

#* segment running at sampling frequency 3MHz
...
decimate    3
...
#* segment running at sampling frequency 1MHz
...
interpolate fs  5
...
#* segment running at sampling frequency 15MHz
```

LOCATION DEPENDENT INSTRUCTION

This instruction modifies deeply the behavior of the simulation. Be careful to place this instruction at the appropriate location in the netlist.

ivar

see paragraph: dvar, ivar.

load

This instruction should be used only for debugging purposes. We will not describe the format of the file to load: it is usually the output of a "*dump*" instruction (***ANSI-C*** code).

The instruction "load" loads the file corresponding to a "*dump*" from a previous simulation. The loaded values override the default initial values of ***NAPA*** at initialization and after each "*restart*". Format is identical to the format of file produced by "*dump*". Only one "*load*" is authorized inside a ***NAPA*** netlist.

Restarting a complete simulation using "*load*" and "*dump*" is possible. Nevertheless, note that neither "dump" nor "load" saves or restores the internal states of user-defined functions and tools, or of the pseudo-random generator.

load <"file_name">

```
load "foo.dmp"
```

LOCATION INDEPENDENT INSTRUCTION

The usage of "*load*" is directly linked to the instruction "*dump*" and should be limited to very specific situations.

This instruction can be located anywhere in the netlist.

napa_version

napa_version <version_id>

This is intended to be a compatibility check. The transpiler stops with an error if the declared version is older than its own.

With the current version:

```
...
napa_version   V5.03
...
```

node

node <nod_nam> <kind> [<parm_if_any...>] [<nod_if_any...>]

node void <kind> [<parm_if_any...>] [<nod_if_any...>]

node (<nod_nam>) <kind> [<parm_if_any...>] [<nod_if_any...>]

A ***NAPA*** node is always the output of a single object. Node and object cannot be dissociated. The ***NAPA*** node is the primitive element of the ***NAPA*** netlist. See chapter presenting the ***NAPA*** concept, for more information.

Each node must be defined once and *only* once. Self-reference is not allowed.

Unused nodes cause a warning. To prevent the warning, use the specific identifier "*void*" as a dummy signal. This identifier "*void*" can be used many times; there is no risk of creating unwanted connections. Another way to avoid such a warning is to place the output node between parentheses; the transpiler will not check the use of the node.

Undetermined nodes cause an error. Undefined inputs or internal loop containing no delay element can leave a node undetermined. As this propagates in the netlist, one undetermined node can leave many others undetermined.

A node and a variable cannot share the same name. A reserved keyword cannot be used as node name.

The nodes are initialized by default to "ANALOG_INI" or "DIGITAL_INI" (values predefined resp. as 0.0 and 0). Nodes can also be initialized using instruction "*init*". In this case, their initialization may depend on variable definitions and their respective positions in the netlist.

Instructions "decimate", "interpolate", "nominal" and "drop" are dividing the ***NAPA*** netlist in separate segments. Segments are sorted separately. Nodes of a segment are not mixed with the nodes of another segment.

NAPA sorts nodes in such a way that the data flow is respected. ***NAPA*** introduces no implicit delay.

NAPA differently handles nodes and user's variables:

NODES "NODE" AND VARIABLES "DVAR", "IVAR" ARE SORTED SEPARATELY BY *NAPA*.
VARIABLES "STRING" ARE NOT SORTED.
NODES ARE UPDATED AFTER VARIABLES IN THE SIMULATION LOOP.

Compare variables to nodes, and see also appendix A, page 155, for more information.

LOCATION DEPENDENT INSTRUCTION

nominal

Keyword "*nominal*" introduces a new segment in the netlist. Nodes and variables are processed by segment (see chapter describing the concept of ***NAPA***). Please note that the keyword "fs" is mandatory.

nominal fs

The following segment runs at the global sampling frequency specified by the instruction "*fs*".

```
fs  8.0e6
...
# segment running at sampling frequency 8.0MHz
...
decimate 4
...
#* segment running at sampling frequency 2.0MHz
...
nominal fs
...
#* segment running at sampling frequency 8.0MHz
```

LOCATION DEPENDENT INSTRUCTION

This instruction modifies deeply the behavior of the simulation. Be careful to place this instruction at the appropriate location in the netlist.

num_initial

DEPRECATED, use “interlude” instruction.

opcode

Opcode is used to describe the ALUs (see node “*alu*”).

opcode <alu_name> <opcode_number> [<function_template>]

Several “opcode” instructions are needed to describe a single ALU. Each line contains an opcode number and its corresponding function. Template uses dummy operands (slots #1, #2, ...).

The number of dummy operands must correspond to the number of inputs of the “alu” node. An empty template corresponds to no operation (hold).

```
opcode  myalu 0                 // hold output, no operation
opcode  myalu 1  #1             // register 1 to output
opcode  myalu 2  #2
opcode  myalu 3  #1 + #2        // sum of register 1 and 2
opcode  myalu 4  #1 - #2
opcode  myalu 5  - #1
opcode  myalu 6  shiftr(#1)
opcode  myalu 7  shiftl(#1)
```

In the example above, it is assumed that “shiftr ()” and “shiftl ()” have been defined as ***ANSI-C*** functions in some header called in the netlist.

LOCATION INDEPENDENT INSTRUCTION

This instruction can be located anywhere in the netlist.

output

Output to file or standard output can be specified by the keyword "*output*" followed by the file name, a variable of type "string" or a stream name between double quotes, followed by the names of the nodes or variables to be output. Several output instructions can coexist inside the same ***NAPA*** netlist if they are outputting into different streams or files. At run-time, the potential collisions between I/O streams are checked by the simulator through a dedicated ***C*** function registering the I/Os.

This instruction may be conditioned by an event using the keyword "*when*" followed by an event variable.

It is *very* important to understand that the output rate is computed from the sampling frequencies of the elements and *not* from the location of this instruction in such a way that this instruction does not introduce any hidden subsampling. A list made only of indexes and times (LOOP_INDEX, TIME...) is written at each step; a list made only of constants is written once, at the start of the simulation, unless the instruction is conditioned by an event.

output <"fil_nam"> <nod_nam | var_nam...>

output <mystring> <nod_nam | var_nam...>

output "stdout" <nod_nam | var_nam...>

output "stderr" <nod_nam | var_nam...>

output <"fil_nam"> <nod_nam | var_nam...> when <event_condition>

output <mystring> <nod_nam | var_nam...> when <event_condition>

output "stdout" <nod_nam | var_nam...> when <event_condition>

output "stderr" <nod_nam | var_nam...> when <event_condition>

<event_condition> is a Boolean combination of events defined by instruction "*event*".

Real type output may be scaled individually by a suffix:

(y)	yocto,	10^{-24}
(z)	zepto,	10^{-21}
(a)	atto,	10^{-18}
(f)	femto,	10^{-15}
(p)	pico,	10^{-12}
(n)	nano,	10^{-09}
(u)	micro,	10^{-06}
(m)	milli,	10^{-03}

(k)	kilo,	10^{+03}
(M)	mega,	10^{+06}
(G)	giga,	10^{+09}
(T)	tera,	10^{+12}
(P)	peta,	10^{+15}
(E)	exa,	10^{+18}
(Z)	zetta,	10^{+21}
(Y)	yotta,	10^{+24}
(%)	per cent,	100.0

Integer type output may be configured individually by a suffix:

(x) or **(X)** hexadecimal representation

NB: the binary prefixes[2] (kibi, mebi, gibi, tebi, pebi, exbi, zebi and yobi) are currently not supported.

It is possible to add units after the scaling prefix provided prefix and unit are separated by '_'. The suffix follows the name without blank: a1(k_Volt), not a1 (k_Volt). It applies to analog outputs, (x) and (X) to digital outputs; a string, an index or a time is output without suffix.

For example, the following netlist writes four outputs: the global variable "LOOP_INDEX" on the standard output, nodes "a1", "a2" and "a3" in the file "somefile.dat", node "d2" on the standard error, and variables and nodes in the file "/home/jdoe/NAPA/project1/mysim.out".

```
...
string filout   "somefile.dat"
string filerr   "stderr"

dvar   ampl1      1000.0
dvar   ampl2         1.0
dvar   ampl3         1.0e-6

node   a1 sine       0.0   ampl1  1000.0  0.0
node   a2 cosine     1.0   ampl2  1234.0  0.0
node   a3 triangle   0.0   ampl3  4321.0  0.0
node   d2 comp a1    0.0

output  "stdout"  LOOP_INDEX
output  filout    a1(k_Volt)  a2  a3(u)
output  filerr    d2
output  "/home/jdoe/NAPA/project1/mysim.out" ...
        LOOP_INDEX ampl1..3 a1..3 d2

terminate  LOOP_INDEX > 1000LL
```

2 Prefixes of standard IEC 60027-2 (2000-11)

Another example with output conditioned by an event:

```
...
event  ev1         LOOP_INDEX > 100LL
event  ev2  (new) (LOOP_INDEX > 200LL)

dvar   ampl  1.0

node   a1 sine    0.0  ampl  1000.0  0.0
node   a2 cosine  0.0  ampl  1234.0  0.0
node   d2 comp    a1 0.0

output "stdout"    LOOP_INDEX a1..2 d2  when ev1

output "once.out"  LOOP_INDEX a1..2 d2  when ev2

terminate  LOOP_INDEX > 1000LL
```

Output formats may be changed using instruction *"format".*

Output files begin with a banner of fifteen lines. This banner contains the necessary information to document the data produced by the simulation. Output data format is compatible with graphics packages like ***Gnuplot***[3]. The output banner can be removed partially or totally by defining directive "*NO_BANNER*" or "*STRICTLY_NO_BANNER*". Of course, directive "*STRICTLY_NO_BANNER*" implies directive "*NO_BANNER*". Another directive "*NO_TIME_OUTPUT*" suppresses the first column of the output.

```
directive  NO_BANNER


directive  STRICTLY_NO_BANNER


directive  NO_TIME_OUTPUT
```

The banner of fifteen lines was chosen as a standard for all user-defined functions or tools outputting to file. These directives may condition the existence of the banner for these functions. However this is under the responsibility of the writer of these user's functions.

Please note that keyword instruction "interlude" has no effect on data produced by "*output*".

NOTE:

This instruction synchronizes automatically with the nodes "itool" and "dtool". When such tools are instantiated in the netlist, this instruction no longer refers to the absolute time of the simulation but to a time relative to the beginning of each packet of data, i.e. to the times when the macro "*TOOL_INDEX*" changes.

CAUTION:
RAM or ROM content cannot be output.

LOCATION INDEPENDENT INSTRUCTION

3 ***Gnuplot*** is a multi-platform graphics package available as freeware.

ping

This instruction 'pings' the functions which are used in the ***NAPA*** netlist. If a function has the appropriate code, the simulator will display the file where this function has been declared. Valid for ***C*** functions or macros, user functions and tools defined in header files.

Ping output file can be specified by a file name, a variable of type "string" or a stream name between double quotes. Default is "stderr".

ping <"file_name">

ping <myvar>

ping ["stderr"]

It is expected, but it is not mandatory, that a user-defined function is accompanied by two lines which will trigger a response to the ping command:

```
 ...
#define myfunction_IS_REGISTERED
PING(myfunction);
```

LOCATION INDEPENDENT INSTRUCTION

This instruction is a declaration which can be located anywhere in the netlist.

post

This instruction allows postprocessing the resulting file of a time domain output (instruction "output") or of a user-defined tool (node "itool" or "dtool"). Please note that standard IO stream "stdout" and "stderr" cannot be postprocessed. This instruction must directly follow the tool or the output to be postprocessed. See "User-Defined Functions and Tools".

post <function_id> <"file_name"> [< parameters...>]

post <label> <function_id> <"file_name"> [< parameters...>]

post <label> void <"file_name">

Without label or with labels all different, the postprocessing refers to the previous output or tool analysis. A sequence of postprocesses with the same label cascades the postprocessing. A label is a strictly positive integer.

See the note below concerning the usage of identifier "void" with "post".

For example, the output of the TSNR analysis will be postprocessed by a function "histo". Histogram and statistics of the TSNR will be computed and results stored in files "tsnr2.out" and "tsnr3.out". Another postprocess, "*margins*", reads the output of tool "*tf*" applied to a loop gain and gives its gain and phase margins. Three general postprocesses complete them: "*compare*" checks an output against a reference file, column by column, within a tolerance; "*csv*" exports it for a spreadsheet; "*resample*" brings it on a regular grid of its abscissa.

```
...
node  a noise  0.0 1.0
node  void itool tsnr "tsnr1.out" a 1.0  100000

post  histo "tsnr2.out" 4                  // process "tsnr1.out"
post  stat  "tsnr3.out" 4                  // process "tsnr1.out"
...
terminate TOOL_INDEX > 100
```

Another example, where output is postprocessed and postprocessings cascaded (please note the labels):

```
...
node  a noise  0.0  0.1
node  b dalgebra    a*a

output  "file1.out"  a  b

post  1  select  "file2.out" 1  0.0 0.3     // process "file1.out"
post  1  stat    "file3.out" 2              // process "file2.out"
post  2  histo   "file4.out" 2              // process "file1.out"

terminate TIME > 1.0
```

Use the identifier “void” to change the file where the postprocessor is pointing:

```
...
node  a noise  0.0  0.1
node  b dalgebra    a*a

output  "file1.out" a  b

post  1  select  "file2.out" 1  0.0 0.3     // process "file1.out"
post  1  stat    "file3.out" 2              // process "file2.out"
post  2  void    "file2.out"                // refer to "file2.out"
post  2  histo   "file4.out" 2              // process "file2.out"

terminate 1.0 < TIME
```

There is another way to use the instruction “post” using the instruction “void”:

```
...
void  "file1.out"                          // to indicate a file
post   select  "file2.out" 1  0.0 0.3      // process "file1.out"
post   stat    "file3.out" 2               // process "file2.out"
...
```

This method is useful to postprocess a file created by another ***NAPA*** run. “post” accepts optional qualifiers. See (option) in chapter Instruction Qualifiers.

It is also interesting to note that files processed by a postprocess are often discarded. Therefore a unique file ID is recommended. Use instruction “string” without definition to create these files.

LOCATION DEPENDENT INSTRUCTION

random_seed

random_seed <[- | +] long_int_num>

By default, the internal random generator uses a seed chosen at transpilation time. This seed is written in the ***C*** code and can be reused in further simulations. To reuse a seed, use the instruction "*random_seed* <num>" where <num> is the seed to introduce. The new simulation then repeats the previous one exactly.

A negative seed produces a stream of antithetic pseudo-random numbers.

In some special situations, you will need to produce a seed at the simulation time (when you give the executable to a third party for instance), use "random_seed 0". If a "*command_line*" instruction is instantiated, this is done by default.

```
random_seed 123456
```

CAUTION:
If you write your own pseudo-random number generator, your code should check the value of the macro "ANTITHETIC". It is set to 'TRUE' when an antithetic stream is requested, 'FALSE' otherwise. Write your code accordingly.

LOCATION INDEPENDENT INSTRUCTION

restart

You can ask for a reset of the nodes and user-defined objects:

restart

Calls the reset function of every user-defined function ("duser", "iuser") and resets the ***NAPA*** nodes. The values loaded by "load" (not those of "input") are loaded again.

CAUTION:

NAPA variables and tools "*itool*" and "*dtool*" are *not* reset. Restart is generally not a safe task and should be reserved to very specific situations.

LOCATION DEPENDENT INSTRUCTION

string

string <str_name> [<"value" >]

Where "*string*" stands for string of characters, i.e. a set of characters terminated by character '\0'.

Please note that there is a special syntax for strings when they are part of the instruction "*command_line*". See this instruction for this special case.

This instruction declares and initializes strings at the same time. The value of a "s*tring*" accepts *only* string constant or indirection. Self-reference is not allowed. The value is not mandatory: default value is the empty string **""**. **The *NAPA* transpiler does *not* sort strings!** Strings cannot be updated.

Indirection is possible: "*string*" may be defined by a string containing one or several identifiers of previously defined variables ("*ivar*", "*dvar*" or "*string*" or sampling frequency *"fs"* and period *"ts"* and *"directive"*) preceded by the character '#'. Indirection will be resolved during the transpilation of the netlist and will not be reevaluated by the simulator during run-time. The processing of the indirection using "*ivar*" or "*dvar*" will be limited to copy the value of the variable, no mathematical evaluation will be performed:

```
...
string  unit "Hz"
dvar    fstart    1000.0
dvar    fstop     10.0*fstart
string  str1 "start frequency is #fstart #unit"
string  str2 "stop  frequency is #fstop  #unit"
...
```

In the example above, the string variables "str1" and "str2" will contain respectively:

"`start frequency is 1000.0 Hz`"
"`stop  frequency is 10.0*fstart Hz`"

A string variable named "Space" is special in "output": its name is replaced by blanks in the column header and its value is written as is. It can serve as a column separator, for example string Space ",". It cannot be an "ivar" or a "dvar".

>>> An empty definition creates a unique ID, used for instance, in the creation of a unique filename as in the example below.

```
...
string     aname
tool  fft  aname sigout 1.0 8.0e3 65536
post  sort "stdout"  4 (down)
...
```

In the example above, a unique filename will be created by the tool "fft". For instance, "YL2431498947_0.out", as a tool adds automatically the suffix '.out' to file name if needed. Here this file is discarded after the execution of the postprocessing "sort".

LOCATION DEPENDENT INSTRUCTION

---

stuck

For fault modeling, it could be necessary to reproduce a fault. It is possible to make a node stuck at a value. The ***C*** expression replaces the definition of the node. Initialization is not affected.

This special instruction must be placed in the MAIN netlist file only. It should be used only in fault modeling and should not be used for other purposes.

stuck <nod_nam> <*C*_expression_returning_a_number>

LOCATION DEPENDENT INSTRUCTION

CAUTION:

C expression is automatically cast to the original type of the node (digital or analog). This expression could contain nodes or variables. In this case, ***NAPA*** will reorganize the ***C*** code to handle these new inputs, the resulting netlist must be a valid ***NAPA*** netlist (loop without delay will cause undetermination).

synchronize

When several tools are used in the same simulation, the synchronizable ones are synchronized by default. Synchronization can be enabled (default) or disabled.

synchronize [(yes)]

synchronize (no)

NAPA automatically builds a synchronization mechanism. This mechanism is based on a mailbox and handshaking. ***NAPA*** and the tools exchange synchronization messages. The instruction "*synchronize (no)*" disables the synchronization mechanism. This is not generally recommended.

LOCATION INDEPENDENT INSTRUCTION

This instruction is a declaration which can be located anywhere in the netlist.

terminate

A netlist states when the simulation stops:

terminate [<*C*_Boolean_expression>]

NAPA simulation stops at the end of the corresponding loop. Only one "*terminate*" instruction may appear in a netlist. The Boolean ***C*** expression determines the termination of the simulation (value 'TRUE'). This instruction can be placed anywhere in the netlist.

Typical termination condition uses macro "LOOP_INDEX" or "TIME". The macro "LOOP_INDEX" counts the number of loops the simulator has already performed. Another possible termination condition uses the macro "TOOL_INDEX". This macro counts the number of sets of tasks the analysis tools have done. Do not use the relative values "REL_LOOP_INDEX" or "REL_TIME" here: they are periodically reset to zero.

```
...
terminate  TIME >= 0.001


...
terminate  LOOP_INDEX >= 1000000LL


...
terminate  TOOL_INDEX >= 1
```

CAUTION:

Double-check the termination condition as ***NAPA*** will not introduce another simulation termination for you.

Default value for the condition is “true”. In this case, the simulator does not need to process any loop. The code corresponding to the main loop is therefore not built.

LOCATION INDEPENDENT INSTRUCTION

The evaluation of the terminate expression does not depend on the segment where this instruction is located. It is evaluated at each cycle of the main loop.

title

The title of the simulation is set (optionally) by:

title <“some one-line text between double quotes”>

For instance:

```
title "Hello world!"
```

User-defined functions or tools to document simulation output can use the title through macros “TITLE” and “SHORT_TITLE”. Output driven by instruction “*output*” uses the title as automatic documentation of the output file. Multiple “*title*” strings are concatenated in a single string. A “title” without string gives an empty title; without any “title”, the title is the name of the input file without extension.

“*title*” is a special string variable. Indirection is possible (see “*string*”). It is not necessary to place the “*title*” after the definition of the variables concerned by the indirection. It will be processed after all the variables have been defined and their possible indirections resolved.

TIPS:

Use “*title*” in a cell to complete the main title if necessary, for example the circuit netlist:

```
title "TSNR analysis"
...
node out cell  dut "/home/jdoe/NAPA/project1/a2d_desc.1" in
node ctr itool tsnr "stdout" ...
...
```

tool

This instruction is expanded into a node “itool”:

For instance:

```
tool  tf  "stdout"  in 1.0 out 1.0 npts
```

is expanded and is equivalent to:

```
node  void itool  tf  "stdout"  in 1.0 out 1.0 npts
```

Refer to the node kind itool for further explanation.

LOCATION DEPENDENT INSTRUCTION

ts

The sampling period is set (optionally) by:

ts [<sampling_period>]

The sampling period must be a positive double precision CONSTANT number. This is the main sampling period of the simulation. Use it instead of "fs". See also instruction *"fs".*

```
ts  1.0e-6
```

Default value is 1.0, with a warning when neither "fs" nor "ts" is given.

```
ts
```

CAUTION:

Sine wave generators and similar node kinds are using the sampling frequency to compute the signal. A common mistake is to forget to define "*ts*". In this situation, ***NAPA*** uses the default value, the simulator risks processing sine or cosine of huge numbers, losing accuracy or simply overflowing.

LOCATION INDEPENDENT INSTRUCTION

This instruction is a declaration which can be located anywhere in the netlist.

update

Variables "*ivar*", "*dvar*" but not "*string*" can be updated during execution. It may be conditioned by an event using the keyword "*when*" followed by an event variable.

Syntax:

update <var_nam> [<*C*_expression>]

update <var_nam> [<*C*_expression>] when <event_condition>

<event_condition> is a Boolean combination of events defined by instruction "*event".*

If no expression is given, "*update*" will reuse the exact expression given during the corresponding variable definition at each execution of the segment. An error is detected if a variable is updated but not declared (use "*ivar*", "*dvar*" to declare these variables). It is allowed to update variables using node values. Variables are always updated before nodes. It is important to note that 'updates' are sorted. Self-reference is allowed in update expression.

Instructions “*decimate*”, “*interpolate*”, “*nominal*” and “*drop*” are dividing the ***NAPA*** netlist in separate segments. Segments are sorted separately. Nodes and variables of a segment are not mixed with the nodes and user's variables of another segment.

In the following example, the variables “n” and “freq” are updated at each simulation loop. As no value is introduced, the update of variable “freq” uses the formula contained in the variable definition:

```
...

ivar  n            0
dvar  fstart  1000.0
dvar  fstop  10000.0
dvar  fstep     10.0
dvar  freq    fstart + (fstep * (R_TYPE) n)
...
node s0 sine 0.0 1.0 freq 0.0
...
update n         n+1
update freq
...
terminate freq > fstop
```

In the next example, the same variables are updated by an event, every 1000 loops after 10 µs:

```
...
event ev    ((LOOP_INDEX % 1000LL) == 0LL) && (TIME > 1.0e-5)
...
ivar  n            0
dvar  fstart  1000.0
dvar  fstop  10000.0
dvar  fstep     10.0
dvar  freq    fstart + (fstep * (R_TYPE) n)
...
node s0 sine 0.0 1.0 freq 0.0
...
update n         n+1   when ev
update freq            when ev
...
terminate freq > fstop
```

In the next example, the same variables are updated once every 1000 loops:

```
...
ivar  n            0
dvar  fstart  1000.0
dvar  fstop  10000.0
dvar  fstep     10.0
dvar  freq    fstart + (fstep * (R_TYPE) n)
...
node s0 sine 0.0 1.0 freq 0.0
...
decimate 1000                // decimation by 1000
...
```

```
update n         n+1
update freq
...
terminate freq > fstop
```

It is also possible to update a variable by adding “&update” to the definition of the variable, in this case it is not possible to change the definition:

File “/mysimu.nap”

```
...
dvar     myvar1   log(TIME)     &update
...
```

See also appendix A, page 155, for more information about the simulation flow.

CAUTION:

A common mistake is to believe that a variable update will take the latest update expression by default. The default value is *always* the variable definition. There is no exception. Another common mistake is to believe that an “update” placed after a node in the same segment is processed after it: variables are always updated before nodes.

LOCATION DEPENDENT INSTRUCTION

void

This instruction is used to precede one or more “post” instructions. It sets the name of the file processed by the following postprocessors. See the “post” instruction.

void <file_name>

```
...
void    "file1.out"
post    stat   "file2.out" 4           // analyze file "file1.out"
post    select "file3.out" 6 0.0 1.0   // analyze file "file1.out"
...
```

LOCATION DEPENDENT INSTRUCTION

warning

This instruction publishes a warning during the parsing of the netlist and writes a message on the “stderr” output.

warning “<some_message>”

(comment)

A line beginning with “**#**” is a comment for the NAPA transpiler. The standard ***NAPA*** flow always runs the macro preprocessor ***MAC*** first, and ***MAC*** issues a warning for such a line: use “**#***” or “**##**” to obtain a comment compatible with both languages. Blank lines are allowed.

 # any one-line text

For example:

```
...
#* this is a whole line of comment
#* a comment does not appear inside the generated C code
#* a comment can contain anything, why not @#$%& ?
...
```

✘ NOTES:

Text after “//” is ignored by the ***NAPA*** transpiler. Do not use this right-hand comment in a ***MAC*** preprocessor directive. ***C***-like comments (/* ... */) are not supported by ***NAPA***.

Instruction Qualifiers

*Qualifiers are reserved keywords used to modify the behavior of some of the **NAPA** instructions.*

after, before

If an "*algebra*", "*dalgebra*", "*ialgebra*", "*test*", "*itool*", "*dtool*", "*iuser*" or "*duser*" node depends on another node being evaluated before it is evaluated itself, but not explicitly in the netlist, the ***NAPA*** transpiler has no hint to sort these nodes properly. You can specify "*after* <node>" or "*before* <node>". This guarantees that the node is properly sorted. With "before", the target node must also be an "*algebra*", "*dalgebra*", "*ialgebra*", "*test*", "*itool*", "*dtool*", "*iuser*" or "*duser*" node.

CAUTION: "*after*" and "*before*" should be used sparingly.

The next example is a typical example of bad programming practices:

Say you have a user-defined function that calculates the average and RMS error of all the values of a storage array. A user-defined function can only return one value, so the other value could be stored in a global variable and then accessed with a "*dalgebra"* statement. You must however make sure to access it after the function is called. In a header file "my_stat.hdr":

```
#ifndef __MYSTAT_HDR__
#define __MYSTAT_HDR__

double rms_error;

/* ** FUNCTION PROTOTYPE ********************************** */

double duser_average_00(int);

/* ** FUNCTION DEFINITION ********************************* */

double duser_average_00(int id) {
  long long i;
  double    x;
  double    y  = 0.0;
  double    y2 = 0.0;
  for (i = 0LL; i < LOOP_INDEX; i++) {
    x   = store_array[i];
    y  += x;
    y2 += x * x;
  }
  y  /= (double) LOOP_INDEX;
```

```
  y2 /= (double) LOOP_INDEX;
  rms_error = sqrt(y2 - y*y);
  return y;
}
void  init_duser_average_00(int id) { return; }
void check_duser_average_00(int id) { return; }
void reset_duser_average_00(int id) { return; }
void close_duser_average_00(int id) { return; }

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

#endif                                          /* __MYSTAT_HDR__ */
```

In the ***NAPA*** netlist file:

```
header <napa.hdr>
header "my_stat.hdr"
...
node avrg duser     average
node rms  dalgebra  after avrg  rms_error
...
```

Now it is certain that node “rms” will be defined and updated AFTER node “avrg” in the ***C*** code produced by the ***NAPA*** transpiler.

See also “*with*” qualifier.

when

Qualifier “*when*” is used in instructions “assert”, “dump”, “output” and “update” to condition the execution of the instruction by an event or a Boolean combination of events. See instruction “event”.

with

The keyword “*with*” may be applied only to nodes. It relocates the node in the segment of the node pointed by this keyword. This is very practical when the analysis of the simulation is grouped in a single file. It guarantees that both nodes are located in the same segment, i.e. running under the same sampling conditions (same sampling rate and same sampling offset). It could make the simulation netlist more readable or more modular.

These ***NAPA*** netlist files are equivalent:

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

fs        1.0e6
```

```
...
node a    sine   0.0 1.0 1234.5 0.0
node void itool fft "file1.out"  a 1.0 1000000          // run @ 1 MHz

interpolate 4

node b    cosine 0.0 1.0 5432.1 0.0
node void itool fft "file2.out"  b 1.0 1000000          // run @ 4 MHz
...

terminate TOOL_INDEX >= 1


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

fs        1.0e6

...
node a    sine   0.0 1.0 1234.5 0.0

interpolate 4

node b    cosine 0.0 1.0 5432.1 0.0

...
node void itool fft "file1.out" a 1.0 1000000  with a    // run @ 1 MHz
node void itool fft "file2.out" b 1.0 1000000            // run @ 4 MHz
...

terminate TOOL_INDEX >= 1
```

See also "*after*" and "before" qualifiers.

(expand) | (noexpand)

These qualifiers are used to control the header expansion in the generated ***C*** code. See instruction "*header*" for more information.

The alternative spellings are expand$ and noexpand$ respectively.

(negative) | (positive) | (dual) | (both)

These qualifiers are used in the definition of the node kinds "trig" and "change" to qualify the trigger type; (both) is accepted by "change" only.

The alternative spellings are negative$, positive$ and dual$ respectively.

(no) | (yes)

Qualifiers (*no)* and (*yes)* are used in instruction "*synchronize*".

(nocheck)

Qualifier (*nocheck)* is used in nodes "*div*" and "*mod*".

(hex)

Qualifier "(*hex*)" is used in instruction "*array*" to declare the array type. It is internally equivalent to "(digital)" but indicates to the ***NAPA*** transpiler that the initialization file of the array contains hexadecimal data (addresses remaining digital type).

It is also used in instruction "*format*".

(digital)

This qualifier is used:

1. In the definition of the node kinds "const" and "dc" to force casting.
2. In instruction "*array*" to declare the ROM or RAM type.
3. In an instruction "*declare*".
4. In an instruction "*format".*

(analog)

This qualifier is used:

1. In the definition of the node kinds "const" and "dc" to force casting.
2. In instruction "*array*" to declare the ROM or RAM array types.
3. In an instruction "*declare*".
4. In an instruction "*format".*

(string)

This qualifier is used in an instruction "*declare*". This is the only usage of this qualifier.

(constant)

This qualifier is used in an instruction "*declare*". This is the only usage of this qualifier.

(true)

This qualifier is used in an instruction "*declare*". This is the only usage of this qualifier.

(arithmetic) | (geometric) | (harmonic) | (rms)

These qualifiers are used in the definition of node kind “average”.

The alternative spellings are arithmetic$, geometric$, harmonic$ and rms$ respectively.

(pointer)

Qualifier *“(pointer)”* is used in instruction “*array*” to declare the POINTER array type. This is the only usage of this qualifier.

(new)

This qualifier is used to modify the definition in instruction “*event*”.

(<option>[::<parm>])

It is possible to define options for the nodes “duser”, “iuser”, “dtool” and “itool” and for the instruction “*post*”. The ***NAPA*** transpiler automatically builds the macros that query the existence of one or several options in the instantiation of such functions. The writer of a user/tool/post function may call the ***NAPA*** macros “ISOPTION( )” and/or “ISNOTOPTION( )” like in this example where itool “wonder” has two options: (do) and (do_not):

File “myfile1.nap”

```
...
node void itool  wonder "fila.out"  s1 1.0 2500.0   (do)
...
node void itool  wonder "filb.out"  s4 1.0 5000.0
...
int itool_wonder_opt[WONDERMAX];             /* options of itool ‘wonder’ */
...
void check_itool_wonder(... , int id) {
  if        (ISOPTION("itool_wonder",id,"do"    ) == TRUE) {
    itool_wonder_opt[id] = 1;
  } else if (ISOPTION("itool_wonder",id,"do_not") == TRUE) {
    itool_wonder_opt[id] = 0;
  } else {
    itool_wonder_opt[id] = 0;                /* default is (do_not)       */
  }
  if        (ISNOTOPTION("itool_wonder",id) == TRUE) {
    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;
}
...
```

The option *(void)* is always ignored. This is useful when an option has to be passed through a cell interface: the cell interface demands a fixed number of parameters, therefore, transmitting a *(void)* option makes it possible to specify that the default option (i.e. no option) is chosen.

A parameter can be added to the option. It must be a constant number (analog or digital):

File “myfile2.nap”

```
...
node void itool  magic "fila.out"  s1 1.0 2500.0   (resolution::10)
...
node void itool  magic "filc.out"  s4 1.0 5000.0   (resolution)
node void itool  magic "fild.out"  s2 1.0 4000.0
...
```

The writer of a user/tool/post function may call the ***NAPA*** macros ISOPTION(f,i,o), ISNOTOPTION(f,i), ISPARAMETER(f,i,o) and GETPARMADDRESS(f,i,o).

Please note that an alternate syntax for *(option)* is *option**$***.

(<real_type_output_scaling>)

These suffixes scale individual analog outputs of the instruction “output”.

(y)	yocto,	10^{-24}
(z)	zepto,	10^{-21}
(a)	atto,	10^{-18}
(f)	femto,	10^{-15}
(p)	pico,	10^{-12}
(n)	nano,	10^{-09}
(u)	micro,	10^{-06}
(m)	milli,	10^{-03}
(k)	kilo,	10^{+03}
(M)	mega,	10^{+06}
(G)	giga,	10^{+09}
(T)	tera,	10^{+12}
(P)	peta,	10^{+15}
(E)	exa,	10^{+18}
(Z)	zetta,	10^{+21}
(Y)	yotta,	10^{+24}
(%)	per cent,	100.0

(<integer_type_output_configuration>)

This suffix configures individual digital outputs of the instruction “output”.

(x) or **(X)** hexadecimal representation

Short Forms

Short forms help to reduce the length of the netlist and to enhance the readability.

&update

It is possible to force the update of a variable directly in its definition. In this case, neither a new definition nor a conditional update is possible.

File “myfile.nap”

```
...
dvar   e   rand_uniform(1.0, s)    &update
dvar   f   rand_uniform(a, 1.0)    &export   &update
...
```

is equivalent to:

File “myfile.nap”

```
...
dvar   e   rand_uniform(1.0, s)
dvar   f   rand_uniform(a, 1.0)    &export
...
update e
update f
...
```

&constant

It is possible to declare a variable constant directly in its definition:

File “myfile.nap”

```
...
ivar  c   16*16      &constant
...
```

is equivalent to:

File “myfile.nap”

```
...
ivar    c   16*16
...
declare (constant) c
...
```

&export

It is possible to export a variable directly in its definition:

```
...
dvar   e   rand_uniform(1.0, s)   &export
dvar   f   rand_uniform(a, 1.0)   &export  &update
...
```

is equivalent to:

File “myfile.nap”

```
...
dvar   e   rand_uniform(1.0, s)
dvar   f   rand_uniform(a, 1.0)   &update
...
export e  f
...
```

&delayed

It is possible to delay any node (except a “delay” node) by one loop of its segment with this short form:

File “myfile.nap”

```
...
node   a  comp  x  y             &delayed
...
```

is equivalent to:

File “myfile.nap”

```
...
node   s  comp  x  y
node   a  delay  s
...
```

Use “init” instruction to perform the initialization if necessary.

Nodes “duser”, “iuser”, “dtool” and “itool” receive a flag, so that their functions can check that they are delayed (macro ISDELAYED).

Special symbols

*Some symbols have a special meaning in **NAPA**.*

$ …

Symbol "$" is used in a cell or data file to flag local variables or nodes.

./ …

Symbol "**./**" in a string refers to the ***NAPA*** file system. It indicates that the pathname is relative to the directory containing the calling cell or data cell (see the ***NAPA*** file system).

~/ …

Symbol "~/" in a string refers to the ***NAPA*** file system. It indicates that the pathname is relative to the directory containing the main netlist (see the ***NAPA*** file system).

< … >

Angle brackets are used in several contexts:

1. To mark the pathname of reusables (see the ***NAPA*** file system). The library pathname is given on the command line of the transpiler. Angle brackets around a file pathname are used to indicate that the file is located in the generic library itself.
2. To declare a width limited register (width casting corresponding to unsigned numbers).

(…)

Besides their use in regular ***C*** expressions, parentheses are used to declare a width limited register (width casting corresponding to signed numbers).

They are used to indicate that a variable or a node could be unused without creating any warning message. They are also part of some qualifiers like (analog), (digital), (hex)...

...

Symbol "#" is used in several contexts:

1. Comment. The ***NAPA*** transpiler ignores all lines beginning with this symbol.
2. Indirection operator in "*string"* or "*title*" definition.
3. In "*opcode*" instruction to flag the dummy operands of the template.

... // ...

Double slashes are used as end of line comment. Any text after "//" is ignored by the ***NAPA transpile***r.

. (one dot)

One dot followed by a number <n> to extract the nth parameter in an array of pointers or in "*ganging*".

.. (two dots)

Two dots are used in iterative identifiers. An iterative identifier is used to replace a list of identifiers.

... (three dots)

Three dots form the continuation mark: they extend an instruction to the next line. Only a right-hand comment may follow them.

:

The bit field extractor is used to access a single bit from a digital node.

::

This operator is used for the 'ganging by name'.

Node Kinds

*Each node kind represents a piece of **C** code tuned to realize a physical element, simple or complex but always written as a customized in-line function.*

adc: N-level signed A/D converter

node <nod_nam> adc <num_lev> <nod_in_nam> <nod_ref_nam>

The input and the reference node must be analog type, and the output node will be digital type. The number of levels must be a constant integer of at least 2. This analog-to-digital converter is signed (for an unsigned converter, see "*uadc*"). A/D output is clipped when the input is outside the input range.

These A/D converters are perfectly symmetrical for an odd number of levels. The A/D converters with an even number of levels are built to correspond exactly to A/D having one level more, but with the upper level missing.

If "y" is the digitized value corresponding to node "x" digitized with 4 allowable levels:

```
node ref dc  1.0
node y   adc 4 x ref
```

If "y" is the digitized value corresponding to node "x" digitized with 5 allowable levels:

```
node ref dc 1.0
node y adc 5 x ref
```

The transfer functions of these A/D converters are:

input A/D 5 levels	digital output	input A/D 4 levels	digital output
-∞ ... -0.75	-2	-∞ ... -0.75	-2
-0.75 ...-0.25	-1	-0.75 ...-0.25	-1
-0.25 ... 0.25	0	-0.25 ... 0.25	0
0.25 ... 0.75	1	0.25 ... ∞	1
0.75 ... ∞	2	NA	NA

algebra: Chameleonic ***C*** expression

This is the most general expression, so ***NAPA*** can check very little in it. Inside the expression, you can mix ***NAPA*** nodes or variables with ***C*** global variables. Node type is determined from the type of the nodes of the ***C*** expression (they must be all analog or all digital).

node <nod_nam> algebra <*C*_expression>

For example, if node "y" is 2 times node "a" plus node "b" then:

```
node y    algebra (2 * a) + b
```

There is one restriction: ***NAPA*** determines the type (analog or digital) of node "y" by the types of "a" and "b" (which must be type consistent), thus you must include at least one node in the algebraic expression from which ***NAPA*** can derive a type.

For example:

```
node pi   algebra 3.14159                    // **** WRONG! ****
```

This is **ILLEGAL** and will be flagged as an error as ***NAPA*** will not know which type to make node "pi".

Instead you should use:

```
node unity dc (analog) 1.0
node pi    algebra (unity * 3.14159)
```

This will result in "pi" being a double. If you tried:

```
node one dc (digital) 1
node pi algebra (one * 3.14159)
```

This would result in "pi" being an integer and thus would have a value of 3!

The simplest way in this case is to force the type by using "*dalgebra*", or in this particular case simply "*dc*"!

```
node pi dalgebra 3.14159


node pi dc (analog) 3.14159
```

CAUTION:

NAPA is unable to track any implicit or explicit casting in an "*algebra*" expression. **Use with care!**

See also "*after*" and "*before*" qualifiers.

alu: User-defined ALU

node <nod_nam> alu <alu_nam> <opcode_num> <input_nod_nam...>

An ALU is described in the templates of "*opcode*" instructions. Node "*alu*" is the instantiation of the ALU in the ***NAPA*** netlist.

Opcode number must be an integer. A run-time error occurs if the current opcode does not correspond to any opcode definition. The ALU name must correspond to the name of at least one opcode instruction. Several "alu" nodes can share the same opcode definitions.

For example:

```
node y  alu  myalu  m  in1 in2
...
opcode  myalu 0                    // nop, hold output
opcode  myalu 1    #1              // register 1 to output
opcode  myalu 2  - #2              // - register 2 to output
opcode  myalu 3    #1 + #2         // sum of registers 1 and 2 to output
...
```

NAPA will determine the type (analog or digital) of node "y" by the types of "in1" and "in2".

CAUTION:

The same "*opcode*" instructions can be shared by an ALU processing floating-point values and an ALU processing integer values. The output type is determined automatically from the input nodes, not by the templates (be careful about possible implicit casting).

For example:

```
...
node n1  dc (digital)  3
node n2  dc (digital)  4
#
node i1 dc (digital)  10
node i2 dc (digital)   9
#
node r1 dc (analog)   10.0
node r2 dc (analog)    9.0
#
node y  alu myalu n1 i1 i2         // y is digital type, value 1
node z  alu myalu n1 r1 r2         // z is analog type,  value 1.111111
node e  alu myalu n2 i1 i2         // e is digital type, value 3

#
opcode  myalu 0  #1 + #2
opcode  myalu 1  #1 - #2
opcode  myalu 2  #1 * #2
opcode  myalu 3  #1 / #2           // opcode currently pointed by 'n1'
opcode  myalu 4  sqrt(#1)          // opcode currently pointed by 'n2'
opcode  myalu 5  #1 * #1
opcode  myalu 6  0
...
```

and: N-input AND element

node <nod_nam> and <nod_nam...>

```
node y and a b c
```

The input nodes *must* be digital type, and output node is always digital type (0 or 1).

average: Average of N inputs

node <nod_nam> average <[- | +] nod_nam...> [(<option>)]

The output node is analog type. Inputs are analog type. This node allows N input nodes. Options are: (arithmetic), (geometric), (harmonic) or (rms). For geometric and harmonic average, negative inputs will be set to 0.0. Default is arithmetic average.

```
node w1  average   x  -y  z
node w2  average   x  -y  z  (harmonic)
```

bshift: Barrel shifter

node <nod_nam> bshift <[- | +] shift_val> <nod_nam>

node <nod_nam> bshift <[- | +] shift_var> <nod_nam>

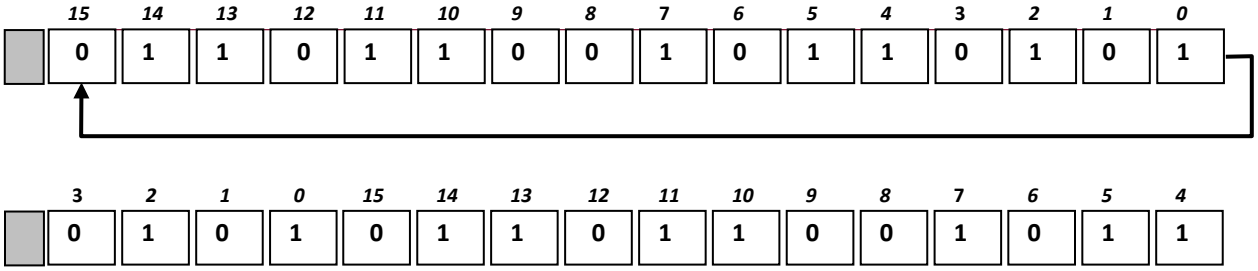
node <nod_nam> bshift <[- | +] shift_nod> <nod_nam>

```
ivar sh4   -4
...
node y bshift 4    x
node z bshift -sh4 x
```

The bits of the input node are rotated, to the right for a positive shift value, to the left for a negative one. In the example above, the values of “y” and “z” are equal to the value of “x” with the bits shifted by 4 positions to the right. Input node and shift value (node, variable or constant) must be digital type. Shift value can be signed. Output is digital type.

The register to shift has the width of the input node, 64 bits by default: in the example below, the input “x” is declared on 16 bits, as the output “n”. The shift value is taken modulo this width (a shift of 70 on a 64-bit register is a shift of 6). The width of the output node, if any, is applied after the rotation.

```
(16) node x ...
(16) node n bshift -4  x
```



btoi: N-bit conversion to unsigned integer

node <nod_nam> btoi <nod_nam | 0 | 1> <nod_nam | 0 | 1...>

```
node y btoi b7 b6 b5 b4 b3 b2 b1 b0
node z btoi  1  1 b5  0  0  0 b1  0
```

Input nodes *must* be digital type (binary value); output node is digital type. Inputs can be nodes or constant 0 or 1 but not variables. They are sorted from MSB to LSB. Node values must be 0 or 1. Other values are accepted but lead to inconsistent results. See also "*itob*".

---

buffer: Non-inverting buffer

node <nod_nam> buffer <nod_nam>

```
node y buffer x
```

The input node *must* be digital type, and output node is always digital type. Output is 0 if input is 0, 1 otherwise. It is therefore *not* equivalent to a *"copy"* node dealing with a digital input!

---

bwand: N-input bitwise AND

node <nod_nam> bwand <nod_nam...>

node <nod_nam> bwand <hexadecimal_constant> <nod_nam...>

```
<8> node a bwand b c d
```

The input node *must* be digital type, and output node is always digital type. Output is the bitwise "and" of the inputs. First input could be either a node or a hexadecimal constant.

To be safe, all bitwise operations should be width limited.

bwbuffer: Bitwise non-inverting buffer

node <nod_nam> bwbuffer <nod_nam>

```
<16> node y bwbuffer x
```

The input node *must* be digital type, and output node is always digital type. Output is a copy of input.

To be safe, all bitwise operations should be width limited.

bwinv: Bitwise inverter

node <nod_nam> bwinv <nod_nam>

```
<12> node y bwinv x
```

The input node *must* be digital type, and output node is always digital type. Output is the bitwise inversion of the input.

To be safe, all bitwise operations should be width limited.

bwnand: N-input bitwise NAND

node <nod_nam> bwnand <nod_nam...>

node <nod_nam> bwnand <hexadecimal_constant> <nod_nam...>

```
<4> node a bwnand b c d
```

The input nodes *must* be digital type, and output node is always digital type. Output is the bitwise "nand" of the inputs. First input could be either a node or a hexadecimal constant.

To be safe, all bitwise operations should be width limited.

bwnor: N-input bitwise NOR

node <nod_nam> bwnor <nod_nam...>

node <nod_nam> bwnor <hexadecimal_constant> <nod_nam...>

```
<9> node a bwnor b c d e
```

The input nodes *must* be digital type, and output node is always digital type. Output is the bitwise "nor" of the inputs. First input could be either a node or a hexadecimal constant.

To be safe, all bitwise operations should be width limited.

bwnot: Bitwise NOT

node <nod_nam> bwnot <nod_nam>

```
<28> node y bwnot x
```

Strictly equivalent to "*bwinv*".

To be safe, all bitwise operations should be width limited.

bwor: N-input bitwise OR

node <nod_nam> bwor <nod_nam...>

node <nod_nam> bwor <hexadecimal_constant> <nod_nam...>

```
<6> node a bwor b c d
```

The input nodes *must* be digital type, and output node is always digital type. Output is the bitwise "or" of the inputs. First input could be either a node or a hexadecimal constant.

To be safe, all bitwise operations should be width limited.

bwxnor: N-input bitwise XNOR

node <nod_nam> bwxnor <nod_nam...>

node <nod_nam> bwxnor <hexadecimal_constant> <nod_nam...>

```
<2> node a bwxnor b c d
```

The input nodes *must* be digital type, and output node is always digital type. Output is the bitwise "xnor" of the inputs. First input could be either a node or a hexadecimal constant.

To be safe, all bitwise operations should be width limited.

bwxor: N-input bitwise XOR

node <nod_nam> bwxor <nod_nam...>

node <nod_nam> bwxor <hexadecimal_constant> <nod_nam...>

```
<3> node a bwxor b c d
```

The input nodes *must* be digital type, and output node is always digital type. Output is the bitwise "xor" of the inputs. First input could be either a node or a hexadecimal constant.

To be safe, all bitwise operations should be width limited.

cell: Subcircuit instantiation from a file

node <nod_nam> cell <inst_nam> <"fil_nam"> <nod_nam | parm...>

Strictly speaking, "cell" is not a node but a pseudo-node: it instantiates a sub-netlist. The sub-netlist is flattened inside the main netlist without any compilation or simulation time penalty. The file "fil_nam" must contain a ***NAPA*** cell netlist. Instantiation name <inst_nam> of the cell must be unique and cannot be reused for another cell. Every cell internal node or variable must begin with character "$". Nodes or parameters not beginning with "$" are considered as global. The first line holds the keyword "cell_interface", then the output node, the optional inputs and the parameters, all prefixed with "$".

Both cell interface and cell formal parameters accept iterative identifiers (see instruction "data").

This pseudo-node "*cell*" does not support the transfer of parameters by name.

A dedicated directory is reserved to store a library of reusable cells. The pathname of this library is indicated to ***NAPA*** through the command line. Using angle brackets "< ... >" is a short form to call a cell contained in this directory. Supposing that the pathname of this library is "/NAPAlib/net", the following expressions are perfectly equivalent:

```
node n3 cell  pls1 <my_cel.net> n1 n2
node n4 cell  pls2 "/NAPAlib/net/my_cel.net" n1 n2
```

The cell instantiation obeys the ***NAPA*** file system. In the following examples, the first call points to a file located in the main directory, the second call points to the current cell directory, the third call points to the root directory, i.e. the working directory from which the ***NAPA*** command was invoked:

```
node n3 cell  pls1 "~/my_cel.net"  n1 n2
node n4 cell  pls2 "./my_cell.net" n1 n2
node n5 cell  pls3 "my_cell.net"   n1 n2
```

Follow ***NAPA*** recommendations, use suffix ".net" for cell netlist (see appendix C page 166).

✖ CAUTION:

1. It is not recommended to use "*decimate*", "*interpolate*", "*nominal*" or "*drop*" instructions inside a cell as they are hidden in cell file but affect the sampling rate of all objects defined after them.

2. Do not forget that your cell should tolerate signed parameters. Use parentheses if possible in the body of the cell (in the example below, we use parentheses with "$parm2" for this reason).

File "ex.net"

```
cell_interface  $out  $in1 $in2  $parm1 $parm2

node $a1  dc   $parm1
node $a2  sum  $in1 $in2  $a1
node $out dalgebra $a2 * sin(_PI_ - ($parm2))
```

File "circuit.nap"

```
node y cell foo1 "ex.net"  a b  2.0 1.0
node z cell foo2 "ex.net"  b -a  p -4.0
...
```

Internal nodes of instantiated cell can be accessed from main netlist using the instance name as prefix: <inst_name>__<nod_nam> (double underscore). It is useful to probe internal nodes ("*output*", "*itool*", "*dtool*"...) but not recommended as it is a tricky way to input or output signals:

In the previous example: foo1__a1, foo1__a2, foo2__a1, foo2__a2

The nodes and parameters given at the instantiation replace those of the interface. Please note that the macro preprocessor ***MAC*** will *never* process these cell internal netlists, as macro preprocessors are not aware of the content of these files.

See also instruction "*data*" and pseudo-node "*generator*".

change: Watchdog

node <nod_nam> change <nod_nam> [(<option>)]

```
node y change x                 // output is -1, 0 or 1
node z change w (dual)          // output is -1, 0 or 1
node z change w (positive)      // output is     0 or 1
node z change w (negative)      // output is     0 or 1
node z change w (both)          // output is     0 or 1
```

Input is digital or analog type. Output node is digital type. Options are (positive), (negative), (both) or (dual). Output is 1 when <nod_nam> changes, otherwise output node is 0, with the exception of option (dual) where output will be +1 or -1 depending on the sign of the change. Default is (dual).

clip: Clip element

node <nod_nam> clip <[- | +] threshold_l> <[- | +] threshold_h> <nod_nam>

```
node y clip 0.5 4.5 x
```

The output node is the same type as the input node (analog or digital). The thresholds can be variables but neither nodes nor expressions.

clock: Digital clock generator

node <nod_nam> clock <“pattern[.pattern]”>

node <nod_nam> clock <“pattern[.pattern]”> <number>

node <nod_nam> clock <“pattern[.pattern]”> <variable>

node <nod_nam> clock <string_variable>

node <nod_nam> clock <string_variable> <number>

node <nod_nam> clock <string_variable> <variable>

```
string pat "0.01"
ivar len  100
node   w  clock "0010 0110"
node   y  clock "010.101100"
node   z  clock "1.1011" 3
node   v  clock pat len
```

Output is digital type with a value 0 or 1 according to the pattern. A pattern is a string of zeroes and ones enclosed by double quotes, spaces are allowed. A period ‘.’ in the pattern separates an initial aperiodic pattern from a periodic pattern. In the examples above, the node w will repeat the sequence ‘00100110’, the node y will start by the sequence ‘010’ followed by a repetition of the sequence ‘101100’. An optional integer number (or a constant integer variable) indicates the repetition of each pattern descriptor. The default value of the repetition factor is 1. In the examples above, the node z will produce the sequence ‘111’ followed by a repetition of the sequence ‘111000111111’.

NB: If a string variable is used as pattern descriptor, it must be defined before the concerned clock definition.

comp: Comparator

node <nod_nam> comp <[- | +] plus_input_node> <[- | +] minus_input_node>

node <nod_nam> comp <[- | +] plus_input_node> <[- | +] variable>

node <nod_nam> comp <[- | +] variable> <[- | +] minus_input_node>

node <nod_nam> comp <[- | +] plus_input_node> <[- | +] number>

node <nod_nam> comp <[- | +] number> <[- | +] minus_input_node>

```
node y comp sigplus sigminus
```

The two inputs must be both analog or both digital. One of the inputs must be a node. Other input could be a node, a variable or a constant number. Output node is digital type. Output is 1 if <sigplus> is greater than or equal to <sigminus>; otherwise output is 0.

A delayed comparator, i.e. delayed by one loop, is easy to model using the short form “&delayed”:

```
node y comp sigplus sigminus  &delayed
```

TIPS:

This comparator has no hysteresis. It is possible to build a hysteresis comparator quite easily. The following example shows an analog differential comparator with a programmable hysteresis encapsulated in a reusable cell. Formal inputs of the cell are input nodes “$p” and “$m”, hysteresis “$h”. Output is “$out”:

File “my_cell.net”

```
cell_interface $out  $p $m  $h

# hysteresis comparator

declare (analog) $p $m $h

node $out comp  $p $th
node $dl  delay $out
node $mx  mux   $dl  $lh $ll
node $th  sum   $m $mx

node $lh  dc    ($h/2.0)
node $ll  dc   -($h/2.0)
```

const: Constant

node <nod_nam> const [<(digital)>] <*C*_expression>

node <nod_nam> const <(analog)> <*C*_expression>

The value of the ***C*** expression *must* be a number. By default “*const*” node is considered as digital type if no explicit casting is done. Node will be declared internally as “I_TYPE” or “R_TYPE” according to casting. Please note that casting *(hex)* is not supported here.

Traditional warnings concerning casting remain valid!

```
node y1   const  123
node y2   const  (digital) 10
node nqst const  (analog)  2.0 * value
```

CAUTION:

The “*const*” nodes are *only* evaluated at initialization (before that, their initial value is DIGITAL_INI or ANALOG_INI, as for the other node kinds). It is thus forbidden to refer to other nodes. The use of variables is allowed. Of course the “*const*” nodes will use the variable definition and will ignore any variable updates.

copy: Signed copy

node <nod_nam> copy <[- | +] nod_nam>

```
node y copy -x
```

Output is a simple signed copy of input. Type of output is identical to type of the input.

cosine: Cosine wave generator

node <nod_nam> cosine <[- | +] offset> <ampl> <freq> <[- | +] phase>

```
node y cosine 0.0 1.0 1000.0 _PI_
```

Similar to "*sine*". Output is analog type.

This node is built to support amplitude, frequency and phase modulation.

TIPS:
If there is no need of frequency nor phase modulation, prefer to use the node "*osc*" with an additional phase of π/2 ("_PI2_"), as trigonometric functions are pretty long to compute and impact badly the speed of the simulations.

dac: N-level signed D/A converter

node <nod_nam> dac <num_lev> <nod_in_nam> <nod_ref_nam>

The input node must be digital type, the reference node must be analog type and the output node will be analog type. The number of levels must be a constant integer of at least 2. This digital-to-analog converter is signed (for an unsigned converter, see "*udac*").

D/A output is clipped when input is outside dynamic input range.

These D/A converters are perfectly symmetrical for an odd number of levels. The D/A converters with an even number of levels are built to correspond exactly to D/A having one level more, but with the upper level missing.

If "y" is the analog value corresponding to node "x" quantized with 8 allowable levels:

```
node ref dc  1.0
node y   dac 8   x ref
```

If "y" is the analog value corresponding to node "x" quantized with 9 allowable levels:

```
node ref dc  1.0
node y   dac 9   x ref
```

The transfer functions of these D/A converters are:

input D/A 9 levels	analog output	input D/A 8 levels	analog output
n <= -4	-1.00	n <= -4	-1.00
-3	-0.75	-3	-0.75
-2	-0.50	-2	-0.50
-1	-0.25	-1	-0.25
0	0.00	0	0.00
1	0.25	1	0.25
2	0.50	2	0.50
3	0.75	n >= 3	0.75
n >= 4	1.00	NA	NA

dalgebra: *C* expression cast to real type

node <nod_nam> dalgebra <*C*_expression>

```
node y dalgebra rand_normal(0.0, 0.10) + off
```

This is just like an "*algebra*" node, but ***NAPA*** does not try to determine the type, it is automatically **cast** to double. Thus this type of node does not need to refer to any other node, as a plain algebra node does.

Inside the expression, you can mix ***NAPA*** nodes or variables with ***C*** global variables.

Traditional warnings concerning casting remain valid!

See also: "after" qualifier page 81.

dc: DC source

node <nod_nam> dc [<(analog)>] <*C*_expression>

node <nod_nam> dc <(digital)> <*C*_expression>

The value of the ***C*** expression *must* be a number. By default "*dc*" node is considered as analog type if no explicit casting is done. Node will be declared internally as "I_TYPE" or "R_TYPE" according to casting.

Please note that casting *(hex)* is not supported here.

Traditional warnings concerning casting remain valid!

```
node y1   dc  1.23
node y2   dc  (digital) 10
node nqst dc  (analog)  2.0 * freq
```

CAUTION:
The "*dc*" nodes are *only* evaluated at initialization (before that, their initial value is ANALOG_INI or DIGITAL_INI, as for the other node kinds). It is thus forbidden to refer to other nodes. The use of variables is allowed. Of course the "*dc*" nodes will use the variable definition and will ignore any variable updates.

delay: Single or multiple delay

The output node is the same type as the input node (analog or digital type).

node <nod_nam> delay <[- | +] nod_nam>

node <nod_nam> delay <number> <[- | +] nod_nam>

node <nod_nam> delay <ivar_name> <[- | +] nod_nam>

For instance: "y" is a delayed value of "x":

```
node y delay x
```

Consider initialization!

The delay represents a kind of memory. Use the instruction "*init*" to initialize the output node of the delay if necessary.

Here is an example of an analog delay with initialization.

```
node y delay x
init y 1.0 + sin(0.85)
```

TIPS:
Delays are used quite often. Delay can be used for instance to build integrators and differentiators: Here is an example of cell realizing a delayed integrator with an initial value:

File "my_integrator1.net"

```
cell_interface $o  $i  $start
init $o $start
node $o delay  $a
node $a sum    $o $i
```

The non-delayed integrator with an initial value:

File "my_integrator2.net"

```
cell_interface $o  $i  $start
init $d $start
node $d delay  $o
node $o sum    $d $i
```

A differentiator cell can be built as easily:

File “my_differentiator.net”

```
cell_interface $o  $i
node $d delay $i
node $o sub   $i $d
```

Please note that these cells are *chameleonic* as the output takes the type of the input.

Multiple delays: the output node is the same type as the input node (analog or digital). The number of delays must be an unsigned integer constant (0 gives a simple copy) or an “*ivar*” with a constant value of at least 1 (i.e. not updated, checked at run time). The multiple delay is translated efficiently into $\boldsymbol{C}$ (circular buffer).

For instance, y is x delayed by 16 sampling periods:

```
node y delay 16  x
```

and z is x delayed by 2 sampling periods:

```
ivar n 2
node z delay  n  x
```

The multiple “*delay*” node represents an efficient way to describe multiple delays. Use the instruction “*init*” to initialize the output node of the “*delay*” if necessary. All internal storage elements will be initialized to that value. To give each stored value its own initial value, you need to replace the “*delay*” by a sequence of “*delay*” nodes.

NB: The short form “&delayed” may be used advantageously in many situations.

differentiator: Non-inverting differentiator

node <nod_nam> differentiator <[- | +] nod_nam>

```
node y differentiator -x
```

The output node is the same type as the input node (analog or digital). Output is the non-inverting non-delayed differentiation of the input.

div: Divider element

node <nod_nam> div <[- | +] nod_nam> <[- | +] nod_nam> [(nocheck)]

node <nod_nam> div <[- | +] nod_nam> <[- | +] var_nam> [(nocheck)]

node <nod_nam> div <[- | +] nod_nam> <[- | +] number> [(nocheck)]

If “y” is the division of nodes “a” and “b” then:

```
node y div a b
```

The node “a” and node (or variable or number) “b” *must* be the same type (analog or digital), and output node “y” will be of this type. A division by zero stops the simulation with an error message. Qualifier “*(nocheck)*” suppresses the test of the division by zero.

NOTE:
This node accepts a node, a variable or a constant as second input. There is no specialization like “*gain*” and “*prod*” or “*sum*” and “*offset*”. Very few nodes have the same syntax (see “*mod*”).

dtoi: Analog-to-digital type conversion

node <nod_nam> dtoi <nod_nam>

If “x” is an analog type node, node “y” will be a digital type node with the **rounded** value of node “x”:

```
node y dtoi x
```

“*dtoi*” can be considered as an ideal multi-level A/D without clipping with a step equal to 1.0 (see also “*adc*” and “*uadc*”).

CAUTION: This conversion is not equal to a simple casting! It corresponds to a mathematical rounding. This node is *not* equivalent to ‘node y ialgebra (long long) x’!

dtool: User-defined tool

node <nod_nam> dtool <dtool_nam> [<list>] [(<opt>)]

Similar to “*duser*”, but “*dtool*” is synchronizable and is not reset automatically during a restart. Output node is generally used to control the simulation. If output is not used, use the identifier “void” to avoid a warning. See user-defined functions page 147. See also the instruction “post”.

“*dtool*” accepts optional qualifiers. See (option) in chapter Instruction Qualifiers.

CAUTION: Nodes “*dtool*” and “*duser*” are not equivalent.

See also “*after*” and “*with*” qualifiers.

duser: User-defined function

node <nod_nam> duser <duser_nam> [<list>] [(<opt>)]

This user-defined function returns an analog value. You write a function in ***C*** and put it in the ***C*** header file. You can pass the function any number of nodes or constants, including no arguments at all. You need to follow guidelines to write user-defined functions called by "duser" nodes. As ***NAPA*** automatically calls the check, init, reset and close functions, all of them must be provided.

For a "*duser*" function called "foo" having 5 arguments, you need to provide these ***C*** functions:

```
double     duser_foo_05(..., int id)   function itself
void check_duser_foo_05(..., int id)   called at initialization
void  init_duser_foo_05(..., int id)   called at initialization
void reset_duser_foo_05(..., int id)   called during a restart[4]
void close_duser_foo_05(..., int id)   called at the end
```

Where the last input parameter (id) must be an integer representing the instantiation number of the function, an additional parameter, provided by the ***NAPA*** transpiler. Some of these functions could be empty but must exist.

For example, user-defined function "clipper" could be defined like this:

```
double duser_clipper_02(double x, double limit, int id) {
 if (x > limit) {
    return limit;
  }
  else if (x < -limit) {
    return -limit;
  }
  return x;
}
```

Other related functions have to be provided (mandatory): for example:

```
void check_duser_clipper_02(double x, double limit, int id) {
 if (limit < 0.0) {
    fprintf(stderr, "NAPA run time error (clipper[%d])\n", id);
    fprintf(stderr, "clipping limit cannot be negative\n");
    napa_exit(EXIT_FAILURE);
  }
  return;
}

void  init_duser_clipper_02(double x, double limit, int id) {
  return;
}

void close_duser_clipper_02(double x, double limit, int id) {
  return;
}
```

4 Code in function "reset_duser_...()" must be re-entrant!

```
void reset_duser_clipper_02(double x, double limit, int id) {
  check_duser_clipper_02(x, limit, id);               /* check it again */
  return;
}
```

NAPA calls these functions when necessary. You can then use the function in the netlist. For example, if node “y” is the clipped value of node “x” with a clipping value of 2.5, then:

```
node y duser clipper x 2.5
```

Please note that the instantiation identifier is automatically added by ***NAPA*** itself (0, 1 ...).

Sometimes, it is necessary to send qualifiers to modify the behavior of the user functions. As this qualifier is alphanumeric, there is a risk of collision with existing nodes or variables. ***NAPA*** allows user and tool nodes to place this qualifier between parentheses:

```
node y duser synchro_linsweep 0.0  99.0 100 (shuffle)
```

“*duser*” accepts optional qualifiers. See (option) in chapter Instruction Qualifiers.

CAUTION: Nodes “*dtool*” and “*duser*” are not equivalent.

See also: “after” qualifier page 81.

equal: Equality

node <nod_nam> equal <[- | +] nod_nam> <[- | +] nod_nam> [<precision>]

node <nod_nam> equal <[- | +] nod_nam> <[- | +] variable> [<precision>]

node <nod_nam> equal <[- | +] variable> <[- | +] nod_nam> [<precision>]

node <nod_nam> equal <[- | +] nod_nam> <[- | +] number> [<precision>]

node <nod_nam> equal <[- | +] number> <[- | +] nod_nam> [<precision>]

```
node x equal a1 a5

node y equal a2 a9   0.01
node z equal d3 10   5
```

The two inputs must be both analog or both digital; at least one of them must be a node. Other input could be a node, a variable or a constant number. Output node is digital type. Output is set to 1 if input nodes are equal; otherwise output is set to 0.

For analog types, equality is by default tested within a relative tolerance, |a-b| <= EPSILON*(|a|+|b|) as for macro ISEQUAL. If a precision is added in the instruction, the equality for analog or digital types is tested with the indicated absolute tolerance.

fzand: N-input AND element (Fuzzy logic)

node <nod_nam> fzand <nod_nam...>

```
node y fzand a b c
```

Zadeh operator. Output node is always analog type (between 0.0 and 1.0). Inputs must be analog type.

Formula: y = min(x1, ..., xN), limited to [0, 1], where x1 ... xN are the input nodes.

fzbuffer: Non-inverting buffer (Fuzzy logic)

node <nod_nam> fzbuffer <nod_nam>

```
node y fzbuffer x
```

Zadeh operator. Output node is always analog type (between 0.0 and 1.0). Input must be analog type.

Formula: y = x, limited to [0, 1].

In addition to the function of buffer, this function is also used to limit the signal to the interval [0, 1] compatible with all the fuzzy logic functions.

fzinv: Negation element (Fuzzy logic)

node <nod_nam> fzinv <nod_nam>

```
node y fzinv x
```

Strictly equivalent to “fznot”.

Formula: y = 1 - x, limited to [0, 1].

fznand: N-input NAND element (Fuzzy logic)

node <nod_nam> fznand <nod_nam...>

```
node y fznand a b c
```

Zadeh operator. Output node is always analog type (between 0.0 and 1.0). Inputs must be analog type.

Formula: y = 1 - min(x1, ..., xN), limited to [0, 1], where x1 ... xN are the input nodes.

fznor: N-input NOR element (Fuzzy logic)

node <nod_nam> fznor <nod_nam...>

```
node y fznor a b c
```

Zadeh operator. Output node is always analog type (between 0.0 and 1.0). Inputs must be analog type. Formula: y = 1 - max(x1, ..., xN), limited to [0, 1], where x1 ... xN are the input nodes.

fznot: Negation element (Fuzzy logic)

node <nod_nam> fznot <nod_nam>

```
node y fznot x
```

Zadeh operator. Output node is always analog type (between 0.0 and 1.0). Input must be analog type. Formula: y = 1 - x, limited to [0, 1].

fzor: N-input OR element (Fuzzy logic)

node <nod_nam> fzor <nod_nam...>

```
node y fzor a b c
```

Zadeh operator. Output node is always analog type (between 0.0 and 1.0). Inputs must be analog type. Formula: y = max(x1, ..., xN), limited to [0, 1], where x1 ... xN are the input nodes.

fzxnor: 2 inputs XNOR element (Fuzzy logic)

node <nod_nam> fzxnor <nod_nam> <nod_nam>

```
node y fzxnor a b
```

Zadeh operator. Output node is always analog type (between 0.0 and 1.0). Inputs must be analog type. Formula: y = 1 - min(max(a, b), 1 - min(a, b)), limited to [0, 1].

fzxor: 2 inputs XOR element (Fuzzy logic)

node <nod_nam> fzxor <nod_nam> <nod_nam>

```
node y fzxor a b
```

Zadeh operator. Output node is always analog type (between 0.0 and 1.0). Inputs must be analog type.

Formula: y = min(max(a, b), 1 - min(a, b)), limited to [0, 1], i.e. (a OR b) AND NOT (a AND b).

gain: Gain element

node <nod_nam> gain <[- | +] constant> <nod_nam>

node <nod_nam> gain <[- | +] variable> <nod_nam>

The output node is the same type as the input node (analog or digital). The gain factor can be a constant or a variable but neither a node nor an expression. Variable or constant type must be consistent with node type.

If “y” is equal to “x” times 2.5 then:

```
node y gain  2.5  x
```

A user's variable can be used as gain factor:

```
dvar g pow(10.0, ampldB/20.0)
node y gain  g  x
```

CAUTION: Nodes “*gain*” and “*prod*” are not equivalent.

generator: Subcircuit generation from a file

node <nod_nam> generator <inst_nam> <“fil_nam”> <nod_nam | parm...>

This is not a node but a pseudo-node. “fil_nam” is an executable capable of generating a ***NAPA*** cell file (see “*cell*”). Instantiation name <inst_nam> of the generated cell must be unique and cannot be reused for another cell or generator.

A directory is dedicated to store a library of reusable generators. The pathname of this library is indicated to ***NAPA*** through the command line. Using angle brackets “< ... >” is a short form to call a generator contained in this directory. Supposing that the pathname of this library is “/napalib/gen”, the following expressions are perfectly equivalent:

```
node n1 generator  pls1  <mkcel>  n2 100.0
node n3 generator  pls2  "/napalib/gen/mkcel"  n2 100.0
```

The generator is an executable (or a script) capable of generating a ***NAPA*** cell file. The ***NAPA*** transpiler runs the executable directly, without going through ***a shel***l (function "_spawnv" on the Windows build, "system" elsewhere); no shell metacharacter in a parameter is therefore ever reinterpreted:

spawnv("/napalib/gen/mkcel pls1_0.gen n2 100.0")

The user must therefore write an executable (or a script) taking arguments "pls1_0.gen" corresponding to its output file and strings "n2" and "100.0" and generating a syntactically correct ***NAPA*** cell (see node "*cell*"). Cell should be generated as file "pls1_0.gen", the name received as first argument: the instantiation name, "_" and the number of generators already called in the netlist (here 0), followed by the suffix ".gen", in the directory of the main netlist.

The ***NAPA*** transpiler uses the newly created cell as a regular ***NAPA*** cell, generating automatically an internal call corresponding to:

node n1 cell pls1 "pls1_0.gen" n2 100.0

The cell file "pls1_0.gen" will contain a netlist written by the generator like:

File "pls1_0.gen"

```
cell_interface  $o $i  $parm
...
node $o ...
...
```

The executable of a generator must be located inside the generator library: ***NAPA*** refuses to run an executable located outside this library, whatever the way it is called. A pathname relative to the directory of the main netlist ("~/....") or to the directory of the calling cell is therefore rejected, as in both following calls:

```
node n1 generator  pls1  "~/mkcel"  n2 100.0
node n3 generator  pls1  "mkcel"    n2 100.0
```

Pseudo-node generator accepts iterative identifiers (see instruction "data").

Follow ***NAPA*** recommendations, use no suffix for the name of the executable under Linux and Unix, use suffix ".exe" under Windows (see appendix C page 166).

hold: Hold and track element

node <nod_nam> hold <ctrl_nod_nam> <nod_nam>

```
node y hold ctr x
```

The control *must* be digital type. The output node is the same type as the input node (analog or digital). Node "y" tracks the input node "x" if the control node is 0 (see also similar node "*track*"). It is held otherwise.

Consider initialization!

ialgebra: *C* expression cast to integer type

node <nod_nam> ialgebra <*C*_expression>

```
node y ialgebra (2 * digit) + 1
```

This is just like an "*algebra*" node, but ***NAPA*** does not try to determine the type, it is automatically **cast** to integer. Thus this type of node does not need to refer to any other node, as a plain algebra node does.

Inside the expression, you can mix ***NAPA*** nodes or variables with ***C*** global variables.

Traditional warnings concerning casting remain valid!

See also: "after" qualifier page 81.

integrator: Non-inverting integrator

node <nod_nam> integrator <[- | +] nod_nam>

```
node y integrator -x
node z integrator  w   &delayed
```

The output node is the same type as the input node (analog or digital). Output is the non-inverting non-delayed integration of the input. But using the short form "&delayed", it is easy to model a delayed integrator!

Binary counter is realized in a one-line ***NAPA*** instruction:

```
<3> node y integrator One
```

Output y is 1 2 3 4 5 6 7 0 1 2 3 4 ..

Consider initialization!

At the initialization by default, output node is equal to "DIGITAL_INI" or "ANALOG_INI". Use the instruction "*init*" to overwrite the default initialization of the output:

```
<3> node y  integrator One
   init y  7
```

Output y is now 0 1 2 3 4 5 6 7 0 1 2 3 ..

inv: Negation element (Boolean logic)

(strictly equivalent node: "*not*").

node <nod_nam> inv <nod_nam>

```
node y inv x
```

The input node *must* be digital type, and output node is always digital type (0 or 1).

itob: Bit extractor from digital node

node <nod_nam> itob <bit_rank> <nod_nam>

Input node *must* be digital type (positive value), output node is digital type (0 or 1). <bit_rank> is the rank of the bit to be extracted from the integer stored in <nod_nam>. The parameter <bit_rank> must be an integer. It is the responsibility of the user to keep the value of <bit_rank> non-negative.

"bseven" is bit 7 of the integer "w" (bit 0 = LSB):

```
node bseven itob 7 w
```

If integer "w" is equal to '001 0**1**0 000 001', "bseven" is equal to 1. See also "*btoi*" and bit field.

CAUTION: Distinguish *"itob"* and bit field:
Bit fields are also used to extract a single bit from a digital node. They use the same method to extract the bit. There is a subtle difference. The bit field creates an automatic bit extraction made in the same segment as the node from which the bit must be extracted. *"itob"* extracts the bit from the node where it is placed. Sampling is made at local sampling frequency!

itod: Digital-to-analog type conversion

node <nod_nam> itod <nod_nam>

If node "x" is digital type node, node "y" will be an analog type node with the same value as "x".

```
node y itod x
```

"*itod*" can be considered as an ideal multi-level D/A without clipping with a step equal to 1.0 (see also "*dac*" and "*udac*").

itool: User-defined tool

node <nod_nam> itool <itool_nam> [<list>] [(<opt>)]

Similar to "*iuser*", but "*itool*" is synchronizable and is not reset automatically during a restart. Output node is generally used to control the simulation. If output is not used, consider using identifier "*void*",

to avoid unwanted warning message. See user-defined functions page 147. See also the instruction “post”.

“*itool*” accepts optional qualifiers. See (option) in chapter Instruction Qualifiers.

TIPS: instruction “tool” is expanded into an “itool” node returning “*void*”.

TIPS: for time-frequency analysis, i.e. a spectrum that evolves with time (settling, lock acquisition, chirps, intermittent tones), the library provides the tools “*spectro*” (spectrogram, short-time Fourier transform) and “*cwt*” (continuous wavelet transform, Morlet wavelet). Their usage is documented in their headers “Tool/spectro.hdr” and “Tool/cwt.hdr”, gathered in “Hdr/Header.rtf”.

TIPS: measurement tools of the library: for clocks and oscillators, “*allan*” (Allan deviation) and “*pnoise*” (phase noise L(f) in dBc/Hz), next to “*jitter*”; for a step response, “*stepresp*” (delay, rise time, overshoot, settling time); for converters, “*adclin*” (DNL, INL and missing codes by the code density method), and “*tsnr*”, which also prints the SFDR and the ENOB on a comment line. Each tool is documented in its header, gathered in “Hdr/Header.rtf”.

CAUTION: Nodes “*itool*” and “*iuser*” are not equivalent.

See also “*after*” and “*with*” qualifiers.

iuser: User-defined function

node <nod_nam> iuser <iuser_nam> [<list>] [(<opt>)]

This user-defined function returns a digital value. You write a function in ***C*** and put it in the ***C*** header file. You can pass the function any number of nodes or constants, including no arguments at all. You need to follow guidelines to write user-defined functions called by “iuser” nodes. As ***NAPA*** automatically calls the check, init, reset and close functions, all of them must be provided.

For an “iuser” function called “foo” with 3 arguments, you need to provide:

```
I_TYPE     iuser_foo_03(..., int id)   function itself
void check_iuser_foo_03(..., int id)   called at initialization
void  init_iuser_foo_03(..., int id)   called at initialization
void reset_iuser_foo_03(..., int id)   called during a restart[5]
void close_iuser_foo_03(..., int id)   called at end of simulation
```

Where the last input parameter (id) must be an integer representing an additional parameter, the instantiation number of the function provided by the ***NAPA*** transpiler. Some of these functions could be empty but must exist.

For example, if you want to use a function “FourLevelADC” in a ***NAPA*** netlist, you could define the following functions in a header file (***C*** language):

```
I_TYPE iuser_FourLevelADC_02 (double x, double ref, int id) {
/* "id" is an identifier of the instantiation handled       */
/* directly by NAPA (0, 1, ...)                             */
  if        (x < -1.25*ref) {
    return 0;
```

5 Code in function “reset_iuser_...()” must be re-entrant!

```
 } else if (x <   0.0*ref) {
    return 1;
  } else if (x <  1.25*ref) {
    return 2;
  } else {
    return 3;
  }
}

void  init_iuser_FourLevelADC_02 (double x, double ref, int id) {
  /* code executed during the initialization                */
  return;
}

void check_iuser_FourLevelADC_02 (double x, double ref, int id) {
  /* code executed during the check phase                   */
  if (ref <= 0.0) {
    fprintf(stderr, "NAPA run time Error (FourLevelADC[%d])", id);
    fprintf(stderr," reference must be positive\n");
    napa_exit(EXIT_FAILURE);
  }
  return;
}

void reset_iuser_FourLevelADC_02 (double x, double ref, int id) {
  /* code executed during the reset                         */
  return;
}

void close_iuser_FourLevelADC_02 (double x, double ref, int id) {
  /* code executed at the end of the simulation             */
  return;
}
```

These functions are called automatically by NAPA when necessary. You could then use this in the netlist. For example if node “y” is the digitized value of node “x”, the header file being “adc4.hdr”:

```
header "adc4.hdr"
node y iuser FourLevelADC x 1.00
```

Instantiation identifier “id” is automatically added by NAPA (id = 0, 1...).

Sometimes, it is necessary to send qualifiers to modify the behavior of the user functions. As this qualifier is alphanumeric, there is a risk of collision with existing nodes or variables. ***NAPA*** allows user and tool nodes to place this qualifier between parentheses:

```
node y iuser sequence  1 10 (up)
```

“*iuser*” accepts optional qualifiers. See (option) in chapter Instruction Qualifiers.

CAUTION: Nodes “*itool*” and “*iuser*” are not equivalent.

See also “*after*” qualifier.

latch: SR Latch

node <nod_nam> latch <set_input> <reset_input>

```
node y latch sigset sigreset
```

Both inputs must be digital. Output node is digital type. Output is 0 or 1. Output is 1 if last signal to be at 1 was <set_input>. Output is 0 if last signal to be at 1 was <reset_input>. In case of conflicting transitions giving an undetermined output, simulation stops with an appropriate error message.

Consider initialization!

At the initialization, latch output node is equal to "DIGITAL_INI" by default. Use the instruction "*init*" to overwrite the default initialization of the output:

```
node y latch sigset sigreset
init y 1
```

lshift: Left shift element

node <nod_nam> lshift <shift_val> <nod_nam>

node <nod_nam> lshift <shift_var> <nod_nam>

node <nod_nam> l shift <shift_nod> <nod_nam>

```
ivar sh4  4
...
node y lshift 4   x
node z lshift sh4 x
```

In the example above, the values of "y" and "z" are equal to the value of "x" with the bits shifted by 4 positions to the left. Input node and shift value (node, variable or constant) must be digital type. Output is digital type. A constant shift value must lie between 0 and the width of the input node minus 1 (63 by default); a shift value given by a node or a variable is not checked and must remain between 0 and 63. The shift value cannot be signed and the input node cannot be negated. The bits shifted out on the left are lost: declare the width of the output node to limit the result.

max: Maximum of N inputs

node <nod_nam> max <[- | +] nod_nam...>

If "y" is the maximum of nodes "a", "b", and "-c" then:

```
node y max a b -c
```

The nodes “a”, “b”, and “c” *must* be the same type (analog or digital), and output node “y” will be of this type.

merge: N-input multiplexer from exclusive loop segments

node <nod_nam> merge <[- | +] nod_nam...> [(nocheck)]

The input nodes must be either all analog type or all digital type. The output node will be the same type as input. The output merges the inputs, which must come from mutually exclusive segments, otherwise an appropriate error message is generated. Qualifier “*(nocheck)*” suppresses this test.

In this example below, the node “y” gets the value of node “a1”, the value of node “-a2” or the value of “a3” when the segment corresponding respectively to these nodes is activated. Result will be a sequence of values of “a1, a3, -a2, a1, a3, -a2, a1, ...”

```
...
decimate fs  3         // running 1 over 3 (with an offset 0)
node a1 ...

decimate fs  3  2      // running 1 over 3, with an offset 2
node a2 ...

decimate fs  3  1      // running 1 over 3, with an offset 1
node a3 ...

nominal  fs
node y   merge a1 -a2  a3
...
```

min: Minimum of N inputs

node <nod_nam> min <[- | +] nod_nam...>

If “y” is the minimum of nodes “a”, “b”, and “-c” then:

```
node y min a b -c
```

The nodes “a”, “b”, and “c” *must* be the same type (analog or digital), and output node “y” will be of this type.

mod: Modulo divider element

node <nod_nam> mod <[- | +] nod_nam> <[- | +] nod_nam> [(nocheck)]

node <nod_nam> mod <[- | +] nod_nam> <[- | +] var_nam> [(nocheck)]

node <nod_nam> mod <[- | +] nod_nam> <[- | +] number> [(nocheck)]

If “y” is “a” modulo “-b”:

```
node y mod a -b
```

The node “a” and the node (or variable or number) “b” *must* be the same type (analog or digital), and output node “y” will be of this type. Division by zero generates an error message and the exit of the simulation. Qualifier “*(nocheck)*” suppresses the test of the division by zero.

NOTE:
This node accepts a node, a variable or a constant as second input. There is no specialization like “*gain*” and “*prod*” or “*sum*” and “*offset*”. Very few nodes have the same syntax (see “*div*”).

muller: C Muller element, N inputs (Boolean logic)

node <nod_nam> muller <nod_nam...>

```
node y muller a b
```

The input nodes *must* be digital type, and output node is always digital type (0 or 1). The output of a *C* Muller element follows the inputs if they are all ‘TRUE’ or all ‘FALSE’. It retains its previous value in the other cases. This is a digital hysteresis element.

Consider initialization!

This node can be initialized (default is 0):

```
node y muller a b
...
init y 0
```

mux: N-input multiplexer controlled by integer levels

node <nod_nam> mux <ctrl_nod_nam> <[- | +] nod_nam | void...>

node <nod_nam> mux <ctrl_var_nam> <[- | +] nod_nam | void...>

```
node y mux ctrl x1 -x2 x3 x4
```

The input nodes must be either all analog type or all digital type. The output node will be the same type as input. The control (node or variable) must be digital. When control equals 0, output is connected to first input node, when control equals 1, output is connected to second node ...etc. If control is out of range, the simulation stops with an error message.

This node accepts "void" inputs.

```
node z mux ctrl x1 -x2 void x4
```

nand: N-input NAND element (Boolean logic)

node <nod_nam> nand <nod_nam...>

```
node y nand a b c
```

The input nodes *must* be digital type, and output node is always digital type (0 or 1).

noise: Source of noise

node <nod_nam> noise <[- | +] DC_level> <noise_density_level>

```
node y noise dclev 0.01
```

The inputs must be analog type (constant or variable), and output node is always analog type. The DC level can be signed, the noise density (RMS / √Hz) value cannot. Output is a normally distributed pseudo random noise. See also instruction "*random_seed*".

nor: N-input NOR element (Boolean logic)

node <nod_nam> nor <nod_nam...>

```
node y nor a b c
```

The input nodes *must* be digital type, and output node is always digital type (0 or 1).

not: Negation element (Boolean logic)

(strictly equivalent node: "*inv*").

node <nod_nam> not <nod_nam>

```
node y not x
```

The input node *must* be digital type, and output node is always digital type (0 or 1). Output is 1 if input is 0, 0 otherwise.

offset: DC level shifter element

node <nod_nam> offset <[- | +] constant> <nod_nam>

node <nod_nam> offset <[- | +] variable> <nod_nam>

The output node is the same type as the input node (analog or digital). The DC level-shift can be a constant or a variable but neither a node nor an expression. Variable or constant type must be consistent with node type.

If “y” is equal to “x” plus 2.5 then:

```
node y offset  2.5  x
```

A user's variable can be used as DC level-shift:

```
dvar off 10
node y offset  off  x
```

CAUTION: Nodes “*offset*” and “*sum*” are not equivalent.

or: N-input OR element (Boolean logic)

node <nod_nam> or <nod_nam...>

```
node y or a b c
```

The input nodes *must* be digital type, and output node is always digital type (0 or 1).

osc: Oscillator

node <nod_nam> osc <[- | +] offset> <ampl> <freq> <[- | +] phase>

```
node y osc 0.0 1.0 1000.0 _PI_
```

Similar to "*sine*" but much faster to compute. Output is analog type.

This node is built to support amplitude modulation but not frequency nor phase modulation. In this case, use the node "*sine*".

poly: Polynomial of order N

node <nod_nam> poly <[- | +]coeffN> [<[- | +]coeffi> ...] <nod_nam>

The coefficients are given from the highest degree down to degree 0: "y" is the polynomial of a*x^2 + b*x - c, where "a", "b", and "-c" are variables and x the input node:

```
node y poly a b -c  x
```

The output node will be the type of the input node. Variables must be of the same type as the input node.

prod: N-input multiplier element

node <nod_nam> prod <[- | +] nod_nam...>

"y" is the product of nodes "a", "b", and "-c":

```
node y prod a b -c
```

The nodes "a", "b", and "c" *must* be the same type (analog or digital), and output node "y" will be of this type.

CAUTION: Nodes "*gain*" and "*prod*" are not equivalent.

quant: Quantizer

node <nod_nam> quant <nod_nam> <[- | +] nod_nam>

node <nod_nam> quant <parameter> <[- | +] nod_nam>

node <nod_nam> quant <constant> <[- | +] nod_nam>

"y" is the quantization of "x", with a quantization step "stp":

```
node y quant stp x
```

The output node has the type of the input node. The quantization step must have the same type as the input node and must be strictly positive.

ram: Random access memory

node <nod_nam> ram <ram_nam‘[’addr_nod‘]’> <CS> <RW_nod> <nod_nam>

Where the address <addr_nod>, the chip select control <CS> and the read/write control <RW_nod> are digital type. The type of output (corresponding to the RAM read port) and of the input node (corresponding to the RAM write port) must be compatible with the “array” declaration corresponding to the RAM <ram_nam>. It is thus perfectly possible to define an analog RAM in ***NAPA***!

The RAM has two unidirectional communication ports: read and write. The RAM is addressed only if the chip select input <CS> is different from zero. When the chip select is equal to zero, neither input nor output is possible and read port is on hold. Read/Write control at 1 activates the read port. At 0, it activates the write port. During a write access, the read port follows the written value. Address is verified during run-time to remain inside the array limits (see instruction “*array*”). It is possible to initialize a RAM using an initialization file (see instruction “array”).

The simulator will track the read access to uninitialized data and will issue a warning indicating the loop index of the last occurrence (if any) at the end of the simulation.

```
node rp ram myram[m] cs rw wp
...
array (digital) myram[23]
```

How to simulate two RAMs connected to a common bus? Use a node “mux” with appropriate control to model the common read port. In the following example, the bus is modeled by signals “rp” and “wp”, respectively for the read and write lines:

```
node cs1 inv    ctr
node cs2 buffer ctr
...
node rp1 ram myram1[n] cs1 rw1 wp         // selected when ctr is 0
node rp2 ram myram2[n] cs2 rw2 wp         // selected when ctr is 1
node rp  mux ctr rp1 rp2
...
array (digital) myram1[32]
array (digital) myram2[32]
```

There is another type of RAM, the dual port “ram2”.

ram2: Dual port random access memory

node <nod_nam> ram2 <ram_nam‘[’addr_nod‘]’> <CS> <RW_nod> <nod_nam>

Similar to node “ram”. Use two instances of the node to implement the dual port.

```
node rp1 ram2 myram[m] cs rw wp1
node rp2 ram2 myram[n] cs rw wp2
...
array (digital) myram[128]
```

rect: Rectifier element

node <nod_nam> rect <nod_nam>

"y" is the absolute value of node "x":

```
node y rect x
```

Output node conforms to the type of the input node.

register: Data register

node <nod_nam> register <ctrl_nod_nam> <nod_nam>

```
node y register ctr x
```

The control *must* be digital type. The output node is the same type as the input node (analog or digital). Node "y" follows the input node "x" if the control node is not 0. Value is held otherwise.

This node is strictly equivalent to "*track*".

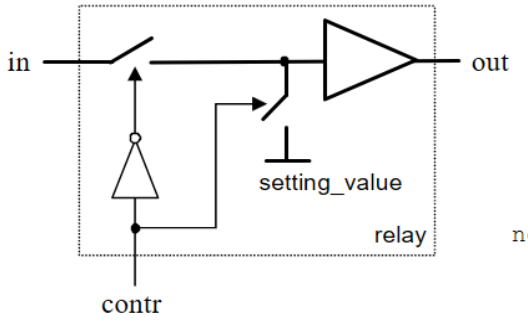
Consider initialization!

relay: One input relay, normally closed

node <nod_nam> relay <ctrl_nod_nam> <nod_nam>
node <nod_nam> relay <ctrl_var_nam> <nod_nam>
node <nod_nam> relay <ctrl_constant> <nod_nam>
node <nod_nam> relay <ctrl_nod_nam> <nod_nam> <set_var_nam>
node <nod_nam> relay <ctrl_var_nam> <nod_nam> <set_var_nam>
node <nod_nam> relay <ctrl_constant> <nod_nam> <set_var_nam>
node <nod_nam> relay <ctrl_nod_nam> <nod_nam> <set_constant>
node <nod_nam> relay <ctrl_var_nam> <nod_nam> <set_constant>
node <nod_nam> relay <ctrl_constant> <nod_nam> <set_constant>

```
node y1 relay ctrl  x
node y2 relay ctrl  x  v
node y3 relay ctrl  x  1.5
node y4 relay 0     x  n
```

The input node can be analog type or digital type. The output node will be the same type as input. Control type must be digital type. When control equals 0, output is connected to input node. If not, output is set to setting value (default 0 or 0.0).




```
node out relay contr in setting_value
```

An example of cell using a relay for reset:

File “my_integrator.net”

```
cell_interface $out $in $reset

#* delayed integrator with reset (chameleonic cell)

node $b   delay  $a
node $out relay  $reset $b    // here using default: setting to zero
node $a   sum    $in    $out
```

rip: Bitwise rip bus

node <nod_nam> rip <mask> <nod_nam>

```
node b rip 0x0F50  a
```

Output node is a ripped value of input node: the extracted bits are aligned on the right. Mask must be an unsigned hexadecimal constant written with the prefix 0x; it cannot select a bit beyond the width of the input node.

```
node a          ABCD EFGH IJKL MNOP QRST
Mask            0000 0000 1111 0101 0000      (i.e.  0x0F50)
node b          0000 0000 0000 00IJ KLNP
```

rom: Read only memory

node <nod_nam> rom <rom_nam‘[’addr_nod‘]’> <CS>

Where the address <addr_nod> and the chip select control <CS> are digital type. The type of output node (corresponding to the ROM read port) must be compatible with the “*array*” declaration corresponding to the ROM <rom_nam>. It is thus perfectly possible to define an analog ROM in ***NAPA***!

The ROM has one unidirectional communication port. The ROM is addressed only if the chip select input is different from zero. When the chip select <CS> is equal to zero, no output operation is possible and read port is on hold. Address is verified during run-time to remain inside the array limits (see instruction “*array*”).

```
node  rport rom  rom1[m] cs
...
array (hex) rom1[1024]  "program.rom"
```

How to simulate two ROMs connected to a common bus? Use a node “mux” with appropriate control to model the common read port. In the following example, the bus is modeled by the read line “rp”:

```
node cs1 inv    ctr
node cs2 buffer ctr
...
node rp1 rom myrom1[n] cs1                // selected when ctr is 0
node rp2 rom myrom2[n] cs2                // selected when ctr is 1
node rp  mux ctr rp1 rp2
...
array (digital) myrom1[100]  "program1.rom"
array (digital) myrom2[100]  "program2.rom"
```

There is another type of ROM, the dual port “rom2”.

rom2: Dual port read only memory

node <nod_nam> rom2 <rom_nam‘[’addr_nod‘]’> <CS>

Similar to node “rom”. Use two instances of the node to implement the dual port.

```
node rport1 rom2  myrom[m] cs
node rport2 rom2  myrom[n] cs
...
array (hex) myrom[1024]  "program.rom"
```

rshift: Right shift element without rounding

node <nod_nam> rshift <shift_val> <nod_nam>

node <nod_nam> rshift <shift_var> <nod_nam>

node <nod_nam> rshift <shift_nod> <nod_nam>

```
ivar sh2      2
...
node y rshift 2   x
node z rshift sh2 x
```

In the example above, the values of “y” and “z” are equal to the value of “x” with the bits shifted by 2 positions to the right. Input node and shift value (node, variable or constant) must be digital type. Output is digital type. The shift is arithmetic: the sign bit is copied, so that the result is the value divided by 2 to the power of the shift, rounded toward minus infinity (-3 shifted by 1 gives -2). Same limits as “lshift” for the shift value and the input node.

rshift1: Right shift element, rounding half away from zero

node <nod_nam> rshift1 <shift_val> <nod_nam>

node <nod_nam> rshift1 <shift_var> <nod_nam>

node <nod_nam> rshift1 <shift_nod> <nod_nam>

```
ivar sh2       2
...
node y rshift1 2   x
node z rshift1 sh2 x
```

In the example above, the values of “y” and “z” are equal to the value of “x” with the bits shifted by 2 positions to the right and rounded to the nearest integer, a half being rounded away from zero: 1.5 gives 2, -1.5 gives -2. Input node and shift value (node, variable or constant) must be digital type. Output is digital type. The computation is exact on 64 bits. Same limits as “rshift” for the shift value.

rshift2: Right shift element, rounding half to even

node <nod_nam> rshift2 <shift_val> <nod_nam>

node <nod_nam> rshift2 <shift_var> <nod_nam>

node <nod_nam> rshift2 <shift_nod> <nod_nam>

```
ivar sh2       2
...
node y rshift2 2   x
node z rshift2 sh2 x
```

In the example above, the values of “y” and “z” are equal to the value of “x” with the bits shifted by 2 positions to the right and rounded to the nearest integer, a half being rounded to the even integer (round half to even): 1.5 gives 2, 2.5 gives 2, -2.5 gives -2. Input node and shift value (node, variable or constant) must be digital type. Output is digital type. The computation is exact on 64 bits. Same limits as “rshift” for the shift value.

sign: Sign of signal

node <nod_nam> sign <nod_nam>

```
node y sign x
```

Input is digital or analog type. Output node is digital type. Output is +1 for a positive input, -1 for a negative one, 0 for zero. For analog signal, a signal with an absolute value smaller than ***NAPA*** constant EPSILON is considered to be zero.

sine: Sine wave generator

node <nod_nam> sine <[- | +] offset> <ampl> <freq> <[- | +] phase>

```
node y sine 2.5 5.0 freq 0.0
```

The DC offset, amplitude (peak value), frequency and phase can be constants, variables or nodes but not expressions. Phase is expressed in radians. All these parameters are mandatory. The output is analog.

This node is built to support amplitude, frequency and phase modulation.

TIPS:
If there is no need of frequency nor phase modulation, prefer to use the node “*osc*” as trigonometric functions are pretty long to compute and impact badly the speed of the simulations.

square: Square wave source

node <nod_nam> square <[- | +] off> <ampl> <freq> <delay> [<duty_cycle>]

```
node y square 0.0 1.0 freq 0.0 0.33
```

The DC offset, amplitude, frequency, delay and duty cycle can be variables or constants but not nodes nor expressions. All parameters are mandatory, duty cycle excepted. This is quite similar to node kind “sine” (Imagine a rectangularized sine wave) but this node does not allow any node as input.

Default duty cycle is 0.50.

CAUTION:
Be aware that the square wave is sampled at the local sampling frequency. If sampling is not coherent with the transitions of the square wave, effective duty cycle can be modified.

It is the responsibility of the user to keep the delay non-negative.

Take care that this node is not appropriate to make FM modulation: changing the frequency causes a phase discontinuity.

step: Step function source

node <nod_nam> step <[- | +] lvl1> <[- | +] lvl2> <timestep1> [<timestep2>]

```
fs   10.0e6
...
node y1 step  0.0 1.0 1.0e-6
node y2 step -1.0 1.0 1.0e-6 5.0e-6
```

Node will take the value of <lvl1> for current time strictly lower than <timestep1> or strictly larger than optional parameter <timestep2>, and will take value <lvl2> otherwise. The levels and the time steps can be variables but not nodes nor expressions. All parameters must be analog type. Output is analog type.

CAUTION:
Be aware that the step function is sampled at the local sampling frequency. If sampling is not coherent with the transitions of the step, effective time transitions will be modified.

sub: Subtraction element

node <nod_nam> sub <[- | +] nod_nam> <[- | +] nod_nam>

node <nod_nam> sub <[- | +] nod_nam> <[- | +] number>

If “y” is the subtraction of nodes “a” and “b” then:

```
node y sub a b
```

The input nodes *must* be the same type (analog or digital), and output node will be of this type.

sum: N-input summing element

node <nod_nam> sum <[- | +] nod_nam...>

If “y” is the sum of nodes “a”, “b”, and “-c” then:

```
node y sum a b -c
```

The input nodes *must* be the same type (analog or digital), and output node will be of this type. It is important to note that overflow in digital processing depends on the precise architecture of the adder. A strict conformity to the real implementation is therefore necessary (a series of two adders with 2 inputs does not correspond strictly to an adder with 3 inputs).

CAUTION: Nodes “*offset*” and “sum” are not equivalent.

test: ***C*** **condition, output 0 or 1**

node <nod_nam> test <*C*_expression>

```
node y test (1.23 < msig) && ctr
```

Output is 1 if the ***C*** expression is non-zero, 0 otherwise: a Boolean, unlike “ialgebra”, which truncates the value.

Inside the expression, you can mix ***NAPA*** nodes or variables with ***C*** global variables.

See also: “after” qualifier page 81.

toggle: Toggle flip-flop

node <nod_nam> toggle

node <nod_nam> toggle <nod_nam>

```
node y toggle x
```

The input and output nodes are digital type. Output node will toggle (level 0, 1, each time input x is different from zero).

```
node z toggle
```

The output node is digital type. Output node toggles at each loop.

Consider initialization!

At the initialization, output node is equal to "DIGITAL_INI" by default. Use the instruction "*init*" to overwrite the default initialization.

track: Track and hold element

node <nod_nam> track <ctrl_nod_nam> <nod_nam>

```
node y track ctr x
```

The control *must* be digital type. The output node is the same type as the input node (analog or digital). Node "y" follows the input node "x" if the control node is not 0 (see also similar node "*hold*"). It is held otherwise. This node is perfectly equivalent to node "*register*".

Consider initialization!

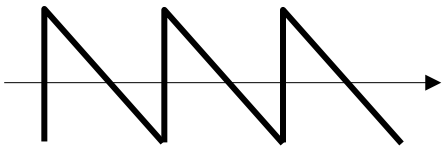
triangle: Triangle wave source

node <nod_nam> triangle <[- | +] off> <ampl> <freq> <delay> [<duty_cycle>]

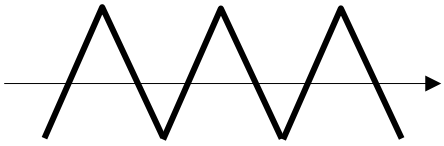
```
node y triangle 0.0 1.0 1000.0 0.0
```

The DC offset, amplitude, frequency and delay can be variables or constants but neither nodes nor expressions. This is quite similar to node kind "sine" (Imagine a triangularized sine wave) but this node does not allow any node as input. An optional parameter controls the duty cycle. This parameter can be a variable or a constant but can be neither a node nor an expression.

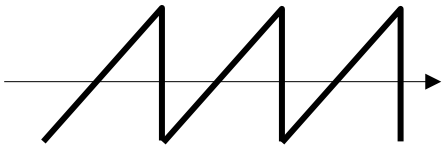
Default duty cycle is 0.50.



Duty Cycle = 0.0



Duty Cycle = 0.5



Duty Cycle = 1.0

CAUTION:
It is the responsibility of the user to keep the delay non-negative and the duty cycle between 0.0 and 1.0.
Take care that this node is not appropriate to make FM modulation: changing the frequency causes a phase discontinuity.

trig: Trigger (dual, positive or negative edge trigger)

node <nod_nam> trig <[- | +] trig_value> <nod_nam> [(dual)]

node <nod_nam> trig <[- | +] trig_value> <nod_nam> (positive)

node <nod_nam> trig <[- | +] trig_value> <nod_nam> (negative)

In the following example, we fix the threshold at 2.5:

```
node y1 trig 2.50 x
node y2 trig 2.50 x (dual)
node y3 trig 2.50 x (positive)
node y4 trig 2.50 x (negative)
```

<trig_value> can be a constant or a variable but not a node nor an expression. Input node can be digital type or analog type. Output node is digital type (value 0 or 1). Default trigger mode is "(*dual)*".

NOTE:
Trigger is disabled during the very first loop to avoid unwanted triggering.

uadc: N-level unsigned A/D converter

node <nod_nam> uadc <num_lev> <nod_in_nam> <nod_ref_nam>

The input node and the reference node must be analog type and the output node will be digital type. The number of levels must be a constant integer of at least 2. This analog-to-digital converter is unsigned (for a signed converter, see "*adc*").

A/D output is clipped when input is outside dynamic input range.

If "y" is the digitized value corresponding to node "x" digitized with 11 allowable levels:

```
node ref dc 1.0
node y uadc 11 x ref
```

input A/D 11 levels	digital output
-∞ ... 0.05	0
0.05 ... 0.15	1
0.15 ... 0.25	2
0.25 ... 0.35	3
0.35 ... 0.45	4
0.45 ... 0.55	5
0.55 ... 0.65	6
0.65 ... 0.75	7
0.75 ... 0.85	8
0.85 ... 0.95	9
0.95 ... ∞	10

udac: N-level unsigned D/A converter

node <nod_nam> udac <num_lev> <nod_in_nam> <nod_ref_nam>

The input node must be digital type, the reference node must be analog and the output node will be analog type. The number of levels must be a constant integer of at least 2. This digital-to-analog converter is unsigned (for a signed converter, see "*dac*").

D/A output is clipped when input is outside dynamic input range.

```
node ref dc 1.0
node y udac 11 x ref
```

input D/A 11 levels	analog output
n <= 0	0.00
1	0.10
2	0.20
3	0.30
4	0.40
5	0.50
6	0.60
7	0.70
8	0.80
9	0.90
n >= 10	1.00

wsum: Weighted sum of N inputs

node <nod_nam> wsum <[- | +] weight> <nod_nam> [... <[- | +] weight> <nod_nam>]

The output node is the same type as the input nodes (analog or digital). The weight factor can be a constant or a variable but not a node nor an expression. Variable or constant type must be consistent with node type.

This node takes N pairs (weight, node).

```
node w  wsum   2.5 x   1.5 y   7.8 z
```

A user's variable can be used as weight factor:

```
dvar g  pow(10.0, ampldB/20.0)
node w  wsum  g x  1.25 y
```

xnor: N-input XNOR element (Boolean logic)

node <nod_nam> xnor <nod_nam...>

```
node y xnor a b c
```

The input nodes *must* be digital type, and output node is always digital type (0 or 1).

xor: N-input XOR element (Boolean logic)

node <nod_nam> xor <nod_nam...>

```
node y xor a b c
```

The input nodes *must* be digital type, and output node is always digital type (0 or 1).

zero: Insertion of zeroes

node <nod_nam> zero <decimation_factor> <decimation_offset> <nod_nam>

```
node y zero 8 3 x
```

The input node can be analog type or digital type. The output node will be the same type as input. The factor and the offset must be digital constants. Output is equal to input once every decimation factor; otherwise it is zero. Offset shifts the output pattern: the input is transmitted at the local steps k (k = 0, 1, ...) where k + 1 modulo the factor equals the offset.

NOTE: Another way to introduce zeroes is to use the node "*relay*".

NAPA Constants and Types

The simulator defines global constants available to the user.

NAPA Constants

Several constants and macro constants are defined at the very beginning of the output ***C*** code and can be used by user's functions or tools:

TRUE	1	
FALSE	0	
YES	1	
NO	0	
START	1LL	
STOP	0LL	
BLOCK_SEPARATOR	"\n\n"	
ANALOG_INI	0.0	
DIGITAL_INI	0LL	
EPSILON	2.000000000000000e-15	
_pi_, _PI_	3.141592653589793238	(double) / (long double)
_pi2_, _PI2_	1.570796326794896619	(double) / (long double)
_pi4_, _PI4_	0.7853981633974483096	(double) / (long double)
_pi8_, _PI8_	0.3926990816987241548	(double) / (long double)
_2pi_, _2PI_	6.283185307179586477	(double) / (long double)
_e_, _E_	2.718281828459045235	(double) / (long double)

Other constants depend on the netlist:

RANDOM_SEED	(contains the seed value which will be used to initialize the pseudo-random generator of the simulator)
SHORT_TITLE	(contains the current first line of the title as defined by "*title*")
TITLE	(contains the current title as defined by "*title*")
FS	(contains the value of the main sampling frequency as defined by "*fs*")
SIM_RATE	(contains the rate of the main loop of the simulator, that is the value defined by "*fs*" modified by the "*interpolate*" or "*decimate*" factor of the first segment)
FSL	(contains current value of the local sampling frequency as defined by "*fs*" instruction modified by "*decimate*", "*interpolate*" and "*nominal*" instructions)
PERIODIC	('TRUE' if periodic sampling is ensured by construction, 'FALSE' in a "drop" segment)
NUM_OF_SEGMENTS	(contains the number of segments contained in the netlist)
NAPA_VERSION	(contains the current ***NAPA*** transpiler version number)
PLATFORM	(contains the name of the platform running the ***NAPA*** transpiler)
SOURCE	(contains the name of the source file processed by the ***NAPA*** transpiler)
ORIGIN	(contains the name of the main netlist without its extension, or, for a stand-alone simulator built with "command_line", the user name with blanks replaced by '_')
CODE	(contains the name of the ***C*** file produced by the ***NAPA*** transpiler)
USER	(contains the user name if any)
CREATED	(contains the date and time of creation of ***C*** code by the ***NAPA*** transpiler, followed by the user name if any)
NAPA_JOB_ID	(contains the date and time of creation of ***C*** code by the ***NAPA*** transpiler, in a compact form)
CWD	(contains the name of the working file directory where the 'exe' code is run)
COMMAND_LINE	(contains the command line the standalone simulator was invoked with, rebuilt from its arguments; defined only when the netlist uses "*command_line*")
COMMAND_PARMS	(contains the pathname of the simulator followed by the names of the parameters declared by "*command_line*", as printed in the usage message)
TERMINATE	(contains the termination condition of the simulation)
ANTITHETIC	('TRUE' for antithetic pseudo-random stream, 'FALSE' otherwise)
SYNCHRONIZE	(global status of the synchronization of the tools)

A few **enum** type constants are defined to control the "IO_MANAGER()" or the "napa_record_manager()" included in the ***C*** code produced by the ***NAPA*** transpiler: their names are obtained with macro function IO_COMMAND_NAME()

UNKNOWN	
CLOSE	
OPENAPPEND	
OPENREAD	
OPENWRITE	
OPENAPPEND_BINARY	(equivalent to OPENAPPEND on UNIX-like operating systems)
OPENREAD_BINARY	(equivalent to OPENREAD on UNIX-like operating systems)
OPENWRITE_BINARY	(equivalent to OPENWRITE on UNIX-like operating systems)
QUERY	
REWIND	
REWRITE	
ALLOCATE	
DELETE	
DEBUG	

Some other macros exist only if the instruction “export” is used:

EXPORT	
V_HEAD	(string containing the name of the first exported variable)
V_NAME	(first exported variable)
V_FORMAT	(print format for the first exported variable)
V_TYPE	(type of the first exported variable)
E_HEAD	(string containing the list of the names of the exported variables)
E_LIST	(list of exported variables)
E_FORMAT	(print format for the list of exported variables)

Constant Types

RANDOM_SEED	Macro constant, long long integer.
SHORT_TITLE	Macro constant, string.
TITLE	Macro constant, string.
NUM_OF_SEGMENTS	Macro constant, integer.
NAPA_VERSION	Macro constant, string.
PLATFORM	Macro constant, string.
ORIGIN	Macro constant, string.
SOURCE	Macro constant, string.

CODE	Macro constant, string.
USER	Macro constant, string.
CREATED	Macro constant, string.
NAPA_JOB_ID	Macro constant, string.
CWD	Macro constant, string.
FSL	Macro constant, double precision.
FS	Macro constant, long double.
SIM_RATE	Macro constant, long double.
COMMAND_LINE	Macro constant, string (with “command_line” only).
COMMAND_PARMS	Macro constant, string (with “command_line” only).
TERMINATE	Macro, Boolean expression.
PERIODIC	Macro constant, ‘TRUE’ or ‘FALSE’.
ANTITHETIC	Macro constant, ‘TRUE’ or ‘FALSE’.
SYNCHRONIZE	Macro constant, YES or NO.
EXPORT	Macro constant, defined or undefined.

CAUTION: Do not change the value of these constants as you could corrupt the simulation.

Generic Types and Output Formats

Node and variable types are available for user-defined procedures:

I_TYPE	***NAPA*** digital type (***C*** long long integer)
R_TYPE	***NAPA*** analog type (***C*** double precision)
***C*_TYPE**	***NAPA*** char type (***C*** char)

Node and variable output formats are available for user-defined procedures:

I_FORMAT	***NAPA*** digital type output format
R_FORMAT	***NAPA*** analog type output format
S_FORMAT	***NAPA*** string type output format

Global Variables

There are a few global variables and macro variables that you should be aware of:

long	**SEGMENT**	*(macro)*
long long	**LOOP_INDEX**	*(macro)*
long long	**ABS_LOOP_INDEX**	*(macro)*
long long	**REL_LOOP_INDEX**	*(macro)*
int	**TERMINATE**	*(macro)*
long long	**TOOL_INDEX**	*(macro)*
double	**TIME**	*(macro)*
double	**ABS_TIME**	*(macro)*
double	**REF_TIME**	*(macro)*
double	**REL_TIME**	*(macro)*
double	**WALL_CLOCK**	*(macro)*
int	**ASSERT_FLAG**	*(macro)*
int	**DUMP_FLAG**	*(macro)*
int	**ERROR_FLAG**	*(macro)*

SEGMENT: the current segment number.

LOOP_INDEX: used in main loop of program, this index counts from 0 to infinity. Not reset at a "*restart*". This counter counts the loops as defined in the netlist. Real number could be higher as the simulator is running at the lowest common multiple value of the segment frequencies.
It is typically used to set a condition to stop the simulation. "LOOP_INDEX" is coded as a 'long long' to avoid a wrap-around.

ABS_LOOP_INDEX: absolute loop counter counting the simulation loops.

REL_LOOP_INDEX: relative loop counter, reset synchronously with "TOOL_INDEX" changes.

TERMINATE: this macro contains the termination condition of the simulator.

TOOL_INDEX: used in main loop of program, this index counts from 0 to infinity the number of tasks executed by the synchronized tools. Not reset at a "*restart*". It is typically used to set a condition to stop the simulation (see also "terminate" 51). It is important to note that this macro variable exists only if a tool is instantiated and synchronization not disabled.

ABS_TIME, TIME: these macros are synonyms and are copies of the internal variable “napa_abs_time”. You can use these in user-defined functions that are functions of time. Do not attempt to change the value of the internal variable “napa_abs_time” as it will cause disastrous side effects to the simulation.

REF_TIME: this macro is a synonym and is a copy of the internal variable “napa_ref_time”. You can use it in user-defined functions that are functions of time. Do not attempt to change the value of the internal variable “napa_ref_time” as it will cause undesirable side effects to the simulation.

REL_TIME: relative value, reset synchronously with “TOOL_INDEX” changes.

WALL_CLOCK: this macro returns the wall clock time elapsed from the beginning of the simulation.

ASSERT_FLAG: This macro, a copy of internal variable “napa_assert_flag” is a flag indicating if an assert condition has triggered an exit call. This exit call is managed by a “*gateway*” instruction if any. Default value is FALSE.

DUMP_FLAG: This macro, a copy of internal variable “napa_dump_flag” is a flag indicating if a dump condition has triggered an exit call. Default value is FALSE.

ERROR_FLAG: This macro, a copy of internal variable “napa_error_flag” is set when an error occurs during the simulation.

CAUTION:
It is forbidden to update any global variables like “napa_abs_time”, “napa_assert_flag”... as you could corrupt the simulation.

NAPA C Functions and Macro Functions

Available C Macro Functions

Several macros are predefined in the generated ***C*** code. These macros can be used in any ***C*** functions defined by the users.

ABS(x)	absolute value of x
CLIP(x, l, h)	'x' clipped between 'l' and 'h'
SIGN(x)	sign of x
MIN(x, y)	minimum of 'x' and 'y'
MAX(x, y)	maximum of 'x' and 'y'
MODULO(x, y)	generalized x modulo y, double type
ISSMALL(x)	test if value is nearly zero
ISNOTSMALL(x)	test if value is not nearly zero
ISEQUAL(x, y)	test equality of 'x' and 'y'
ISNOTEQUAL(x, y)	test inequality of 'x' and 'y'
ISINSIDE(x, l, h)	test if 'x' is inside interval [l,h]
ISOUTSIDE(x, l, h)	test if 'x' is strictly outside interval [l,h]
ISTIME(t)	test if 't' is inside interval [TIME-0.5/FSL, TIME+0.5/FSL)
ISODD(x)	test if 'x' is odd
ISEVEN(x)	test if 'x' is even
ISINTEGER(x)	test if number 'x' is an integer
ISNAN(x)	test for NaN value
LIN2DB(x, r)	linear to dB conversion of 'x', 'r' being the reference
DB2LIN(x, r)	dB to linear conversion of 'x', 'r' being the reference
POW2DB(x, r)	power to dB conversion of 'x', 'r' being the reference
DB2POW(x, r)	dB to power conversion of 'x', 'r' being the reference

LOG(x)	natural logarithm of ‘x’ (‘x’ bounded below by EPSILON)
POW(x, y)	power ‘y’ of ‘x’ (‘x’ bounded below by 0.0)
ROOT(x, y)	root ‘y’ of ‘x’ (‘x’ bounded below by 0.0)
LOG10(x)	logarithm in base 10 of ‘x’ (‘x’ bounded below by EPSILON)
POW10(x)	power ‘x’ of 10
POWEROF2(n)	power ‘n’ of 2, integer type (returns 1 if ‘n’ is not positive; ‘n’ must be smaller than 63)
DEG2RAD(x)	degree to radian conversion
RAD2DEG(x)	radian to degree conversion
SIN(x)	sine of ‘x’ using long double function “sinl()”
COS(x)	cosine of ‘x’ using long double function “cosl()”
SQR(x)	square of ‘x’
SQRT(x)	square root of ‘x’ if ‘x’ is positive, 0.0 otherwise
LENGTH(s)	length of the string ‘s’
B2A(b)	converts a Boolean to the string ‘true’ or ‘false’
RAND_01( )	uniformly distributed pseudo-random number in interval [0..1]
RAND_01_X( )	uniformly distributed pseudo-random number in the open interval (0, 1)
D2I(x)	converts a double float number to a long long integer, rounded to the nearest
I2D(n)	converts a long long integer to a double float number
LINDOMAIN(c, b, e)	value linearly proportional to real ‘c’ between ‘b’ and ‘e’ when ‘c’ is included between 0 and 1
LOGDOMAIN(c, b, e)	value logarithmically proportional to real ‘c’ between ‘b’ and ‘e’ when ‘c’ is included between 0 and 1
LINSWEEP(c, b, e, n)	linear sweep from ‘b’ to ‘e’ controlled by integer ‘c’, ‘n’ points
LOGSWEEP(c, b, e, n)	logarithmic sweep from ‘b’ to ‘e’ controlled by integer ‘c’, ‘n’ points
FSS(n)	local sampling frequency in segment ‘n’
STS(n)	local offset of sampling frequency in segment ‘n’
PS(n)	local period of segment ‘n’
ISDELAYED(f,i)	check if a user-defined function or tool has been delayed
ISOPTION(f,i,o)	check for option in user-defined functions, tools and posts
ISNOTOPTION(f,i)	true if an option was given that no previous ISOPTION recognized
ISPARAMETER(f,i,o)	check for parameter in user-defined functions, tools and posts
GETPARMADDRESS(f,i,o)	get parameter address in user-defined functions, tools and posts
IO_MANAGER(c,f,n,s,t)	manage IO
IO_COMMAND_NAME(n)	get command name
PING(f)	ping function ‘f’ in included header files

TIMER(n)	pause execution for 'n' seconds

Available C Functions

Several functions are predefined in the generic header file "napa.hdr". They adapt the simulator to a particular environment. These functions *must* exist but they can be empty.

```
void  napa_init(void)
void  napa_close(void)
```

Several other functions may be defined in the generated ***C*** code.

```
void  napa_control_init(void)
void  napa_end(void)
void  napa_exit(int)
void  napa_reset_nodes(void)
void  napa_reset_variables(void)
void  napa_array_setup(int)
void  napa_record_setup(int)
void napa_ping(int)
I_TYPE  napa_default_seed(I_TYPE)
I_TYPE  napa_rand_integer(I_TYPE)
I_TYPE  napa_default_root(I_TYPE)
I_TYPE  napa_default_rand(void)
double *napa_check_for_option(const char*, long, const char*, int)
IO_COMMAND  napa_record_manager(IO_COMMAND,
   DATA_RECORD*, long, long, long, const char*)
```

Macros *"ISDELAYED()", "ISOPTION()", "ISNOTOPTION()"* and the function "**napa**_*check_for_option( )*" are used internally by user-defined functions to handle options.

It is sometimes useful, for debugging purposes, to publish the content of the arrays and records defined by the ***NAPA*** instruction "*array*". Use ***NAPA*** instruction "*call*":

File "my_NAPA_file.nap"

```
header <napa.hdr>
fs ...
...
call void napa_record_setup(DEBUG)     // for arrays of pointers
call void napa_array_setup(DEBUG)      // for RAM and ROM
...
terminate ...
```

A specific function providing the management of the IO stream (open, close IO files) is also defined in the generated ***C*** code. This resource manager is used by the "*output*" instruction and is available for the user-defined functions or tools.

IO_COMMAND napa_IO_manager(IO_COMMAND c, FILE **fp, char *fl, char *s, char *t)

which users call through the macro "IO_MANAGER(...)".

All standard mathematical ***ANSI-C*** functions are available (as defined in ***C*** header file "math.hdr").

Other functions are defined in the generic "napa.hdr" header file (see this file for details). This header could be configured to adapt the compiler to a particular platform. These functions are then available for the users (see file "napa.hdr" in generic header directory for more information).

User C Functions

The user can define other ***C*** functions inside a ***NAPA*** header and include them as resources using the header instruction. If a function is used explicitly in a ***NAPA*** netlist and is not a standard ***C*** mathematical function as defined in the ***ANSI-C*** header file "math.hdr", the transpiler defines the macro COMPILE_<function>.

If the expression "a * foo(b)" is used in the terminate condition of the ***NAPA*** netlist, ***NAPA*** writes the ***C*** code:

```
#define COMPILE_foo
```

If the function is not used explicitly, but is hidden inside some header files, ***NAPA*** has no way to detect it and no directive will be issued.

These macro definitions help the user to compile the headers selectively using the classical:

```
#ifdef COMPILE_foo
...
#endif
```

to enclose the code relative to the function to be compiled. This is the simplest mechanism offered by NAPA to handle user-defined functions. More sophisticated concepts exist and will be explained in the next chapter.

User-Defined Functions and Tools

The Concept

There are several ways to write user-defined functions. Some of them are plain ***C*** functions used as regular ***C*** functions inside "*ialgebra*", "*dalgebra*", "*algebra*", "*test*", "*drop*", ... instructions. None requires special care.

Others are more sophisticated and are intended to be used as "*dtool*", "*itool*", "*duser*" or "*iuser*". As they interact directly with the ***NAPA*** transpiler some guidelines have to be followed. Each time such an instruction is instantiated, the ***NAPA*** transpiler performs several other actions. For example, for an "itool" called "foo", instantiated twice, with three arguments "name", "node" and "num":

This tool is described in a header file "foo.hdr" as:

```
I_TYPE   itool_foo_03(char* fnam, double nod, long num, int id)

void check_itool_foo_03(char* fnam, double nod, long num, int id)
void  init_itool_foo_03(char* fnam, double nod, long num, int id)
void reset_itool_foo_03(char* fnam, double nod, long num, int id)
void close_itool_foo_03(char* fnam, double nod, long num, int id)
```

We will use this ***NAPA*** netlist as example:

File "my_NAPA_file.nap"

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

fs    1.0e6
ivar  n   1000
ivar  m   n/2
...
node  in  ...
node out  ...
node   a  itool  foo  "result.out"  in  n
node   b  itool  foo  "stdout"     out  m
...
terminate ...
```

The ***C*** code produced by the ***NAPA*** transpiler corresponding to the instantiation of these tools is:

File “my_NAPA_file.c”

```
...
#define COMPILE_itool_foo   2
...
#include "/Simulate/NapaDos/Hdr/napa.hdr"
#include "foo.hdr"
...
int main(void) {
  ...
  check_itool_foo_03("result.out", dnode_in,  ivar_n, 0);
  check_itool_foo_03("stdout",     dnode_out, ivar_m, 1);
  ...
  init_itool_foo_03("result.out", dnode_in,  ivar_n, 0);
  init_itool_foo_03("stdout",     dnode_out, ivar_m, 1);
  ...
  do {
    napa_abs_time = napa_abs_loop * 1.000000000000000e-06L;
    ...
    dnode_in  = ... ;
    inode_a   = itool_foo_03("result.out", dnode_in,  ivar_n, 0);
    ...
    dnode_out = ... ;
    inode_b   = itool_foo_03("stdout",     dnode_out, ivar_m, 1);
    ...
  } while (!TERMINATE);
  ...
  close_itool_foo_03("result.out", dnode_in,  ivar_n, 0);
  close_itool_foo_03("stdout",     dnode_out, ivar_m, 1);
  ...
  return EXIT_SUCCESS;
}
...
```

The macro “COMPILE_itool_foo” and the function calls correspond exactly to the ***NAPA*** netlist. A full example of the ***C*** code of such a tool is shown at the end of this chapter.

Postprocessing of time domain output (from instruction “*output*”) or output of a user’s defined tool is also possible. Instruction “*post*” is placed directly after the output or the tool. Each time such an instruction is instantiated, the ***NAPA*** transpiler performs several other actions. For example, for a “*post*” called “analyze” of the output of a “tsnr” tool, postprocess instantiated one time, with two arguments “file_out” and “num”:

Define a macro:

```
#define  COMPILE_post_analyze   1
```

where ‘1’ is the number of instantiations of “analyze”. For each instantiation, several functions are called; the last argument is the instance number, from 0 (here the first and only instance):

Preparation at the initialization of the simulation:

```
prepare_post_analyze_02("tsnr", "prev_out", "file_out", num, 0)
```

Execution at the end of the simulation:

```
execute_post_analyze_02("tsnr", "prev_out", "file_out", num, 0)
```

NAPA automatically passes the origin of the data (output or tool type), the output file to process and places them in front of the parameters provided by the user. These functions will be placed respectively after the tool initialization functions and tool close functions.

It is mandatory that the functions “prepare...” and “execute...” are predefined (empty or not). The number of user’s arguments (here 2) is part of the resulting name of the functions.

Tool Synchronization

Simpler user-defined functions (“*duser*” and “*iuser*”) are reserved to create new primitives to describe the network. More sophisticated user-defined functions (“*itool*” and “*dtool*”) are used to describe analysis tools (like FFT, TSNR, Linearity, Distortion ...). These tools play a very important role in the simulation. As the same analysis can be repeated during the same simulation, and as several analysis tools may be used, it is particularly important that these tools can be automatically synchronized.

NAPA provides a simple synchronization mechanism based on messages, through a pointer (“napa_msg”) to a structure called “mailbox”. The simulator handles this mechanism, transparently for the user. These pointers point to the mailbox reserved for each instantiation of the tools. The simulator, by setting the message corresponding to “napa_`msg->i`”, indicates to the tool that it is authorized to process the signal. The tool itself resets the message corresponding to the same pointer at the end of an analysis. To send a status of its analysis back to the simulator, the tool uses a second message corresponding to “napa_`msg->o`”. A few other variables are stored in the mailbox. The variable “napa_msg->n” for instance is set to 0 during initialization and is available to store an integer value. It is currently used to store the state of the state machine describing the cycle of the tools if any. The simulator analyzes the resulting messages coming from the tools and sends an adequate message to start or to hold new analysis. Next paragraph shows an example of a synchronizable tool. A nice feature is that a tool without synchronization code will not block the simulation nor will be affected by the synchronization messages.

An Example of Tool

Examples of “*duser*” and “*iuser*” functions have already been described (see corresponding nodes). Here is a template of the most sophisticated structure of ***NAPA***: a synchronizable “SMART TOOL” able to check, collect and process data by itself, able to check, open, fill and close an output file, while controlling the simulation flow. You can of course imagine other ways to build user functions; keep the headers consistent with each other.

This function, called here “template”, is only an EXAMPLE you can use to build your own user-defined tool. Variations are of course possible and will depend on the application. Consult the generic ***NAPA*** header library for other examples. This tool uses resource managers to simplify the code. See next paragraph for details.

File “template.hdr”

```
#ifndef __TEMPLATE_HDR__
#define __TEMPLATE_HDR__

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

/*   EXAMPLE OF ITOOL FUNCTION ACCUMULATING NUM ANALOG DATA BEFORE PROCESSING,      */
/*   PROCESSING AND OUTPUT BY "PACKETS" IN A FILE.                                  */
/*   SEVERAL TOOLS CAN BE INSTANTIATED INSIDE A SIMULATION.                         */
```

```
/*   THIS ITOOL USES A DYNAMIC MEMORY ALLOCATION MANAGER SUPPOSED TO BE DEFINED IN  */
/*   FILE "../Resource/dm.hdr": 'DA_resources_manager()' AND THE IO RESOURCES       */
/*   MANAGER 'IO_MANAGER()' DEFINED IN THE OUTPUT C SOURCE.                         */

/*   USAGE:  node <no> itool template <"filnam"> <ni> ... <num>                     */

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

/*   THIS TOOL IS SYNCHRONIZABLE.                                                   */


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

/* NAPA itool defined functions:  "template"                                        */

/* ** FUNCTION HEAD TRIMMING,  3 ARGUMENTS FUNCTION ***************************** */

/* use the ANSI-C preprocessor to screen, complete or modify the parameter list     */

#define check_itool_template_03(a,b,c,d)  check_itool_template(a,b,#b,c,d)
#define reset_itool_template_03(a,b,c,d)  reset_itool_template(a,b,#b,c,d)
#define  init_itool_template_03(a,b,c,d)   init_itool_template(a,b,#b,c,d)
#define close_itool_template_03(a,b,c,d)  close_itool_template(a,b,#b,c,d)
#define       itool_template_03(a,b,c,d)        itool_template(a,b,#b,c,d)


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

void check_itool_template(char*, double, char*, long, int);
void reset_itool_template(char*, double, char*, long, int);
void  init_itool_template(char*, double, char*, long, int);
void close_itool_template(char*, double, char*, long, int);
I_TYPE     itool_template(char*, double, char*, long, int);

void template_compute_and_print(long, int);
void template_my_function(long, int);

...   all global function names must begin with  "template_" to avoid name collision

/* ** PING INFORMATION ********************************************************** */

#define itool_template_IS_REGISTERED

PING(itool_template);
...   tools, when registered and PING() macro called, are able to respond to a 'ping'
      command from the simulator. This is optional but highly recommended.

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

#include "../Resource/dm.hdr"                     /* dynamic memory manager        */


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

#ifdef COMPILE_itool_template                     /* compilation control directive */

#ifndef TEMPLATE_MAX
#define TEMPLATE_MAX   MAX_NUMBER                 /* max number of tool "template" */
#endif


#define TEMPLATE_NUM_INITIAL_IS_REGISTERED

#if defined(TEMPLATE_NUM_INITIAL_IS_EMPTY) || !defined(TEMPLATE_NUM_INITIAL)
  #undef  TEMPLATE_NUM_INITIAL
  #define TEMPLATE_NUM_INITIAL       0LL
```

```
#endif
...


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

double *template_in_array[TEMPLATE_MAX];
double *template_out_array[TEMPLATE_MAX];

long    template_num[TEMPLATE_MAX];               /* store counter                 */
long    template_np[TEMPLATE_MAX];                /* number of points              */
FILE   *template_fp[TEMPLATE_MAX];                /* output file pointer           */

...   all global variable names must begin with  "template_" to avoid name collision


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

void check_itool_template(char *filnam, double x, char *name, long acc, int id) {
  if (id >= TEMPLATE_MAX) {
    fprintf(stderr, "NAPA run time Error: (template[%d])\n", id);
    fprintf(stderr, " Only %d TEMPLATE tools are allowed in a netlist\n", TEMPLATE_MAX);
    napa_exit(EXIT_FAILURE);
  }
  if (acc < 8L) {
    fprintf(stderr, "NAPA run time Error: (template[%d])\n", id);
    fprintf(stderr, " A minimum of 8 points must be accumulated\n");
    napa_exit(EXIT_FAILURE);
  }
  return;
}


void reset_itool_template(char *filnam, double x, char *name, long acc, int id) {
  ...
  return;
}


void init_itool_template(char *filnam, double x, char *name, long acc, int id) {
  template_np[id] = acc;                 /* FIXED as memory is allocated accordingly */
  DA_MANAGER(ALLOCATE,  &(template_in_array[id]),  acc,         "template");
  DA_MANAGER(ALLOCATE,  &(template_out_array[id]), acc,         "template");
  IO_MANAGER(OPENWRITE, &(template_fp[id]),     filnam, ".out", "template");
#ifndef NO_BANNER                                    /* output file banner control  */
  fprintf(template_fp[id], "# %s\n", SHORT_TITLE);
  fprintf(template_fp[id], "# (tool                ) template\n");
  fprintf(template_fp[id], "# (transpiler version  ) %s\n",    NAPA_VERSION);
  fprintf(template_fp[id], "# (source file         ) %s\n",    SOURCE);
  fprintf(template_fp[id], "# (random seed         ) %lld\n",  RANDOM_SEED);
  fprintf(template_fp[id], "# (signal              ) %s\n", name);
  fprintf(template_fp[id], "# (number of samples   ) %ld\n",   acc);
  fprintf(template_fp[id], "# (sampling frequency  ) %g Hz\n", FSL);
  fprintf(template_fp[id], "# \n");
  fprintf(template_fp[id], "# \n");          /* lines beginning with "# " are        */
  fprintf(template_fp[id], "# \n");          /* ignored by software such as GNUPLOT  */
  fprintf(template_fp[id], "# \n");
  fprintf(template_fp[id], "# \n");
  fprintf(template_fp[id], "# %s\n", CREATED);
  fprintf(template_fp[id], "# packet ...");
#ifdef EXPORT
  fprintf(template_fp[id], "%s", E_HEAD);    /* column header for exported variables */
#endif
  fprintf(template_fp[id], "\n");
  fflush(template_fp[id]);
#endif
  napa_msg->n = TOOL_WAIT;                   /* initialize the tool state machine    */
  return;
}
```

```
void close_itool_template(char *filnam, double x, char *name, long acc, int id) {
  DA_MANAGER(FREE,  &(template_in_array[id]),  template_np[id], "template");
  DA_MANAGER(FREE,  &(template_out_array[id]), template_np[id], "template");
  IO_MANAGER(CLOSE, &(template_fp[id]),         filnam, ".out", "template");
  return;
}


I_TYPE itool_template(char *filnam, double x, char *name, long acc, int id){
  static long packet[TEMPLATE_MAX];
  long *num;
  long *pak;
  num = &(template_num[id]);
  pak = &(packet[id]);
  switch (napa_msg->n) {
  case TOOL_WAIT:
    if (napa_msg->i == FALSE) {
      break;
    }
    *num = -TEMPLATE_NUM_INITIAL;
    reset_itool_template(filnam, x, name, acc, id);
    napa_msg->n = TOOL_COUNTDOWN;
  case TOOL_COUNTDOWN:
    if (*num < 0L) {
      (*num)++;
      break;
    }
    napa_msg->n = TOOL_ACCUMUL;
  case TOOL_ACCUMUL:
    template_in_array[id][*num] = x;                      /* sampling input variable */
    (*num)++;
    if (*num < template_np[id]) {
      break;
    }
    napa_msg->n = TOOL_COMPUTE;
  case TOOL_COMPUTE:
    template_compute_and_print(*pak, id);
    fflush(template_fp[id]);
    (*pak)++;
    napa_msg->i = FALSE;
    napa_msg->n = TOOL_WAIT;
    break;
  }
  napa_msg->o = *pak;
  return napa_msg->o;
}

void template_compute_and_print(long packet, int id) {
  ...
  template_my_function(packet, id);
  ...
  fprintf(template_fp[id], " %3ld % 14e ", packet, ...);
#ifdef EXPORT
  fprintf(template_fp[id], E_FORMAT, E_LIST);             /* exported variables      */
#endif
  fprintf(template_fp[id], "\n");
  return;
}

void template_my_function(long packet, int id) {
  ...
  return;
}

...
#endif                                                    /* COMPILE_itool_template  */

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

#endif                                                         /* __TEMPLATE_HDR__   */
```

You can then use the "*itool*" "template" in a ***NAPA*** netlist.

For example:

```
header <napa.hdr>
header "/user/jdoe/template.hdr"                // CALL RESOURCES
...
node ctr itool template "ofile.out" s1 128      // USE THE SMART TOOL
tool template "stdout" a4 1000                  // short form of the same
...
```

NAPA now handles the whole process. ***NAPA*** keeps the instance identifiers: give every parameter except "id".

Use, if you want, the node "ctr" or the macro "TOOL_INDEX" pointing to the number of batches of tasks the tool has already completed to control the variable(s) driving your simulation, like the amplitude for a TSNR analysis, the frequency for a frequency tracking analysis, etc... See appendix D, page 167, for a complete example.

It is possible to build more complex tools based on this principle. In these cases, the previous template does not apply and must be reworked. But up to now, this template has been used in 80% of the existing applications.

Resource Managers

Several resource managers help the users to create their own user-defined function or tool. One of these resource managers is part of the ***C*** source produced by the ***NAPA*** transpiler: "*napa_IO_manager(...)*" and its corresponding macro function "*IO_MANAGER(...)*". This resource manager is responsible for opening and closing an IO stream, handling necessary verifications for the user (collision, access...).

```
USAGE: to open or close an IO stream

(void) IO_MANAGER(CLOSE,      &fp,     filnam,   "suffix", "tag")
(void) IO_MANAGER(OPENAPPEND, &fp,     filnam,   "suffix", "tag")
(void) IO_MANAGER(OPENREAD,   &fp,     filnam,   "suffix", "tag")
(void) IO_MANAGER(OPENWRITE,  &fp,     filnam,   "suffix", "tag")
(void) IO_MANAGER(REWIND,     &fp,     filnam,   "suffix", "tag")
(void) IO_MANAGER(REWRITE,    &fp,     filnam,   "suffix", "tag")
(void) IO_MANAGER(DELETE,     &fp,     filnam,   "suffix", "tag")
(void) IO_MANAGER(DEBUG,      &fp,     filnam,   "suffix", "tag")
 n  =  IO_MANAGER(QUERY,      &fp,     filnam,   "suffix", "tag")

Where "fp" is a FILE pointer (passed by its address "&fp"), "filnam" is the
name of the stream ('stdout', 'stderr', 'stdin' or a file pathname), "tag"
is a string identifying the user-defined function or tool being used to
document error messages if any. "suffix" is the default suffix of the file
to be processed, the resource manager will add automatically this suffix to
the file name if it is missing.
```

There is another command, used by the simulator itself:

```
(void) IO_MANAGER(-1,       NULL, NULL, NULL, NULL)
```

It releases allocated memory of unused I/Os and compacts internal registers.

Other resource managers are currently part of the standard header library (“hdr”). The use of these resource managers is not mandatory but they are built to ease and secure the writing of complex user-defined functions or tools.

See the generic header library for other examples.

APPENDIX A

NAPA Simulation Flow. Order of Execution

Initialization

Call ***NAPA*** initialization function.
Set up the tool synchronization if any.
Initialize user's variables and call “init” functions.
Open I/O files if any.
Initialize nodes not yet initialized by “init”.
Call user's and tool's check functions if any.
Call user's and tool's initialization functions if any.
Prepare postprocessing functions if any.
Write the banner of the output files if any.
Load initialization file if any.

Main Loop

Repeat the loop...

Compute current time
Compute the segment processing conditions

Processing of **block 1** (update variables) *segment after segment*

Assert statement if any.
Update user's variables and calls, input variables if any.

Processing of **block 2** (update nodes) *segment after segment*
Update nodes if any.

Processing of **block 3** (time domain output) *segment after segment*
Output values if any.

Update synchronization messages if any.

Dump internal states if required.
Increment loop index....

then check the termination condition and exit the loop if requested.

Termination

Close I/O files if any.
Call the close functions of the user functions and tools, if any.

Execute postprocessing functions if any.

Call the ***NAPA*** close function.

Program exit.

RECOMMENDATION:
*If you have any doubt concerning the simulation flow, the only file to consult is the **C** code file produced by the **NAPA** transpiler.*

APPENDIX B

NAPA Reserved Identifiers

NAPA writes ***C***: some identifiers are therefore forbidden, others discouraged, as node or variable names. The ***NAPA*** transpiler issues warning or error messages.

Please note that the *NAPA* transpiler cannot verify that you are misusing *C* keywords.

In the following list, an "E" means that an error is produced and ***NAPA*** transpilation exited if you use the keyword as a node or variable identifier. A "**W**" means that a warning message is produced.

The classification between error and warning was chosen from an estimation of the risk caused by an identifier misunderstanding.

$_

$assert$	internal use	E
$call$	internal use	E
$init$	internal use	E
$null$	internal use	E
$restart$	internal use	E

A_

abs	***C*** function	E
ABS	macro function	E
ABS_LOOP_INDEX	macro	E
ABS_TIME	macro	E
acos	***C*** function	E
acosh	***C*** function	E
acosl	***C*** function	E
adc	node	W
after	qualifier	E
algebra	node	W
alias	instruction	W
ALLOCATE	enum type	E
alu	node	W
analog	qualifier	W
ANALOG_INI	macro	E
and	node	W
ANTITHETIC	macro	E
arithmetic	qualifier	W
array	instruction	W
asin	***C*** function	E
asinh	***C*** function	E

asinl	***C*** function	E
assert	instruction	W
ASSERT_FLAG	macro	E
atan	***C*** function	E
atan2	***C*** function	E
atan2l	***C*** function	E
atanh	***C*** function	E
atanl	***C*** function	E
average	node	W

B_

B2A	macro function	E
before	qualifier	E
BLOCK_SEPARATOR	macro	E
bshift	node	W
btoi	node	W
buffer	node	W
bwand	node	W
bwbuffer	node	W
bwinv	node	W
bwnand	node	W
bwnor	node	W
bwnot	node	W
bwor	node	W
bwxnor	node	W
bwxor	node	W

C_

C_TYPE	typedef	E
call	instruction	W
ceil	***C*** function	E
cell	node	W
cell_interface	interface	W
CELLS_LIB	macro	E
change	node	W
char	***C*** keyword	E
clip	node	W
CLIP	macro function	E
clock	node	W
CLOSE	enum type	E
CODE	macro	E
command_line	instruction	W
COMMAND_LINE	macro	E
COMMAND_PARMS	macro	E
comment	instruction	W
COMMENT	macro	E
comp	node	W
COMPILE_...	reserved prefix	E
const	node	W
constant	qualifier	W
copy	node	W
cos	***C*** function	E
COS	macro	E
cosh	***C*** function	E
cosine	node	W
cosl	***C*** function	E
CREATED	macro	E

D_

D2I	macro function	E
dac	node	W
dalgebra	node	W
data	instruction	W
data_interface	interface	W
DATA_RECORD	typedef	E
DB2LIN	macro function	E
DB2POW	macro function	E
dc	node	W
debug	instruction	W
DEBUG	enum type	E
DEBUG_...	reserved prefix	E
decimate	instruction	W
declare	instruction	W
DEG2RAD	macro function	E
delay	node	W
DELETE	enum type	E
differentiator	node	W
digital	qualifier	W
DIGITAL_INI	macro	E
directive	instruction	W
div	node	W
double	***C*** keyword	E
drop	instruction	W
dtoi	node	W
dtool	node	W
dual	qualifier	W
dump	instruction	W
DUMP_FLAG	macro	E
duser	node	W
dvar	instruction	W

E_

_e_, _E_	global constants	E
E_FORMAT	macro	E
E_HEAD	macro	E
E_LIST	macro	E
EPSILON	global constant	E
equal	node	W
erf	***C*** function	E
erfc	***C*** function	E
error	instruction	W
ERROR_FLAG	macro	E
event	instruction	W
exp	***C*** function	E
expand	qualifier	W
export	instruction	W
EXPORT	macro	E

F_

fabs	***C*** function	E
false	global constant	E
FALSE	macro	E
float	***C*** keyword	E
floor	***C*** function	E

fmod	***C*** function	E
format	instruction	W
fprintf	***C*** function	E
FREE	enum type	E
fs	instruction	E
FS	global constant	E
FSL	macro	E
FSS	macro function	E
fzand	node	W
fzbuffer	node	W
fzinv	node	W
fznand	node	W
fznor	node	W
fznot	node	W
fzor	node	W
fzxnor	node	W
fzxor	node	W
G_		
gain	node	W
ganging	instruction	W
gateway	instruction	W
generator	node	W
GENERATORS_LIB	macro	E
geometric	qualifier	W
GETPARMADDRESS	macro function	E
Ground	automatic node	E
H_		
harmonic	qualifier	W
header	instruction	W
HEADERS_LIB	macro	E
hex	qualifier	W
hold	node	W
I_		
I2D	macro function	E
I_FORMAT	macro	E
I_TYPE	typedef	E
ialgebra	node	W
init	instruction	W
inject	instruction	W
input	instruction	W
int	***C*** keyword	E
integer	qualifier	W
integrator	node	W
interface	instruction	W
interlude	instruction	W
interpolate	instruction	W
inv	node	W
IO_COMMAND_NAME	macro function	E
IO_RECORD	typedef	E
IONAME	macro function	E
ISDELAYED	macro function	E
ISEQUAL	macro function	E
ISEVEN	macro function	E
ISINSIDE	macro function	E

ISINTEGER	macro function	E
ISNAN	macro function	E
ISNOTEQUAL	macro function	E
ISNOTOPTION	macro function	E
ISNOTSMALL	macro function	E
ISODD	macro function	E
ISOPTION	macro function	E
ISOUTSIDE	macro function	E
ISPARAMETER	macro function	E
ISSMALL	macro function	E
ISTIME	macro function	E
itob	node	W
itod	node	W
itool	node	W
iuser	node	W
ivar	instruction	W

L_

labs	***C*** function	E
latch	node	W
LENGTH	macro function	E
LIN2DB	macro function	E
LINDOMAIN	macro function	E
LINSWEEP	macro function	E
load	instruction	W
log	***C*** function	E
LOG	macro function	E
log10	***C*** function	E
LOG10	macro function	E
log2	***C*** function	E
LOGDOMAIN	macro function	E
LOGSWEEP	macro function	E
long	***C*** keyword	E
LOOP_INDEX	macro	E
lshift	node	W

M_

MAILBOX	macro	E
MAILBOX_RECORD	typedef	E
max	node	W
MAX	macro function	E
merge	node	W
min	node	W
MIN	macro function	E
mod	node	W
MODULO	macro function	E
muller	node	W
mux	node	W

N_

nand	node	W
napa_...	***C*** simulator identifiers	E
napa_...	*C* simulator macros	E
napa_version	instruction	W
negative	qualifier	W
new	qualifier	W
no	qualifier	E

NO	macro	E
node	instruction	W
noexpand	qualifier	W
noise	node	W
nominal	instruction	W
nor	node	W
normal	qualifier	W
not	node	W
NUM_OF_SEGMENTS	macro	E
NUM_OF_TIME_OUTPUTS	macro	E
O_		
offset	node	W
One	automatic node	E
opcode	instruction	W
OPENAPPEND	enum type	E
OPENAPPEND_BINARY	enum type	E
OPENREAD	enum type	E
OPENREAD_BINARY	enum type	E
OPENWRITE	enum type	E
OPENWRITE_BINARY	enum type	E
or	node	W
ORIGIN	macro	E
osc	node	W
output	instruction	W
PING	macro function	E
PING_FAIL	macro function	E
post	instruction	W
P_		
_2pi_, _2PI_	global constants	E
_pi_, _PI_	global constants	E
_pi2_, _PI2_	global constants	E
_pi4_, _PI4_	global constants	E
_pi8_, _PI8_	global constants	E
P_TYPE	typedef	E
PACKET	macro	E
PERIODIC	macro	E
ping	instruction	W
PLATFORM	macro	E
pointer	qualifier	W
poly	node	W
positive	qualifier	W
pow	***C*** function	E
POW	macro function	E
POW10	macro function	E
POW2DB	macro function	E
POW2LIN	macro function	E
POWEROF2	macro function	E
prod	node	W
PS	macro function	E
Q_		
quant	node	W
QUERY	enum type	E
R_		

R_FORMAT	macro	E
R_TYPE	typedef	E
RAD2DEG	macro function	E
ram	node	W
ram2	node	W
RAM2_I_RECORD	typedef	E
RAM2_R_RECORD	typedef	E
RAM_I_RECORD	typedef	E
RAM_R_RECORD	typedef	E
RAND_01	macro function	E
RAND_01_X	macro function	E
random_seed	instruction	W
RANDOM_SEED	macro	E
rect	node	W
REF_TIME	macro	E
register	node	W
REL_LOOP_INDEX	macro	E
REL_TIME	macro	E
relay	node	W
RESET	enum type	E
restart	instruction	W
REWIND	enum type	E
REWRITE	enum type	E
rip	node	W
rms	qualifier	W
rom	node	W
rom2	node	W
ROM2_I_RECORD	typedef	E
ROM2_R_RECORD	typedef	E
ROM_I_RECORD	typedef	E
ROM_R_RECORD	typedef	E
ROOT	macro function	E
rshift	node	W
rshift1	node	W
rshift2	node	W

S_

S_FORMAT	macro	E
sampling	instruction	W
SEGMENT	macro	E
short	***C*** keyword	E
SHORT_TITLE	macro	E
sign	node	W
SIGN	macro function	E
SIM_RATE	macro	E
sin	***C*** function	E
SIN	macro function	E
sine	node	W
sinh	***C*** function	E
sinl	***C*** function	E
sizeof	***C*** keyword	E
SOURCE	macro	E
SQR	macro function	E
sqrt	***C*** function	E
SQRT	macro function	E
square	node	W
START	macro	E

step	node	W
STL	macro	E
STOP	macro	E
string	instruction & qualifier	W
STS	macro function	E
stuck	instruction	W
sub	node	W
sum	node	W
synchronize	instruction	W
SYNCHRONIZE	macro	E
T_		
tan	***C*** function	E
tanh	***C*** function	E
tanl	***C*** function	E
terminate	instruction	W
TERMINATE	macro	E
test	node	W
TIME	macro	E
TIMER	macro function	E
TITLE	macro	E
title	instruction	W
toggle	node	W
tool	instruction	E
TOOL_INDEX	macro	E
track	node	W
triangle	node	W
trig	node	W
true	global constant	E
TRUE	macro	E
ts	instruction	W
U_		
uadc	node	W
udac	node	W
uniform	qualifier	W
UNKNOWN	enum type	E
unsigned	***C*** keyword	E
update	instruction	W
USER	macro	E
V_		
V_FORMAT	macro	E
V_HEAD	macro	E
V_NAME	macro	E
V_TYPE	typedef	E
void	instruction	E
W_		
WALL_CLOCK	macro	E
warning	instruction	W
when	qualifier	E
with	qualifier	E
wsum	node	W
X_		
X_FORMAT	macro	E

xnor	node	W
xor	node	W
Y_		
yes	qualifier	E
YES	macro	E
Z_		
zero	node	W
Zero	automatic node	E

APPENDIX C

NAPA File Naming Recommendation

To share information between users, a common file naming is recommended:

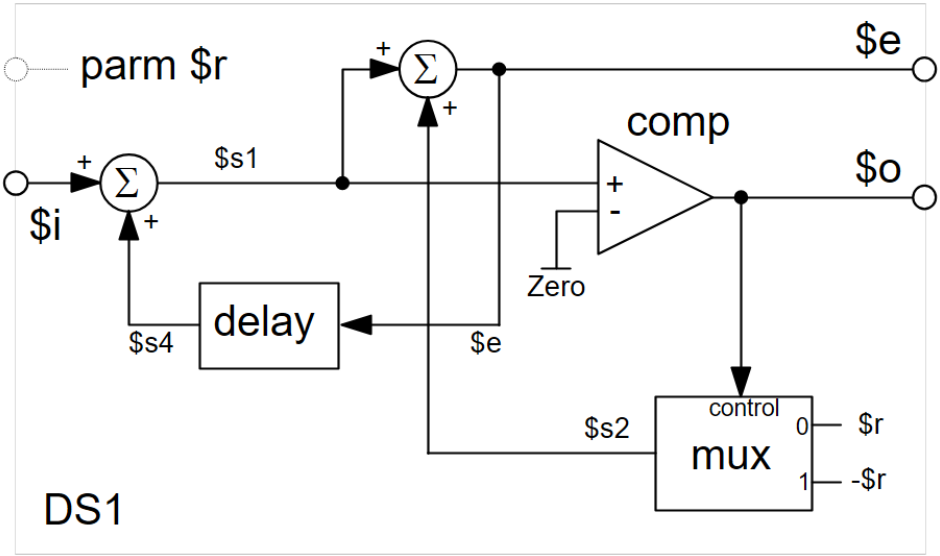
NAPA main netlist file	xxxx**.nap**	
NAPA netlist of a cell	xxxx**.net**	
NAPA schematic	xxxx.s**ch**	(Schematic Editor)
MAC output	xxxx**.tmp**	(***NAPA*** preprocessor)
MAXIMA package	xxxx.**mac**	
C code generated by NAPA	xxxx**.c**	
Executable binary code	xxxx**.bin**	(Linux, Unix)
	xxxx**.exe**	(Windows)
NAPA header file and user's profile	xxxx**.hdr**	
NAPA data cells	xxxx.**dat**	
Simulation input file	xxxx**.in**	
Simulation output file	xxxx**.out**	
NAPA log file	xxxx**.log**	
NAPA dump file	xxxx**.dmp**	
NAPA load file	xxxx**.ini**	
NAPA ping file	xxxx.ping	
NAPA RAM initialization file	xxxx**.ram**	
NAPA ROM description file	xxxx**.rom**	
NAPA generator (executable)	no suffix	(Linux, Unix)
	xxxx**.exe**	(Windows)
NAPA generator (output cell)	xxxx**.gen**	
Graphics front end directives	xxxx**.plt**	(Gnuplot...)

APPENDIX D

NAPA Netlist Example

This example is using a user's SMART TOOL "*tsnr*" currently defined in the header library file "fft2.hdr". As many other tools, "*tsnr*" output node is used to analyze and control the simulation. This tool is accumulating the data automatically inside its internal arrays. During data accumulation, output node is kept at a constant integer value (starting at 0). As soon as specified number of data samples is reached, "*tsnr*" computes FFT and "*tsnr*", stores the results in the specified output file or stream, resets its internal arrays, increments the output node by 1, and begins the accumulation of the next data. Output node is used to control the amplitude of the input signal of the circuit by user's variables "ampldb" and "ampl".

The circuit is based on a cell instantiated twice:



The corresponding netlist of this cell is placed in file "ds1.net":

File "ds1.net"

```
cell_interface   $o   $i  $e  $r

#* DS1, subcell of a digital Sigma-Delta simulator


declare (digital)  $i $r             // recommended for documentation and error tracking
```

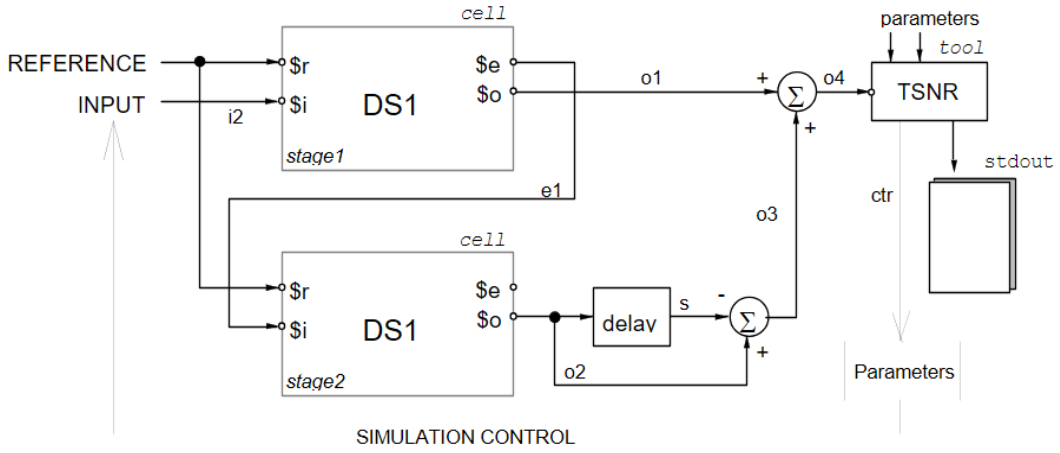
```
#*  $o   output of the cell
#*  $i   input of the cell
#*  $e   quantization error
#*  $r   digital reference

node $rf  dc (digital) $r

node $rm  copy -$rf              // negative reference
node $rp  copy  $rf              // positive reference

node $s1  sum   $i   $s4
node $o   comp  $s1  Zero
node $s2  mux   $o   $rp  $rm
node $e   sum   $s1  $s2
node $s4  delay $e
```

The complete circuit uses this cell to build a ΣΔ modulator. A SMART TOOL "*tsnr*" is connected to analyze the circuit behavior and to control the simulation.



The main netlist corresponding to this network is placed in a file "example.nap":

File "example.nap"

```
title   "SIGMA-DELTA MODULATOR (MASH 2) analyzed at frequency #freq Hz"

header <napa.hdr>                                       // generic NAPA resources
header <napatool.hdr>                                   // resources for TSNR analysis

directive  WINDOW  BLACKMAN_HARRIS                      // user's directive for FFT

dvar   adbstart -60.0
dvar   adbstop    0.0

fs     1.0e6

export        ampl_db                                   // exported to TSNR output
interlude     100

ivar numpts   POWEROF2(14)                              // number of points (FFT)

dvar ref      1.0                                       // reference
dvar freq     1000.0                                    // signal frequency
dvar ampl_db  LINSWEEP(ctr, adbstart, adbstop, 61)      // amplitude  (dB)
dvar ampl     DB2LIN(ampl_db, 1.0)                      // amplitude (lin)

#* Activation

node i0 dc    ref
node i1 sine   0.0  ampl  freq  0.0
```

```
node i2 adc   1024   i1  i0

#* MASH2 Modulator

node o1 cell stage1 "./ds1.net"  i2 e1 512                // first stage
node o2 cell stage2 "./ds1.net"  e1 e2 512                // second stage
node o3 sum  -s o2
node o4 sum  o1 o3
node s  delay   o2

#* Analysis

node ctr itool tsnr "stdout" o4 ref 10000.0 numpts        // TSNR analysis on 10kHz

#* Variables update

update ampl_db
update ampl

terminate    ampl_db > adbstop
```

The corresponding netlist is placed in the file “example.nap”. This is the main ***NAPA*** source netlist.

This netlist requests a TSNR analysis (Total Signal to Noise Ratio) on a bandwidth of 10.0 kHz. The TSNR analysis will be performed for 61 input amplitudes, each computed from a 16384-point FFT.

The ***NAPA*** transpiler writes a file “example.c”. This is the ***C*** source of the simulator corresponding to the netlist “example.nap”. After compilation of file “example.c” with an ***ANSI-C*** compiler, an executable file “example.bin” (or “example.exe” under Windows) is produced.

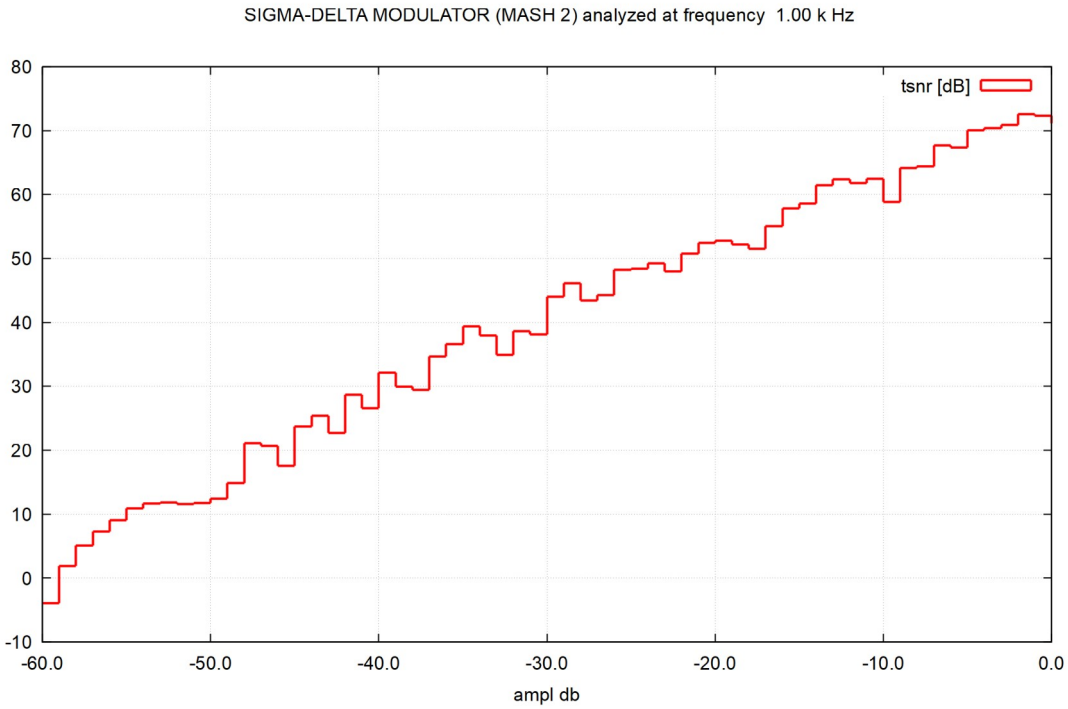
THIS EXECUTABLE IS THE SIMULATOR CORRESPONDING TO THE *NAPA* NETLIST.

Results are directed to standard output, as requested in the ***NAPA*** source netlist “example.nap”:

File “example.out”

```
# SIGMA-DELTA MODULATOR (MASH 2) analyzed at frequency  1.00 k Hz
# (tool                ) tsnr - frequency domain analysis
# (transpiler version  ) NAPA V5.03 for Win64
# (source file         ) example.tmp
# (random seed         ) 894740326
# (normalization       ) o4 / 1
# (bandwidth           ) 0.00000  Hz .. 10.0000 kHz
# (samples             ) 16384
# (sampling frequency  ) 1.00000 MHz
# (frequency resolution) 61.0352  Hz
# (window              ) Blackman Harris 4-Term
# (window resolution   ) 732.422  Hz
#
# Tue Sep 15 15:57:32 2026 by Yleduc
# packet        freq(Hz)    sig_RMS(dB)  noise_RMS(dB)       tsnr(dB)           ampl_db
 0          9.765625e+02  -7.446674e+01  -7.053082e+01  -3.935921e+00  -6.000000000e+01
 1          9.765625e+02  -6.731440e+01  -6.917045e+01   1.856051e+00  -5.900000000e+01
 2          9.765625e+02  -6.512889e+01  -7.023651e+01   5.107614e+00  -5.800000000e+01
 3          9.765625e+02  -6.394195e+01  -7.122887e+01   7.286923e+00  -5.700000000e+01
 4          9.765625e+02  -6.318386e+01  -7.223131e+01   9.047445e+00  -5.600000000e+01
 5          9.765625e+02  -6.267844e+01  -7.357854e+01   1.090011e+01  -5.500000000e+01
 6          9.765625e+02  -6.231186e+01  -7.400046e+01   1.168860e+01  -5.400000000e+01
 7          9.765625e+02  -6.202770e+01  -7.384894e+01   1.182124e+01  -5.300000000e+01
 8          9.765625e+02  -6.182962e+01  -7.337617e+01   1.154656e+01  -5.200000000e+01
 9          9.765625e+02  -6.167589e+01  -7.344078e+01   1.176488e+01  -5.100000000e+01
 10         9.765625e+02  -5.864847e+01  -7.110631e+01   1.245784e+01  -5.000000000e+01
 11         9.765625e+02  -5.744968e+01  -7.231812e+01   1.486844e+01  -4.900000000e+01
 12         9.765625e+02  -5.678510e+01  -7.791497e+01   2.112987e+01  -4.800000000e+01
 13         9.765625e+02  -5.634818e+01  -7.698814e+01   2.063996e+01  -4.700000000e+01
 14         9.765625e+02  -5.501663e+01  -7.260972e+01   1.759309e+01  -4.600000000e+01
 15         9.765625e+02  -5.375895e+01  -7.750911e+01   2.375017e+01  -4.500000000e+01
 16         9.765625e+02  -5.312502e+01  -7.849956e+01   2.537454e+01  -4.400000000e+01
```

```
 17         9.765625e+02  -5.192534e+01  -7.461913e+01   2.269379e+01  -4.300000000e+01
 18         9.765625e+02  -5.095841e+01  -7.967847e+01   2.872006e+01  -4.200000000e+01
 19         9.765625e+02  -5.004144e+01  -7.660177e+01   2.656033e+01  -4.100000000e+01
 20         9.765625e+02  -4.900618e+01  -8.112143e+01   3.211525e+01  -4.000000000e+01
 21         9.765625e+02  -4.793488e+01  -7.783995e+01   2.990507e+01  -3.900000000e+01
 22         9.765625e+02  -4.718983e+01  -7.659335e+01   2.940353e+01  -3.800000000e+01
 23         9.765625e+02  -4.605976e+01  -8.069889e+01   3.463913e+01  -3.700000000e+01
 24         9.765625e+02  -4.501174e+01  -8.163995e+01   3.662821e+01  -3.600000000e+01
 25         9.765625e+02  -4.402193e+01  -8.343553e+01   3.941360e+01  -3.500000000e+01
 26         9.765625e+02  -4.304816e+01  -8.098042e+01   3.793226e+01  -3.400000000e+01
 27         9.765625e+02  -4.210052e+01  -7.704882e+01   3.494830e+01  -3.300000000e+01
 28         9.765625e+02  -4.100655e+01  -7.964794e+01   3.864140e+01  -3.200000000e+01
 29         9.765625e+02  -4.007329e+01  -7.817637e+01   3.810308e+01  -3.100000000e+01
 30         9.765625e+02  -3.903097e+01  -8.302884e+01   4.399787e+01  -3.000000000e+01
 31         9.765625e+02  -3.803261e+01  -8.414232e+01   4.610971e+01  -2.900000000e+01
 32         9.765625e+02  -3.705506e+01  -8.046306e+01   4.340800e+01  -2.800000000e+01
 33         9.765625e+02  -3.601648e+01  -8.024276e+01   4.422628e+01  -2.700000000e+01
 34         9.765625e+02  -3.502734e+01  -8.322414e+01   4.819680e+01  -2.600000000e+01
 35         9.765625e+02  -3.402274e+01  -8.244786e+01   4.842512e+01  -2.500000000e+01
 36         9.765625e+02  -3.304045e+01  -8.226545e+01   4.922500e+01  -2.400000000e+01
 37         9.765625e+02  -3.203604e+01  -7.997437e+01   4.793834e+01  -2.300000000e+01
 38         9.765625e+02  -3.102667e+01  -8.174812e+01   5.072144e+01  -2.200000000e+01
 39         9.765625e+02  -3.002783e+01  -8.247731e+01   5.244949e+01  -2.100000000e+01
 40         9.765625e+02  -2.903096e+01  -8.177370e+01   5.274274e+01  -2.000000000e+01
 41         9.765625e+02  -2.803809e+01  -8.018814e+01   5.215005e+01  -1.900000000e+01
 42         9.765625e+02  -2.703639e+01  -7.857728e+01   5.154089e+01  -1.800000000e+01
 43         9.765625e+02  -2.603347e+01  -8.107058e+01   5.503711e+01  -1.700000000e+01
 44         9.765625e+02  -2.502860e+01  -8.284141e+01   5.781280e+01  -1.600000000e+01
 45         9.765625e+02  -2.403008e+01  -8.259046e+01   5.856039e+01  -1.500000000e+01
 46         9.765625e+02  -2.302925e+01  -8.449578e+01   6.146654e+01  -1.400000000e+01
 47         9.765625e+02  -2.202983e+01  -8.445193e+01   6.242210e+01  -1.300000000e+01
 48         9.765625e+02  -2.102904e+01  -8.285115e+01   6.182211e+01  -1.200000000e+01
 49         9.765625e+02  -2.003228e+01  -8.248316e+01   6.245088e+01  -1.100000000e+01
 50         9.765625e+02  -1.903120e+01  -7.788692e+01   5.885572e+01  -1.000000000e+01
 51         9.765625e+02  -1.803082e+01  -8.214553e+01   6.411471e+01  -9.000000000e+00
 52         9.765625e+02  -1.702993e+01  -8.141058e+01   6.438065e+01  -8.000000000e+00
 53         9.765625e+02  -1.602907e+01  -8.371147e+01   6.768240e+01  -7.000000000e+00
 54         9.765625e+02  -1.503084e+01  -8.241164e+01   6.738081e+01  -6.000000000e+00
 55         9.765625e+02  -1.403081e+01  -8.412336e+01   7.009256e+01  -5.000000000e+00
 56         9.765625e+02  -1.303072e+01  -8.343503e+01   7.040431e+01  -4.000000000e+00
 57         9.765625e+02  -1.203083e+01  -8.294084e+01   7.091000e+01  -3.000000000e+00
 58         9.765625e+02  -1.103125e+01  -8.358762e+01   7.255637e+01  -2.000000000e+00
 59         9.765625e+02  -1.003111e+01  -8.237126e+01   7.234015e+01  -1.000000000e+00
 60         9.765625e+02  -9.031263e+00  -8.020363e+01   7.117237e+01   0.000000000e+00
# end
```



The tool writes the first fifteen lines.
They begin with symbol "**#**" as a comment marker (***Gnuplot***[6] compatible).

First line is a copy of the title defined in the ***NAPA*** netlist. Second line is the name of the tool producing the output. Following lines describe the main tool settings, including windowing function used by the "*tsnr*" SMART TOOL (the default is defined inside the resource header file "*winname.hdr*" and was reset to a Blackman-Harris window by the user's directive "WINDOW"). Last comment line describes the content of the output data.

Data are organized following specifications described in the user-defined tool "*tsnr*". The data include the exported variable "ampl_db" as specified in the ***NAPA*** source file "example.nap".

A detailed example of a user-defined tool was given in chapter "User-Defined Functions and Tools" (see page 147).

6 ***Gnuplot*** is a multi-platform graphics package available as freeware.

APPENDIX E

Quick Reference: *NAPA* instructions

#	<any comment on one line>
alias	<name> <target_node>
	<name> <target_var>
array	(analog) <array_name> '['<arr_size>']' [<"file_pathname">]
	(digital) <array_name> '['<arr_size>']' [<"file_pathname">]
	(pointer) <array_name> '['<arr_size>']' <nod_nam | var_nam | array_name...>
	(pointer) <array_name> '[ ]' <node_nam | var_nam | array_name...>
assert	<"text_message"> <C_Boolean_expression>
call	void <C_function_returning_void>
	<return_value> <C_expression>
cell_interface	<$node | $var | $parm...>
command_line	<var_nam...> | fs | ts | void
comment	<"text_message">
data	<"file_name"> <parameters...>
data_interface	<$var | $parm...>
debug	[<debug_level_number | identifier...>]
decimate	[fs] <decimation_factor> [<decimation_initial_value>]
declare	(analog) <identifier...>
	(digital) <identifier...>
	(string) <identifier...>
	( ) <identifier...>
	(constant) <identifier...>
	(true) <a_function_returning_a_boolean...>
directive	<C_preprocessor_macro_identifier> [<value>]

drop	[fs] <C_boolean_expression>
dump	<“file_pathname”> [when <C_boolean_expression_of_events>]
dvar	<var_nam> [<initial_value>]
error	<“text_message”>
event	<event_name> [<C_expression_returning_a_boolean>]
	<event_name> (new) [<C_expression>]
export	<global_variable_name>
	<nod_nam>
	<var_nam>
format	(analog) <“C_double_output_format”> | S | M | L
	(digital) <“C_long_long_output_format”> | S | M | L
	(string) <“C_string_output_format”> | S | M | L
fs	[<sampling_frequency>]
ganging	<array_name> ‘[’<array_size>‘]’ <nod_nam | var_nam | array_nam...>
	<array_name> ‘[ ]’ <nod_nam | var_nam | array_nam...>
gateway	[<count_down>]
header	<“file_pathname”> [(noexpand)]
	<“file_pathname”> (expand)
init	void <*C*_function_returning_void>
	<var_nam> <C_function>
	<nod_nam> <C_expression>
inject	<nod_nam> <C_function>
input	<“file_pathname”> <var_nam...>
	“stdin” <var_nam...>
interlude	<const1> [<const2>]
interpolate	[fs] <interpolation_factor>
ivar	<var_nam> [<“initial_value”>]
load	<“file_pathname”>
napa_version	<version_id>
node	<nod_nam> <node_kind> <nod_nam | var_nam | parameter...>
	void <node_kind> <nod_nam | var_nam | parameter...>
nominal	fs
opcode	<alu_name> <opcode_number> [<template>]
output	<“path_nam”> <node_nam | var_nam...> [when <boolean_expr_of_events>]
	<string_var_nam> <node_nam | var_nam...> [when <boolean_expr_of_events>]
	“stderr” <node_nam | var_nam...> [when <boolean_expr_of_events>]

	“stdout” <node_nam | var_nam...> [when <boolean_expr_of_events>]
ping	[“stderr”]
	<“path_nam”>
	<string_var_nam>
post	[<label>] <function_id> <file_name> [<parameters>]
	<label> void <file_name>
random_seed	<[- | +] seed_number>
restart	
string	<var_nam> [<“initial_value”>]
stuck	<nod_nam> <C_expression_returning_a_number>
synchronize	(yes) | (no)
terminate	[<C_boolean_expression>]
title	<“some one-line text”>
tool	<user_defined_itool> <parameters>
ts	[<sampling_period>]
update	<var_nam> [<C_expression>] [when <boolean_expr_of_events>]
void	<file_name>
warning	<“text_message”>

APPENDIX F

Quick Reference: Node Syntax

adc	<num_level> <input_node> <reference_node>	*R → I*
algebra	<C_expression>	*Chameleonic*
alu	<alu_name> <opcode node> <input_node...>	*Chameleonic*
and	<input_node> <input_node...>	*I → I*
average	<[- | +]input_node> <[- | +]input_node...>	*R → R*
bshift	<number> <input_node>	*I → I*
	<[- | +]shift_var> <input_node>	*I → I*
	<[- | +]shift_node> <input_node>	*I → I*
btoi	<input_node> <input_node...>	*I → I*
buffer	<input_node>	*I → I*
bwand	<hexavalued_mask> <input_node...>	*I → I*
	<input_node> <input_node...>	*I → I*
bwbuffer	<input_node>	*I → I*
bwinv	<input_node>	*I → I*
bwnand	<hexavalued_mask> <input_node...>	*I → I*
	<input_node> <input_node...>	*I → I*
bwnor	<hexavalued_mask> <input_node...>	*I → I*
	<input_node> <input_node...>	*I → I*
bwnot	<input_node>	*I → I*
bwor	<hexavalued_mask> <input_node...>	*I → I*
	<input_node> <input_node...>	*I → I*

bwxnor	<hexavalued_mask> <input_node...>	*I → I*
	<input_node> <input_node...>	*I → I*
bwxor	<hexavalued_mask> <input_node...>	*I → I*
	<input_node> <input_node...>	*I → I*
cell	<instance_name> <“file_name”> <parameter_list>	*N/A*
change	<input_node>	*X → I*
clip	<[- | +]threshold_low> <[- | +]threshold_high> <input_node>	*Chameleonic*
clock	<“[pattern_descriptor_aperiodic.]pattern_descriptor_periodic”>	*→ I*
comp	<[- | +]positive_input_node> <[- | +]negative_input_node>	*X → I*
	<[- | +]positive_input_node> <[- | +]variable>	*X → I*
	<[- | +]variable> <[- | +]negative_input_node>	*X → I*
	<[- | +]positive_input_node> <[- | +]number>	*X → I*
	<[- | +]number> <[- | +]negative_input_node>	*X → I*
const	[(digital)] <C_expression>	*→ I*
	(analog) <C_expression>	*→ R*
copy	<[- | +]input_node>	*Chameleonic*
cosine	<[- | +]offset> <amplitude> <frequency> <[- | +]phase>	*X → R*
dac	<num_level> <input_node> <reference_node>	*I → R*
dalgebra	<C_expression_cast_to_real>	*X → R*
dc	[(analog)] <C_expression>	*→ R*
	(digital) <C_expression>	*→ I*
delay	<[- | +]input_node>	*Chameleonic*
	<number> <[- | +]input_node>	*Chameleonic*
	<delays_var> <[- | +]input_node>	*Chameleonic*
differentiator	<[- | +]input_node>	*Chameleonic*
div	<[- | +]input_node> <[- | +]input_node>	*Chameleonic*
	<[- | +]input_node> <[- | +]variable>	*Chameleonic*
	<[- | +]input_node> <[- | +]number>	*Chameleonic*
dtoi	<input_node>	*R → I*
dtool	<dtool_name> [<parameter_list>]	*X → R*
duser	<duser_name> [<parameter_list>]	*X → R*

equal	<[- | +]input_node> <[- | +]input_node>	*X → I*
	<[- | +]input_node> <[- | +]variable>	*X → I*
	<[- | +]variable> <[- | +]input_node>	*X → I*
	<[- | +]input_node> <[- | +]number>	*X → I*
	<[- | +]number> <[- | +]input_node>	*X → I*
fzand	<input_node> <input_node...>	*R → R*
fzbuffer	<input_node>	*R → R*
fzinv	<input_node>	*R → R*
fznand	<input_node> <input_node...>	*R → R*
fznor	<input_node> <input_node...>	*R → R*
fznot	<input_node>	*R → R*
fzxnor	<input_node> <input_node>	*R → R*
fzor	<input_node> <input_node...>	*R → R*
fzxor	<input_node> <input_node>	*R → R*
gain	<[- | +]number> <input_node>	*Chameleonic*
	<[- | +]gain_var> <input_node>	*Chameleonic*
generator	<instance_name> <“generator_name”> <parameter_list>	*N/A*
hold	<control_node> <input_node>	*Chameleonic*
ialgebra	<C_expression_cast_to_int>	*X → I*
integrator	<[- | +]input_node>	*Chameleonic*
inv	<input_node>	*I → I*
itob	<bit_rank> <input_node>	*I → I*
itod	<input_node>	*I → R*
itool	<itool_name> [<parameter_list>]	*X → I*
iuser	<iuser_name> [<parameter_list>]	*X → I*
latch	<set_input_node> <reset_input_node>	*I → I*
lshift	<number> <input_node>	*I → I*
	<shift_var> <input_node>	*I → I*
	<shift_node> <input_node>	*I → I*
max	<[- | +]input_node> <[- | +]input_node...>	*Chameleonic*

merge	<[- | +]input_node_from_seg_a> <[- | +]input_node_from_seg_b...>	*Chameleonic*
min	<[- | +]input_node> <[- | +]input_node...>	*Chameleonic*
mod	<[- | +]input_node> <[- | +]input_node>	*Chameleonic*
	<[- | +]input_node> <[- | +]variable>	*Chameleonic*
	<[- | +]input_node> <[- | +]number>	*Chameleonic*
muller	<input_node> <input_node...>	$I \rightarrow I$
mux	<control_node> <[- | +]input_node_0> <[- | +]input_node_1...>	*Chameleonic*
	<control_var> <[- | +]input_node_0> <[- | +]input_node_1...>	*Chameleonic*
nand	<input_node> <input_node...>	$I \rightarrow I$
noise	<[- | +]DC_level> <noise_density_level>	$\rightarrow R$
nor	<input_node> <input_node...>	$I \rightarrow I$
not	<input_node>	$I \rightarrow I$
offset	<[- | +]number> <input_node>	*Chameleonic*
	<[- | +]offset_var> <input_node>	*Chameleonic*
or	<input_node> <input_node...>	$I \rightarrow I$
osc	<[- | +]offset> <amplitude> <frequency> <[- | +]phase>	$X \rightarrow R$
poly	<[- | +]coeff$_0$> [<[- | +]coeff$_i$> ...] <[- | +]input_node>	*Chameleonic*
prod	<[- | +]input_node> <[- | +]input_node...>	*Chameleonic*
quant	<input_node> <[- | +]input_node>	*Chameleonic*
	<variable> <[- | +]input_node>	*Chameleonic*
	<constant> <[- | +]input_node>	*Chameleonic*
ram	<name‘[’addr_node‘]’> <CS_node> <control_node> <W_node>	*Declared*
ram2	<name‘[’addr_node‘]’> <CS_node> <control_node> <W_node>	*Declared*
rect	<input_node>	*Chameleonic*
register	<control_node> <input_node>	*Chameleonic*
relay	<control_node> <input_node>	*Chameleonic*
	<control_var > <input_node>	*Chameleonic*
	<control_node> <input_node> < [- | +] setting_var>	*Chameleonic*
	<control_var> <input_node> < [- | +] setting_var>	*Chameleonic*
	<control_node> <input_node> < [- | +] setting_constant>	*Chameleonic*

	<control_var> <input_node> < [- | +] setting_constant>	*Chameleonic*
rip	<hexavalued_mask> <input_node>	*I → I*
rom	<name‘[’addr_node‘]’> <CS_node>	*Declared*
rom2	<name‘[’addr_node‘]’> <CS_node>	*Declared*
rshift	<number> <input_node>	*I → I*
	<shift_var> <input_node>	*I → I*
	<shift_node> <input_node>	*I → I*
rshift1	<number> <input_node>	*I → I*
	<shift_var> <input_node>	*I → I*
	<shift_node> <input_node>	*I → I*
rshift2	<number> <input_node>	*I → I*
	<shift_var> <input_node>	*I → I*
	<shift_node> <input_node>	*I → I*
sign	<input_node>	*X → I*
sine	<[- | +]offset> <amplitude> <frequency> <[- | +]phase>	*X → R*
square	<[- | +]offset> <amplitude> <frequency> <delay>[<duty_cycle>]	*→ R*
step	<level1> <level2> <transition_time> [<transition_time>]	*→ R*
sub	<[- | +]input_node> <[- | +]input_node>	*Chameleonic*
	<[- | +]input_node> <[- | +]number>	*Chameleonic*
sum	<[- | +]input_node> <[- | +]input_node...>	*Chameleonic*
test	<C_expression_cast_to_int>	*X → I*
toggle	[<input_node>]	*I → I*
track	<control_node> <input_node>	*Chameleonic*
triangle	<[- | +]offset> <amplitude> <frequency> <delay>[<duty_cycle>]	*→ R*
trig	<number> <input_node> [(dual)]	*X → I*
	<number> <input_node> (positive)	*X → I*
	<number> <input_node> (negative)	*X → I*
uadc	<num_level> <input_node> <reference_node>	*R → I*
udac	<num_level> <input_node> <reference_node>	*I → R*
wsum	<weight> <input_node> ... <weight> <input_node>	*Chameleonic*
xnor	<input_node> <input_node...>	*I → I*

xor	<input_node> <input_node...>	*I → I*
zero	<decimation_factor> <decimation_offset> <input_node>	*Chameleonic*

APPENDIX G

Quick Reference: The *NAPA* File System

Absolute reference	**“/ ”**
Reference to a **generic** library	**< >**
Reference to the **root** directory	**“ ”**
Reference to the **main** directory	**“~/”**
Reference to the **current cell** directory	**“./”**