NAPA User's Guide - Summary

What Is a Transpiler?

A transpiler (source-to-source compiler) translates source code in one language into source code in another language - or a different form of the same language - at a similar level of abstraction, rather than compiling down to machine code or bytecode. NAPA fits this exactly: it turns a .nap netlist into ANSI-C source, which a separate real compiler (gcc) then compiles to machine code.

NAPA User's Guide – Summary (Version 5.02, November 2026)

📌 Overview and Purpose

NAPA (Netlist to ANSI-C Transpiler) is a high-level simulation framework designed for **mixed-signal (analog/digital) sampled-data networks**, such as digital filters, analog modulators, or complex IC designs. It transpiles a **netlist**—a textual description of a circuit—into **optimized ANSI-C code**, which is then compiled into a fast, cycle-based simulator. NAPA abstracts low-level C programming, allowing designers to focus on **system architecture** while ensuring **type safety, portability, and performance**.

Key goals:

- **Rapid prototyping** of mixed-mode systems.
- **Automatic type handling** (analog/digital) via netlist analysis.
- **Cycle-accurate simulation** with minimal overhead.
- **Extensibility** via user-defined functions, cells, and tools.

🔧 Core Concepts

1. The NAPA Node

- A **node** represents a **unidirectional primitive** (e.g., logic gates, arithmetic operations, memory elements) with **one output** and **one or more inputs**.
- Nodes are **sampled at a defined frequency** (default: 1.0 Hz).
- **Types**:
    - **Analog**: Internally represented as `double` (double-precision floating-point).
    - **Digital**: Internally represented as `long long int` (64-bit signed integer).
    - **Chameleonic**: Adapts to input types (e.g., `algebra`, `sum`, `gain`).
- **Automatic type determination**: NAPA infers node types from their **kind** (e.g., `and` → digital) or **connections**.

2. The Netlist

- A **textual description** of the circuit, composed of **instructions** (e.g., `node`, `dvar`, `fs`).
- **Segments**: Divided by **decimation** (`decimate 2` → runs at `fs/2`) or **interpolation** (`interpolate 16` → runs at `16*fs`).
- **Loops**: Must contain **at least one delay** to avoid undetermined nodes.

3. Transpilation and Compilation Process

1. **Netlist Parsing**: NAPA reads the netlist, resolves dependencies, and checks for **undetermined nodes** (loops without delays).
2. **Node Sorting**: Nodes are ordered to respect data flow (e.g., inputs before outputs).
3. **C Code Generation**: Produces a **flattened C program** (no function calls for nodes) with:
    - **Initialization** (e.g., `NAPA_reset_nodes()`).
    - **Main loop** (executes nodes at their sampling rates).
    - **Termination** (e.g., cleanup, output).
4. **C Compilation**: The generated C code is compiled with an **ANSI-C compiler** (e.g., GCC) into an executable simulator.

4. Sampling Frequency (`fs`)

- **Main frequency** (`fs`) defines the **base simulation rate** (default: 1.0 Hz).
- **Local adjustments**:
  - **Decimation** (`decimate N`): Reduces frequency by factor `N` (e.g., `decimate 2` → `fs/2`).
  - **Interpolation** (`interpolate N`): Increases frequency by factor `N` (e.g., `interpolate 16` → `16*fs`).
- **Segment-based execution**: Each segment runs at its own rate, nested within the main loop.

⚙️ Key Features

1. Node Types (100+ Primitives)

NAPA provides a **rich library of node kinds**, categorized by function:

Category	Examples	Type
Logic Gates	`and`, `or`, `not`, `xor`, `nand`	Digital (`I→I`)
Arithmetic	`sum`, `sub`, `prod`, `div`, `gain`	Chameleonic
Memory	`delay`, `register`, `ram`, `rom`	Digital/Analog
Conversions	`dtoi` (R→I), `itod` (I→R), `btoi`	Type-specific
Signal Sources	`dc`, `sine`, `cosine`, `square`, `noise`	Analog/Digital
Comparators	`comp`, `equal`, `sign`	`X→I`
Bitwise Ops	`bwand`, `bwxor`, `lshift`, `rshift`	Digital (`I→I`)
User-Defined	`alu`, `duser`, `iuser`, `dtool`	Custom
Fuzzy Logic	`fzand`, `fzor`, `fznot`	Analog (`R→R`)
Special	`cell` (subcircuit), `generator`	Pseudo-node

- **Chameleonic nodes** (e.g., `algebra`, `sum`) adapt to input types.
- **Pseudo-nodes** (`cell`, `generator`) instantiate subcircuits from files.

2. User Variables

- **Purpose**: Control simulation parameters (e.g., gain, frequency) or external inputs.
- **Types**:
    - `dvar`: Analog (double-precision).
    - `ivar`: Digital (long long integer).
    - `string`: Text (char array).
- **Usage**:

```
dvar R 1.0e6
ivar npts POWEROF2(20)
string filin "somefile.dat"
```

- **Register Arithmetic**: Supports **width-limited** digital nodes/variables (e.g., `(16) ivar x 100` for 16-bit signed).
    - **2's complement** (parentheses: `(N)`) or **unsigned** (angle brackets: `<N>`).
    - Emulates **fixed-width arithmetic** (e.g., overflow, roll-off).

3. Simulation Control

Instruction	Purpose
`fs 1.0e6`	Set main sampling frequency (1 MHz).
`decimate 2`	Create a segment running at `fs/2`.
`interpolate 16`	Create a segment running at `16*fs`.
`drop (LOOP_INDEX % 1000)`	Conditionally skip a segment.
`nominal fs`	Reset segment to main frequency.
`terminate when ev1`	Stop simulation when event `ev1` is true.
`assert "Error!" x>0`	Halt simulation if condition fails (debugging).
`dump "file.dmp"`	Save simulation state (for debugging).
`load "file.dmp"`	Restore simulation state.
`restart`	Reset simulation to initial state.
`event ev1 (new) x`	Define a trigger (e.g., when `x` changes).
`synchronize`	Sync tools across segments.

4. Input/Output

- **Input**:
  - Read variables from files or `stdin`:

    ```
    dvar value1
    dvar value2
    ivar value3
    input "myfile.dat" value1 value2 value3
    ```

  - Supports **command-line arguments** (`command_line fs infile`).
- **Output**:
  - Export nodes/variables to files:

    ```
    output "stdout" a b c d when ev0
    output "result.out" in out
    ```

  - **Formatting**: Customize output with `format` (e.g., `format (analog) "% .9e"`).
  - **Tools**: User-defined **smart tools** (e.g., FFT, SNR analysis) via `itool`/`dtool`.

5. Hierarchy and Reusability

- **Cells**: Reusable subcircuits (`.net` files) instantiated via:

  ```
  node out cell lp <CT_Domain/Butterworth/LP2.net> in 1.0e3
  ```

- **Generators**: Dynamic subcircuit generation (`.gen` files).
- **Headers**: Include C code (`.hdr` files) for user-defined functions.
- **File System**:
  - **Absolute**: `/path/to/file`.
  - **Generic Library**: `<file.hdr>` (searches in `-h`, `-n`, `-g` directories).
  - **Relative**: `"/file.net"` (main dir), `"./file.net"` (current cell dir).

📁 File Structure and Workflow

1. Typical Project Layout

`/Simulate/NapaDos/Hdr`	generic header files	`*.hdr`
`/Simulate/NapaDos/Net`	generic cell files	`*.net`
`/Simulate/NapaDos/Gen`	generic generator files	`*.exe`
`/Simulate/NapaDos/Dos`	the NAPA transpiler and scripts	
`/jdoe/NAPA/project_1`	project's netlists, headers, cells, simulation output	`*.nap, *.rom, *.hdr, *.net, *.dat, *.c, *.exe, *.out`

2. Transpilation and Compilation Workflow

1. **Write netlist** (`myfile.nap`):

   ```
   title "$F"
   header <napatool.hdr>
   fs 1.0e6
   ...
   ```

2. **Transpile with NAPA**:

   ```
   MACDOS.exe myfile.nap -c -l > myfile.tmp
   NAPADOS.exe myfile.tmp -u "JDoe" -d /Simulate/NapaDos > myfile.c
   ```

3. **Compile C code**:

   ```
   gcc myfile.c -o myfile.exe
   ```

4. **Run simulation**:

   ```
   myfile.exe > myfile.out
   ```

From the NAPA editor, the four steps are run by pressing **Alt+R**.

3. Portability

- **OS Support**: 64-bit Windows (transpiler and simulators built with the MinGW-w64 GCC toolchain).
- **Requirements**: the GCC compiler shipped with NAPA (MinGW-w64, in `\Simulate\UtilDos`).
- **File Naming**: NAPA pathnames follow the **UNIX syntax**, with forward slashes, also on Windows (e.g., `/Simulate/NapaDos/Net`).

Best Practices and Warnings

✅ Do:

- **Initialize memory nodes**: Use `init` for `delay`, `register`, `integrator`, etc.
- **Use width-limited nodes** for fixed-point arithmetic:

```
(22) node s sum a1 a2
<16> node ptr16 ialgebra thermometric(ptr4, 16)
```

- **Leverage chameleonic nodes** for type flexibility (e.g., `algebra`, `sum`).
- **Modularize designs** with `cell` and `generator`.
- **Validate types** with `declare`:

```
declare (analog) fmin fmax fchirp
```

❌ Avoid:

- **Undetermined nodes**: Ensure all loops contain **at least one delay**.
- **Implicit casting**: Prefer dedicated nodes (e.g., `dtoi` over `ialgebra`) to avoid C casting pitfalls.
- **Overusing `dump`**: Generates large files; use only for debugging.
- **Mixing node/variable names**: A name cannot be both a node and a variable.
- **Forgetting `fs`**: Default is 1.0 Hz, which may cause accuracy issues in signal generators.

🔍 Debugging Tools

- **`debug`**: Add C preprocessor macros (e.g., `debug 2` → `#define DEBUG_MODE_2`).
- **`assert`**: Halt on conditions (e.g., `assert "Overflow!" x < 1000`).
- **`dump` + `load`**: Save/restore simulation state.
- **`list` option**: List nodes/variables without generating C code (`NAPADOS.exe myfile.nap -l`).

🎯 Conclusion

NAPA is a **powerful, flexible, and portable** tool for **mixed-signal simulation**, bridging the gap between high-level design and low-level C implementation. Its **automatic type handling**, **extensive node library**, and **hierarchical support** make it ideal for **IC designers, DSP engineers, and system architects**. By following **best practices** (e.g., proper initialization, type declarations, and modularization), users can build **efficient, reliable, and maintainable** simulations.

For advanced use, NAPA supports **custom C functions**, **smart tools**, and **networked execution**, though users must be mindful of **C limitations** (e.g., casting, side effects). The **two-stage process** (NAPA transpilation → C compilation → executable) ensures **optimized performance** while maintaining **readability and extensibility**.

📚 **Further Reading**: Refer to the full ***NAPA User's Guide*** **(v5.02)** for detailed node descriptions, advanced features (e.g., `ram`, `rom`, `tool` synchronization), and troubleshooting.