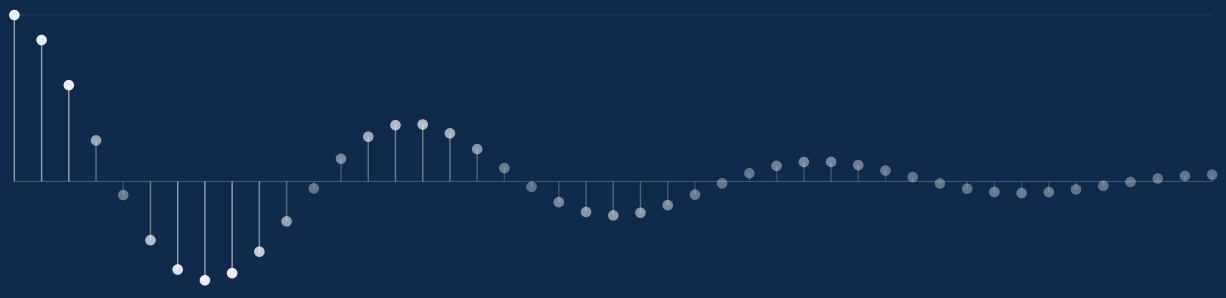
TDSE



USER GUIDE

TDSE™ SDK

Discrete time-domain operator simulation.

BUILDER · RUNTIME · ADAPTER CIRCUIT · CLI · PLUGIN SYSTEM

VERSION	RELEASE	BUILD
1.0.0-rc1	May 24, 2026	9c50c96c

Contents

CHAPTER 1

Preface

1.1 How To Use This Manual

This is the official user guide for the TDSE SDK. It follows the path most teams actually take: get a first model running, understand what Runtime is doing, and then harden the integration for regular use.

If you only remember one reading rule, use this one:

- read Hands-On Tutorial first when you need confidence
- read a task chapter first when you already know your starting artifact

For a closed-source RC evaluation, the shortest path is:

1. first 10 minutes: finish Installation and prove the package is intact
2. first 30 minutes: complete the hands-on tutorial and prove Builder -> Runtime once
3. first day: choose one example that matches your real host architecture

The handbook is written for four common readers:

Reader	Start here	Then usually continue with
technical evaluator or procurement reviewer	this Preface	Platform Notes and Plugin System
SDK integration engineer	Installation	Hands-On Tutorial, Runtime API Summary, and Runtime chapters
EDA / RTDS / HIL architect	this Preface	Theory and Concepts, Builder and Data Contracts, and Runtime chapters
verification or support engineer	Hands-On Tutorial	Troubleshooting, Profiler, and Backend and Performance

1.2 Start Here

Use this short map before you commit to a chapter:

If you are trying to do this	Start here	Then continue with
Get the fastest trustworthy first run	Installation	Hands-On Tutorial
Find the closest runnable reference after Hands-On Tutorial	Examples Guide	the linked workflow chapter
Package validated `H` or `IR` data into a `.pack`	Builder and Data Contracts	Runtime Lifecycle
Embed TDSE in a host simulation loop	Runtime Lifecycle	Step Execution
Convert circuit or RAW inputs into Builder-ready artifacts	Adapter Circuit	CLI Reference or Examples Guide

If you are trying to do this	Start here	Then continue with
Tune backend selection or collect performance evidence	Profiler	Backend and Performance
Triage a failure	Troubleshooting	the chapter named by the symptom path

The TDSE SDK consists of four cooperating layers:

- **Builder** ingests validated operator artifacts and writes a runtime pack
- **Runtime** loads that pack and provides the lifecycle and per-step API calls your simulator uses
- **Adapter** converts domain-specific inputs into Builder-ready artifacts. The current adapter module is Adapter Circuit, which handles circuit netlists and RAW files
- **CLI** provides a unified `tdse` command-line interface for adapter, profiler, and SDK workflows

In practical product terms, teams adopt TDSE SDK when they want to:

- pre-package a large linear time-invariant subsystem once instead of re-solving it globally at every host step
- keep the host solver in control of timestep, acceptance, and scheduling while TDSE supplies `op`, `hr`, `ir`, and `dr`
- embed reduced linear dynamics into an existing simulator, EMT loop, or co-simulation framework through a small Runtime API surface

TDSE is usually a good fit when your host architecture already has:

- a clear boundary between a linear time-invariant subsystem and the nonlinear or event-driven host solve
- a stable port/interface definition between those subsystems
- a reason to turn validated FRF, impulse-response, netlist, or RAW inputs into a reusable runtime artifact

The book is organized in the same order most integrations mature:

- **Onboarding:** Preface, Installation, Hands-On Tutorial, and Examples Guide
- **Core model flow:** Theory and Concepts, Builder and Data Contracts, and the Runtime chapters
- **Source-data workflows:** Adapter Circuit, RAW Import, and the CLI reference
- **Advanced integration:** variable `dt`, backend selection, plugins, telemetry, and profiler workflows
- **Support material:** Troubleshooting, Appendix, and Platform Notes

Read the book in two passes:

1. First pass: Installation -> Hands-On Tutorial -> Examples Guide
2. Second pass: the chapter cluster that matches your integration path

This manual focuses on SDK usage and integration. It shows you how to:

- Install and configure the TDSE SDK
- Build and consume runtime packs using the Builder and Runtime APIs
- Integrate the Runtime into host simulators and step loops
- Use the Adapter layer (currently Adapter Circuit) to convert domain inputs
- Apply the CLI for common workflows and diagnostics

- Tune performance through backend selection and runtime plans
- Diagnose and resolve common integration issues

This manual does not cover:

- Internal implementation details of the TDSE algorithms
- Mathematical derivations beyond the conceptual overview in Theory and Concepts
- Generated API symbol reference (see the separate API documentation)
- Platform-specific build system configuration beyond installation
- Third-party tool integration beyond documented adapter interfaces

Check these scope boundaries early before you commit to an integration path:

- TDSE Runtime is an execution engine for packaged linear models; it is not a general-purpose circuit simulator by itself
- circuit-domain ingestion is currently through Adapter Circuit, whose supported netlist subset is documented in Element Reference
- qualified Linux support has a specific host baseline and optional accelerator scope; see Platform Notes
- real-time and HIL success still depends on host-side scheduling, timing-budget measurement, and target-machine qualification

Use this guide for concepts, workflows, and integration choices. Use the generated API reference when you need exact symbol details.

1.3 How the SDK Fits Together

The four layers form a pipeline: domain inputs enter through the Adapter layer, Builder packages them, and Runtime executes the result.

```
Domain Input          Adapter              Builder             Runtime
(RAW, netlist,  →     Circuit      →     pack       →     step loop,
 FRF data)            converts and        (.pack file)       op/hr/ir/commit
                      validates
```

- **You use the Adapter** when your starting point is a circuit netlist or RAW file.
- **You use the Builder** when you have FRF data or impulse responses ready to package.
- **You use the Runtime** in your host simulator's step loop — it's the part you link into your own application.
- **You use the CLI** (`tdse` command) for quick exploration, profiling, and benchmarking.

Skip ahead to Installation if you want to start coding now. The terminology reference below is here when you need it.

1.4 Document Conventions

Typographic conventions used throughout this guide:

Convention	Meaning	Example
Bold	Emphasis, important terms on first use	The **Runtime** executes the pack
Italic	New terms, parameter names in prose	the *runtime pack* is read-only

Convention	Meaning	Example
`Monospace`	Code, API symbols, file paths, command-line input	call `tdse_model_create()`
`Monospace bold`	Commands to type exactly as shown	**`cmake --build build`**

The guide uses four callout styles. Each appears here exactly as it does in the rest of the book:

> **WARNING**
> Critical information that prevents data loss or incorrect results.

> **IMPORTANT**
> Information essential to correct usage.

> **NOTE**
> Supplementary information or clarification.

> **TIP**
> Shortcut or recommended practice.

1.5 Key Terminology

The terminology below covers the technical abbreviations and runtime terms used most often in the handbook.

1.5.1 Core Terms

Term	Meaning
H	Impulse Response - Time-domain impulse response (delayed-history operator)
IR	Independent Response - Pre-computed independent response sequence (optional)
op	Instantaneous Operator - Immediate response term (G0 or R0) at current timestep
hr	History Response - Delayed-history contribution from past timesteps
ir	Independent Response Term - Contribution from pre-computed IR sequence
dr	Direct Response - Post-commit direct response term
Builder	Builder Component - Creates runtime packs from validated input data
Runtime	Runtime Component - Executes runtime packs in simulation loops
pack	Runtime Pack - Compressed binary model file created by Builder
RuntimeCore	Product/distribution slice centered on Runtime packaging and execution support

Term	Meaning
trial step	Current not-yet-committed evaluation context created by `tdse_step_begin(...)`
committed step	Last accepted step after `tdse_step_commit(...)`
port	Interface variable pair defining coupling between linear and nonlinear subsystems
impulse response	Time-domain response of the linear subsystem to a unit impulse at a port

1.5.2 Lifecycle Terms

Term	Meaning
close	Explicit shutdown request that returns immediately with a status
destroy	Recommended shutdown call with a caller-chosen wait budget
release	Final cleanup path, usually used in RAII or finally-style code
ownership handoff	Moment a caller must stop treating a handle as locally owned
bounded destroy	Destroy with an explicit wait budget and a result the caller can inspect

1.5.3 Diagnostics Terms

Term	Meaning
create diagnostics	Request-scoped data returned through `tdse_model_create_diagnostics_t`
model info	Static metadata snapshot for a live handle
state info	Dynamic execution-state snapshot for a live handle
last error info	Sticky snapshot of the most recent non-OK runtime result
pack error token	Symbolic name derived from `create_diag.pack_error_code`

1.5.4 Concurrency Terms

Term	Meaning
same-handle concurrent use	Overlapping runtime API entry on the same `tdse_model_t*`
execution thread	Logical worker or simulation thread that owns one runtime handle
finalizer cleanup	Cleanup path that does not itself define host lifecycle policy

1.6 Revision History

Version	Date	Description
1.0.0-rc1 documentation refresh	2026-05-14	Full Linux SDK support, merged adapter library, updated examples
1.0.0-rc1	2026-04-20	Initial release candidate

CHAPTER 2

Installation

Use this chapter to get the SDK onto the machine, confirm headers and libraries are discoverable, and prove that a tiny program can link against the installed package.

The SDK provides libraries (Runtime, Builder, Adapter) that you link into your application, plus a command-line tool (`tdse`) for benchmarking and profiling.

Once the verification step succeeds, continue directly to Hands-On Tutorial. That chapter is the shortest trustworthy first run.

For a first customer-style evaluation, keep the goal narrow:

1. prove the shipped package is intact
2. prove headers and libraries are discoverable from a downstream consumer
3. prove one tiny program links and runs before you move on to Builder or host integration

When this chapter is complete, you should be ready to start the hands-on tutorial without changing tools, package paths, or library names.

2.1 System Requirements

Every SDK user needs a compiler and CMake to build against the SDK libraries.

- **Windows:** Windows 10 or later, 64-bit. MSVC 2019 or later.
- **Linux:** x86_64 GNU/Linux, glibc-based (tested on Ubuntu 24.04). GCC 10+ or Clang 14+.
- **CMake:** 3.21 or later. On Linux, `pkg-config` is also supported.
- **Disk space:** ~200 MB.

Prebuilt SDK packages include all third-party dependencies. You do not need to install SuiteSparse, OpenBLAS, LAPACK, or CUDA runtime libraries separately through your system package manager.

This chapter assumes a shipped closed-source RC evaluation package rather than source access to TDSE itself.

2.2 Package Contents

2.2.1 Linux

```
tdse-sdk/
├── bin/
│   ├── tdse                          (benchmark and profiling CLI)
```

```
│   └── tdse_plugin_doctor          (plugin diagnostics)
├── include/
│   ├── tdse/
│   │   ├── tdse.h                  (Runtime C API)
│   │   ├── tdse.hpp                (Runtime C++ wrapper)
│   │   └── ...
│   ├── tdse_builder.h              (Builder C API)
│   ├── tdse_builder.hpp
│   ├── tdse_perf.h                 (backend selection)
│   ├── tdse_adapter_circuit.h
│   └── tdse_adapter_circuit/
│       ├── common.h
│       ├── compile.h
│       ├── compute.h
│       └── ...
├── lib/
│   ├── libtdse.so                  (Runtime, always shared)
│   ├── libtdse_builder.so          (Builder, .a for static builds)
│   ├── libtdse_adapter_circuit.so  (Adapter Circuit, .a for static builds)
│   ├── plugins/
│   │   └── sim/
│   │       ├── libtdse_sim_cpu_dense.so
│   │       ├── libtdse_sim_cpu_sparse.so
│   │       ├── libtdse_sim_cuda.so
│   │       └── plugin_manifest.json
│   ├── tdse/third_party/           (bundled: SuiteSparse, OpenBLAS, CUDA runtime)
│   ├── cmake/tdse/                 (CMake package)
│   └── pkgconfig/tdse.pc           (pkg-config)
└── share/tdse/
    ├── profiler/
    ├── third_party_notices/
    ├── tdse_sdk_variant.json
    └── tdse_runtimecore_variant.json
```

Third-party shared libraries (SuiteSparse, OpenBLAS, CUDA runtime) are bundled under `lib/tdse/third_party/` so the SDK is self-contained.

2.2.2 Windows

```
tdse-sdk/
├── bin/
│   ├── tdse.exe                    (benchmark and profiling CLI)
│   ├── tdse_plugin_doctor.exe      (plugin diagnostics)
│   ├── tdse.dll                    (Runtime)
│   └── *.dll                       (third-party dependencies)
├── include/
│   ├── tdse/
│   │   ├── tdse.h                  (Runtime C API)
│   │   ├── tdse.hpp                (Runtime C++ wrapper)
│   │   └── ...
```

```
│   ├── tdse_builder.h              (Builder C API)
│   ├── tdse_builder.hpp
│   ├── tdse_perf.h                 (backend selection)
│   ├── tdse_adapter_circuit.h
│   └── tdse_adapter_circuit/
│       ├── common.h
│       ├── compile.h
│       ├── compute.h
│       └── ...
├── lib/
│   ├── tdse.lib                    (Runtime import library)
│   ├── tdse_builder.lib            (Builder import library)
│   ├── tdse_adapter_circuit.lib    (Adapter Circuit)
│   ├── plugins/
│   │   └── sim/
│   │       ├── tdse_sim_cpu_dense.dll
│   │       ├── tdse_sim_cpu_sparse.dll
│   │       ├── tdse_sim_cuda.dll
│   │       └── plugin_manifest.json
│   ├── cmake/tdse/                 (CMake package)
│   └── pkgconfig/tdse.pc           (pkg-config)
└── share/tdse/
    ├── profiler/
    ├── third_party_notices/
    ├── tdse_sdk_variant.json
    └── tdse_runtimecore_variant.json
```

Third-party DLLs are placed in `bin/` alongside the CLI executable — adding `bin/` to your PATH makes everything discoverable.

2.3 Install the SDK

2.3.1 Linux

Extract the tarball and verify the package is intact:

```
mkdir -p /opt/tdse-sdk
tar -xf tdse-sdk-*-linux-x86_64.tar.gz -C /opt/tdse-sdk --strip-components=1
/opt/tdse-sdk/bin/tdse sdk version --json-out -
```

Expected output: a JSON line containing `"status":"ok"`. If you see `error while loading shared libraries`, add the library directory:

```
export LD_LIBRARY_PATH=/opt/tdse-sdk/lib:$LD_LIBRARY_PATH
```

For convenience, add the CLI to your PATH:

```
export PATH="/opt/tdse-sdk/bin:$PATH"
```

2.3.2 Windows

Extract the zip archive to a directory of your choice, e.g. `C:\tdse-sdk`, then verify:

```
C:\tdse-sdk\bin\tdse sdk version --json-out -
```

For convenience, add `bin\` to your PATH via **System Properties → Environment Variables**.

2.4 Set Up Your Project

2.4.1 CMake

Point `CMAKE_PREFIX_PATH` to the SDK root and link the targets your application needs:

```
find_package(tdse REQUIRED)
target_link_libraries(my_app PRIVATE tdse::tdse tdse::tdse_builder)
```

```
cmake -DCMAKE_PREFIX_PATH=/path/to/tdse-sdk ..
```

Available targets:

Target	What it provides
`tdse::tdse`	Runtime (step loop, lifecycle, diagnostics)
`tdse::tdse_builder`	Builder (pack creation)
`tdse::tdse_adapter_circuit`	Adapter (circuit domain)

If your application only needs the Runtime, use `find_package(tdseRuntimeCore)` which exports `tdse::tdse` alone.

2.4.2 pkg-config (Linux)

```
export PKG_CONFIG_PATH=/opt/tdse-sdk/lib/pkgconfig:$PKG_CONFIG_PATH
g++ -std=c++20 my_app.cpp $(pkg-config --cflags --libs tdse) -o my_app
```

2.4.3 Manual Flags

If you are not using CMake or pkg-config:

Linux:

```
g++ -std=c++20 my_app.cpp \
    -I/opt/tdse-sdk/include \
    -L/opt/tdse-sdk/lib \
    -ltdse_builder -ltdse -lm \
    -Wl,-rpath,/opt/tdse-sdk/lib \
    -o my_app
```

Windows:

```
cl my_app.c ^
  /I"C:\tdse-sdk\include" ^
  /link "C:\tdse-sdk\lib\tdse.lib" "C:\tdse-sdk\lib\tdse_builder.lib"
```

2.5 Verify Your Setup

Compile and run a minimal program to confirm your project can find headers and link against both Runtime and Builder.

```
// test_sdk.cpp
#include <tdse/tdse.h>
#include <tdse_builder.h>
#include <cstdio>

int main() {
    const char* v = tdse_version_string();
    if (!v) return 1;
    std::printf("TDSE %s\n", v);
    std::printf("builder status: %s\n", tdse_builder_status_message(0));
    return v ? 0 : 1;
}
```

Linux:

```
g++ -std=c++20 test_sdk.cpp \
    -I/opt/tdse-sdk/include \
    -L/opt/tdse-sdk/lib \
    -ltdse_builder -ltdse -lm \
    -Wl,-rpath,/opt/tdse-sdk/lib \
    -o test_sdk
./test_sdk
```

Windows:

```
cl test_sdk.cpp ^
  /I"C:\tdse-sdk\include" ^
  /link "C:\tdse-sdk\lib\tdse.lib" "C:\tdse-sdk\lib\tdse_builder.lib"
test_sdk.exe
```

Expected output: `TDSE 1.0.0-rc1` (or your installed version).

2.5.1 Check Individual Components

Use these commands to confirm specific SDK components are present. They are not required for normal use — they help diagnose setup problems.

Headers:

```
# Linux
ls /opt/tdse-sdk/include/tdse/tdse.h
ls /opt/tdse-sdk/include/tdse_builder.h
```

```
rem Windows
dir "C:\tdse-sdk\include\tdse\tdse.h"
dir "C:\tdse-sdk\include\tdse_builder.h"
```

Libraries:

```
# Linux
ls /opt/tdse-sdk/lib/libtdse.so
ls /opt/tdse-sdk/lib/libtdse_builder.so
```

```
rem Windows
dir "C:\tdse-sdk\lib\tdse.lib"
dir "C:\tdse-sdk\lib\tdse_builder.lib"
```

Plugins (required for Adapter Circuit commands):

```
# Linux
ls /opt/tdse-sdk/lib/plugins/sim/libtdse_sim_cpu_dense.so
ls /opt/tdse-sdk/lib/plugins/sim/libtdse_sim_cpu_sparse.so
ls /opt/tdse-sdk/lib/plugins/sim/plugin_manifest.json
ls /opt/tdse-sdk/bin/tdse_plugin_doctor
```

```
rem Windows
dir "C:\tdse-sdk\lib\plugins\sim\tdse_sim_cpu_dense.dll"
dir "C:\tdse-sdk\lib\plugins\sim\plugin_manifest.json"
dir "C:\tdse-sdk\bin\tdse_plugin_doctor.exe"
```

2.6 Inspect Installed Variant

The installed package manifests tell you which public targets and optional features are present. Read these before guessing whether a machine has sparse CPU, CUDA, full SDK headers, or only RuntimeCore.

Linux:

```
cat /opt/tdse-sdk/share/tdse/tdse_sdk_variant.json
cat /opt/tdse-sdk/share/tdse/tdse_runtimecore_variant.json
```

Windows:

```
type C:\tdse-sdk\share\tdse\tdse_sdk_variant.json
type C:\tdse-sdk\share\tdse\tdse_runtimecore_variant.json
```

Typical `tdse_sdk_variant.json` fields include:

- `build_variant` - package flavor such as `full-feature`
- `package_profile` - always `sdk`
- `features` - optional slices such as `sparse_cpu` and `cuda`
- `recommended_adapter_targets` - the target list used by `-DTDSE_ADAPTER_TARGETS=AUTO`
- `exported_targets` - the exact installed CMake targets

Typical `tdse_runtimecore_variant.json` fields include:

- `build_variant` - the RuntimeCore bundle flavor for this release
- `package_profile` - always `runtimecore`
- `config_package` - the installed CMake package name, typically `tdseRuntimeCore`
- `exported_targets` - the exact installed RuntimeCore CMake targets

Use these manifests when:

- a downstream consumer needs to choose between RuntimeCore and the full SDK
- a build system wants to enable sparse CPU or CUDA only when installed
- you need to confirm that `AUTO` target selection is doing what you expect

If you consume the SDK through CMake, the `package_consumer_targets` example described in Examples Guide demonstrates the intended out-of-tree install-tree flow. If you consume the full SDK through `pkg-config`, the feature slices are exposed through variables such as `sparse_libs` and `cuda_libs`.

2.7 Acceptance Ladder

Use this ladder when you need more confidence than "the files extracted correctly."

1. **Basic link proof** Run `tdse sdk version --json-out -` and compile the tiny program in this chapter. This proves the CLI, headers, and core libraries are discoverable.
2. **Installed-package consumer proof** Build one of the out-of-tree package-consumer examples against the installed SDK: `package_consumer_targets` for CMake or `package_consumer_pkgconfig` for `pkg-config`.
3. **Runtime and plugin proof** Run `tdse_plugin_doctor` against the installed manifest, then complete the Hands-On Tutorial path to prove Builder -> Runtime execution still works on this machine.

For Linux packaging and release-candidate validation, the broader install, package, and clean-host gates are summarized in Platform Notes.

If your next question is "can a downstream build consume this install tree cleanly?", stop here and run one package-consumer example before reading deeper SDK chapters. That is a better first PoC signal than jumping straight into advanced Runtime or Adapter details.

2.8 Compatibility Promises

These are the public compatibility rules you can rely on when integrating against the installed SDK:

- **Pack format:** Runtime accepts supported pack versions and reports incompatibility at create time through normal diagnostics and validation APIs.
- **Public structs:** most extensible Runtime, Builder, Adapter, and plugin-facing request/config/diagnostic structs use a `struct_size` field for forward-compatible growth. Intentionally frozen payloads are documented as exceptions.
- **Plugin ABI:** simulation plugins follow major/minor compatibility. Same major plus same or older minor is accepted by the host.
- **Installed-package consumption:** the shipped install tree is exercised through CMake/package-export and `pkg-config` smoke paths, not only through ad hoc same-machine linking.

2.9 Troubleshooting

Symptom	Likely cause	Fix
`tdse/tdse.h: No such file or directory`	Include path not set	Add `-I<prefix>/include`
`libtdse.so: cannot open shared object file`	Library path not set (Linux)	`export LD_LIBRARY_PATH=<prefix>/lib`
`LNK1181: cannot open input file 'tdse.lib'`	Linker path not set (Windows)	`/LIBPATH:<prefix>\lib` or use CMake
`find_package(tdse)` fails	CMake cannot locate the SDK	`-DCMAKE_PREFIX_PATH=<prefix>`
`No simulation engine plugin loaded`	Plugin artifact missing or manifest path wrong	Check `<prefix>/lib/plugins/sim/` and `plugin_manifest.json`
`tdse: command not found`	CLI not on PATH	Add `<prefix>/bin` to PATH

In production deployments:

- point `TDSE_PLUGIN_MANIFEST` at `<prefix>/lib/plugins/sim/plugin_manifest.json`
- enable `TDSE_PLUGIN_STRICT_MODE=1`
- use `tdse_plugin_doctor` as the first support or field diagnostic command

2.10 Next Step

If the install and verification steps succeeded:

- go to Hands-On Tutorial for the first end-to-end Builder -> Runtime run
- then use Examples Guide to find the closest runnable reference for your real workflow

CHAPTER 3

Hands-On Tutorial

Use this chapter for the shortest trustworthy TDSE bring-up: build a minimal 3-port model from scratch, write one pack, load it into Runtime, and verify a small step loop.

Prerequisite: Complete the Installation chapter first.

This is the first-run chapter referenced elsewhere in the handbook as **Hands-On Tutorial**.

The tutorial has four steps:

1. **Create the Builder input data** — a C program that defines an impulse response and writes a runtime pack.
2. **Compile and run the Builder** — produce `tutorial.pack`.
3. **Run the Runtime step loop** — load the pack, create a model, and execute five steps.
4. **Verify the output** — confirm the model metadata and step results match expectations.

By the end, you will have exercised the core SDK pipeline: Builder -> pack -> Runtime -> step loop. All code is runnable as-is.

For most installed-package evaluations, this chapter should take about 15-30 minutes. If the installed SDK is already on your machine and the commands below run unchanged, you should end the session with one pack, one successful model create, and one clean create-step-destroy path.

Expected outputs from this chapter:

- `tutorial.pack`
- a successful `tdse_model_create(...)`
- five successful runtime steps
- a clean `tdse_model_destroy(...)`

Before you start, confirm these are already true:

- `tdse sdk version --json-out -` succeeds from the installed package
- the tiny compile-and-link check from Installation already works
- you are using the same SDK prefix, compiler family, and library names from Installation

After Hands-On Tutorial, use the Examples Guide to choose the closest runnable reference for your real integration path.

3.1 Step 1: Create the Builder Input Data

This program does three things: (1) defines a 3-port impulse response with 4 taps, (2) configures the Builder with shape parameters, and (3) writes a `.pack` file.

Create `tutorial_data.c`:

```c
#include <tdse/tdse.h>
#include <tdse_builder.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    const size_t np = 3;
    const size_t nh = 4;
    const double dt = 1e-3;

    /* Identity-like H: h[k] decays with k */
    double h[4 * 3 * 3] = {
        1, 0, 0,  0, 1, 0,  0, 0, 1,       /* h[0]: instantaneous */
        0.2, 0, 0, 0, 0.2, 0, 0, 0, 0.2,   /* h[1] */
        0.1, 0, 0, 0, 0.1, 0, 0, 0, 0.1,   /* h[2] */
        0.05, 0, 0, 0, 0.05, 0, 0, 0, 0.05 /* h[3] */
    };

    /* Uniform tap axis: tau[k] = k * dt */
    double tau[4] = {0.0, 1.0e-3, 2.0e-3, 3.0e-3};

    /* Configure and build */
    tdse_builder_t* b = NULL;
    tdse_builder_options_t opt = tdse_builder_options_init();
    opt.dt = dt;
    opt.nh = nh;
    opt.np = np;
    opt.nq = np;

    if (tdse_builder_create(&b) != TDSE_BUILDER_OK) return 1;

    tdse_h_desc_t h_desc;
    memset(&h_desc, 0, sizeof(h_desc));
    h_desc.struct_size = sizeof(h_desc);
    h_desc.np = (int32_t)np;
    h_desc.nq = (int32_t)np;
    h_desc.nh = (uint64_t)nh;
    h_desc.data = h;
    h_desc.tau = tau; /* optional explicit tau; h must already match that axis */

    if (tdse_builder_configure_ex(b, &opt) != TDSE_BUILDER_OK) return 1;
    if (tdse_builder_apply_h(b, &h_desc) != TDSE_BUILDER_OK) return 1;
    if (tdse_builder_write_pack(b, "tutorial.pack") != TDSE_BUILDER_OK) return 1;

    tdse_builder_destroy(b);
```

```
    printf("Pack written to tutorial.pack\n");
    return 0;
}
```

This is a **runnable example:** compile it against the SDK headers and libraries.

3.2 Step 2: Run the Builder

Compile and run (assuming SDK is installed at `/opt/tdse-sdk`):

```
cc tutorial_data.c \
    -I/opt/tdse-sdk/include \
    -L/opt/tdse-sdk/lib \
    -ltdse_builder -lm \
    -o tutorial_build
LD_LIBRARY_PATH=/opt/tdse-sdk/lib ./tutorial_build
```

Windows (MSVC):

```
cl tutorial_data.c ^
  /I"C:\tdse-sdk\include" ^
  /link "C:\tdse-sdk\lib\tdse_builder.lib"
tutorial_build.exe
```

Success: you see `Pack written to tutorial.pack`.

3.3 Step 3: Run the Runtime Step Loop

Create `tutorial_run.c`:

```
#include <tdse/tdse.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

static int read_file(const char* path, unsigned char** out, size_t* out_len) {
    FILE* f = fopen(path, "rb");
    long sz; size_t n; unsigned char* p;
    if (!f) return 1;
    fseek(f, 0, SEEK_END); sz = ftell(f); fseek(f, 0, SEEK_SET);
    p = (unsigned char*)malloc((size_t)sz);
    n = fread(p, 1, (size_t)sz, f); fclose(f);
    *out = p; *out_len = (size_t)sz;
    return n != (size_t)sz;
}
```

```c
int main(void) {
    const size_t np = 3;
    const double dt = 1e-3;
    const double t0 = 0.0;
    const size_t nsteps = 5;
    unsigned char* pack = NULL;
    size_t pack_len = 0;
    tdse_model_t* m = NULL;

    if (read_file("tutorial.pack", &pack, &pack_len) != 0) return 1;

    /* Create model with diagnostics */
    tdse_model_create_diagnostics_t diag = tdse_model_create_diagnostics_init();
    tdse_status_t st = tdse_model_create(pack, pack_len, &diag, &m);
    free(pack);
    if (st != TDSE_OK || m == NULL) {
        printf("Create failed: %s (pack_error=%u)\n",
               tdse_status_message(st), diag.pack_error_code);
        return 1;
    }

    /* Log metadata */
    tdse_model_info_t info = tdse_model_info_init();
    tdse_model_info(m, &info);
    printf("Model: np=%zu nq=%zu nh=%zu dt=%.6f\n",
           (size_t)info.np, (size_t)info.nq, (size_t)info.nh, info.dt);

    /* Prime at n = -1 */
    double primary0[3] = {0.0, 0.0, 0.0};
    tdse_step_begin(m, t0 - dt, dt);

    tdse_dense_block_t op_blk;
    memset(&op_blk, 0, sizeof(op_blk));
    double op[9] = {0};
    op_blk.struct_size = sizeof(op_blk);
    op_blk.rows = (int32_t)np; op_blk.cols = (int32_t)np;
    op_blk.ld = (int32_t)np; op_blk.layout = TDSE_LAYOUT_ROW_MAJOR;
    op_blk.data = op;
    tdse_step_op(m, &op_blk);
    tdse_step_commit(m, primary0);

    /* Main step loop */
    double hr[3], ir[3], dr[3], y[3];
    for (size_t n = 0; n < nsteps; ++n) {
        double primary[3] = {sin(0.1*n), cos(0.1*n), 0.5};

        tdse_step_begin(m, t0 + n * dt, dt);
```

```
        tdse_step_hr(m, hr);
        tdse_step_ir(m, ir);

        /* Host composes: y = op * primary + hr + ir */
        for (size_t r = 0; r < np; ++r) {
            y[r] = hr[r] + ir[r];
            for (size_t c = 0; c < np; ++c)
                y[r] += op[r * np + c] * primary[c];
        }

        tdse_step_commit(m, primary);
        tdse_step_dr(m, dr);

        printf("step=%zu y=[%+.6f %+.6f %+.6f] dr=[%+.6f %+.6f %+.6f]\n",
               n, y[0], y[1], y[2], dr[0], dr[1], dr[2]);
    }

    /* Bounded shutdown */
    tdse_model_destroy_options_t dopt = tdse_model_destroy_options_init();
    tdse_model_destroy_result_t dres = tdse_model_destroy_result_init();
    dopt.wait_timeout_ms = 250.0;
    st = tdse_model_destroy(m, &dopt, &dres);
    printf("Destroy: %s\n", tdse_status_message(st));
    return st == TDSE_OK ? 0 : 1;
}
```

Compile and run (assuming SDK is installed at `/opt/tdse-sdk`):

```
cc tutorial_run.c \
    -I/opt/tdse-sdk/include \
    -L/opt/tdse-sdk/lib \
    -ltdse -lm \
    -o tutorial_run
LD_LIBRARY_PATH=/opt/tdse-sdk/lib ./tutorial_run
```

Windows (MSVC):

```
cl tutorial_run.c ^
  /I"C:\tdse-sdk\include" ^
  /link "C:\tdse-sdk\lib\tdse.lib"
tutorial_run.exe
```

3.4 Step 4: Verify the Output

Expected output:

```
Model: np=3 nq=3 nh=4 dt=0.001000
step=0 y=[+0.000000 +0.000000 +0.000000] dr=[+... +... +...]
```

```
step=1 y=[+... +... +...] dr=[+... +... +...]
step=2 y=[+... +... +...] dr=[+... +... +...]
step=3 y=[+... +... +...] dr=[+... +... +...]
step=4 y=[+... +... +...] dr=[+... +... +...]
Destroy: OK
```

Success checklist:

- `tdse_model_create` returns `TDSE_OK` and `m != NULL`
- `tdse_model_info` reports `np=3, nq=3, nh=4, dt=0.001`
- Every step loop call returns `TDSE_OK`
- `tdse_model_destroy` returns `TDSE_OK`

If any step fails, see the failure table in Common First-Run Failure Modes and Troubleshooting.

If all four steps succeed, do not add more complexity here. Move directly to Examples Guide and pick the closest real integration path.

3.5 After This Tutorial

Once the first run works, do not keep extending this file as your main reference. Switch to the chapter that matches your real starting point:

Your next task	Go here next	Why
find the closest runnable reference	Examples Guide	fastest way to map from Hands-On Tutorial to real code
package real `H` or `IR` data	Builder and Data Contracts	production Builder checks and handoff rules
wire Runtime into a host simulator	Runtime Lifecycle and Step Execution	lifecycle, ownership, and step semantics
convert netlists or RAW data	Adapter Circuit	end-to-end circuit workflow
investigate a failure	Troubleshooting	symptom-first triage

For a compact mental model, keep these five handoffs in mind:

1. source data becomes Builder input
2. Builder writes a `.pack`
3. Runtime creates a model from that pack
4. the host runs prime, trial, and commit
5. shutdown happens after the step loop is quiescent

3.6 Key Patterns

The tutorial above exercises two patterns you will use repeatedly.

The Builder → Runtime pipeline:

```
create builder → configure → apply H → write pack → create model → step loop → destroy
```

The step loop (minimal form):

```
tdse_step_begin(m, t0 - dt, dt);   /* prime at n = -1 */
tdse_step_op(m, &op);
tdse_step_commit(m, primary_minus1);

for (uint64_t n = 0; n < nsteps; ++n) {
  tdse_step_begin(m, t0 + n * dt, dt);
  tdse_step_hr(m, hr);
  tdse_step_ir(m, ir);
  /* host solve: y = op * primary + hr + ir */
  tdse_step_commit(m, primary);
}
```

Production shutdown: use bounded destroy with an explicit wait budget.

```
tdse_model_destroy_options_t opt = tdse_model_destroy_options_init();
opt.wait_timeout_ms = 250.0;
tdse_model_destroy(m, &opt, &result);
```

For the mathematical meaning behind `H`, `IR`, `hr`, `ir`, and `op`, see Theory and Concepts.

3.6.1 C++ Wrapper

The C++ wrapper (`tdse::Model`) follows the same contract. See the Examples Guide for runnable C++ examples.

3.6.2 Common First-Run Failure Modes

When the API bring-up fails, it is usually one of these:

Symptom	Most Likely Cause	First Check
create fails immediately	bad pack bytes or missing diagnostics init	`create_diag`, pack validation, pack token
`step_ir` fails mid-run	IR horizon too short	simulation step index and `ir_nsteps`
`destroy` times out	another same-handle API is still active	thread ownership and wait budget
`INVALID_STATE` on `dr`	`dr` queried before commit or during active trial step	lifecycle order

CHAPTER 4

Examples Guide

> **Audience:** Anyone looking for a runnable reference implementation that matches their integration scenario. Start here after Installation and the first hands-on tutorial run.

Use this chapter to match the shipped evaluation examples to real integration paths. If you are not sure which chapter to read next, choose the example closest to your source data and follow the linked workflow from there.

Use this chapter in two passes:

1. pick the example family that matches your starting artifact
2. use the linked chapter as the durable reference after the example runs

For a one-day minimal integration, the usual path is:

1. run Hands-On Tutorial once so create, step, and destroy are already proven on your machine
2. choose exactly one example that matches your real starting point
3. switch from the example to the linked chapter and keep that chapter as the contract source for production code

4.1 Start Here

If your starting point looks like this	Best first example	Then read
I just finished Hands-On Tutorial and want one more minimal Runtime example	`runtime_bare_metal`	Runtime Lifecycle and Step Execution
I already have FRF data	`builder_frf_1p` or `builder_frf_2p`	Builder and Data Contracts
I need circuit-to-pack workflow	`workflow_cpp_quickstart`	Adapter Circuit
I need host embedding patterns	`mna_block_embed_2p`	Adapter Circuit
I need RAW conversion	`raw_import_api` or `raw_import_cpp`	Adapter Circuit and CLI Reference
I need observability or perf work	`telemetry_basic`	Telemetry and Profiler

Fastest evaluation picks:

- `runtime_bare_metal` when you want one more Runtime-only proof after Hands-On Tutorial
- `workflow_cpp_quickstart` when you want the shortest `netlist -> pack` PoC path
- `package_consumer_targets` or `package_consumer_pkgconfig` when you need proof that a downstream build can consume the installed SDK cleanly

If you are still in the first 30 minutes of an evaluation, ignore the full index below and choose only one of those three paths.

4.2 Example Index

4.2.1 Core Runtime

Example	What It Does
`runtime_bare_metal`	Load a pack, create a model, run a step loop — no Builder or Adapter involved
`variable_dt`	Pass different `dt` values at each step; demonstrates non-uniform time-stepping

4.2.2 Builder

Example	What It Does
`builder_frf_1p`	Build a pack from synthetic 1-port FRF data, then verify via Runtime step loop
`builder_frf_2p`	Same for a 2-port passive admittance
`builder_irc`	Apply IRC compression (MULTIRES_V1, MULTIRES_V2_MOMENT1) and compare tap counts

4.2.3 Adapter Circuit

Example	What It Does
`circuit_adapter_1p`	Minimal host circuit solver: stamp TDSE into a 1-port MNA system, solve, step
`circuit_adapter_2p`	Same for 2-port with explicit KCL stamping
`mna_block_embed_2p`	Embed TDSE 2-port contributions into a 4-node MNA network
`adapter_cpp_quickstart`	C++ wrapper: compile a netlist, compute Y matrix, AC probe, adaptive plan, build H
`adapter_cpp_threaded_lanes`	Multi-threaded pattern: per-worker handles with compile and request policy overrides
`workflow_cpp_quickstart`	High-level `netlist_to_pack` workflow: one call from SPICE file to .pack file
`netlist_to_pack_np`	Low-level Adapter-to-Builder handoff with manual/adaptive sweep planning (advanced)
`raw_import_api`	Convert PSS/E RAW to netlist via C API
`raw_import_cpp`	Same RAW conversion using the C++ wrapper (`raw::fileToNetlist`)
`nport_y2p_ac`	AC probe with an NPORT element loaded from a Touchstone `.y2p` file
`tail_scan`	Run `scanTailVsNfreq` to determine the required frequency resolution for a circuit
`region_prepare`	Resolve named circuit regions into explicit port definitions
`try_api`	Use the `try*` no-throw C++ API variants with manual error-code checking

4.2.4 Telemetry

Example	What It Does
`telemetry_basic`	Initialize telemetry service, attach to model, run steps, inspect JSON output

4.2.5 Deployment

Example	What It Does
`package_consumer_targets`	Consume TDSE via CMake `find_package(tdse)`
`package_consumer_pkgconfig`	Consume TDSE via `pkg-config`

4.3 Choosing the Right Example

```
What are you trying to do?
├─ "I just want to see the Runtime step loop"
│  → runtime_bare_metal
├─ "I have FRF data and want to build a pack"
│  → builder_frf_1p (1 port) or builder_frf_2p (2+ ports)
├─ "I want to compress a long impulse response"
│  → builder_irc
├─ "I have a circuit netlist and want end-to-end"
│  → circuit_adapter_1p or circuit_adapter_2p
├─ "I want to embed TDSE in my host solver"
│  → mna_block_embed_2p
├─ "I want to use the C++ wrapper API"
│  → adapter_cpp_quickstart
├─ "I want the fastest netlist-to-pack path"
│  → workflow_cpp_quickstart
├─ "I need full control of the Adapter-to-Builder pipeline"
│  → netlist_to_pack_np
├─ "I want to convert a PSS/E RAW file in code"
│  → raw_import_api (C) or raw_import_cpp (C++)
├─ "I want to check causality / find the right nfreq"
│  → tail_scan
├─ "I want to resolve a circuit region into ports"
│  → region_prepare
├─ "I want non-throwing error handling in C++"
│  → try_api
├─ "I want to run with variable time steps"
│  → variable_dt
├─ "I need multi-threaded adapter usage"
│  → adapter_cpp_threaded_lanes
├─ "I need step-level telemetry in production"
│  → telemetry_basic
├─ "I need to package TDSE for my build system"
│  → package_consumer_targets (CMake) or package_consumer_pkgconfig (pkg-config)
```

4.4 After The Tutorial

If you came here directly from Hands-On Tutorial, the cleanest next moves are:

- stay in Core Runtime if you are proving host loop ownership and shutdown behavior
- move to Builder if your real input is already validated FRF or impulse-response data
- move to Adapter Circuit if your real input starts as a netlist, RAW case, or NPORT workflow
- move to Telemetry or Deployment only after a normal create-step-destroy path is already stable

4.5 From Example To Real Integration

Use the example only long enough to prove the path. Then switch to the chapter that owns the real contract for production code.

If the example proved...	Durable chapter to keep open	Avoid jumping to this too early
pack create / destroy and one host step loop	Runtime Lifecycle and Step Execution	Variable `dt` or multi-model topics before the fixed-step path is stable
Builder packaging from FRF or `H` / `IR` data	Builder and Data Contracts	low-level Adapter flows you do not actually need
netlist or RAW input can become a qualified `.pack`	Adapter Circuit	direct low-level Builder tuning before the input path is trusted
package consumption works in a downstream build	Installation and Platform Notes	performance or plugin tuning before the install surface is stable

4.6 Building the Examples

If your evaluation package includes example projects, build them from the delivered example root:

```
cmake -S <example-root> -B <example-root>/build -DCMAKE_BUILD_TYPE=Release
cmake --build <example-root>/build -j"$(nproc)"
```

To build a specific example, use its documented CMake target name. For example:

```
cmake --build <example-root>/build --target tdse_example_builder_frf_1p
```

For package consumption examples, install the SDK first:

```
cmake -S <package-consumer-example-root> -B build -DCMAKE_PREFIX_PATH=/opt/tdse-sdk
cmake --build build
```

CHAPTER 5

Theory and Concepts

> **Audience:** Anyone who wants to understand what TDSE computes and why, before reading the API chapters. If you only need the fastest path to a working build, skip ahead to Getting Started and come back when you need to understand what the step-loop terms mean.

Use this chapter when you want the math behind TDSE to feel concrete enough to guide implementation choices. It links each concept to the Builder and Runtime operations you see later in the handbook, so the theory stays connected to code.

If you have already finished Hands-On Tutorial but the roles of `op`, `hr`, `ir`, `dr`, trial, and commit still feel abstract, read this chapter before you go deeper into the Runtime chapters.

5.1 Why TDSE Exists

5.1.1 The Problem with Conventional Solvers

Many engineering simulations involve dynamic systems whose time-domain behavior is described by differential equations. After spatial discretization or lumped-parameter modeling, these systems are typically expressed as ordinary differential equations (ODEs) or differential-algebraic equations (DAEs).

When a large linear subsystem is re-solved at **every simulation time step**, even though its linear characteristics have not changed, the repeated processing can introduce three compounding problems:

1. **Excessive computational cost.** For dense direct solves, a global linear solve can scale roughly as $O(N^3)$ in the number of degrees of freedom. Sparse and structured solvers reduce that cost, but repeated factorization or solve work can still dominate large coupled simulations.
2. **Numerical error accumulation.** Repeated matrix factorizations and global solves accumulate rounding errors, particularly in the presence of high-frequency dynamics or stiff coefficients.
3. **Scalability limits.** Model-reduction techniques such as modal truncation and static condensation reduce cost, but the accuracy trade-off must be checked against the frequency range and coupling behavior that matter to the host simulation.

5.1.2 The Impulse-Response Approach

TDSE takes a different path. Instead of re-solving the linear subsystem at every step, it processes the linear part **once** during a pre-computation phase and then reuses the result throughout the simulation through a convolution operation.

The key insight is:

- The linear subsystem is time-invariant. Its response to any port input can be fully characterized by a single impulse-response sequence.
- Cross-subsystem coupling can be decomposed into three computable terms: an independent-response term, a historical convolution term, and an instantaneous term.
- These terms are incorporated into the governing equation of the nonlinear subsystem, forming an **equivalent equation** that can be solved without ever re-opening the linear subsystem.

5.1.3 Complexity Reduction

Approach	Per-Step Cost	Notes
Conventional dense global solve	$O(N^3)$	N = total system degrees of freedom; sparse or structured solves can scale differently
TDSE convolution	$O(P^2 \cdot L)$	P = number of ports, L = effective impulse-response length

In practical engineering systems, P is often much smaller than N (for example, tens of ports versus thousands of internal states). This is the main reason TDSE can reduce per-step work while retaining the frequency-domain detail represented by the input data and fitting choices.

5.1.4 When TDSE Is the Right Choice

TDSE is designed for dynamic systems that can be partitioned into:

- a **linear time-invariant** subsystem whose characteristics do not change during simulation
- a **nonlinear and/or time-varying** subsystem that must be solved step-by-step

Typical application domains include:

Domain	Examples
Electrical / Electronic	power electronics, EMI/EMC, transmission-line networks
Mechanical / Structural	structural dynamics, multibody systems, vibration analysis
Hydraulic / Pneumatic	servo systems, fluid power networks
Thermal	heat transfer, thermo-mechanical coupling
Electromagnetic	transient EM, electromagnetic compatibility
Fluid dynamics	fluid motion, thermo-fluid-solid coupling
Control systems	controller-in-the-loop, plant-controller co-simulation
Multi-physics	any coupled system mixing the above domains

5.2 Core Concepts

The terms below are the ones that matter most when you move between theory, Builder inputs, and the Runtime step loop. Each definition is paired with the SDK operation that makes it visible in practice.

5.2.1 Dynamic System

A physical, engineering, or computational system whose state variables evolve over time. Described by time-domain equations relating state variables, their time derivatives, and inputs.

SDK mapping: the entire model you build and run through Builder + Runtime.

5.2.2 Generalized Differential Equations

The mathematical formulation used to describe dynamic system behavior during numerical simulation. Includes ODEs, DAEs, and semi-discrete equations obtained after spatial discretization (finite-element, finite-volume, finite-difference, boundary-element, or spectral-element methods).

SDK mapping: the equations that the host solver integrates. TDSE does not solve them directly; it provides operator terms that the host incorporates into its own solve.

5.2.3 System Partitioning

Splitting a coupled dynamic system into two interacting parts for numerical analysis:

- **First subsystem** (linear, time-invariant): processed once; its characteristics are captured as precomputed data
- **Second subsystem** (nonlinear, time-varying, event-driven): solved step-by-step by the host

Interactions between subsystems are represented through **port variables**.

SDK mapping: this is a modeling decision made upstream of the SDK. The SDK assumes partitioning is already complete when you provide `H` and `IR` artifacts.

5.2.4 Port Function

A functional relationship that describes coupling between subsystems through one or more interface variables. A port function represents how an output quantity of one subsystem influences an input quantity of another.

Port variables are physically meaningful conjugate pairs representing energy or signal exchange: voltage/current in electrical systems, force/displacement in mechanical systems, pressure/flow in hydraulic systems.

SDK mapping: the dimension `np` (number of primary inputs / ports) that you configure in Builder and that the host primary vector must match.

5.2.5 Impulse-Response Sequence (H)

A discrete-time sequence `h[n]` that characterizes how the linear subsystem responds over time to a unit impulse applied at a port. It captures the time-domain transfer behavior of the linear subsystem.

For multi-port systems, the impulse response is a three-dimensional array indexed by `(time lag, output port, input port)`.

SDK mapping: the `H` tensor you pass to `tdse_builder_apply_h()`. Layout: `[nh][nq][np]`. At runtime, `H[0]` becomes the instantaneous operator (`op`), and `H[1..nh-1]` contribute to the delayed-history term (`hr`).

5.2.6 Independent-Response Sequence (IR)

A discrete-time sequence `x0[n]` that characterizes the intrinsic dynamic behavior of the linear subsystem under a zero-input condition (i.e., when coupling inputs from other subsystems are not applied). It represents the "free response" of the subsystem.

SDK mapping: the `IR` sequence you optionally pass to `tdse_builder_apply_ir()`. At runtime, it is queried via `tdse_step_ir()`.

5.2.7 Historical Term

A contribution generated through time-domain convolution of the impulse-response sequence with past values of port input quantities. It represents accumulated effects of prior subsystem interactions over time.

In the convolution sum:

```
historical = sum_{k=0}^{n-1} h[n-k] * y[k]
```

SDK mapping: the `hr` vector returned by `tdse_step_hr()`. Length is `nq`.

5.2.8 Instantaneous Term

A contribution associated with the zero-time component of the impulse-response sequence. It represents the immediate coupling effect at the current simulation time.

In the convolution sum:

```
instantaneous = h[0] * y[n]
```

SDK mapping: the `op` operator returned by `tdse_step_op()`. Shape is `nq x np` (full view) or `np x np` (square view).

5.2.9 Equivalent Equation

The reformulated governing equation of the nonlinear subsystem, constructed by incorporating the three terms above:

```
g(y, dy/dt, ..., t) = x0[n] + (historical) + (instantaneous) * y[n]
```

This equivalent equation can be solved using standard time-stepping integration **without ever re-solving the linear subsystem**.

SDK mapping: the trial-solve relation that the host evaluates each step:

```
y_trial = op * primary_trial + hr + ir
```

5.2.10 Numerical Time-Stepping Integration

The process of advancing a numerical solution of time-domain equations forward in time using discrete time steps. The time steps may be fixed or variable, and the integration may employ explicit, implicit, or semi-implicit schemes.

SDK mapping: the host-owned simulation loop that calls `tdse_step_begin/op/hr/ir/commit` each step. TDSE supplies the operator terms; the host owns the integration strategy.

5.3 The Four-Step Method

The TDSE framework is organized around four processing steps. Steps 1-3 happen during pre-computation (before the simulation loop). Step 4 is the runtime loop itself.

5.3.1 Step 1. System Partitioning

What happens: The coupled dynamic system is divided into a linear time-invariant first subsystem and a nonlinear/time-varying second subsystem. Port functions define the coupling interface.

In practice: This is a modeling decision. You decide which portions of your system are linear and extract their frequency response or time-domain characteristics using your own tools or the TDSE Adapter layer (currently Adapter Circuit for circuit-domain workflows).

SDK entry points: Adapter Circuit commands (`tdse adapter circuit matrix`, `tdse adapter circuit series`) produce the FRF/port data that feeds into Builder.

5.3.2 Step 2. Linear Subsystem Characteristic Computation

What happens: The computing device computes two discrete sequences for the linear subsystem:

1. **Independent-response sequence** `x0[n]` — the free response under zero coupling input
2. **Impulse-response sequence** `h[n]` — the response to a unit impulse at each port

SDK entry points: If you have frequency-domain data, `tdse_builder_h_from_spectrum()` performs the frequency-to-time conversion internally. If you already have time-domain `H`, pass it directly to `tdse_builder_apply_h()`.

5.3.3 Step 3. Equivalent Equation Construction

What happens: Using the impulse-response sequence, cross-subsystem coupling is analytically decomposed into three terms and incorporated into the governing equation of the nonlinear subsystem:

Term	Meaning	Pre-computable?
Independent-response	free response of the linear subsystem	yes
Historical	convolution of past port inputs with impulse response	yes (from past steps)
Instantaneous	current-step coupling via `h[0]`	combined with host solve

The instantaneous term is folded into the left-hand-side operator of the nonlinear subsystem, so the equivalent equation retains the same structural form as the original DAE/ODE.

SDK entry points: This decomposition happens automatically inside Builder when it writes the pack. At runtime, `tdse_step_op()`, `tdse_step_hr()`, and `tdse_step_ir()` expose the three terms.

5.3.4 Step 4. Time-Domain Numerical Solution

What happens: The host advances the equivalent equation through time-stepping integration. Because the linear dynamics have already been incorporated via convolution, each step only needs to solve the nonlinear subsystem locally — no global linear solve is required.

SDK entry points: The runtime step loop: `tdse_step_begin` -> `tdse_step_op` / `tdse_step_hr` / `tdse_step_ir` -> host solve -> `tdse_step_commit`.

5.4 Partitioning Your System

System partitioning is the single most important modeling decision when using TDSE. The guidance below is meant to help you make that split once, cleanly, before you build a pack.

5.4.1 Linear vs Nonlinear: The Split

The first subsystem should contain everything that is:

- **linear**: superposition applies; no state-dependent coefficients
- **time-invariant**: coefficients do not change with time or with operating point

Everything else goes into the second subsystem: nonlinear elements, time-varying parameters, switches, event-driven components, control logic.

5.4.2 Port Variables

The interface between subsystems is defined by **port variables**. Each port carries a pair of conjugate physical quantities:

Domain	Effort variable	Flow variable
Electrical	voltage	current
Mechanical	force	displacement
Hydraulic	pressure	flow rate
Thermal	temperature	heat flow

The number of ports (`np` in the SDK) is the number of independent interface channels. Choosing fewer ports reduces per-step cost ($O(P^2 \cdot L)$) but may coarsen the coupling representation.

5.4.3 Practical Partitioning Guidelines

1. **Put all linear passive networks in the first subsystem.** Transmission lines, RLC networks, structural matrices, and thermal conductance networks are typical candidates.

2. **Put switches, nonlinear devices, and controllers in the second subsystem.** Power electronics switches, diode characteristics, contact elements, and controller logic change behavior during simulation and cannot be pre-characterized.
3. **Choose ports at natural physical interfaces.** Wherever you would place a measurement probe or a boundary condition is usually a good port location.
4. **Minimize port count when possible.** If two ports always carry the same signal (redundant coupling), consider merging them. Cost scales with P^2.

5.4.4 What If the System Cannot Be Cleanly Partitioned?

Some systems have linear elements whose parameters depend on operating-point quantities computed by the nonlinear subsystem (e.g., a temperature-dependent resistance). In such cases:

- If the parameter variation is slow relative to the simulation dynamics, treat it as approximately constant and re-build the pack when the operating point changes significantly.
- If the variation is fast, consider placing the element in the nonlinear subsystem.

5.5 Obtaining the Impulse Response

The impulse-response sequence `H` is the central pre-computed artifact in the TDSE framework. There are three paths to obtain it.

5.5.1 Path Comparison

Path	Input Data	Best When	SDK Support
Time-domain	Direct simulation of impulse excitation	You already have a time-domain solver for the linear subsystem	Direct `H` via `tdse_builder_apply_h()`
Frequency-domain	Frequency response function (FRF) at discrete frequencies	Your upstream tools produce FRFs (e.g., AC sweep from circuit simulator)	`tdse_builder_h_from_spectrum()`
Modal decomposition	Eigenstructure or modal parameters	You have dominant mode information from FEA or modal analysis	Convert to `H` externally, then `tdse_builder_apply_h()`

5.5.2 Frequency-Domain Path in Detail

When your source data is a frequency response function H(w), the conversion to a time-domain impulse response involves:

1. **Frequency sampling.** Define a positive-frequency grid `omega[k]` covering the bandwidth of interest.
2. **Inverse transform.** Compute the discrete inverse Fourier transform to obtain the time-domain sequence.
3. **Causality correction.** Band-limited frequency data can produce non-causal (pre-response) artifacts in the time domain. The computing device applies correction to enforce physical causality.
4. **Truncation and windowing.** The resulting sequence may need to be truncated to a practical length and windowed to suppress high-frequency ringing.

Steps 2-4 are handled internally by `tdse_builder_h_from_spectrum()`.

5.6 Causality Correction Guide

When converting frequency-domain data to time-domain impulse responses, the finite frequency sampling bandwidth can introduce non-physical artifacts. Causality correction restores physical consistency.

The Builder exposes four correction methods. The goal here is to show what each one changes and how to choose without guesswork.

5.6.1 Two Families of Correction

Correction methods fall into two families, depending on which part of the frequency response you use to recover the full complex spectrum.

Family A: Real/Imaginary Component Methods

For a causal linear time-invariant system, the real and imaginary parts of the frequency response satisfy reciprocal Hilbert-transform relationships (Kramers-Kronig). Given one component, the other can be reconstructed.

Family B: Magnitude/Phase Methods

For minimum-phase systems, logarithmic magnitude and phase are coupled analytically. Given either magnitude or phase, the other can be reconstructed via a Hilbert transform of the logarithmic magnitude.

5.6.2 Builder Correction Methods

Method	Family	Input	What It Does	When to Use
Reconstruct from Real (`TDSE_BUILDER_CORRECTION_RECONSTRUCT_FROM_REAL`)	A (real/imag)	Real part of FRF	Recovers the full complex spectrum from the real part via Kramers-Kronig. Preserves total energy and DC component.	General-purpose default; works well when the real part is reliable
Reconstruct from Imag (`TDSE_BUILDER_CORRECTION_RECONSTRUCT_FROM_IMAG`)	A (real/imag)	Imaginary part of FRF	Recovers the full complex spectrum from the imaginary part. Suppresses even-symmetric response components that cause pre-response artifacts.	When the imaginary part is more reliable, or when FRF data shows symmetric artifacts
Reconstruct from Magnitude (`TDSE_BUILDER_CORRECTION_RECONSTRUCT_FROM_MAG`)	B (mag/phase)	Magnitude only	Derives phase from log-magnitude via minimum-phase Hilbert transform, recovering the full complex spectrum.	When only magnitude data is available or reliable; phase is missing or noisy
Reconstruct from Phase (`TDSE_BUILDER_CORRECTION_RECONSTRUCT_FROM_PHASE`)	B (mag/phase)	Phase only	Derives magnitude from phase via Hilbert transform, recovering the full complex spectrum.	When only phase data is available or reliable; magnitude is missing or noisy

5.6.3 Selection Guide

```
What frequency-domain data do you have?
├─ Real part of FRF → Reconstruct from Real (general-purpose, energy-preserving)
├─ Imaginary part of FRF → Reconstruct from Imag (suppresses symmetric artifacts)
├─ Magnitude only → Reconstruct from Magnitude
└─ Phase only → Reconstruct from Phase
```

5.6.4 Correction Method vs. Pack Quality

The correction method directly affects the numerical behavior of the resulting pack:

- **Wrong choice** can produce: non-causal pre-responses, energy drift, or spectral distortion.
- **Right choice** ensures: causal impulse response, energy consistency, and stable convolution behavior.

If you are unsure, start with **Reconstruct from Real** and compare runtime results against a known reference. Switch only when you can verify the change improves accuracy for your specific system.

> **IMPORTANT**
> Once a pack is written, Runtime sees only the resulting `H` tensor. It has no knowledge of which correction method was used. Support triage should always separate the question "did Builder construct the intended H?" from "did Runtime execute the pack correctly?"

5.7 Practical Considerations

5.7.1 Truncation and Windowing

The impulse response of a linear subsystem typically has a finite but non-strictly-bounded duration. Directly using an unprocessed sequence within a finite numerical window can introduce:

- **High-frequency ringing** from abrupt truncation
- **Spectral leakage** in subsequent convolution operations

Mitigation strategies applied during pre-computation:

- **Truncation:** limit the impulse response to `nh` taps (the `nh` parameter in Builder configure). Choose `nh` large enough to capture the significant dynamic content.
- **Windowing:** apply a tapering window (e.g., exponential decay, half-cosine) to the tail of the sequence to smooth the truncation boundary and suppress ringing.

SDK mapping: `nh` in `tdse_builder_options_t` controls the history depth. Larger `nh` captures more dynamic content but increases per-step cost.

5.7.2 Resampling and Interpolation

The impulse-response sequence is precomputed at a fixed sampling interval, but the host integrator may use a different step size (especially under variable-step schemes).

When the convolution time grid does not align with the integration grid, the computing device interpolates or resamples the sequence to maintain:

- **Time alignment** between convolution terms and the numerical integrator
- **Energy consistency** of accumulated convolution contributions
- **Numerical stability** of the overall stepping process

SDK mapping: When `h_desc.tau` is provided, the H tensor must already be weighted for the explicit time axis. Use `tdse_h_uniform_to_tau_1d()` or `tdse_h_uniform_to_piecewise_tau_1d()` to convert uniform-grid taps to an explicit axis before passing them to Builder.

5.7.3 Variable Time-Step Integration

In variable-step integration, the step size may change from one step to the next. TDSE accommodates this through the resampling mechanism: before each step, the impulse-response and independent-response sequences are re-evaluated on the current integration grid.

The practical implication for host integrators:

- The `dt` passed to `tdse_step_begin()` can differ from step to step
- Convolution terms remain properly aligned across different time scales
- Energy consistency is maintained automatically

5.7.4 Multi-Port Convolution

For systems with multiple coupled ports, the impulse response becomes a three-dimensional array where each input port has its own set of response traces at all output ports.

During simulation, convolution is performed in parallel along the port dimension. The per-step computational cost scales as $O(P^2 \cdot L)$ where P is the number of ports and L is the effective impulse-response length.

SDK mapping: the H tensor shape `[nh][nq][np]` directly encodes this multi-port structure. At runtime, `tdse_step_op()` returns the full `nq x np` operator, and `tdse_step_hr()` returns the `nq`-length history vector that already accounts for all port interactions.

5.7.5 First Subsystem State Reconstruction

After completing the simulation of the nonlinear subsystem, the time-domain response of the linear subsystem can be reconstructed using the convolution expression:

```
x[n] = x0[n] + sum_{k=0}^{n-1} h[n-k] * y[k]
```

This is a post-processing step: substitute the solved `y[n]` (from the second subsystem) into the convolution formula to recover the linear subsystem state trajectory.

SDK mapping: TDSE Runtime does not perform this reconstruction automatically. The host can implement it using the committed `primary` vectors and the original `H` and `IR` data.

5.8 Pack Representation: Y+ISC vs Z+VOC

The pack stores not only the numerical `H` and `IR` arrays but also a representation label that tells downstream consumers what physical meaning the port variables carry. Understanding this label is essential when your model contains an independent-response sequence (IR).

5.8.1 Two Canonical Representations

Representation	Form	Port Input (primary)	Port Output (secondary)	Relationship
Y + ISC	`FLOW_FROM_EFFORT`	effort variable (e.g., voltage)	flow variable (e.g., current)	`i = Y * v + ISC`
Z + VOC	`EFFORT_FROM_FLOW`	flow variable (e.g., current)	effort variable (e.g., voltage)	`v = Z * i + VOC`

The terminology comes from electrical engineering but the concept generalizes:

Domain	Effort (Y+ISC input)	Flow (Y+ISC output)
Electrical	voltage	current
Mechanical (translational)	displacement	force
Hydraulic	pressure	flow rate
Thermal	temperature	heat flow

5.8.2 Why It Matters

The representation determines how the host must interpret the step-loop terms:

- In **Y + ISC** mode, `primary` is the effort vector, `op` is an admittance, `hr + ir` form the short-circuit current contribution, and `dr` gives the current response.
- In **Z + VOC** mode, `primary` is the flow vector, `op` is an impedance, `hr + ir` form the open-circuit voltage contribution, and `dr` gives the voltage response.

The math inside Runtime is identical in both cases — the same convolution engine runs — but the host must know which physical convention applies to correctly wire `primary`, interpret `dr`, and couple the result back into the nonlinear subsystem.

5.8.3 Setting the Representation

```
tdse_builder_set_pack_meta(b,
    TDSE_PACK_FORM_FLOW_FROM_EFFORT,  /* or TDSE_PACK_FORM_EFFORT_FROM_FLOW */
    TDSE_PACK_DOMAIN_ELECTRICAL);     /* domain hint for downstream tooling */
```

When `IR` is present, setting the representation is **not optional**. The IR sequence has different physical meaning depending on whether it represents ISC (short-circuit current) or VOC (open-circuit voltage).

5.8.4 When Representation Is Ambiguous

For packs without IR, `TDSE_PACK_FORM_UNKNOWN` is acceptable because the convolution math is representation-independent. But once IR is attached, leaving the form as `UNKNOWN` creates a support hazard: downstream users cannot tell whether the IR term should be added to an admittance equation or an impedance equation.

If your team cannot answer which representation the pack carries, do not write the pack yet. Resolve the ambiguity at the Builder stage — it cannot be fixed later at the Runtime stage.

5.9 EMT Simulator Integration

The mapping below is for readers who already think in EMT solver equations and want to line those terms up with the TDSE step loop.

5.9.1 Norton Equivalent Interpretation

The TDSE step-loop trial equation:

```
y_trial = op * primary_trial + hr + ir
```

takes a direct Norton equivalent form in the EMT nodal equation:

```
G_eff * v(n+1) = J_history + J_source
```

The mapping is:

EMT Term	TDSE Term	Meaning
`G_eff`	`op` (from `tdse_step_op`)	Effective nodal conductance matrix (instantaneous operator `H[0]`)
`J_history`	`hr` (from `tdse_step_hr`)	History current source from delayed convolution of past port voltages
`J_source`	`ir` (from `tdse_step_ir`)	Independent-response current source (ISC: short-circuit current under zero coupling voltage)
`v(n+1)`	`primary_trial`	Port voltage vector at the next time point
`i(n+1)`	`y_trial`	Port current vector (the Norton injection)

In Y+ISC representation (admittance form):

```
i(n+1) = Y_eff * v(n+1) + ISC_history + ISC_source
```

Where:

- `Y_eff = op` is the instantaneous admittance looking into the linear subsystem
- `ISC_history = hr` is the convolution history current
- `ISC_source = ir` is the independent-source current (the linear subsystem's free response)

In Z+VOC representation (impedance form), the roles swap:

```
v(n+1) = Z_eff * i(n+1) + VOC_history + VOC_source
```

5.9.2 Integration with EMT Nodal Solve

A typical EMT solver builds a global nodal equation at each time step:

```
G_global * v_global(n+1) = J_global(n+1)
```

When TDSE represents a linear subsystem, the host injects the TDSE Norton equivalent into the global equation:

1. **Stamp `op` into the global conductance matrix.** Add `op[row, col]` to the corresponding entries of `G_global` at the port node positions. This is identical to stamping a conductance matrix from a conventional companion model.
2. **Stamp `hr + ir` into the global RHS vector.** Add `hr[row] + ir[row]` to `J_global` at the port node positions. This is the history plus source injection.

3. **Solve the global system.** The EMT solver solves `G_global * v = J_global` as usual.
4. **Extract the accepted port voltages.** After the global solve, extract the port voltage solution as `primary_accepted`. Call `tdse_step_commit(model, primary_accepted)`.
5. **Advance history.** The commit call updates internal convolution history. The next `tdse_step_hr` call will incorporate the just-accepted port voltages.

5.9.3 Worked Example: Single-Port Transmission Line

Consider a 500 kV, 100 km single-phase transmission line with characteristic admittance `Y_c` and propagation delay `tau`:

```
i(t) = Y_c * v(t) + h_history(t)
```

In TDSE terms:

- `np = nq = 1` (single port)
- `op = Y_c` (the instantaneous characteristic admittance)
- `hr = h_history` (the reflected-wave history from past terminal voltages)
- `ir = 0` or a known source injection (if the remote end has a known waveform)

The integration into a 50 us EMT step loop:

```c
/* Inside the EMT solver main loop, at the transmission line port: */
tdse_step_begin(model, t, dt);

double op_val;
tdse_dense_block_t op_blk = { ... };
tdse_step_op(model, &op_blk);         /* op = Y_c */

double hr_val, ir_val;
tdse_step_hr(model, &hr_val);         /* hr = reflected wave contribution */
tdse_step_ir(model, &ir_val);         /* ir = remote-end source (if any) */

/* Stamp into global EMT nodal equation */
G_global[port_node][port_node] += op_val;
J_global[port_node] += hr_val + ir_val;

/* After global solve: */
double v_port = v_solution[port_node];  /* extracted from global solution */
tdse_step_commit(model, &v_port);

/* Optional: read direct response */
double dr_val;
tdse_step_dr(model, &dr_val);           /* dr = Y_c * v_port */
```

Key points:

- The TDSE model replaces the conventional Bergeron/traveling-wave companion model
- No explicit propagation delay bookkeeping in the host – TDSE handles it through the convolution history
- The host only stamps conductance and current, then extracts the solution voltage

- Multi-port lines or cables use `np > 1` with the same pattern: `op` is `np x np` and `hr`, `ir` are `np`-length vectors

5.9.4 EMT Integration Anti-Patterns

Avoid these common mistakes when embedding TDSE in an EMT solver:

1. **Double-stamping conductance.** If the host already has a companion model for the same port nodes, remove it before stamping `op`. The TDSE operator replaces the conventional companion, it does not supplement it.
2. **Confusing Y+ISC with Z+VOC.** In Y+ISC mode, `primary` is voltage and `y_trial` is current (injection). In Z+VOC mode, `primary` is current and `y_trial` is voltage. Stamping the wrong form into the nodal equation produces silently incorrect results.
3. **Committing before the global solve.** The commit must use the *accepted* port voltages from the global solution, not the trial voltages. The sequence is: query terms -> global solve -> extract solution -> commit.
4. **Ignoring the prime step.** EMT solvers typically start from steady-state initial conditions. The TDSE prime step at `n = -1` establishes the initial history. If the host has known initial port voltages, use those as `primary_minus1` in the prime commit.

5.9.5 Multi-Port Subsystem Integration

For N-port subsystems (e.g., a multi-conductor cable or a transformer equivalent):

```
G_global[node_i][node_j] += op[row][col]     /* for each port pair */
J_global[node_i] += hr[row] + ir[row]         /* for each port */
```

Where the mapping between TDSE port indices (`row`, `col`) and EMT global node indices (`node_i`, `node_j`) depends on how ports were defined during system partitioning.

The `nq > np` case allows the host to access additional output equations beyond the port count. In EMT terms, this corresponds to internal measurements (e.g., receiving-end voltage of a transmission line) without adding extra port nodes to the global system.

5.10 Numerical Accuracy Considerations

The goal here is to turn the accuracy story into engineering guidance: where error enters the TDSE pipeline, and which parts usually matter most.

5.10.1 Spectrum-to-H Conversion Accuracy

The frequency-domain-to-time-domain conversion performed by `tdse_builder_h_from_spectrum()` introduces error from three sources:

Error Source	Magnitude	Mitigation
Finite bandwidth truncation	depends on spectral content beyond `w_max = pi/dt`	ensure `dt` captures highest significant frequency
Causality correction	method-dependent (see below)	choose correction method based on data completeness
Windowing/truncation at `nh`	`O(h[nh])` – the tail value at truncation	verify `H[nh-1]` is negligible

Correction method accuracy comparison:

Method	Typical Error	Best Case	Notes
None	high (non-causal pre-response visible)	N/A	Not recommended for production
Reconstruct from Real	< 1% energy error for well-sampled FRF	< 0.1% for dense uniform grids	General-purpose default
Reconstruct from Imag	< 0.5% for data with even-symmetric artifacts	< 0.1%	Better when FRF has symmetric truncation artifacts
Reconstruct from Magnitude	5-15% magnitude error typical	< 2% for minimum-phase systems	Phase accuracy depends on Hilbert transform quality
Reconstruct from Phase	5-15% phase error typical	< 2%	Only when phase is missing/unreliable

Verification: After conversion, compare the time-domain impulse response against the expected physical behavior. The first sample `H[0]` should be real-valued (no imaginary component for a real system). The tail `H[nh-1]` should decay to near the machine epsilon or at least below the application's accuracy requirement.

5.10.2 IRC Compression Accuracy

Builder IRC compresses the tail of the impulse response into exponential decay parameters. The accuracy is controlled by `tail_tolerance`:

`tail_tolerance`	Typical Pack Size Reduction	Accuracy
1e-4	50-70%	moderate; acceptable for initial screening
1e-6	30-50%	good; suitable for most power-system applications
1e-8	15-30%	high; recommended for production accuracy
1e-10	5-15%	very high; marginal compression benefit

The error bound is: the compressed tail differs from the original by at most `tail_tolerance` per tap in the L2 sense.

Verification: Run the Profiler IRC scan (`tdse profiler irc-scan`) to compare compressed vs. uncompressed results before committing to production.

5.10.3 FP32 vs FP64 History Precision

When `tdse_compute_precision_set(model, TDSE_COMPUTE_PRECISION_FP32)` is used, the history convolution uses single-precision accumulation:

Precision	Per-tap rounding error	Cumulative error for `nh` taps	Recommended when
FP64 (default)	~2.2e-16	negligible	always preferred for accuracy
FP32	~1.2e-7	up to `nh * 1.2e-7`	`nh < 1000` and application tolerates ~1e-4 relative error

Guideline: Use FP32 only when `nh` is large enough that the performance benefit outweighs the accuracy cost, and when the host application's error budget can absorb the additional `~nh * 1.2e-7` perturbation in the history term.

5.10.4 Truncation and Windowing

The impulse response is truncated to `nh` taps. The truncation error is:

```
E_trunc = sum_{k=nh}^{infinity} |h[k]|^2
```

Practical rule: verify that `|H[nh-1]|` is at least 40 dB below `|H[0]|` (i.e., the tail has decayed by a factor of 100 or more). If not, increase `nh` and rebuild the pack.

5.10.5 Accuracy Verification Workflow

1. Build pack with generous `nh` (e.g., 2x the expected minimum)
2. Run a reference simulation with known analytical solution
3. Measure the error: `|y_TDSE - y_exact| / |y_exact|`
4. If error exceeds tolerance:
   - Check if `H[nh-1]` has decayed sufficiently
   - Try a different causality correction method
   - Compare compressed (IRC) vs. uncompressed results
   - Switch from FP32 to FP64 history precision
5. Archive the accuracy verification alongside the pack for release evidence

5.11 Now Continue Here

Once the conceptual model is clear, use the next chapter based on the question you actually need to answer:

If you need to...	Go next
choose the right Runtime API family quickly	Runtime API Summary
package validated `H` or `IR` data into a pack	Builder and Data Contracts
understand create / destroy / ownership behavior	Runtime Lifecycle
understand the trial / commit simulation loop	Step Execution

CHAPTER 6

Runtime Integration

Use this chapter as the Runtime lookup page when the real question is not “how does the step loop work?” but “which API family should I reach for first?” It is intentionally compact: a map into the Runtime APIs, not a replacement for the generated API Reference or the deeper lifecycle chapters.

Read this chapter first when you already know TDSE belongs in your host loop but do not yet know whether the next stop should be Builder, lifecycle, step execution, or the generated API reference.

> **Related Chapters** - For lifecycle rules, see Runtime Lifecycle. - For trial and commit semantics, see Step Execution. - For Builder APIs, see Builder and Data Contracts.

6.0.1 How This Runtime Part Is Split

The Runtime material stays split into short sections on purpose. Most customer teams do not read Runtime linearly; they jump to the contract surface that matches the host question in front of them.

If your question is…	Read this chapter first
which API family do I need	this chapter
who owns the handle and how do I shut it down	Runtime Lifecycle
what exact calls happen every accepted step	Step Execution
what happens under overlap, races, or teardown contention	Concurrency and Shutdown
how many models, threads, or GPU shares can I afford	Threading and Scaling and Multi-Model Patterns

6.0.2 Scope Boundary

This chapter covers Runtime APIs only:

- core model lifecycle
- runtime snapshots and diagnostics
- trial/commit step execution
- runtime-facing `perf` / `ext` controls that operate on `tdse_model_t*`
- C++ wrappers for the same runtime-facing surface

It does not list Builder or Adapter Circuit APIs. For those chapters, use Builder and Data Contracts and Adapter Circuit.

6.0.3 Core Runtime Lifecycle APIs

API	First Job	Use It When	Details Live In
`tdse_model_create(...)`	create a runtime handle from pack bytes	every normal create path	Runtime Lifecycle
`tdse_model_close(...)`	request a close without waiting for teardown	you need an immediate lifecycle answer	Runtime Lifecycle
`tdse_model_destroy(...)`	shut down with an explicit wait policy	normal host-managed shutdown	Runtime Lifecycle
`tdse_model_release(...)`	perform terminal cleanup in finalizer-style paths	destructor, guard, or finally cleanup	Runtime Lifecycle
`tdse_model_reset(...)`	clear committed state before intentional reuse	you are reusing a live handle on purpose	Runtime Lifecycle

6.0.4 Runtime Version And Introspection APIs

API	First Job	What It Returns	Notes
`tdse_version_string(...)`	get full version text	full semantic version string	stable runtime version text
`tdse_version_major(...)`	get major version	semantic version major component	integer component
`tdse_version_minor(...)`	get minor version	semantic version minor component	integer component
`tdse_version_patch(...)`	get patch version	semantic version patch component	integer component
`tdse_pack_inspect(...)`	inspect pack dimensions before create	`tdse_model_info_t` decoded from pack bytes	no runtime handle allocation
`tdse_pack_inspect_ex(...)`	inspect pack metadata before create	`tdse_pack_summary_t` with pack version and payload summary	metadata-only path

6.0.5 Runtime Snapshot And Diagnostics APIs

API	First Job	What It Returns	Notes
`tdse_model_info(...)`	read static model metadata	model dimensions and fixed properties	`out_info->struct_size` must be set
`tdse_model_state_info(...)`	read live execution state	current runtime state	`out_state->struct_size` must be set
`tdse_model_last_error_info(...)`	read the last failure snapshot	most recent non-OK runtime result on the handle	sticky until next failure or `tdse_model_reset(...)`
`tdse_pack_error_token(...)`	decode create-time pack errors	readable token for a pack-error code	uses `tdse_model_create_diagnostics_t::pack_error_code`
`tdse_status_message(...)`	decode runtime status codes	readable text for a runtime status code	status text only

6.0.6 Step Execution APIs

Use this table to find the right step API quickly. For the full trial/commit model, sequencing rules, and caller buffer contracts, switch to Step Execution.

API	First Job	Typical Output	Details Live In
`tdse_step_begin(...)`	start or re-enter a trial step	none	Step Execution

API	First Job	Typical Output	Details Live In
`tdse_step_op(...)`	read the instantaneous operator	dense block `Np x Np` or `Nq x Np`	Step Execution
`tdse_step_hr(...)`	read delayed-history contribution	vector length `Nq`	Step Execution
`tdse_step_ir(...)`	read independent-response contribution	vector length `Nq`	Step Execution
`tdse_step_commit(...)`	advance committed history	none	Step Execution
`tdse_step_dr(...)`	read committed direct response	vector length `Nq`	Step Execution

6.0.7 Runtime-Facing Perf And Ext APIs

6.0.7.1 `perf` APIs

API	First Job	Notes
`tdse_backend_registry_count(...)` / `tdse_backend_registry_get(...)`	enumerate available backends	capability discovery
`tdse_backend_set(...)`	choose a backend before stepping	pre-step only
`tdse_backend_get_active(...)`	inspect the active backend	reads last effective backend on the model
`tdse_backend_apply_plan(...)`	apply a JSON backend plan	pre-step only
`tdse_cuda_backend_set_config(...)` / `tdse_cuda_backend_get_config(...)`	set or inspect CUDA options	setter is pre-step only
`tdse_compute_precision_set(...)` / `tdse_compute_precision_get(...)`	set or inspect CPU history precision	setter is pre-step only
`tdse_local_threads_set(...)` / `tdse_local_threads_get(...)`	set or inspect local CPU thread override	setter is pre-step only
`tdse_backend_id_name(...)` / `tdse_backend_id_from_name(...)`	map backend ids and names	token lookup helpers
`tdse_perf_get_build_features_json(...)`	export build/runtime feature flags	pass `out_json = NULL` to query required bytes

6.0.7.2 `ext` APIs

API	First Job	Notes
`tdse_ext_set_log_callback(...)`	register a structured log sink	global setting
`tdse_ext_set_log_level(...)` / `tdse_ext_get_log_level(...)`	set or inspect log threshold	global setting
`tdse_ext_log_emit(...)`	emit a structured log record	integration helper
`tdse_ext_set_deterministic_mode(...)` / `tdse_ext_get_deterministic_mode(...)`	control reproducibility mode	global setting
`tdse_ext_set_runtime_guard_config(...)` / `tdse_ext_get_runtime_guard_config(...)`	control runtime guard behavior	warnings only; no numerical change
`tdse_ext_get_runtime_guard_metrics(...)` / `tdse_ext_reset_runtime_guard_metrics(...)`	inspect or clear guard metrics	metrics snapshot helpers
`tdse_ext_update_runtime_guard_metrics(...)`	update guard metrics	public for completeness, mainly runtime-internal

API	First Job	Notes
`tdse_ext_status_from_runtime(...)` / `tdse_ext_status_from_builder(...)` / `tdse_ext_status_from_circuit_ac(...)`	normalize module-local status codes	status mapping helpers
`tdse_ext_status_message(...)` / `tdse_ext_pack_error_name(...)`	decode unified status and pack errors	readable token helpers

6.0.8 C++ Wrapper Equivalents

C++ API	Runtime Surface	Notes
`tdse::Model::fromMemory(...)`	`tdse_model_create(...)`	request-scoped create diagnostics overload available
`tdse::Model::empty()` / `operator bool()`	wrapper state only	use after move, close, destroy, or release to gate business calls
`tdse::Model::info()`	`tdse_model_info(...)`	returns `tdse_model_info_t`
`tdse::Model::stateInfo()`	`tdse_model_state_info(...)`	returns `tdse_model_state_info_t`
`tdse::Model::lastErrorInfo()`	`tdse_model_last_error_info(...)`	sticky failure snapshot
`tdse::Model::destroy(options)`	`tdse_model_destroy(...)`	canonical bounded destroy
`tdse::Model::release()`	`tdse_model_release(...)`	finalizer cleanup
`tdse::Model::beginStep()` / `stepOp()` / `stepHr()` / `stepIr()` / `commit()` / `stepDr()`	runtime step APIs	`stepOp()` defaults to `Np x Np`; `full=true` selects `Nq x Np`; wrapper enforces `Np` / `Nq` buffer sizes
`tdse::perf::backendCount()` / `backendGet()`	backend registry APIs	C++ registry enumeration
`tdse::perf::setBackend()` / `activeBackend()` / `applyPlan()`	backend routing APIs	runtime-facing perf control
`tdse::perf::setCudaConfig()` / `getCudaConfig()`	CUDA config APIs	per-model CUDA options
`tdse::perf::setComputePrecision()` / `getComputePrecision()`	precision APIs	CPU history precision
`tdse::perf::setLocalThreads()` / `getLocalThreads()`	local thread APIs	local CPU thread override
`tdse::perf::getBuildFeaturesJson(...)`	`tdse_perf_get_build_features_json(...)`	returns JSON through `std::string`
`tdse::ext::setLogCallback()` / `setLogLevel()` / `logLevel()`	ext logging APIs	logging wrapper coverage
`tdse::ext::setDeterministicMode()` / `deterministicMode()`	deterministic-mode APIs	global runtime toggle
`tdse::ext::setRuntimeGuardConfig()` / `runtimeGuardConfig()` / `runtimeGuardMetrics()` / `resetRuntimeGuardMetrics()`	runtime guard APIs	stable-guard wrapper coverage
`tdse::ext::statusMessage()` / `packErrorName()` / `statusFromRuntime(...)` / `statusFromBuilder(...)` / `statusFromCircuitAc(...)`	status mapping APIs	readable status and token helpers

6.0.9 Recommended Usage Map

Situation	Recommended API
create a model	`tdse_model_create(...)`
check runtime version	`tdse_version_string(...)`

Situation	Recommended API
inspect static shape	`tdse_model_info(...)`
inspect live state	`tdse_model_state_info(...)`
inspect last runtime failure	`tdse_model_last_error_info(...)`
compute trial output terms	`tdse_step_begin/op/hr/ir`
advance committed state	`tdse_step_commit(...)`
inspect committed direct response	`tdse_step_dr(...)`
choose backend before stepping	`tdse_backend_set(...)` or `tdse_backend_apply_plan(...)`
inspect available backend capabilities	`tdse_backend_registry_count/get`
query build/runtime feature flags	`tdse_perf_get_build_features_json(...)`
normal host shutdown	`tdse_model_destroy(...)`
terminal cleanup path	`tdse_model_release(...)`

6.0.10 Quick Memory Aid

If you remember only three runtime ideas, remember these:

1. `create` is request-scoped and diagnostic-rich
2. `commit` is the only state-advancing step call
3. perf setters are pre-step only; after the first successful `begin`, configuration is frozen

6.1 Lifecycle and Ownership

Use this section when you need to know who owns a Runtime handle, which API ends that ownership, and which shutdown call fits your host code. This section is about handle lifetime only; the per-step loop itself belongs in Step Execution.

> **Related Chapters** For Builder configuration and pack generation, see Builder and Data Contracts. For the step-loop execution model, see Step Execution. For concurrency rules, see Concurrency and Shutdown.

Use this section when your host integration question is primarily:

- who owns the `tdse_model_t*` handle right now
- which shutdown path fits the host's policy and wait budget
- when local ownership is gone and all further calls must stop

If your question is instead "what exact calls happen every accepted step," jump to Step Execution first.

6.1.1 Why Lifecycle Comes First

The most important Runtime rule is not the math. It is lifecycle. If you understand when a handle is live, closing, destroying, released, or no longer locally owned, the rest of the API becomes much easier to use correctly.

6.1.2 Where This Fits In A Host Integration

For a minimal host integration, lifecycle is usually one of three patterns:

Host pattern	Runtime lifecycle concern
single model owned by one solver thread	ordinary create -> destroy path
worker pool with one model per thread	ownership must stay per-handle, not per process
supervisory or fault-handling shutdown	`close` or timeout-bearing `destroy` policy matters

This chapter only answers the ownership side of those patterns. It does not define the numerical step loop.

6.1.3 The Runtime Lifecycle In One Table

API	Primary Use	Wait Behavior	Returns Status	Recommended For Host Code
`tdse_model_create(...)`	create a model from pack bytes	n/a	yes	yes
`tdse_model_close(...)`	explicit non-blocking close attempt	does not wait for an in-flight same-handle API	yes	situational
`tdse_model_destroy(...)`	explicit bounded destroy	caller-controlled wait policy	yes	yes
`tdse_model_release(...)`	terminal cleanup / finalizer path	may wait until destroy ownership is acquired	yes	no

Short version:

- use `create` to create
- use `destroy` for ordinary host-managed shutdown
- use `close` when you need an immediate status-bearing close attempt
- use `release` only for terminal cleanup paths that are not driving lifecycle policy

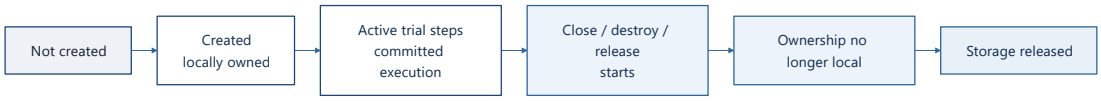


Figure 6.1 High-level lifecycle state model

6.1.4 Create Semantics

The standard create path is:

```
tdse_model_create_diagnostics_t diag = tdse_model_create_diagnostics_init();
tdse_model_t* model = NULL;
tdse_status_t st = tdse_model_create(pack_data, pack_size, &diag, &model);
```

Key rules:

- pass `tdse_model_create_diagnostics_t` so create failures remain request-scoped
- treat `diag.pack_error_code` as part of the normal error path, not as optional trivia
- after successful create, the caller owns the handle locally

When the host needs metadata before deciding whether to create, prefer:

- `tdse_pack_inspect(...)` for a `tdse_model_info_t` snapshot decoded from pack bytes
- `tdse_pack_inspect_ex(...)` for version and payload summary fields in `tdse_pack_summary_t`

These APIs validate pack header and required metadata chunks without creating a `tdse_model_t`.

Owning the handle locally means:

- your thread or wrapper may enter the handle according to the Runtime rules
- your code is responsible for selecting the shutdown path
- another thread has not yet taken over close or destroy

6.1.5 Live-Handle Query APIs

These queries are safe snapshot-style reads on a live handle:

- `tdse_model_info(...)`
- `tdse_model_state_info(...)`
- `tdse_model_last_error_info(...)`

They are intended for metadata, state, and failure diagnosis. If close or destroy has already started on the handle, these queries return `TDSE_ERR_INVALID_STATE`.

6.1.6 Lifecycle States In Practice

The runtime does not publish a first-class enum for every human-readable lifecycle state, but product integrations should reason in these terms:

Practical State	Meaning	Safe Actions
locally owned and idle	handle exists, no active trial step	query info/state, start a step, destroy, close
trial step active	`tdse_step_begin(...)` succeeded and commit has not yet ended the trial	query `op`, `hr`, `ir`, commit
destroy pending locally	caller has entered `destroy` and is waiting or finishing	observe destroy result only
ownership already handed off	another thread already started close or destroy	stop using the handle locally
terminally cleaned up	storage has been released	no further API use is valid

6.1.7 Step-Lifecycle Ownership

Inside a live handle, the runtime lifecycle is:

1. `tdse_step_begin(...)`
2. one or more trial-safe queries:
   - `tdse_step_op(...)`
   - `tdse_step_hr(...)`
   - `tdse_step_ir(...)`
3. `tdse_step_commit(...)`
4. optional `tdse_step_dr(...)`

Two rules matter:

- `tdse_step_commit(...)` is the only state-advancing call
- if a trial solve fails, do not call `commit`; committed state remains unchanged

Lifecycle bugs and step-order bugs usually appear together. If ownership or sequencing is wrong, the first visible symptom is often a bad step call rather than a clearly named lifecycle error.

6.1.8 Close Semantics

`tdse_model_close(...)` is the explicit non-blocking lifecycle endpoint.

Use it when:

- the caller needs a synchronous lifecycle result immediately
- the caller does not want to wait for an in-flight same-handle API

Interpretation:

- `TDSE_OK`: close completed and local ownership ended
- `TDSE_ERR_CONCURRENT_API_USE`: another same-handle API is still in flight
- `TDSE_ERR_INVALID_STATE`: another thread already started close or destroy

`close` is not the recommended production shutdown path. It is the immediate-answer path.

Typical uses:

- supervisory code wants a fast lifecycle answer
- a wrapper wants overlap detection without entering a wait path
- tests intentionally exercise ownership conflicts

Close is best understood as a guardrail API, not as a general shutdown API.

6.1.9 Destroy Semantics

`tdse_model_destroy(...)` is the recommended host-managed shutdown path.

Use it when:

- the caller owns lifecycle policy
- timeout matters
- the caller wants structured result details

Typical usage:

```
tdse_model_destroy_options_t opt = tdse_model_destroy_options_init();
tdse_model_destroy_result_t result = tdse_model_destroy_result_init();
opt.wait_timeout_ms = 250.0;

tdse_status_t st = tdse_model_destroy(model, &opt, &result);
```

Interpretation:

- `TDSE_OK`: destroy completed
- `TDSE_ERR_TIMEOUT`: destroy did not acquire ownership before the budget expired; the handle is still valid

- `TDSE_ERR_INVALID_STATE`: another thread already started close or destroy; local ownership should be treated as gone

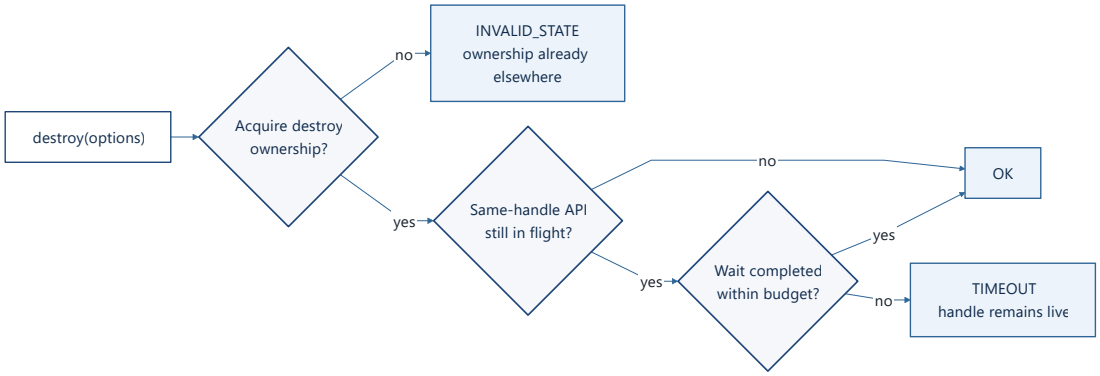


Figure 6.2 Destroy decision model

6.1.9.1 Destroy Result Matrix

Status	Ownership After Return	Handle Still Live?	Typical Host Action
`TDSE_OK`	no longer local	no	clear references and continue
`TDSE_ERR_TIMEOUT`	still local	yes	retry later, escalate, or change policy
`TDSE_ERR_INVALID_STATE`	no longer local	unknown to this thread; assume not locally usable	stop using the handle locally

6.1.10 Release Semantics

`tdse_model_release(...)` exists so C and C++ have an explicit finalizer target.

Use it when:

- a destructor, guard, or finally block must make a best-effort terminal cleanup
- the caller is not trying to implement a bounded lifecycle policy

Do not use it as the primary shutdown path in ordinary host code.

Interpretation:

- `TDSE_OK`: release completed
- `TDSE_ERR_INVALID_STATE`: another thread already started close, destroy, or release

This API is intentionally secondary to `tdse_model_destroy(...)`.

Its purpose is very narrow and very important:

- it gives destructors and unwind paths a clean terminal target
- it avoids turning finalizer code into a policy engine

6.1.10.1 Release Result Matrix

Status	Interpretation
`TDSE_OK`	finalizer cleanup completed
`TDSE_ERR_INVALID_STATE`	another thread already owns terminal cleanup; local ownership is gone

6.1.11 C++ Wrapper Interpretation

The C++ wrapper follows the same lifecycle rules:

- `tdse::Model::fromMemory(...)` wraps `tdse_model_create(...)`
- `tdse::Model::close()` wraps `tdse_model_close(...)`
- `tdse::Model::destroy(options)` wraps `tdse_model_destroy(...)`
- `tdse::Model::release()` wraps `tdse_model_release(...)` but is noexcept and ignores the returned status
- runtime status-bearing failures surface as `tdse::Error`, which preserves both the API name and `tdse_status_t`
- `tdse::Model::empty()` / `operator bool()` are the supported way to check moved-from or already-cleaned-up wrappers
- plain C integrations can normalize runtime failures in two steps:
  1. build a unified code with `tdse_status_code_make(TDSE_STATUS_DOMAIN_RUNTIME, st)`
  2. use `tdse_status_code_classify(...)` and `tdse_status_code_message(...)` for cross-module logging

Important rule:

- `tdse::Model::destroy(...)` requires explicit destroy options so the wait policy is always intentional
- moved-from wrappers may be destroyed, checked with `empty()` / `if (model)`, or assigned again; business APIs still throw on use

That requirement is deliberate. Product code should not drift into implicit infinite-wait shutdown through a convenient wrapper default.

6.1.12 Ownership Handoff Rules

When local ownership is gone, stop using the handle immediately. This is especially important when:

- `tdse_model_close(...)` returns `TDSE_OK`
- `tdse_model_destroy(...)` returns `TDSE_OK`
- `tdse_model_destroy(...)` returns `TDSE_ERR_INVALID_STATE`
- `tdse_model_release(...)` returns `TDSE_OK`
- `tdse_model_release(...)` returns `TDSE_ERR_INVALID_STATE`

The Runtime contract is:

- if another thread has already started close or destroy, ownership is no longer local
- the old pointer must not be treated as a reusable live handle

Ownership handoff is the place where many host bugs become latent use-after-destroy bugs. The safest policy is simple: once handoff happens, clear local references immediately.

6.1.13 C And C++ Mapping Table

Intent	C API	C++ Wrapper
create with request-scoped diagnostics	`tdse_model_create(...)`	`tdse::Model::fromMemory(..., &diag)`
fast close attempt	`tdse_model_close(...)`	`tdse::Model::close()`
canonical bounded shutdown	`tdse_model_destroy(...)`	`tdse::Model::destroy(options)`
terminal finalizer cleanup	`tdse_model_release(...)`	`tdse::Model::release()` and destructor

6.1.14 Anti-Patterns

Avoid these patterns even if they appear to work in a small local test:

1. using `tdse_model_release(...)` as the primary host shutdown API
2. retrying destroy locally after `TDSE_ERR_INVALID_STATE`
3. treating `TDSE_ERR_TIMEOUT` as if the handle were already gone
4. continuing to use a handle after ownership handoff
5. assuming `close` is just a faster `destroy`

6.1.15 A Good Mental Checklist

Before every lifecycle decision, ask:

1. Do I still own this handle locally?
2. Do I need a status-bearing shutdown answer?
3. Do I need a bounded wait budget?
4. Am I in business logic or finalizer cleanup?

If the answer is “business logic and I need a bounded answer,” the API is almost always `tdse_model_destroy(...)`.

6.1.16 Worked Lifecycle Scenarios

6.1.16.1 Scenario A. Clean Business Shutdown

1. caller owns a live handle
2. no same-handle API is in flight
3. caller invokes `tdse_model_destroy(...)`
4. runtime returns `TDSE_OK`
5. caller drops all references

This is the ideal host-managed path.

6.1.16.2 Scenario B. Destroy Races With A Worker Step

1. worker thread is still inside a same-handle step call
2. supervisory code calls `tdse_model_destroy(...)`
3. destroy waits within its configured budget
4. runtime returns either `TDSE_OK` or `TDSE_ERR_TIMEOUT`

This is why destroy carries explicit wait policy and result details.

6.1.16.3 Scenario C. Another Thread Already Took Ownership

1. thread A starts close or destroy
2. thread B also attempts close, destroy, or release
3. thread B receives `TDSE_ERR_INVALID_STATE`
4. thread B must stop using the handle locally

This is not a local retry condition. It is ownership handoff.

6.1.17 Recommended Patterns

Recommended:

- ordinary host shutdown: `tdse_model_destroy(...)`
- timeout-aware supervision: `tdse_model_destroy(...)` plus an explicit policy for `TDSE_ERR_TIMEOUT`
- destructor or guard cleanup: `tdse_model_release(...)`
- immediate non-waiting close attempt: `tdse_model_close(...)`

Avoid:

- using `tdse_model_release(...)` as the first-choice business-logic shutdown API
- relying on infinite-wait destroy as the default production pattern without documenting why
- continuing to use a handle after destroy, release, or ownership handoff

6.2 Step Execution Model

Use this section when you are wiring TDSE Runtime into a simulator step loop. It explains what each step call means, how trial state differs from committed state, how retries behave when a trial is rejected, and what to record when the loop does not behave as expected.

> **Related Chapters** For lifecycle and shutdown semantics, see Runtime Lifecycle. For concurrency rules, see Concurrency and Shutdown.

If the lifecycle section answers "who owns the handle," this section answers "what exact state is the handle in between `begin`, query calls, `commit`, and `dr`?"

For most customer integrations, this section is the contract source for one of these three host patterns:

Host pattern	What to focus on here
fixed-step solver loop	canonical step order and prime-step pattern
adaptive / trial-rejecting solver loop	rejected-trial semantics and state snapshots
real-time / HIL loop	committed-state discipline, `dr` timing, and one-handle-per-thread rules

6.2.1 The Three Invariants

If you remember only three step-loop rules, remember these:

1. `tdse_step_begin(...)` creates a trial context at one exact `(t, dt)`.
2. These trial query calls are query-only within that trial context:

- `tdse_step_op(...)`
- `tdse_step_hr(...)`
- `tdse_step_ir(...)`

3. `tdse_step_commit(...)` is the only call that advances committed history.

Everything else in this chapter follows from those three rules.

6.2.2 Canonical Step Order

Per ordinary step, the runtime flow is:

1. `tdse_step_begin(model, t, dt)`
2. `tdse_step_op(model, op_out)` as needed
3. `tdse_step_hr(model, hr_out)`
4. `tdse_step_ir(model, ir_out)` when IR exists and is still in range
5. host-side trial solve or trial update
6. `tdse_step_commit(model, primary_accepted)` if the trial is accepted
7. optional `tdse_step_dr(model, dr_out)` after commit

`tdse_step_op(...)` is an instantaneous operator query. Hosts may:

- fetch it once and cache it under LTI assumptions
- fetch it per step when host policy wants explicit freshness

Host/TDSE ownership split in this loop:

- the host owns `t`, `dt`, trial acceptance, solver state, and the accepted `primary` vector
- TDSE owns trial-local query state, committed history, and the resulting `op` / `hr` / `ir` / `dr` surfaces

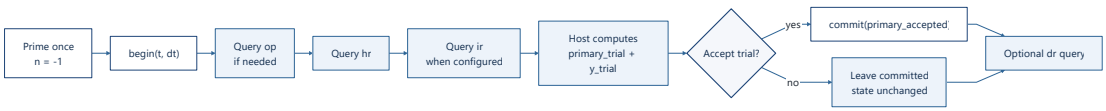


Figure 6.3 Canonical per-step timeline

6.2.3 Prime-Step Pattern

The standard integration pattern primes at `n = -1`:

```
tdse_step_begin(m, t0 - dt, dt);
tdse_step_op(m, &op_square);
tdse_step_commit(m, primary_minus1);
```

Why this matters:

- it establishes committed history before the first ordinary simulation step
- it makes later `hr` evaluation align with the expected discrete-time history model
- it gives `dr` a valid first committed state immediately after prime

If you skip the prime step without redesigning initialization assumptions, the first visible issue often looks numerical even though the real bug is committed-history alignment.

Prime-step design review question:

- what exact accepted primary vector should the runtime treat as the prehistory state?

If that answer is still implicit, the integration is not really complete.

6.2.4 Mathematical Contract

The trial-side relation is:

- `y_trial = op * primary_trial + hr + ir`

Where:

- `op` is the instantaneous operator
- `hr` is the delayed-history contribution
- `ir` is the independent-response contribution

The committed direct-response relation is:

- `dr[n] = op * primary_accepted[n]`

Figure 6.4 Trial versus committed views

6.2.5 Legality Matrix

This is the fastest table to use during code review or triage:

API	Requires Live Handle	Requires Active Trial Step	Requires Prior Commit	Advances State
`tdse_step_begin(...)`	yes	no, unless exact re-entry	no	creates or re-enters trial context
`tdse_step_op(...)`	yes	yes	no	no
`tdse_step_hr(...)`	yes	yes	no	no
`tdse_step_ir(...)`	yes	yes	no	no
`tdse_step_commit(...)`	yes	yes	no	yes
`tdse_step_dr(...)`	yes	no	yes	no

6.2.6 What `tdse_model_state_info(...)` Means During A Step Loop

The most useful runtime snapshot during integration is `tdse_model_state_info(...)`. Its high-value fields are:

Field	Operational Meaning
`step_active`	nonzero while a trial step is currently active
`has_committed_step`	nonzero after at least one successful `commit`
`committed_steps`	number of accepted commits
`committed_t` / `committed_dt`	time coordinates of the latest committed step
`active_t` / `active_dt`	active trial-step coordinates, or zero when no trial is active
`sim_time`	accumulated committed simulation time
`dr_last_valid`	whether `tdse_step_dr(...)` is currently queryable

This snapshot reports dynamic execution state only. It is the quickest way to distinguish these three cases:

- no trial has started yet
- a trial is active but not committed
- a committed step exists and `dr` is valid

6.2.7 State Snapshot Transition Table

The manual previously named these fields, but host teams usually need the exact transition pattern. Use this table as the expected runtime fingerprint.

Phase	`step_active`	`has_committed_step`	`committed_steps`	`active_t` / `active_dt`	`committed_t` / `committed_dt`	`dr_last_valid`
newly created handle	0	0	0	0 / 0	0 / 0	0
after `begin(t, dt)`	1	unchanged	unchanged	`t` / `dt`	unchanged	unchanged
after trial queries only	1	unchanged	unchanged	`t` / `dt`	unchanged	unchanged
after accepted `commit(...)`	0	1	increments by 1	0 / 0	becomes the just-committed `t` / `dt`	1
after rejected trial with no commit	1 until caller leaves the active trial lifecycle	unchanged	unchanged	still the active trial coordinates	unchanged	unchanged
after `dr(...)`	0	1	unchanged	0 / 0	unchanged	1

Two practical readings matter:

- if `step_active=1` and `committed_steps` is not moving, the runtime is still in a trial context
- if `dr_last_valid=1`, at least one accepted commit already exists and no new commit is required just to read the latest committed direct response

6.2.8 `begin` Semantics And Re-Entry

`tdse_step_begin(...)` does three user-visible things:

1. validates that the handle and `(t, dt)` are legal
2. freezes execution-affecting runtime controls on the first successful begin
3. records the active trial coordinates and marks `step_active=1`

The important re-entry rule is strict:

- re-entering `tdse_step_begin(...)` while a trial is already active is valid only when `t` and `dt` match the already-active trial step within runtime tolerance
- mismatched re-entry fails with `TDSE_ERR_INVALID_STATE`

This means `begin` is not a generic "start over" button. It is an idempotent re-entry only for the same trial coordinates.

6.2.8.1 What Re-Entry Is For

Exact re-entry is useful when:

- a host wrapper retries a query path but is still on the same trial step
- instrumentation or layered call sites may invoke the same "ensure step active" helper twice

It is not valid for:

- changing `t` mid-trial
- changing `dt` mid-trial
- silently converting a rejected trial into a different step without resolving the active state

6.2.9 Rejected Trials And Retry Semantics

Rejected trials are normal in real host integrations. The runtime contract is intentionally simple:

1. start the trial with `begin`
2. query `op`, `hr`, and optional `ir`
3. host decides the trial solve is not acceptable
4. do not call `commit`
5. treat committed history as unchanged

The one rule that matters most is this:

- no `commit` means no committed-state movement

Operational consequences:

- `committed_steps` does not increase
- `committed_t` and `committed_dt` do not change
- `sim_time` does not advance
- `dr_last_valid` stays tied to the previous committed step

6.2.9.1 How To Retry Cleanly

If the host rejects the trial and wants to try again:

- keep the retry anchored to the same step coordinates if the same trial is being reconsidered
- do not pretend a new committed step exists

- do not read the lack of state movement as a runtime failure; it is the intended rejection model

If the host instead wants a different `(t, dt)`, it must follow the documented lifecycle rather than mismatched `begin` re-entry.

6.2.10 Worked Snapshot Trace

This is the most useful support trace to memorize.

6.2.10.1 Before Prime

Expected snapshot:

- `step_active=0`
- `has_committed_step=0`
- `committed_steps=0`
- `dr_last_valid=0`

Interpretation:

- no committed history exists yet
- `dr(...)` is invalid at this point

6.2.10.2 After Prime `begin(t0 - dt, dt)`

Expected snapshot:

- `step_active=1`
- `active_t=t0 - dt`
- `active_dt=dt`
- `has_committed_step=0`

Interpretation:

- the runtime now has an active trial context
- committed history still does not exist until `commit`

6.2.10.3 After Prime `commit(primary_minus1)`

Expected snapshot:

- `step_active=0`
- `has_committed_step=1`
- `committed_steps=1`
- `committed_t=t0 - dt`
- `committed_dt=dt`
- `dr_last_valid=1`

Interpretation:

- history now exists
- `dr(...)` is legal
- the first ordinary step can use a properly anchored committed past

6.2.10.4 During Ordinary Step n

After `begin(t_n, dt)` and before `commit(...)`:

- `step_active=1`
- `active_t=t_n`
- `active_dt=dt`
- `committed_steps` still equals the previous accepted count
- `committed_t` still points to the previous accepted step

This is the point where support should ask:

- are we still in a legitimate trial?
- is the host expecting committed values too early?

6.2.10.5 After Accepted Commit At Step n

Expected snapshot:

- `step_active=0`
- `committed_steps` increments by one
- `committed_t=t_n`
- `committed_dt=dt`
- `sim_time` increases by `dt`
- `dr_last_valid=1`

Interpretation:

- committed history has advanced
- future `hr` calls will now see the newly accepted state in their delayed-history basis

6.2.10.6 After Rejected Trial At Step n

If the host decides not to commit:

- `committed_steps` remains unchanged
- `committed_t` remains the previous committed time
- `sim_time` remains unchanged
- `dr_last_valid` still refers to the previous committed step

This is the exact fingerprint of "trial rejected, committed history preserved."

6.2.11 Worked Host Pattern

The intended control split is:

1. Runtime supplies `op`, `hr`, and optional `ir`
2. host code computes or solves the trial equation
3. host decides whether the trial primary vector is accepted
4. Runtime advances history only when the host commits

This is why Runtime remains auditable inside larger simulator loops.

6.2.11.1 Minimal Integration Recipes

Use the smallest recipe that matches your host:

Host type	Minimal recipe
fixed-step simulator	prime once, then `begin -> op/hr/ir -> host solve -> commit -> dr`
adaptive simulator	same loop, but host may reject a trial and skip `commit`
RT/HIL loop	fixed accepted-step policy, one handle per execution thread, `dr` only after commit

For code review, the key question is not “did the host call the APIs” but “which side owns acceptance, timing, and history movement?”

6.2.11.2 C Example

```c
tdse_step_begin(model, t, dt);
tdse_step_op(model, &op);
tdse_step_hr(model, hr);

tdse_status_t ir_st = tdse_step_ir(model, ir);
if (ir_st == TDSE_ERR_IR_STEP_OUT_OF_RANGE) {
  /* Host policy decides whether this is a hard stop or planned IR horizon exit. */
}

solve_trial(op, hr, ir, primary_trial, y_trial);

if (trial_is_accepted(primary_trial, y_trial)) {
  tdse_step_commit(model, primary_trial);
  tdse_step_dr(model, dr);
}
```

The key handbook reading is:

- the host owns acceptance
- the runtime owns committed-history mutation

6.2.11.3 Fixed-Step Host Skeleton

This is the simplest integration that should be proven before any adaptive or real-time embellishment:

```c
prime_once(model, t0, dt, primary_minus1);

for (size_t n = 0; n < nsteps; ++n) {
  tdse_step_begin(model, t0 + n * dt, dt);
  tdse_step_op(model, &op);
  tdse_step_hr(model, hr);
  tdse_step_ir(model, ir);   /* when IR exists */
  host_solve_trial(op, hr, ir, primary_trial, y_trial);
```

```
    tdse_step_commit(model, primary_trial);
    tdse_step_dr(model, dr);
}
```

If this path is not already stable, stop here before adding retries, adaptive `dt`, or multi-threading.

6.2.11.4 Adaptive-Step Ownership Rule

For adaptive hosts, keep one rule explicit in design notes:

- the host may retry or reject a trial
- TDSE must not see history advancement unless the host actually commits

That single rule prevents most accidental state-drift bugs.

6.2.12 `IR` Horizon Behavior Inside The Loop

`tdse_step_ir(...)` is query-only, but it is still a contract boundary. If the current step time lies beyond configured `IR` support, the API returns `TDSE_ERR_IR_STEP_OUT_OF_RANGE`.

Read that status as:

- the runtime is telling you the packaged `IR` sequence does not extend to this step
- the runtime is not silently extrapolating `IR`

What to record:

- archive `committed_steps`, `committed_t`, `dt`, and model `ir_nsteps`
- verify whether the host advanced beyond the packaged horizon by design or by mistake

6.2.13 Rectangular Versus Square Operator View

`tdse_step_op(...)` supports:

- square view: `np x np`
- full rectangular view: `nq x np`

Recommended practice:

- choose one operator-view policy per integration
- document that policy in the host design
- do not switch views ad hoc across call sites without a clear reason

Using the wrong view usually appears later as a shape or interpretation bug, not at the moment the host design drifted.

6.2.14 What to Capture for Step Incidents

When a step-loop issue shows up, collect these details in this order:

1. failing API name and returned status
2. `tdse_model_state_info(...)`

3. `tdse_model_last_error_info(...)`
4. exact `t`, `dt`, and host step index
5. whether the host intended to accept or reject the trial
6. whether prime was performed
7. whether the host expected square or rectangular `op`

This sequence usually resolves three common ambiguities immediately:

- lifecycle misuse versus numerical rejection
- missing prime versus wrong-history interpretation
- `IR` horizon exit versus unrelated step failure

6.2.15 Common Step-Loop Mistakes

Mistake	Why It Is Wrong	Correct Pattern
calling `dr` during an active trial step	`dr` is post-commit only	query `dr` after commit, with no active trial step
treating `hr` as state-advancing	`hr` is query-only	call `commit` to advance history
skipping prime without redesigning history assumptions	later history alignment shifts	prime at `n = -1` or document a different initialization contract
re-entering `begin` with different `t` or `dt`	active trial context must remain one exact step	finish the active trial lifecycle before changing coordinates
assuming rejected trial implies hidden state rollback logic	no commit means no advancement happened	read snapshots and keep retry logic explicit
mixing square and rectangular operator views ad hoc	host matrix assumptions drift	choose and document one operator-view policy

6.2.16 Anti-Patterns

Avoid these integration patterns even if they seem harmless in local testing:

1. using `state_info` only after failures instead of also learning the normal expected snapshot
2. inferring committed advancement from successful `hr` or `ir` queries
3. treating `dr_last_valid=1` as proof that the current trial was committed
4. retrying a rejected trial by changing `(t, dt)` under an already-active trial step
5. explaining a history bug as “numerical noise” before verifying prime and commit sequencing

CHAPTER 7

Builder and Data Contracts

Use this chapter when you already have validated `H` data, optional `IR` data, or source data that Builder can convert, and you need to produce a pack that Runtime can load. It explains the Builder setup flow, the shape rules that matter at the handoff, and the checks worth running before you blame Runtime for a bad pack. This chapter stops at the pack-to-runtime handoff; the full runtime lifecycle and step loop live in their own chapters.

If your starting point is already FRF data, impulse-response data, or a Builder-ready workflow output, start here. If your starting point is a circuit netlist, RAW case, or Touchstone-driven circuit flow, start with Adapter Circuit instead.

> **Related Chapters** For Runtime lifecycle semantics after pack creation, see Runtime Lifecycle. For step execution inside the simulation loop, see Step Execution.

Use this chapter when your integration starts from one of these Builder-side artifacts:

Starting artifact	Builder role
validated `H` tensor	attach `H`, write pack, hand off to Runtime
validated `H` plus optional `IR`	attach both, preserve shape and horizon contracts
matrix or circuit workflow output	accept Builder-ready handoff from Adapter or workflow API

7.1 Prerequisites

- Built SDK artifacts (Runtime `tdse`, Builder `tdse_builder`, headers).
- A generated pack input (`H` required, `IR` optional).
- Build/test workspace ready:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build --config Release
```

7.2 Quick Start

Both paths below end with a written pack and a small handoff check. Use the minimal path for prototyping; switch to the production path before release.

7.2.1 Minimal path

```
tdse_builder_t* b = NULL;
tdse_builder_options_t cfg = tdse_builder_options_init();
cfg.dt = dt;
cfg.nh = nh;
cfg.np = np;
cfg.nq = nq;
tdse_builder_create(&b);
tdse_builder_configure_ex(b, &cfg);        /* np/nq/nh/dt */
tdse_builder_apply_h(b, &h_desc);          /* required */
tdse_builder_apply_ir(b, &ir_desc);        /* optional */
tdse_builder_write_pack(b, "model.pack");
tdse_builder_destroy(b);
```

Then hand the pack to Runtime Lifecycle for create/destroy behavior and to Step Execution for the simulation loop.

Minimal host-side interpretation:

- Builder owns pack construction and pack metadata
- Runtime owns create, step, and destroy after pack bytes are handed off
- the host owns artifact selection, pack storage, and when to start Runtime

7.2.2 Production path

For production integration, add these practices on top of the minimal flow:

1. Validate pack bytes before runtime create (`tdse_pack_validate`).
2. Inspect the pack summary before create so shape and metadata mismatches are visible early.
3. Archive a Builder snapshot (`tdse_builder_info(...)`) next to the pack or release artifact.
4. Run one deterministic runtime smoke check: create the model, read `tdse_model_info(...)`, and verify a known-good step path.
5. Archive `tdse_model_create_diagnostics_t` on non-OK create paths.
6. Run threading stress and contract tests before promoting binaries.

Runtime handoff note:

- use Runtime Lifecycle for `create`, `close`, `destroy`, and `release`
- use Step Execution for prime, trial, commit, and post-commit queries
- use Runtime API Summary when you need a fast symbol map instead of a narrative walk-through

7.2.3 Smallest Supported Builder → Runtime Handoff

If your team needs the shortest credible integration contract, treat the handoff as these four checks:

1. Builder writes one pack successfully
2. `tdse_pack_validate` accepts that pack

3. Runtime create succeeds and `tdse_model_info(...)` matches expected `np`, `nq`, `nh`, and `dt`
4. one prime-and-step smoke path completes without status errors

7.2.4 Validation Checklist

☐ `tdse_pack_validate` passes on release pack.
☐ Minimal step loop returns `TDSE_OK` end-to-end.
☐ One failure-injection case per major status is covered.
☐ Threading stress case is green on target build.
☐ Runtime outputs are stable across repeated runs with same input.
☐ Linux deployment claims match the current RuntimeCore Linux support scope when shipping on Linux.

7.3 Parameter Cookbook

Parameter	Meaning	Recommended Baseline	Notes
`np`	primary input count	system-dependent	Must match host primary vector width.
`nq`	operator row count	`>= np`	Use full rectangular view when `nq > np`.
`nh`	history taps	start with calibrated value	Higher `nh` raises history cost.
`dt`	step interval	fixed per model	Keep builder/runtime `dt` consistent.
`IR` length	independent sequence horizon	cover whole simulation window	Out-of-range returns `TDSE_ERR_IR_STEP_OUT_OF_RANGE`.
runtime handles	parallel threads	one handle per worker	Never step same handle concurrently.

7.4 Data Contracts

TDSE Core is strict about dimensions, layout, and ownership. Most early integration failures are shape mistakes rather than numerical mistakes.

7.4.1 Primary Dimensions

Four dimensions define most Builder-to-Runtime compatibility:

Symbol	Meaning	Practical Rule
`np`	primary input count	must match host primary vector width
`nq`	output-equation count	must satisfy `nq >= np`
`nh`	history tap length	must match the supplied `H` tensor depth
`dt`	model step interval	must be consistent across Builder and Runtime expectations

7.4.2 API Families

At the Builder boundary, the main API families are:

7.4.2.1 Builder lifecycle

- `tdse_builder_create`
- `tdse_builder_configure_ex`
- `tdse_builder_apply_h`
- `tdse_builder_apply_ir`
- `tdse_builder_write_pack`
- `tdse_builder_destroy`

7.4.2.2 Pack validation and inspection

- `tdse_pack_validate`
- `tdse_pack_inspect`
- `tdse_pack_inspect_ex`
- `tdse_pack_error_token`

7.4.2.3 Runtime handoff surface

- `tdse_model_create`
- `tdse_model_info`

For full Runtime lifecycle and step APIs, switch to Runtime Lifecycle, Step Execution, or Runtime API Summary.

7.4.3 Ownership Model

- Builder descriptors such as `tdse_h_desc_t` and `tdse_ir_desc_t` are borrowed views
- Pack-validation and pack-inspection outputs are caller-owned result structs
- No Core API transfers ownership of caller-owned `H`, `IR`, or step-output buffers

7.4.4 Compatibility Equation

Builder-side and Runtime-side assumptions are compatible only when all of these are true:

1. Builder `np`, `nq`, `nh`, and `dt` match the intended runtime model
2. `H` is laid out exactly as declared by its descriptor
3. optional `IR` is laid out exactly as declared by its descriptor
4. runtime host buffers are sized from runtime-reported dimensions rather than from memory or guesswork

7.4.5 `H` Tensor Contract

`H` uses tap-major row-major storage: shape `[nh][nq][np]`, linear index `tap * nq * np + row * np + col`, layout enum `TDSE_H_LAYOUT_TAP_MAJOR_ROW_MAJOR`.

- `H[0]` is the instantaneous operator returned by `tdse_step_op(...)`
- `H[1..nh-1]` contribute to the delayed-history term returned by `tdse_step_hr(...)`

Explicit-`tau` contract:

- `h_desc.tau == NULL` means delayed taps live on the implicit uniform axis `tau[k] = k * dt`
- `h_desc.tau != NULL` means Runtime evaluates history on the supplied explicit time axis

- when `tau` is explicit, `H[1..nh-1]` must already be weighted for that axis
- if your source taps came from a uniform grid, convert them first with `tdse_h_uniform_to_tau_1d(...)` or `tdse_h_uniform_to_piecewise_tau_1d(...)`

7.4.6 IR Sequence Contract

`IR` is step-major: shape `[nsteps][nq]`, linear index `step * nq + row`, layout enum `TDSE_IR_LAYOUT_STEP_MAJOR`. Runtime queries `IR` by current step time. If the query falls outside the configured support window, `tdse_step_ir(...)` returns `TDSE_ERR_IR_STEP_OUT_OF_RANGE`.

7.4.7 Dense Operator Contract

`tdse_step_op(...)` writes into a `tdse_dense_block_t`. Supported views: square (`np x np`) or full (`nq x np`). `cols` must always equal `np`; `rows` must equal either `np` or `nq`.

7.4.8 Shape Worked Example

Assume `np = 3`, `nq = 4`, `nh = 8`, `ir_nsteps = 100`. Then:

Query	Required Length / Shape
primary vector passed to `commit`	length `np = 3`
`hr_out` buffer	length `nq = 4`
`ir_out` buffer	length `nq = 4`
square `op` view	`3 x 3`
full `op` view	`4 x 3`
committed `dr_out` buffer	length `nq = 4`

The common bug: sizing `hr`, `ir`, or `dr` buffers to `np` because the host thinks in terms of ports rather than equations. These buffers are `nq`-sized outputs.

7.4.9 Step-Term Contract

> For the mathematical definitions, see Theory and Concepts.

Term	Meaning	Side Effect
`op`	instantaneous operator	none
`hr`	delayed-history contribution	none
`ir`	independent-response contribution	none
`dr`	committed-step direct response	query-only, post-commit

Read these equations literally: `y_trial = op * primary_trial + hr + ir`; `dr[n] = op * primary_accepted[n]`. `dr` is the direct-response slice on the committed step, not "the whole committed output."

7.4.10 Runtime Handoff Contract

Builder is done once the pack is internally consistent, validated, and clearly labeled. The runtime handoff should answer three questions before any simulation work starts:

1. does the pack validate cleanly
2. do inspected dimensions and metadata match what the host expects
3. does Runtime create the model without reporting pack or compatibility errors

After that point, move to Runtime Lifecycle and Step Execution instead of continuing to reason about Runtime behavior from the Builder chapter.

7.4.11 Host-Side Assertions

In production, assert these once near the integration boundary:

1. `model_info.np` matches host primary-vector width
2. `model_info.nq` matches host equation/output width
3. `model_info.nh` matches intended history depth
4. `model_info.dt` matches host time-step contract

7.4.12 Common Shape Mistakes

- host primary vector width does not match `np`
- `hr`/`ir` buffers sized to `np` instead of `nq`
- square operator storage used when host required `nq x np`
- Builder `dt` and simulation `dt` assumed to match without verification
- inferring `np`/`nq` from old code paths instead of `tdse_model_info(...)`
- treating `IR` as a runtime side channel instead of packaged model content

7.5 Power Systems Guide

7.5.1 Typical Parameters by Scenario

Scenario	dt (s)	nh	nfft	np	nq	Notes
Transmission line (100 km, 500 kV)	1e-6 to 10e-6	1000-5000	2 × nh	2 (1 port) or 4 (2-port)	np or np+1	Large nh for propagation delay
Transformer (50 MVA)	1e-6 to 50e-6	200-1000	2 × nh	2-6	np	Moderate nh for winding capacitance
Distribution cable (underground)	1e-6 to 10e-6	500-2000	2 × nh	2-4	np	nh depends on cable length
EMI/EMC (wideband)	10e-9 to 100e-9	1000-4000	4 × nh	1-10	np	Very fine dt for high-frequency content
Power electronics (switching ~100 kHz)	10e-9 to 100e-9	500-2000	2 × nh	1-4	np	Fine dt for switching transients

7.5.2 Choosing nh

The history depth `nh` must be large enough that `H[nh-1]` has decayed to near zero:

1. Build the pack with an initial `nh` estimate
2. Inspect `H[k]` for the last few taps (k near nh-1)
3. If `|H[nh-1]|` is not negligible compared to `|H[0]|`, increase `nh`
4. Typical transmission lines: `nh ≈ round(propagation_delay / dt) + margin`

For a 500 kV, 100 km line with propagation speed ~2.8×10^8 m/s and dt = 50 µs: propagation delay ≈ 357 µs → nh ≈ 9. For dt = 1 µs: nh ≈ 367.

7.5.3 When to Use nq > np

Set `nq > np` when the model needs measurement equations beyond the port count: multi-port models with internal measurements, mixed Y/Z representations, or host simulators needing both terminal currents and internal state outputs. For standard Y+ISC or Z+VOC, `nq = np` is typical.

7.6 Builder Flow

Builder is the handoff layer between validated source artifacts and Runtime. A clear Builder boundary makes later debugging much easier: you can tell whether a problem comes from the source data, Builder configuration, pack generation, or Runtime execution.

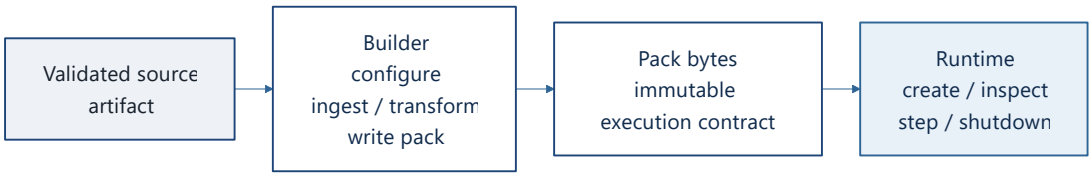


Figure 7.1 Builder-to-Runtime responsibility split

7.6.1 Builder Responsibility

Builder owns the configured dimensions and `dt`, attachment of required `H` and optional `IR`, optional conversion from frequency-domain data into time-domain `H`, optional Builder-side shaping such as IRC, and the final pack metadata and pack write.

Builder does not own upstream artifact validity beyond descriptor and shape checks, host-specific port order or matrix-family interpretation, Runtime step execution, or Runtime shutdown and concurrency policy.

The main rule is simple: Builder should emit the final artifact, and Runtime should execute it as-is. Runtime is not expected to reconstruct or repair pack content later.

7.6.2 Builder State Machine

Important rules: `tdse_builder_configure_ex(...)` is the recommended configure entry-point, re-configuring replaces dimensions and clears previously attached `H` and `IR`, `tdse_`

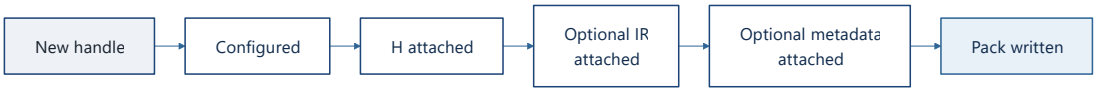


Figure 7.2 Builder state machine

`builder_info(...)` is the best snapshot of the current Builder state, and a successful pack write does not transfer handle ownership.

7.6.3 Builder Contract Table

Contract Item	Set By	Why It Matters Downstream
`dt`	`tdse_builder_configure_ex(...)`	Runtime timing and `IR` step addressing
`np`	`tdse_builder_configure_ex(...)`	host primary vector width must match
`nq`	`tdse_builder_configure_ex(...)`	`hr`, `ir`, `dr`, and full `op` row count
`nh`	`tdse_builder_configure_ex(...)`	delayed-history horizon and `H` tensor depth
`H` layout	`tdse_h_desc_t`	wrong layout produces plausible but incorrect packs
explicit `tau` axis	`tdse_h_desc_t`	malformed nonuniform timing rejected before pack write
`IR` layout and horizon	`tdse_ir_desc_t`	runtime `tdse_step_ir(...)` legality
pack form/domain metadata	`tdse_builder_set_pack_meta(...)`	support and consuming hosts interpret the pack

7.6.4 Direct `H` Ingestion

Use when the upstream artifact is already a validated time-domain kernel. Sequence: configure → populate `tdse_h_desc_t` → `tdse_builder_apply_h(...)` → snapshot with `tdse_builder_info(...)` → write pack. Post-apply: verify `info.configured`, `info.has_h`, and dimension/`tau` expectations match.

7.6.5 Spectrum-to-`H` Conversion

Use `tdse_builder_h_from_spectrum(...)` when the upstream artifact is frequency-domain data.

The sequence: define positive-frequency grid → map source matrix into `tdse_builder_cplx_mat_view_t` → choose correction method → run conversion → attach → snapshot before write.

Correction methods:

- `TDSE_BUILDER_CORRECTION_NONE`
- `_RECONSTRUCT_FROM_REAL`
- `_RECONSTRUCT_FROM_IMAG`
- `_RECONSTRUCT_FROM_MAG`
- `_RECONSTRUCT_FROM_PHASE`

These options change how Builder computes `H`; Runtime only sees the finished pack. When this path looks wrong, ask two separate questions: did Builder produce the intended `H`, and did Runtime execute that pack correctly?

7.6.5.1 Grid Planning: dt, nh, and nfft

These three parameters must satisfy `nfft >= 2 * nh` (FFT covers full impulse response without aliasing), `dt` must capture the highest source frequency, and `nh * dt` must capture full decay. Use `tdse_builder_compute_consistent_grid(&grid)` to compute consistent values from hints.

Scenario	dt	nh	nfft	Rationale
Power electronics (~100 kHz)	10-100 ns	500-2000	2 × nh	Fine dt for switching edges
Transmission line (long delay)	0.1-1 ns	2000-10000	2 × nh	Large nh for delay + reflections
Structural dynamics (< 1 kHz)	1-10 us	200-1000	2 × nh	Coarse dt sufficient
EMI/EMC (wideband)	0.01-0.1 ns	1000-4000	4 × nh	Large nfft for spectral resolution

Common mistakes: `nh` too small → truncated impulse response; `dt` too large → aliasing; `nfft < 2*nh` → wrap-around corruption.

7.6.5.2 FRF Data Layout

The source frequency-domain data must be organized as a row-major complex matrix: `H_frf[freq_idx * np * nq + row * np + col]`. The grid must be monotonically increasing, include near-DC, and use approximately uniform spacing. The Builder expects Y or Z parameters (not S-parameters). Multi-port ordering must be consistent across all frequency points.

7.6.6 Builder IRC (Impulse Response Compression)

IRC reduces effective history depth by compressing the tail of the impulse-response tensor. Two modes: V1 (single decay rate, most common) and V2 (per-tap adaptive decay, higher fidelity). Both retain the first `prefix_len` taps exactly and fit an exponential envelope to the tail.

```
tdse_builder_irc_options_t irc_opt = tdse_builder_irc_options_init();
irc_opt.mode = TDSE_BUILDER_IRC_MODE_V1;
irc_opt.prefix_len = 64;
irc_opt.tail_tolerance = 1e-8;
tdse_builder_apply_irc(b, &irc_opt);
```

Keep the IRC parameters next to the pack so later comparisons are reproducible. Use the Profiler IRC scan to explore compression quality before production use.

7.6.7 Optional `IR`

`IR` is optional, but once attached it becomes part of the pack Runtime will load. Keep Builder `dt` and `IR` `dt` aligned, keep `IR` dimensions aligned with `np` and `nq`, make sure the `IR` support length covers the simulation horizon, and set pack form metadata intentionally (`FLOW_FROM_EFFORT` = Y+ISC, `EFFORT_FROM_FLOW` = Z+VOC, `UNKNOWN` only for packs without `IR`).

7.6.8 Builder Inspection

`tdse_builder_info(...)` is the fastest way to confirm what Builder currently has attached. Use it after configure and after every attach or clear. Fields worth logging are `configured`, `dt/nh/np/nq`, `has_h/has_h_tau/h_layout`, `has_ir/ir_nsteps/ir_dt/ir_layout`, and `pack_form/pack_domain`. Once the pack is written, `tdse_model_info(...)` becomes the matching Runtime-side confirmation point.

7.6.9 Write Gate Checklist

Before `tdse_builder_write_pack(...)`, confirm that Builder is configured, `H` is attached, dimensions match, optional `IR` is either absent or correctly attached, pack metadata is explicit, and the output path is stable. After the write, validate the pack, inspect it, then hand off to Runtime for create and loop verification.

7.7 Worked Paths

7.7.1 Path A: Direct-`H` Pack

Highest-confidence path: configure once → attach `H` → inspect → write → validate → create Runtime model. Fewest transformations; easiest to isolate failures.

7.7.2 Path B: Frequency-Domain Source To Pack

Archive the frequency grid and matrix family → run `tdse_builder_h_from_spectrum(...)` → attach → inspect → write → validate. Needs more release discipline because a successful write does not prove the conversion policy was correct.

7.7.3 Path C: `H` + `IR` With Explicit Pack Meaning

Configure → attach `H` → attach `IR` → `tdse_builder_set_pack_meta(...)` intentionally → inspect → write → validate → create. Most likely to confuse downstream users if metadata is omitted.

7.8 Triage

7.8.1 Failure Modes

Failure	Symptom	Root Cause	Fix
`TDSE_ERR_IR_STEP_OUT_OF_RANGE`	`tdse_step_ir` fails mid-run	Simulation time exceeds IR support window	Extend IR sequence or clamp simulation horizon
`TDSE_ERR_CONCURRENT_API_USE`	sporadic non-OK in multi-thread run	same handle stepped by multiple threads	Serialize per handle; one handle per thread
`TDSE_ERR_INVALID_ARG`	immediate failure on create/step	null/shape mismatch in inputs	Validate pointers, dimensions, struct sizes
pack validation failure	create rejects model bytes	corrupted or incompatible pack	regenerate pack, inspect `tdse_model_create_diagnostics_t`

7.8.2 Builder Failure Classes Worth Catching Early

Failure Class	Typical Root Cause	Best Stage	Support Clue
dimension mismatch	`np`/`nq`/`nh` disagreement	configure/apply	Builder snapshot vs. source planning sheet
malformed descriptor	null pointer, bad struct size, invalid layout	apply	attach call fails immediately
tau-axis issue	invalid explicit nonuniform timing	apply `H`	`has_h_tau` vs. source timing mismatch
wrong matrix family	host supplied wrong physical meaning	preflight	pack validates but numerical behavior is wrong
IR horizon issue	sequence shorter than intended run	preflight/attach	Runtime fails at `tdse_step_ir(...)`
metadata ambiguity	form/domain not set intentionally	pre-write review	cannot tell `Y+ISC` from `Z+VOC`
write-path issue	unstable output path or file handling	pack write	configure/attach succeeded but no artifact exists

7.8.3 Builder-To-Runtime Failure Isolation

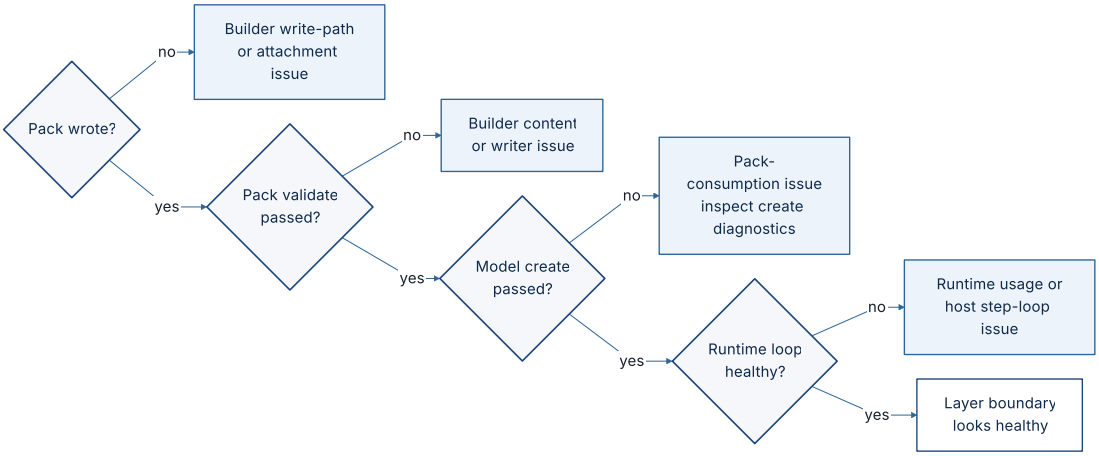


Figure 7.3 Builder-to-Runtime failure isolation

When a runtime test fails after a fresh pack write, split the investigation: did Builder produce the intended artifact, or did Runtime execute it incorrectly? This separation keeps teams from debugging the wrong layer.

7.8.4 Troubleshooting Decision Flow

Rapid checks: prefer `tdse_model_create(...)` over process-global create diagnostics; log `tdse_model_info` once at model create; log `t`, `dt`, and step index on each failed call; keep one deterministic repro input for local and automated runs.

7.8.5 Anti-Patterns

1. re-configuring a handle and assuming previous `H`/`IR` is still attached
2. treating `tdse_builder_write_pack(...)` as proof of semantic correctness
3. attaching `IR` without deciding pack representation metadata

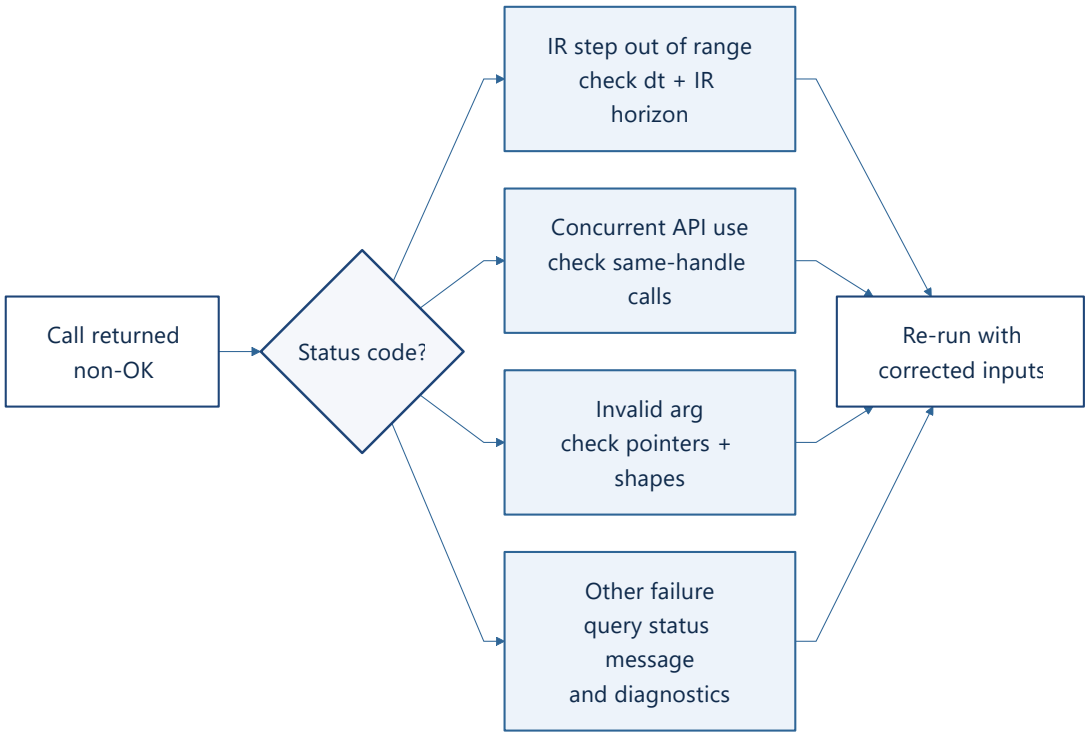


Figure 7.4 Runtime troubleshooting decision flow

4. debugging Runtime first when Builder snapshots were never captured
5. letting Runtime compensate for Builder-side source interpretation mistakes
6. using ad hoc output paths that break pack provenance

7.8.6 Pack Incident Triage

When a pack validation or create issue is reported, collect:

- Builder configure inputs (`dt`, `nh`, `np`, `nq`)
- Builder snapshot from `tdse_builder_info(...)`
- whether the pack came from direct `H` or spectrum conversion
- any conversion or tail-processing settings
- whether `IR` was attached and what pack-form metadata was used
- result of `tdse_pack_validate(...)`
- result and diagnostics from `tdse_model_create(...)`

If the first five items are missing, do not start by blaming Runtime.

CHAPTER 8

Troubleshooting

Use this chapter when something fails and you need the shortest path from a symptom to a likely cause. It starts with the fastest checks, then moves into diagnostics, common failure modes, and the information worth saving before you escalate an issue.

This chapter is useful before a failure happens too. Use it while designing host-side error handling, evidence capture, and support workflows so the first real incident does not force you to invent a diagnostics contract under time pressure.

For PoC or customer-facing evaluations, save one small evidence bundle for every blocking issue: the failing input artifact, exact command or API sequence, returned status, and the minimum diagnostic bundle named in this chapter. That habit makes later reproduction and support much faster.

8.1 Start With the Symptom

Use this table before reading the detailed status-code reference.

Symptom	First place to look	What to collect
Model create fails	create diagnostics and pack validation	failing API, status, `pack_error_code`, pack provenance
First step output looks wrong	prime-step order and Builder dimensions	`tdse_model_info`, step index, `t`, `dt`, first accepted primary vector
`step_ir` fails during a run	IR horizon and committed step count	`ir_nsteps`, committed step index, intended simulation horizon
`destroy` or shutdown hangs	same-handle ownership and wait policy	active worker thread, destroy wait budget, `tdse_model_destroy_result_t`
Adapter cannot parse or solve input	adapter diagnostics and CLI JSON output	command line, `--json-out -` output, netlist or RAW provenance
Performance is worse than expected	backend selection and profiler output	backend info, thread count, profiler recommendation JSON

8.2 Use This Chapter In Two Passes

When you are under time pressure, do not read this chapter from top to bottom.

Use it in two passes:

1. use the symptom table above, then jump to Minimum Diagnostic Bundle and First Five Minutes: Evidence Order

2. return to the deeper diagnostics or reference tails only if the fast path did not explain the failure

The back half of this chapter is intentionally deeper reference material. It is there to support difficult incidents, not to slow down first-response troubleshooting.

8.3 First Response Checklist

When a failure first appears, this is the shortest reliable response path:

1. identify the failing API or command
2. capture the returned status or CLI error
3. gather the minimum diagnostic bundle
4. classify the problem as create, step-loop, concurrency, shutdown, shape, adapter, or performance
5. use the symptom playbook below before reading the deeper reference tails

If that does not explain the failure, use the diagnostic interpretation sections that follow.

8.4 Pick The Right Failure Surface

Many incident reports get slower than they need to because the team is debugging the wrong layer first. Use this split before going deeper:

If the failure starts here	Treat it first as	Go next
`tdse_model_create(...)`, step APIs, or destroy/close	Runtime lifecycle or step failure	Symptom Playbook
`tdse_builder_*` or pack validation	Builder or handoff failure	Builder and Data Contracts and this chapter's create/shape symptoms
`tdse adapter circuit ...` command or adapter API	Adapter input, solver, or conversion failure	Adapter Circuit and CLI Reference
profiler command or runtime-plan mismatch	performance qualification failure	Profiler and Backend and Performance

As a rule of thumb, do not start with Runtime if the adapter never produced trustworthy output, and do not start with Builder if the first bad signal is already a live step-loop failure.

8.5 Diagnostic Interpretation

TDSE Core exposes structured diagnostics so you do not have to infer every failure from logs alone. The sections below show where to look first, how to read the snapshots, and how to turn them into the next troubleshooting step.

8.5.1 The Main Diagnostics Surfaces

When Builder or Runtime fails, these are usually the most useful places to look first:

- Builder and runtime status codes
- `tdse_status_message(...)` and `tdse_builder_status_message(...)`

- `tdse_model_create_diagnostics_t`
- `tdse_model_info(...)`
- `tdse_model_state_info(...)`
- `tdse_model_last_error_info(...)`
- exact step index, `t`, and `dt` at failure time
- destroy wait budget and `tdse_model_destroy_result_t` when shutdown is involved

For most Runtime failures, read them in this order:

1. the failing API name and returned status code
2. the create call's diagnostics struct, if create failed
3. state snapshot
4. sticky last-error snapshot
5. logs and any saved diagnostics

Figure 8.1 First-response diagnostics order

8.5.2 What Each Surface Answers

Three snapshot APIs matter most:

API	Use It For	Operational Question
`tdse_model_info(...)`	static metadata such as `np`, `nq`, `nh`, `dt`, IR presence, and `ir_nsteps`	what model did I actually load?
`tdse_model_state_info(...)`	current active-step and committed-step execution state	what state is the handle in right now?
`tdse_model_last_error_info(...)`	most recent non-OK runtime result on the handle	what was the last runtime failure worth remembering?

Recommended baseline:

- capture `tdse_model_info(...)` once at model create
- capture `tdse_model_state_info(...)` and `tdse_model_last_error_info(...)` on every non-OK runtime path
- archive the failing API name, status, step index, `t`, and `dt` next to those snapshots

The key difference is time perspective:

- `tdse_model_info(...)` is static model metadata
- `tdse_model_state_info(...)` is the current state at query time
- `tdse_model_last_error_info(...)` is the failure-time state captured when the last non-OK runtime call happened

If the current state and the last-error snapshot disagree, that is usually informative rather than suspicious. It often means the host continued running after the failure or a different thread changed ownership.

8.5.3 Create Diagnostics

When `tdse_model_create(...)` fails, start with its diagnostics struct:

```
tdse_model_create_diagnostics_t diag = tdse_model_create_diagnostics_init();
tdse_status_t st = tdse_model_create(pack_data, pack_size, &diag, &model);
```

Check:

- `st`
- `diag.status`
- `diag.pack_error_code`
- `tdse_pack_error_token(diag.pack_error_code)`

This is the standard create failure path. Do not build new integrations around historical process-global create diagnostics.

Recommended support record for every create failure:

- pack identity and provenance
- pack validation result on the same bytes
- `diag.status`
- `diag.pack_error_code`
- `tdse_pack_error_token(...)`
- whether diagnostics came from `tdse_model_create_diagnostics_init()`

8.5.4 Current-State Snapshot Semantics

`tdse_model_state_info_t` reports the current execution state only. It does not tell you what failed earlier on the handle.

The highest-value fields are:

Field	Meaning	Support Interpretation
`step_active`	a trial step is active right now	query-side state is inside a begun-but-not-yet-committed step
`has_committed_step`	at least one commit has succeeded	`dr` can become valid only after this is true
`committed_steps`	number of accepted commits	use as the committed-history index, not the number of attempted trials
`committed_t` / `committed_dt`	latest committed sample time and step size	anchor for the last accepted step
`active_t` / `active_dt`	active trial step coordinates	meaningful only while `step_active != 0`
`sim_time`	accumulated committed simulation time	should advance only through successful commit
`dr_last_valid`	direct-response query is currently legal	`1` means post-commit `dr` is queryable with no active trial step

Read the fields as a coherent state, not as isolated booleans. For example:

- `step_active=1` and `has_committed_step=0` means the handle is in its first trial before any accepted history exists

- `step_active=0`, `has_committed_step=1`, and `dr_last_valid=1` means the handle has committed history and is in the normal post-commit idle state
- `step_active=1` with `dr_last_valid=0` is the expected shape during an ordinary trial step and explains why `tdse_step_dr(...)` would be rejected

8.5.5 Sticky Last-Error Snapshot Semantics

`tdse_model_last_error_info_t` is a per-handle sticky snapshot of the most recent non-OK runtime result.

Two rules matter in practice:

1. successful runtime calls do not clear it
2. `tdse_model_reset()` clears it

That means the last-error snapshot answers "what most recently failed on this handle?" not "what is failing right now?"

The highest-value fields are:

Field	Meaning	Support Interpretation
`valid`	a failure snapshot exists	`0` means the handle has no recorded runtime failure since create/reset
`status`	most recent non-OK runtime status	classify the failure family first from this
`api_kind`	runtime API that returned `status`	tells you whether failure came from begin, query, commit, close, or destroy
`step_active` / `has_committed_step`	lifecycle snapshot at failure time	tells you whether failure happened pre-step, mid-trial, or post-commit
`committed_steps`	accepted-history depth at failure time	shows whether this was startup, first-step, or late-run behavior
`committed_t` / `committed_dt`	last accepted step at failure time	useful for post-commit incidents
`active_t` / `active_dt`	trial step coordinates at failure time	useful for begin/query/commit incidents
`sim_time`	committed simulation time at failure time	useful when the host kept running after the fault
`dr_last_valid`	whether committed `dr` was queryable at failure time	often separates pre-commit misuse from a post-commit query

`api_kind` is especially valuable because it collapses ambiguity fast. Common values to watch are:

`api_kind`	Meaning
`TDSE_RUNTIME_API_STEP_BEGIN`	failure while creating or re-entering a trial step
`TDSE_RUNTIME_API_STEP_HR` / `TDSE_RUNTIME_API_STEP_IR`	query failure during a trial
`TDSE_RUNTIME_API_STEP_COMMIT`	accepted-primary handoff failed
`TDSE_RUNTIME_API_STEP_DR`	direct-response queried in the wrong lifecycle state
`TDSE_RUNTIME_API_MODEL_CLOSE`	explicit close raced with in-flight same-handle work
`TDSE_RUNTIME_API_MODEL_DESTROY`	bounded destroy timed out or otherwise failed

8.5.6 Pairing Current State With Last Error

The strongest diagnostics read comes from comparing the current state snapshot with the sticky last-error snapshot.

Use this interpretation table:

Current State	Last Error	Likely Story
idle, no active step	invalid-state from `STEP_HR` or `STEP_IR` with no committed step	caller queried step terms before `begin`
active trial at `(t, dt)`	last error from `STEP_DR` with `step_active=1`	caller tried to read `dr` during a live trial
post-commit idle	last error still points to an older `STEP_HR` failure	host recovered or continued; sticky snapshot has not been reset
query APIs now return `TDSE_ERR_INVALID_STATE`	last error shows `MODEL_DESTROY` timeout or `MODEL_CLOSE` overlap	shutdown or ownership transition has begun
committed step count unchanged across retries	repeated query/solve failures without commit	host is retrying trials without advancing committed history

The practical support question is:

- is the host looking at the present handle state, or at the last place the handle failed?

Issue reports often go sideways when those two are confused.

8.5.7 High-Value Status Codes

The most important Runtime statuses for most integrations are:

Status	Typical Meaning	First Action
`TDSE_ERR_INVALID_ARG`	shape, pointer, or struct-size problem	validate caller inputs, buffer dimensions, and `struct_size` tags
`TDSE_ERR_INVALID_STATE`	API is valid in general but wrong for the current lifecycle state	inspect call order, shutdown ownership, and whether a trial is active
`TDSE_ERR_CONCURRENT_API_USE`	same handle was entered concurrently	inspect thread ownership and same-handle overlap
`TDSE_ERR_IR_STEP_OUT_OF_RANGE`	current step falls outside IR support	inspect `ir_nsteps`, model `dt`, and host horizon
`TDSE_ERR_TIMEOUT`	bounded destroy exceeded its wait budget	inspect in-flight same-handle work and shutdown policy

8.5.8 Status Families

It helps to group statuses by the question they answer:

Family	Representative Status	Meaning
caller contract	`TDSE_ERR_INVALID_ARG`	the caller provided illegal shape, pointer, or struct input
lifecycle contract	`TDSE_ERR_INVALID_STATE`	the API is valid in general but wrong for the current handle state
ownership / concurrency	`TDSE_ERR_CONCURRENT_API_USE`	the same handle was entered concurrently
model horizon	`TDSE_ERR_IR_STEP_OUT_OF_RANGE`	runtime time advanced beyond packaged IR support
shutdown policy	`TDSE_ERR_TIMEOUT`	bounded destroy did not complete within budget

8.5.9 Worked Snapshot Interpretations

8.5.9.1 Worked Example: Query Before `begin`

Observed behavior:

- failing API: `tdse_step_hr(...)`
- returned status: `TDSE_ERR_INVALID_STATE`
- current state: `step_active=0`, `has_committed_step=0`, `committed_steps=0`
- last error: `valid=1`, `status=TDSE_ERR_INVALID_STATE`, `api_kind=TDSE_RUNTIME_API_STEP_HR`

Interpretation:

- this is not a numerical problem
- the host never established a trial context before querying step terms
- because both current state and last-error state show no active or committed step, the incident is pure lifecycle misuse

Corrective action:

1. ensure `tdse_step_begin(model, t, dt)` succeeds before `tdse_step_hr(...)`, `tdse_step_ir(...)`, or `tdse_step_op(...)`
2. keep the API name with the status in logs so the misuse is visible immediately

8.5.9.2 Worked Example: `dr` Called During A Trial

Observed behavior:

- failing API: `tdse_step_dr(...)`
- returned status: `TDSE_ERR_INVALID_STATE`
- current state: `step_active=1`, `has_committed_step=1`, `dr_last_valid=0`
- last error: `api_kind=TDSE_RUNTIME_API_STEP_DR`, `step_active=1`

Interpretation:

- committed history exists, but the handle is currently inside a trial step
- `dr` is a post-commit query, not a trial-side query
- the failure is about timing inside the step lifecycle, not about missing history

Corrective action:

1. move `dr` after `commit`
2. do not query `dr` while `step_active != 0`

8.5.9.3 Worked Example: `IR` Horizon Miss Mid-Run

Observed behavior:

- failing API: `tdse_step_ir(...)`
- returned status: `TDSE_ERR_IR_STEP_OUT_OF_RANGE`
- model info: `has_ir=1`, `ir_nsteps=<finite value>`, `dt=<model dt>`
- state snapshot: `committed_steps` close to or beyond the supported IR horizon
- last error: `api_kind=TDSE_RUNTIME_API_STEP_IR`, `active_t` and `active_dt` identify the failing trial

Interpretation:

- Runtime is telling you the current trial step lies outside packaged IR support
- the most common causes are host horizon growth or Builder/runtime `dt` mismatch
- because `api_kind` is `STEP_IR`, do not waste time on operator or history math first

Corrective action:

1. compare host step index against `ir_nsteps`
2. confirm Builder and Runtime `dt` match
3. regenerate the pack if the simulation window legitimately grew

8.5.9.4 Worked Example: Close Overlap During Active Work

Observed behavior:

- failing API: `tdse_model_close(...)`
- returned status: `TDSE_ERR_CONCURRENT_API_USE`
- last error:
  - `valid=1`
  - `status=TDSE_ERR_CONCURRENT_API_USE`
  - `api_kind=TDSE_RUNTIME_API_MODEL_CLOSE`
- current state may still show an active trial or may already be post-commit by the time you inspect it

Interpretation:

- close was attempted while another same-handle runtime API was in flight
- this is an ownership bug or a shutdown sequencing bug, not a random close failure
- if the current state later looks idle, that only means the conflicting call finished after the incident

Corrective action:

1. identify which thread owns close/destroy initiation
2. stop treating close as a background-safe cleanup signal
3. move ordinary host shutdown to `tdse_model_destroy(...)` with an explicit wait policy

8.5.9.5 Worked Example: Destroy Timeout And Ownership Handoff

Observed behavior:

- `tdse_model_destroy(...)` returns `TDSE_ERR_TIMEOUT`
- `tdse_model_destroy_result_t.timed_out == 1`
- last error: `status=TDSE_ERR_TIMEOUT`, `api_kind=TDSE_RUNTIME_API_MODEL_DESTROY`
- later diagnostics may still succeed because the handle remains live after timeout

Interpretation:

- destroy did not acquire terminal ownership within the caller's wait budget
- the timeout does not mean the handle was destroyed
- the host must choose a next action explicitly: retry destroy, defer to finalizer cleanup, or report the issue

Support rule:

- record both the configured wait budget and the observed `wait_ms`
- if a follower thread later receives `TDSE_ERR_INVALID_STATE` from destroy or release, interpret that as ownership no longer being local to that thread

8.5.10 Typical Failure Patterns

8.5.10.1 Pack Rejected At Create

First check:

- `tdse_pack_validate(...)`
- `tdse_model_create_diagnostics_t`
- the pack token from `tdse_pack_error_token(...)`

Typical cause:

- malformed pack bytes
- incompatible pack structure
- mismatch between expected and supplied payload

Evidence to collect:

- exact pack token
- output of pack validation
- artifact provenance or source route

8.5.10.2 Invalid Lifecycle State

First check:

- whether `begin` was called before step queries
- whether `dr` was queried before the first commit
- whether a query was attempted after close or destroy began
- whether the host is reading a stale sticky last-error snapshot as if it were current state

8.5.10.3 Same-Handle Concurrent Use

First check:

- whether one handle is shared across threads
- whether the host wrapper allows overlapping entry on the same model handle
- whether shutdown paths can race with step APIs

8.5.10.4 Destroy Timeout

First check:

- the destroy wait budget
- the observed destroy wait result
- whether another same-handle API was still in flight
- whether the host expected `destroy` to behave like `release`

8.5.11 Symptom-To-Surface Table

Symptom	First Surface To Read	Second Surface	Typical Class
create fails	`tdse_model_create_diagnostics_t`	pack token	pack or compatibility issue
runtime query fails	failing API + returned status	`tdse_model_state_info(...)`	lifecycle or concurrency
destroy times out	destroy result	`tdse_model_last_error_info(...)`	shutdown overlap
direct response invalid	`tdse_model_state_info(...)`	`tdse_model_last_error_info(...)`	misuse of `dr`
wrong dimensions suspected	`tdse_model_info(...)`	host allocation site	shape mismatch
logs look stale or contradictory	compare state snapshot with last-error snapshot	failing API timeline	sticky snapshot confusion

8.5.12 Diagnostics By Lifecycle Phase

Phase	Highest-Value Diagnostics
create	create diagnostics, pack token, pack validation
steady execution	model info, state info, last-error snapshot, failing API name, `t`, `dt`, step index
shutdown	destroy result, wait budget, state snapshot, last-error snapshot, ownership interpretation

8.5.13 Production Archive Baseline

For production integrations, archive at least:

- `tdse_model_create_diagnostics_t` for non-OK create paths
- `tdse_model_info(...)`
- `tdse_model_state_info(...)`
- `tdse_model_last_error_info(...)`
- the failing API name
- the exact `t`, `dt`, and step index

If perf controls are in use, also archive the active execution-plan context.

And when shutdown behavior matters, also archive:

- destroy wait budget
- destroy wait result
- whether close, destroy, or release was used
- which thread or component initiated shutdown

8.5.14 Minimum Diagnostic Bundle

A minimally useful diagnostic bundle for runtime incidents contains:

1. pack identity and validation result
2. create diagnostics
3. model info
4. current state snapshot
5. sticky last-error snapshot
6. failing API name and returned status

7. `t`, `dt`, and step index if execution is involved
8. destroy wait budget and result if shutdown is involved

If any one of those is missing, it becomes much harder to reconstruct facts that the host already knew at failure time.

8.5.15 Troubleshooting Workflow

Use this sequence when triaging a new Runtime issue:

1. identify the failing API
2. capture the returned status
3. if create failed, read the create diagnostics first
4. if runtime execution failed, read current state and sticky last-error snapshots together
5. verify host dimension assumptions against `tdse_model_info(...)`
6. check whether the issue is lifecycle, concurrency, timing, shutdown, or pack-related

8.5.16 Diagnostic Anti-Patterns

Avoid:

1. logging only the status text without the API name
2. dropping create diagnostics on failed create paths
3. triaging shape issues without `tdse_model_info(...)`
4. reading only logs when snapshot APIs already explain the state
5. assuming the last-error snapshot clears itself after later success
6. treating `release` behavior as proof of destroy-policy correctness
7. reporting a destroy timeout without recording `wait_timeout_ms` and observed `wait_ms`

8.5.17 When To Report

Report an issue quickly when:

- the same qualified pack regresses on an unchanged supported host
- create diagnostics indicate a pack-compatibility surprise on a previously qualified route
- threading stress or destroy-timeout behavior changes across the same release line
- state and last-error snapshots imply a runtime contract break rather than host misuse

Keep triaging locally first when:

- the symptom is clearly a shape mismatch
- the host thread-ownership model is still ambiguous
- the support bundle does not yet contain both runtime snapshots

8.5.18 Validation And Testing

Recommended user-facing validation order:

1. validate a representative pack
2. create a runtime model successfully
3. run one minimal step loop end to end
4. confirm `op`, `hr`, and `ir` dimensions

5. intentionally exercise one known failure path
6. confirm last-error stickiness and reset behavior in one local repro

For production confidence, keep these checks green:

- threading stress
- `IR` contract replay
- Builder `IR` ingest contract
- runtime direct-response contract
- lifecycle diagnostics around reset, close, and destroy

8.6 Symptom Playbook

Users usually start from a symptom, not from an API name. This section turns the runtime rules into a field guide you can use under pressure.

8.6.1 How To Use This Section

For each problem:

1. match the visible symptom
2. collect the named evidence
3. classify the issue
4. apply the first corrective action
5. return to the Diagnostics and Error Handling section above only if needed

8.6.2 First Five Minutes: Evidence Order

Under support pressure, collect evidence in this order before changing the host code:

1. failing API name and returned `tdse_status_t`
2. `tdse_model_state_info(...)`
3. `tdse_model_last_error_info(...)`
4. create diagnostics or destroy result, depending on phase
5. exact `t`, `dt`, and step index from the host log

This order matters because it separates “what just failed” from “what state the handle was in” before logs or assumptions start to drift.

8.6.2.1 Minimal Incident Snapshot

Use this capture template on every non-OK path:

```
tdse_model_state_info_t state = tdse_model_state_info_init();
tdse_model_last_error_info_t last = tdse_model_last_error_info_init();

const tdse_status_t state_st = tdse_model_state_info(model, &state);
const tdse_status_t last_st = tdse_model_last_error_info(model, &last);
```

Archive together:

- failing API name
- returned status from the failing API
- `state_st` and `last_st`
- `state.step_active`, `state.has_committed_step`, `state.committed_steps`
- `last.valid`, `last.status`, `last.api_kind`
- host-side `t`, `dt`, and step index

If shutdown is involved, also archive `tdse_model_destroy_result_t`.

8.6.3 How To Read Runtime Snapshots

The fastest support skill is reading `state_info` and `last_error_info` as one combined record.

8.6.3.1 State Snapshot Interpretation

Snapshot Field Pattern	What It Usually Means	First Question
`step_active = 1`	a trial step is still open	which thread still owns the active trial?
`has_committed_step = 0`	no commit has succeeded yet	did the host skip the prime step or first ordinary commit?
`dr_last_valid = 0` with `has_committed_step = 1`	committed history exists, but direct response is not currently queryable	did the host start a new trial before querying `dr`?
`committed_steps` stops increasing while host step index advances	host is retrying or rejecting trials without commit	is that intentional or is commit being skipped accidentally?

8.6.3.2 Last-Error Snapshot Interpretation

Snapshot Field Pattern	What It Usually Means	First Action
`last.valid = 0`	no prior non-OK runtime API has been captured on this handle	rely on current failing status and state snapshot
`last.api_kind = TDSE_RUNTIME_API_STEP_IR` with `last.status = TDSE_ERR_IR_STEP_OUT_OF_RANGE`	runtime time exceeded packaged IR window	compare host horizon with `ir_nsteps` and `dt`
`last.api_kind = TDSE_RUNTIME_API_MODEL_DESTROY` with `last.status = TDSE_ERR_TIMEOUT`	bounded destroy timed out before ownership was acquired	inspect in-flight same-handle work and retry policy
`last.api_kind = TDSE_RUNTIME_API_MODEL_CLOSE` with `last.status = TDSE_ERR_CONCURRENT_API_USE`	close raced with active same-handle traffic	stop treating close as a normal shutdown path

Important reading rule:

- `tdse_model_last_error_info(...)` is sticky across later successful calls until reset

That means the snapshot is valuable evidence, but not necessarily proof that the most recent call failed for the same reason. Always pair it with the current failing API and current state snapshot.

8.6.4 Triage Matrix

If You See	Check First	Likely Class	Immediate Containment
create path returns non-OK	create diagnostics + pack token	pack contract	stop runtime triage and validate the pack bytes
step API returns `TDSE_ERR_INVALID_STATE`	`state.step_active` and lifecycle order	sequencing or shutdown overlap	stop issuing more same-handle step calls
shutdown returns `TDSE_ERR_TIMEOUT`	destroy result + `state.step_active`	shutdown overlap	keep the handle live and decide retry vs report
sporadic `TDSE_ERR_CONCURRENT_API_USE`	thread ownership map + last-error api kind	same-handle overlap	isolate one handle per thread
`dr` fails after the host starts next step	`state.step_active`	misuse of post-commit query window	move `dr` earlier or stop depending on it there

8.6.5 Symptom: Model Create Fails

Visible signs:

- `tdse_model_create(...)` returns non-OK
- no usable runtime handle is produced

Collect first:

- `tdse_model_create_diagnostics_t`
- `tdse_pack_error_token(...)`
- pack validation result

Most likely causes:

- malformed pack bytes
- incompatible pack version or structure
- caller forgot to initialize the diagnostics struct

First corrective actions:

1. rerun pack validation on the same bytes
2. log the stable pack token
3. confirm `tdse_model_create_diagnostics_init()` was used

Support note:

- if `tdse_model_create(...)` fails, do not spend time on step-loop or shutdown analysis
- this is a pack or caller-contract problem until proven otherwise

8.6.6 Symptom: Step Loop Runs, Then `step_ir` Fails

Visible signs:

- early steps succeed
- later `tdse_step_ir(...)` returns `TDSE_ERR_IR_STEP_OUT_OF_RANGE`

Collect first:

- current step index
- `t` and `dt`
- `ir_nsteps` from `tdse_model_info(...)`

Most likely causes:

- host simulation horizon exceeds packaged IR support
- Builder and Runtime assumptions about `dt` do not match

First corrective actions:

1. verify the intended simulation horizon
2. compare Builder `dt` with Runtime model `dt`
3. regenerate the pack if the IR support window is too short

Worked interpretation:

- if `last.api_kind` is `TDSE_RUNTIME_API_STEP_IR`
- and `state.committed_steps` is close to or beyond `ir_nsteps`
- the integration has a horizon mismatch, not a random runtime failure

Support-facing rule:

- record the first failing step index and the total intended horizon
- that pair is usually enough to distinguish bad pack content from a bad host loop bound

8.6.7 Symptom: `dr` Is Rejected

Visible signs:

- `tdse_step_dr(...)` returns `TDSE_ERR_INVALID_STATE`

Collect first:

- whether a trial step is still active
- whether at least one commit already happened

Most likely causes:

- `dr` called before the first commit
- `dr` called during an active trial step

First corrective actions:

1. move `dr` after `commit`
2. ensure the trial step has ended before the `dr` query

Fast discriminator:

- `state.step_active = 1` means the host is still inside the trial window
- `state.has_committed_step = 0` means no successful commit exists yet, so `dr` cannot succeed

8.6.8 Symptom: Runtime Calls Fail Sporadically In Multi-Thread Runs

Visible signs:

- intermittent `TDSE_ERR_CONCURRENT_API_USE`
- non-OK statuses appear only under load

Collect first:

- thread ownership model
- handle identity per worker
- failing API name

Most likely causes:

- one live handle is shared across worker threads
- supervisory code overlaps same-handle entry

First corrective actions:

1. enforce one-handle-per-thread
2. document which thread owns create, step, and shutdown
3. rerun threading stress after redesign

Tell-tale evidence pattern:

- the same handle identifier appears in logs from multiple workers
- `last.status = TDSE_ERR_CONCURRENT_API_USE`
- the failing API changes between incidents

That pattern usually means the ownership model is wrong, not that one specific API is buggy.

8.6.9 Symptom: Destroy Times Out

Visible signs:

- `tdse_model_destroy(...)` returns `TDSE_ERR_TIMEOUT`
- `out_result->timed_out` is true

Collect first:

- destroy wait budget
- destroy wait result
- whether another same-handle API was still active

Most likely causes:

- business logic started destroy while a step API was still running
- host assumed destroy behaved like silent release

First corrective actions:

1. inspect in-flight same-handle activity
2. decide whether to retry, report, or defer to finalizer cleanup
3. document the intended shutdown policy instead of relying on defaults

Destroy interpretation matrix:

Result	Meaning	What The Host Should Do Next
`TDSE_OK`	shutdown completed	clear references and continue
`TDSE_ERR_TIMEOUT` + `timed_out = 1`	destroy did not acquire ownership within budget; handle remains live	retry later or report with evidence

Result	Meaning	What The Host Should Do Next
`TDSE_ERR_INVALID_STATE`	another thread already started terminal cleanup	treat ownership as no longer local

Operational rule:

- never reinterpret `TDSE_ERR_TIMEOUT` as "probably destroyed anyway"
- timeout is specifically the case where the host must assume the handle still exists

8.6.10 Symptom: Shapes Look Wrong

Visible signs:

- host buffer sizes do not match what Runtime appears to expect
- operator shape assumptions do not line up with outputs

Collect first:

- `tdse_model_info(...)`
- chosen operator view policy
- host allocation site

Most likely causes:

- confusion between `np` and `nq`
- mixing square and rectangular operator views across call sites

First corrective actions:

1. read `np`, `nq`, and `nh` from `tdse_model_info(...)`
2. pick one explicit operator-view policy
3. update host allocations and asserts accordingly

Support note:

- shape problems are usually integration drift between allocation sites and the runtime contract
- they are rarely fixed by changing numerical tolerances or retry policy

8.6.11 Symptom: Results Differ At The First Ordinary Step

Visible signs:

- later steps look plausible, but the first ordinary step does not

Collect first:

- whether the implementation primes at `n = -1`
- whether history was committed before the ordinary loop

Most likely causes:

- skipped or incorrect prime-step sequence
- misunderstanding of committed-history initialization

First corrective actions:

1. compare implementation with the documented prime-step pattern
2. commit the intended pre-step history explicitly

Useful evidence:

- `state.has_committed_step = 0` at the start of the first ordinary step is a strong sign that the prime-step contract was skipped or never committed

8.6.12 Diagnostic Bundle for Reports

Before reporting an issue, collect:

- pack validation result
- create diagnostics
- model info
- state info
- last error info
- failing API name
- exact `t`, `dt`, and step index
- wait budget and wait result if shutdown is involved

Preferred incident bundle format:

```
api=tdse_step_ir
status=TDSE_ERR_IR_STEP_OUT_OF_RANGE
host_step=4096
t=4.096e-6
dt=1.000e-9
state.step_active=1
state.has_committed_step=1
state.committed_steps=4096
last.valid=1
last.status=TDSE_ERR_IR_STEP_OUT_OF_RANGE
last.api_kind=TDSE_RUNTIME_API_STEP_IR
```

This is the level of detail that lets support answer quickly without guessing.

8.6.13 Anti-Patterns During Triage

Avoid these moves even when the pressure is high:

1. retrying the same failing API without capturing `state_info` first
2. logging only status text without the API name
3. treating a sticky `last_error_info` snapshot as proof of the current failure cause
4. assuming destroy timeout means storage was already released
5. reporting concurrency failures before proving one-handle-per-thread ownership

8.6.14 Worked Diagnostic Scenarios

8.6.14.1 Scenario A. Step Failure That Is Really A Horizon Mismatch

Observed:

- host reports `TDSE_ERR_IR_STEP_OUT_OF_RANGE`
- `last.api_kind = TDSE_RUNTIME_API_STEP_IR`
- `state.committed_steps` is near the configured `ir_nsteps`

Conclusion:

- the pack and runtime are behaving consistently
- the host loop exceeded the packaged IR support window

Best next action:

- shorten the run horizon or rebuild the pack with longer IR support

8.6.14.2 Scenario B. Shutdown Failure That Is Really Ownership Handoff

Observed:

- thread A starts destroy
- thread B later receives `TDSE_ERR_INVALID_STATE` from destroy or release

Conclusion:

- this is not a local retry problem
- terminal cleanup ownership already moved to another thread

Best next action:

- stop same-handle use on thread B and clean up local references only

8.6.14.3 Scenario C. Invalid `dr` Call That Is Really Step-Order Drift

Observed:

- `tdse_step_dr(...)` returns `TDSE_ERR_INVALID_STATE`
- `state.step_active = 1`

Conclusion:

- the host started a new trial before reading committed direct response

Best next action:

- move `dr` before the next `begin`, or stop depending on `dr` in that path

8.6.15 Report Versus Fix Locally

Usually fix locally first when:

- the issue is a visible shape mismatch
- the integration does not yet enforce one-handle-per-thread
- create diagnostics clearly indicate malformed input

Escalate sooner when:

- a previously qualified pack stops creating on a supported host
- the same qualified shutdown path starts timing out on unchanged inputs
- runtime snapshots contradict the expected lifecycle contract

Before the sections below: if you already understand the symptom and only need stable reference material, everything that follows is deep reference rather than first-response guidance.

8.7 Deep Reference: Known Limitations --- Core

The sections below describe what is intentionally out of scope for the current release. Read them as product boundary statements, not as an apology list.

8.7.1 Workflow Boundary Limits

Current TDSE Core boundaries are:

- TDSE Core does not parse or solve domain models directly
- TDSE Core expects validated `H` and optional `IR`, or an already built pack
- adapter layers remain responsible for domain-specific preprocessing and validation

8.7.2 Runtime Execution Limits

Current Runtime limits are:

- same-handle concurrent step entry is not a supported contract
- execution-affecting runtime controls are mutable only before the first successful `tdse_step_begin(...)`
- Runtime does not infer port names, engineering units, or sign conventions
- bounded destroy is explicit; Runtime does not invent host shutdown policy on behalf of the caller

These are deliberate simplifications that keep the runtime contract deterministic and reviewable.

8.7.3 Builder Limits

Current Builder limits are:

- Builder validates dimensions and layouts but does not replace adapter-side semantic validation
- spectrum-to-`H` conversion is limited to documented correction methods and contracts
- pack metadata is descriptive and does not alter Runtime mathematics

8.7.4 Integration Limits

Current integration limits are:

- `IR` access is bounded by the configured sequence horizon
- Runtime plan application depends on optional perf support and recognized plan compatibility
- RuntimeCore support claims are narrower than “all SDK features on all hosts”

8.7.5 Unsupported Assumptions

Do not assume any of the following unless a release note says otherwise:

- same-handle concurrent entry will be serialized for you
- Runtime will reconstruct missing domain semantics
- Builder metadata changes runtime mathematics
- Linux support implies all distro or architecture variants

8.7.6 Documentation Interpretation Rule

If a convenience path, platform promise, or feature stack is not documented in the public guide, treat it as unsupported until a release note or support addendum says otherwise.

8.8 Deep Reference: Known Limitations --- Release

8.8.1 Current Release Limitations

The items below summarize limitations that matter during external evaluation. They are not theoretical restrictions; they reflect the current implementation boundaries described in this guide.

8.8.1.1 Element and Analysis Limits

- `NPORT` is supported in AC small-signal analysis only.
- `NPORT` returns unsupported for DC or transient paths in the current release.
- `SW` / `CSW` switch elements are supported in transient paths only.
- nonlinear `Q` and `M` device support is limited to transient paths.
- AC / `Y` / `Z` / AC-probe paths reject unsupported nonlinear combinations by design.

8.8.1.2 Frequency-Data Import Limits

- `NPORT` is intended for `Y` or `Z` datasets, not `S`-parameter import through the `NPORT` element.
- Touchstone parsing for `NPORT` accepts `.sNp` syntax but `S`-parameter semantics are not accepted for `NPORT` itself.
- `S`-parameter Touchstone usage belongs to the `S` element path, not the `NPORT` path.

8.8.1.3 Parser and Compatibility Limits

- parser compatibility is broad but not equivalent to full general-purpose SPICE compatibility
- `.model SW/CSW` is supported; unrelated `.model` types may be ignored by this adapter parser
- relative file-path resolution depends on whether the source was provided as file input or text input

8.8.1.4 Operational Limits

- environment-variable solver overrides are suitable for experimentation, but explicit policy APIs are preferred for reproducible host integrations
- Builder handoff still requires the caller to choose appropriate spectrum-to-`H` conversion settings; Adapter Circuit does not replace Builder policy decisions
- node-set mode rejects mutual-inductor boundaries that cross only one side of a `K` coupling
- node-set mode rejects partial inclusion of `S` / `NPORT` multiport blocks

8.8.1.5 Evaluation Guidance

Before reporting a limitation as a bug, check whether it falls into one of these intended boundaries:

- AC-only frequency-domain block import
- transient-only nonlinear element support
- solver/backend availability that depends on build configuration

Future releases may extend analysis or device coverage, but until such extensions are documented in the public guide, treat the limits above as the current external boundary.

8.9 Deep Reference: Known Limitations --- General

Current public-boundary limitations include:

- adapter-domain behavior is constrained by documented RAW/netlist scope
- some simulation/probe combinations require exact option sets
- profiler policy quality depends on representative case selection and machine stability
- profiler summaries are not schema-authoritative; JSON report is authoritative

Practical guidance:

- treat command appendices as contract source of truth
- keep replay cases minimal and deterministic
- archive reports with machine/build metadata for release use

8.10 Deep Reference: Status Code System

TDSE uses a structured status code system that spans all SDK modules. Understanding the structure helps you classify errors quickly and route them to the right team.

8.10.1 Status Domains

Every status code belongs to a domain that identifies which SDK module produced it:

Domain	Meaning	Typical Source
`TDSE_STATUS_DOMAIN_RUNTIME`	Runtime execution errors	`tdse_model_create`, step APIs, lifecycle APIs
`TDSE_STATUS_DOMAIN_BUILDER`	Builder pack-generation errors	`tdse_builder_apply_h`, `tdse_builder_write_pack`
`TDSE_STATUS_DOMAIN_CIRCUIT_AC`	Adapter Circuit AC-path errors	circuit compilation, FRF extraction, netlist parsing

Raw status codes from each domain overlap numerically. Always normalize to a unified code before logging or comparing:

```
tdse_ext_status_t unified = tdse_ext_status_from_runtime(st);
```

8.10.2 Classification

Every unified status code has a classification that tells you the error family:

Classification	Meaning	Operational Response
OK	success	continue
Error	permanent failure	inspect inputs, shapes, or lifecycle state
Timeout	time-bounded operation expired	retry or escalate (see destroy timeout)
Unsupported	feature not available in this build or configuration	check build features or backend availability
InvalidState	API called in wrong lifecycle state	check call order and handle ownership

Query the classification:

```
tdse_status_class_t cls = tdse_status_code_classify(unified);
const char* msg = tdse_status_code_message(unified);
```

8.10.3 Human-Readable Messages

Two message APIs exist:

API	Scope	When To Use
`tdse_status_message(st)`	raw domain-local status	quick logging within one module
`tdse_ext_status_message(unified)`	unified cross-module status	production logging across module boundaries

For production integrations, prefer the unified path:

```
tdse_ext_status_t unified = tdse_ext_status_from_runtime(st);
printf("error: %s\n", tdse_ext_status_message(unified));
```

8.10.4 Pack Error Tokens

Create failures produce a `pack_error_code` in the diagnostics struct. Decode it to a readable token:

```
const char* token = tdse_pack_error_token(diag.pack_error_code);
const char* name = tdse_ext_pack_error_name(diag.pack_error_code);
```

Common pack error tokens:

Token	Meaning
`CRC_MISMATCH`	pack bytes are corrupted
`INVALID_TOC`	table of contents is malformed
`MISSING_H_META`	required H metadata chunk is absent
`MISSING_H_DATA`	required H data chunk is absent
`SHAPE_MISMATCH`	declared dimensions contradict data size
`VERSION_UNSUPPORTED`	pack version is not supported by this runtime build

8.10.5 Complete Status Code Reference

8.10.5.1 Runtime Status Codes (`tdse_status_t`)

Value	Constant	Meaning	Recovery
0	`TDSE_OK`	Success	continue
1	`TDSE_ERR_INVALID_ARG`	Invalid argument: null pointer, wrong struct size, bad shape	check caller inputs, buffer dimensions, and `struct_size` fields
2	`TDSE_ERR_OUT_OF_MEMORY`	Memory allocation failed	reduce model count, free unused handles, check GPU memory
3	`TDSE_ERR_INTERNAL`	Internal runtime error	report with full diagnostic bundle
4	`TDSE_ERR_UNSUPPORTED`	Feature not available in this build	check build features with `tdse_perf_get_build_features_json`
6	`TDSE_ERR_IR_STEP_OUT_OF_RANGE`	Step time exceeds IR sequence horizon	extend IR sequence, clamp simulation horizon, or rebuild pack
8	`TDSE_ERR_CONCURRENT_API_USE`	Same handle entered concurrently	enforce one-handle-per-thread ownership
9	`TDSE_ERR_INVALID_STATE`	API called in wrong lifecycle state	check step order, handle ownership, and shutdown sequencing
10	`TDSE_ERR_TIMEOUT`	Bounded destroy exceeded wait budget	check in-flight same-handle work, retry or escalate

8.10.5.2 Builder Status Codes (`tdse_builder_err_t`)

Value	Constant	Meaning	Recovery
0	`TDSE_BUILDER_OK`	Success	continue
1	`TDSE_BUILDER_ERR_INVALID_ARG`	Invalid builder argument	check descriptor fields, struct sizes, and dimension consistency
2	`TDSE_BUILDER_ERR_OUT_OF_MEMORY`	Builder memory allocation failed	reduce nh or np, free builder handle
3	`TDSE_BUILDER_ERR_INTERNAL`	Internal builder error	report with builder snapshot
4	`TDSE_BUILDER_ERR_UNSUPPORTED`	Unsupported builder operation	check SDK version and build configuration
5	`TDSE_BUILDER_ERR_IO`	File I/O failure during pack write	check output path, permissions, and disk space
6	`TDSE_BUILDER_ERR_NP_MISMATCH`	Port count mismatch between configure and descriptor	ensure `np`/`nq` in `h_desc` or `ir_desc` match builder options

8.10.5.3 Extension Status Codes (`tdse_ext_status_t`)

Value	Constant	Meaning	Recovery
0	`TDSE_EXT_STATUS_OK`	Success	continue
1	`TDSE_EXT_STATUS_INVALID_ARG`	Invalid extension API argument	check input parameters
2	`TDSE_EXT_STATUS_OUT_OF_RANGE`	Index or value out of range	check array bounds and valid ranges

Value	Constant	Meaning	Recovery
3	`TDSE_EXT_STATUS_IO_ERROR`	I/O error in extension path	check file paths and permissions
4	`TDSE_EXT_STATUS_FORMAT_ERROR`	Format or parse error	check JSON/plan format
5	`TDSE_EXT_STATUS_NUMERIC_ERROR`	Numerical error in extension operation	check guard metrics and input validity
6	`TDSE_EXT_STATUS_UNSUPPORTED`	Unsupported extension operation	check build features
7	`TDSE_EXT_STATUS_OUT_OF_MEMORY`	Extension memory allocation failed	reduce workload or check resources
8	`TDSE_EXT_STATUS_INTERNAL_ERROR`	Internal extension error	report with diagnostic bundle
9	`TDSE_EXT_STATUS_INVALID_STATE`	Wrong state for extension API	check lifecycle state and call order
10	`TDSE_EXT_STATUS_TIMEOUT`	Extension operation timed out	retry or increase timeout

8.10.5.4 Status Classification and Recovery Guide

Classification	Recoverable?	Typical Response
OK	n/a	continue execution
`INVALID_ARG`	yes, fix caller code	validate inputs before retry
`OUT_OF_MEMORY`	yes, reduce load	free resources, reduce model count
`INTERNAL`	no (report)	collect diagnostics and report
`UNSUPPORTED`	no (design)	check build, use alternative path
`IO`	yes, fix environment	check paths, permissions, disk
`INVALID_STATE`	yes, fix sequence	check lifecycle order
`CONCURRENT_API_USE`	yes, fix ownership	enforce one-handle-per-thread
`TIMEOUT`	yes, retry or escalate	increase wait budget or quiesce first
`IR_STEP_OUT_OF_RANGE`	yes, extend horizon	rebuild pack with longer IR

8.10.6 Cross-Module Logging

For integrations that span Builder, Runtime, and Adapter Circuit, normalize all statuses to the unified system before writing to the same log sink:

```
void log_status(const char* source, tdse_status_t raw_st, int domain) {
    tdse_ext_status_t unified;
    switch (domain) {
        case DOMAIN_RUNTIME: unified = tdse_ext_status_from_runtime(raw_st); break;
        case DOMAIN_BUILDER: unified = tdse_ext_status_from_builder(raw_st); break;
        case DOMAIN_CIRCUIT: unified = tdse_ext_status_from_circuit_ac(raw_st); break;
    }
    printf("[%s] %s (class=%d)\n", source,
           tdse_ext_status_message(unified),
           tdse_status_code_classify(unified));
}
```

8.11 Deep Reference: Deterministic Mode

For reproducibility testing and regression comparison, the SDK provides a deterministic mode that disables non-deterministic parallel and random behavior.

8.11.1 Enabling Deterministic Mode

```
tdse_ext_set_deterministic_mode(1);
```

This is a process-global setting. Call it before any model creation if you need reproducibility.

8.11.2 What It Disables

When deterministic mode is active:

- Internal parallelism is reduced to a single-threaded path for all step operations.
- Any non-deterministic scheduling (e.g., work-stealing, dynamic thread pools) is replaced with deterministic sequential execution.
- Results are bit-identical across runs on the same hardware and build.

8.11.3 What It Does NOT Change

- Pack contents (determinism is a runtime execution property, not a Builder property).
- The mathematical result (the same convolution is computed; only the execution strategy changes).
- Backend selection (you can still use any backend, but internal parallelism is serialized).

8.11.4 Querying Current State

```
int is_deterministic = tdse_ext_get_deterministic_mode();
```

8.11.5 When To Use It

Scenario	Deterministic Mode	Rationale
Regression testing	ON	Bit-exact comparison across runs
Performance benchmarking	OFF	Measures real-world throughput
Debugging numerical issues	ON	Eliminates thread-scheduling as a variable
Production deployment	OFF	Enables full parallelism for throughput

> **IMPORTANT**
> Deterministic mode may significantly reduce throughput. Do not enable it in production unless reproducibility is a hard requirement.

8.12 Deep Reference: Runtime Guard

The runtime guard monitors convolution stability and provides early warning when numerical behavior degrades. It does not change numerical outputs — it only observes and reports.

8.12.1 Configuration

```
tdse_ext_runtime_guard_config_t guard_cfg;
tdse_ext_get_runtime_guard_config(&guard_cfg);
guard_cfg.pivot_warning_threshold = 0.1;
tdse_ext_set_runtime_guard_config(&guard_cfg);
```

8.12.2 Reading Guard Metrics

```
tdse_ext_runtime_guard_metrics_t metrics;
tdse_ext_get_runtime_guard_metrics(model, &metrics);
```

Key metrics:

Metric	Meaning	Watch For
`max_abs_g0`	max absolute value of `h[0]` entries this step	model-dependent; use as trend indicator
`pivot_min`	minimum pivot from the instantaneous operator factorization	significant drop means operator is near-singular
`pivot_ratio`	current `pivot_min` divided by the baseline from the first step	sustained values below 0.1 warrant investigation
`growth_factor`	ratio of `|hr|` this step vs. previous step	sustained values above 1 indicate potential divergence

8.12.3 Interpretation Guide

```
pivot_ratio trend:
├─ > 0.5    → healthy, operator well-conditioned
├─ 0.1-0.5  → watch; may be normal for stiff systems at startup
└─ < 0.1    → investigate; operator may be losing rank

growth_factor trend:
├─ ~1.0     → steady state, history term is stable
├─ 1.0-1.5  → mild growth, may be transient startup behavior
└─ > 1.5    → sustained growth suggests potential divergence
```

8.12.4 Resetting Metrics

```
tdse_ext_reset_runtime_guard_metrics(model);
```

Reset when starting a new simulation phase or after resolving a known transient event.

8.12.5 Relationship To Variable dt

When using variable time-stepping, the guard metrics are especially valuable for detecting whether aggressive dt changes are causing interpolation error accumulation. See Variable Time-Step Integration for the interaction between dt strategy and guard monitoring.

8.13 Deep Reference: Structured Logging

The SDK provides a structured logging system that integrations can route to their own log sinks.

8.13.1 Setting Up A Log Callback

```
void my_log_callback(tdse_log_level_t level, const char* message, void* user_data) {
    if (level >= TDSE_LOG_LEVEL_WARN) {
        fprintf(stderr, "[TDSE %d] %s\n", level, message);
    }
}

tdse_ext_set_log_callback(my_log_callback, NULL);
```

8.13.2 Log Levels

Level	Name	Typical Content
TDSE_LOG_LEVEL_TRACE	Trace	Very detailed internal diagnostics
TDSE_LOG_LEVEL_DEBUG	Debug	Step-level internals, useful during development
TDSE_LOG_LEVEL_INFO	Info	Normal operational messages (model created, backend set, etc.)
TDSE_LOG_LEVEL_WARN	Warning	Guard threshold crossings, near-singular conditions, fallback paths
TDSE_LOG_LEVEL_ERROR	Error	API failures, pack validation errors, resource exhaustion
TDSE_LOG_LEVEL_FATAL	Fatal	Unrecoverable internal errors

8.13.3 Setting Log Level

```
tdse_ext_set_log_level(TDSE_LOG_LEVEL_WARN);  /* only warn and above */
```

Query current level:

```
tdse_log_level_t level = tdse_ext_get_log_level();
```

8.13.4 Emitting Custom Log Messages

Integrations and adapters can use the same structured log system:

```
tdse_ext_log_emit(TDSE_LOG_LEVEL_INFO, "custom_module", "model step completed");
```

8.13.5 Plugin System

For plugin load failures, manifest issues, or backend routing problems, use `tdse_plugin_doctor` to inspect installed plugins and their manifests:

```
TDSE_PLUGIN_MANIFEST=/opt/tdse/lib/plugins/sim/plugin_manifest.json \
  tdse_plugin_doctor
```

The doctor reports ABI compatibility, health status, sha256 verification, and manifest entry matches. Load failure diagnostics use stable categories (`[file_not_found]`, `[abi_mismatch]`, `[hash_mismatch]`, `[manifest_invalid]`) that can be parsed from logs.

Treat the doctor output as first-response evidence, not as the whole incident record. For blocking RC issues, save the exact package version, manifest path, failing backend request, and the smallest reproducer alongside the doctor report.

See the Plugin System chapter for full deployment and troubleshooting details.

8.13.6 Integration Guidance

- Set the log level to `WARN` for production deployments.
- Set to `DEBUG` or `TRACE` only during active debugging — these levels produce significant output.
- Always pair structured logs with status code classification for complete diagnostics.
- Route TDSE log output through your host application's logging infrastructure for unified observability.

CHAPTER 9

Adapter Circuit

Use this chapter when your starting point is a circuit description rather than Builder-ready `H` or FRF data. It shows how Adapter Circuit turns netlists and RAW cases into matrices, source-side waveforms, probes, and Builder-ready handoff data.

Use this chapter for workflow meaning and integration choices. If you only need exact command flags, jump to CLI Reference. If you only need to check whether your input deck is inside the supported subset, jump to Element Reference.

Read this chapter in three passes if you need to:

- get one common task working from the CLI
- embed the adapter through the C API
- tune advanced behavior such as solver policy, planning, or performance

Start with the shortest supported path that matches your question. If your goal is a qualified `.pack` with minimal glue, jump to the workflow path first. If your goal is circuit visibility, start with the CLI matrix / probe flows. Drop to the lower-level C API only when you need explicit phase control or intermediate artifacts.

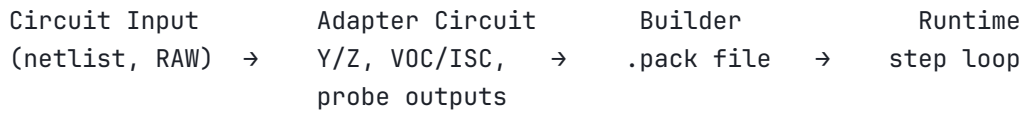
For most integrations, Adapter Circuit falls into one of these roles:

Integration goal	Recommended starting surface
netlist or RAW -> `.pack` as fast as possible	workflow API
inspect matrices or probes before building a pack	CLI matrix / probe path
embed circuit compilation and sweeps inside host code	compile-then-compute C API

9.1 What Adapter Circuit Owns

Imagine you have a circuit — maybe a SPICE netlist from your PDK, maybe a PSS/E RAW case from a grid study, maybe a custom deck with frequency-dependent NPORT elements. You need to get that circuit into TDSE so Builder can produce a runtime pack. But Builder doesn't understand circuits. It understands frequency-domain matrices and time-domain source waveforms.

Adapter Circuit bridges that gap. It takes circuit descriptions, compiles them into a solvable form, and produces the outputs Builder needs: Y and Z matrices over frequency, VOC and ISC time sequences, and probe data for validation. It also handles RAW import — converting PSS/E power-system cases into circuit netlists.



The separation between layers is deliberate:

- **Adapter Circuit** owns circuit parsing, solve orchestration, and circuit-domain validation. It knows about nodes, branches, and MNA stamping. It does not know about Builder packs or Runtime step loops.
- **Builder** converts validated frequency-domain data into a runtime pack. It knows about rational fitting and passivity enforcement. It does not know about circuits.
- **Runtime** executes the pack against a host solver. It knows about time stepping and conductance matrices. It does not know about circuits or fitting.

This layering means you can swap any piece independently. Use a different circuit tool to produce Y matrices? Feed them directly to Builder. Want to use the circuit solver without Builder? Adapter Circuit's probe and series commands give you standalone access.

9.1.1 When to Use It

You should reach for Adapter Circuit when your source data begins as:

- A SPICE-like netlist (`.cir`, `.sp`, or text)
- A PSS/E RAW case file that needs conversion
- A circuit deck containing `NPORT` elements that reference Touchstone files (`.ynp`, `.znp`, `.y2p`, `.s2p`, etc.)

The typical workflow is: compile a netlist, run one or more compute operations, then hand the results to Builder. But you can also use Adapter Circuit standalone — for circuit validation, for exploring frequency response, or for generating probe data.

9.1.2 What It Produces

Every Adapter Circuit operation falls into one of these categories:

- **Frequency-domain matrices**: `Y(jω)` or `Z(jω)` over a user-specified frequency grid. These are the primary input to Builder's `h_from_spectrum`.
- **Source-side time sequences**: Open-circuit voltage (`VOC`) or short-circuit current (`ISC`) waveforms. These define the Norton/Thevenin equivalents that drive the TDSE model in a host solver.
- **Probe outputs**: Internal node voltages or branch currents, in either AC (frequency sweep) or transient (time-domain) form. Probes let you inspect what's happening inside the circuit without extracting full port matrices.
- **Diagnostics**: Structured data describing parser behavior, solver backend selection, matrix conditioning, and policy decisions. Every result struct carries a diagnostics block.

9.1.3 Header Map

Most users only need `tdse_adapter_circuit.h`. Pull in narrower headers only when you are using those features directly:

Header	Purpose
`tdse_adapter_circuit.h`	Umbrella include — pulls in everything
`tdse_adapter_circuit/common.h`	Handle lifecycle, error codes, core data types
`tdse_adapter_circuit/compile.h`	Netlist compilation, compiled info queries

Header	Purpose
`tdse_adapter_circuit/compute.h`	Matrix sweeps, port series, probes, region preparation
`tdse_adapter_circuit/raw.h`	RAW import and conversion (PSS/E → netlist)
`tdse_adapter_circuit/policy.h`	Solver policy and backend selection
`tdse_adapter_circuit/diagnostics.h`	Diagnostics, policy tracing, solver statistics
`tdse_adapter_circuit/planning.h`	Adaptive sweep planning and tail-vs-nfreq analysis
`tdse_adapter_circuit/seq.h`	Sequence network import (.seq files) and fault analysis
`tdse_adapter_circuit/init.h`	Struct initializers, convenience helpers, macros
`tdse_adapter_circuit.hpp`	C++ wrapper with RAII and exception-based errors
`tdse_workflow.h`	High-level workflows (e.g. `tdse_workflow_netlist_to_pack`)

For most integration work, treat this table as a lookup aid rather than required reading.

9.2 Quick Start

Before diving into the full API, here is the shortest path from a netlist to a frequency-domain matrix. If you have a SPICE file called `my_circuit.cir` and you want its Y matrix at DC plus four positive frequencies:

```
tdse adapter circuit matrix \
  --netlist-kind file --netlist my_circuit.cir \
  --matrix y --ports "1,0" \
  --w0-radps 0 --dw-radps 100 --nfreq 5 \
  --out-data matrix.csv --json-out -
```

This single command compiles the netlist, solves the circuit at each frequency, extracts the port admittance, and writes the results to `matrix.csv`. The `--json-out -` flag prints a machine-readable envelope to stdout — you will want this in any automated workflow.

The output CSV contains one row per frequency, with the real and imaginary parts of each Y matrix element in interleaved order: `re(Y11), im(Y11), re(Y12), im(Y12), ...`.

For runnable examples of every CLI command and C API pattern, see the Examples Guide.

9.3 Choose Your Path

Use this table before reading deeper:

If you need to...	Read first	Then use
turn a netlist or RAW case into a `.pack` with minimum glue	Recommended Workflow API Path	Examples Guide
get a Y or Z matrix for Builder	Common CLI Tasks	Builder Handoff
convert a PSS/E RAW case	RAW Import	Builder Handoff
inspect internal circuit behavior	`probe` — Observe Internal Voltages or Currents	Diagnostics
embed the adapter in host code	Core C API Tasks	Embedding in a Host Solver (MNA)
tune backend choice or sweep strategy	Solver Policy	Advanced Tuning And Performance

For exhaustive CLI flags and machine-readable CLI outputs, use CLI Reference. This chapter keeps the command examples and the adapter-side meaning, not every CLI contract detail.

9.4 Common End-To-End Paths

Most users do not need this whole chapter at once. They need one of these short paths:

Starting point	Short path
SPICE-like netlist -> Builder pack	`matrix` -> Builder handoff -> Runtime create
PSS/E RAW case -> Builder pack	`raw-to-netlist` -> `matrix` -> Builder handoff
netlist -> validation data only	`probe` or `series` -> diagnostics review

If your goal is a `.pack` file, stay focused on matrix generation and Builder handoff first. Probe and series workflows are more useful when you are validating the circuit model or debugging a mismatch.

9.4.1 Minimal Integration Decision

Before writing code, decide which side of the boundary needs to own intermediate artifacts:

- if the host only needs a qualified `.pack`, use the workflow path
- if the host must archive or inspect Y/Z, VOC/ISC, or probe outputs, use Adapter directly
- if the host must control every phase boundary, use compile-then-compute and then hand off to Builder explicitly

9.5 Recommended Workflow API Path

If your real goal is "turn this netlist or RAW case into a qualified `.pack` with as little hand-stitched glue as possible," prefer the workflow API before you build your own Adapter -> Builder orchestration.

The public workflow surface covers three common starting points:

- `tdse_workflow_netlist_to_pack(...)`
- `tdse_workflow_raw_to_pack(...)`
- `tdse_workflow_yz_matrix_to_pack(...)`

Why this path is valuable:

- it keeps Builder planning inputs in one request object
- it records whether the grid or sweep plan was auto-derived
- it returns passivity and round-trip verification in the same result
- it gives you one status surface that still preserves underlying Adapter, Builder, Runtime, and N-port outcomes

Use the workflow API when:

- you want the shortest supported path from circuit input to `.pack`
- your host does not need to intercept every intermediate matrix artifact
- you want one qualification result that already includes pack-quality signals

Drop down to manual Adapter + Builder handoff only when you need to:

- archive intermediate Y or Z matrices explicitly
- inject a custom Builder correction or tail-processing sequence outside the workflow defaults
- debug a mismatch by isolating compile, sweep, Builder conversion, and Runtime verification as separate phases

For a runnable public example, start with `workflow_cpp_quickstart` in the Examples Guide.

Workflow ownership split:

- workflow / Adapter / Builder own circuit compilation, planning, and pack construction
- the host owns source-file selection, workflow parameters, artifact archival, and the later Runtime embedding choice

9.6 Mental Model

Before you start writing code, it helps to understand the two fundamental patterns in Adapter Circuit.

The compile-then-compute pattern. You compile a netlist once, which gives you a handle — an opaque object that holds the parsed and analyzed circuit. That handle is then reused across as many compute calls as you need. Compilation is the expensive step (parsing, topology analysis, MNA construction). Compute calls are relatively cheap, and you can run them with different parameters — different frequency grids, different ports, different time steps — against the same handle.

```
compile_from_netlist(&req, &result)  →  handle
    │
    ├── compute_port_fsweep(handle, ...)   →  Y or Z matrix
    ├── compute_port_series(handle, ...)   →  VOC or ISC time series
    ├── compute_probes(handle, ...)        →  internal voltage/current
    └── compute_port_fsweep(handle, ...)   →  another sweep, same handle
```

The two-output pattern. Every Adapter Circuit function that produces data follows a two-call pattern. First call: pass `NULL` for the output buffer and zero for its length — the result tells you how much space you need. Second call: allocate and pass the real buffer. This avoids guessing buffer sizes. The `_init()` functions and the sizing pass handle the details for you.

9.7 Common CLI Tasks

The CLI is the fastest way to validate a netlist, explore a circuit, or generate Builder-facing artifacts. The sections below explain the intent of each command family. Use CLI Reference when you need exhaustive flag or output-field detail.

All adapter circuit commands share two conventions: `--json-out <path|->` for machine-readable output, and `--netlist-kind <file|text|stdin>` for specifying how the netlist is provided.

If you are writing a script, always use `--json-out -` (stdout) and parse the JSON envelope. The human-readable output is for interactive use only — its format may change between versions.

9.7.1 `caps` --- Check Available Backends

Before you run any computation, you need to know what solver backends are available on your machine. The `caps` command answers this: it prints a table of every backend the SDK was built with, and whether each one is usable right now.

```
tdse adapter circuit caps --out-caps - --json-out -
```

This is the first thing to run on a new machine or after upgrading the SDK. A backend might show as unavailable because a required library is missing (CUDA, MKL), because the GPU driver is outdated, or because the backend was not compiled into this build.

Flag	Meaning
`--out-caps <path|->`	Write backend capability table (default: stdout)
`--out-build-features <path|->`	Write build-time feature flags as JSON
`--json-out <path|->`	Machine-readable output envelope

9.7.2 `matrix` --- Compute Y or Z over Frequency

This is the workhorse command. You have a netlist and a set of ports, and you want the admittance (Y) or impedance (Z) matrix at each frequency in a grid. The output goes directly to Builder — the port order you specify here must match the port order you use in `h_from_spectrum`.

```
tdse adapter circuit matrix \
  --netlist-kind file --netlist my_circuit.cir \
  --matrix y --ports "1,0;2,0" \
  --w0-radps 0 --dw-radps 100 --nfreq 5 \
  --out-data matrix.csv --json-out -
```

The frequency grid is defined by three numbers: a start frequency `w0` in rad/s, a step `dw` in rad/s, and a count `nfreq`. The grid covers `w0`, `w0 + dw`, `w0 + 2*dw`, ..., `w0 + (nfreq-1)*dw`. For a DC start use `--w0-radps 0`.

Flag	Meaning	Default
`--netlist-kind file|text|stdin`	How the netlist is provided	(required)
`--netlist <path_or_text>`	Netlist source (omit for stdin)	—
`--matrix y|z`	Admittance (Y) or impedance (Z)	(required)
`--ports "p,n;p,n;..."`	Port definitions by numeric node index. String names in C API	(required)
`--w0-radps`	Start frequency in rad/s. Use `0` for DC	(required)
`--dw-radps`	Frequency step in rad/s	(required)
`--nfreq`	Number of frequency points	(required)
`--dc-policy`	DC frequency resolution policy	`exact_dc_then_fallback`
`--dc-extrapolate-points <N>`	Points used for DC extrapolation	4
`--out-data <path|->`	Output CSV file path	—

DC policy values:

Solving at exactly zero frequency (DC) is harder than it sounds. Inductors become shorts, capacitors become opens, and the MNA matrix can become singular. These policies control how the solver handles ω = 0:

CLI value	Enum constant	Description
`regularized_exact_dc`	`W0_REGULARIZED_EXACT_DC`	Adds small conductance to regularize inductors at DC
`extrapolate_from_positive`	`W0_EXTRAPOLATE_FROM_POSITIVE`	Skips DC, extrapolates from positive-frequency data
`exact_dc_then_fallback`	`W0_EXACT_DC_THEN_FALLBACK`	Exact DC first; regularization fallback if singular

The default (`exact_dc_then_fallback`) is the right choice for most circuits. Switch to `extrapolate_from_positive` if your circuit has no DC path to ground (floating nets, ideal transformers) and regularization isn't helping.

9.7.3 `series` --- Compute VOC or ISC Time Sequences

The `matrix` command gives you frequency-domain behavior. But to drive a time-domain simulation, you often need source waveforms — the open-circuit voltage or short-circuit current seen at each port as a function of time. That's what `series` computes.

```
tdse adapter circuit series \
  --netlist-kind file --netlist my_circuit.cir \
  --series voc --ports "1,0" \
  --method transient --dt 1e-4 --steps 64 \
  --out-data series.csv --json-out -
```

The output is a time series: one row per time step, one column per port. The time at step k is `t0 + k * dt`.

Flag	Meaning	Default
`--series voc|isc`	Open-circuit voltage or short-circuit current	(required)
`--ports "p,n;p,n;..."`	Port definitions	(required)
`--method transient|ifft|tone`	Synthesis method	(required)
`--dt`	Time step in seconds	(required)
`--steps`	Number of time steps	(required)
`--nfft <N>`	FFT size for `ifft` method	—
`--t0 <t0>`	Start time offset	`0`
`--out-data <path|->`	Output CSV file path	—

Choosing a synthesis method:

The method you pick depends on your circuit and what accuracy you need:

Method	When to use
`transient`	General-purpose time-domain solver. Handles nonlinear devices and switching
`ifft`	Frequency-domain sources, linear circuit. Faster than transient. Requires `nfft ≥ 2*(nfreq-1)`

Method	When to use
`tone`	Single-frequency steady-state analysis. Use for AC steady-state verification, not for transient simulation

If you are unsure, start with `transient`. It is slower but always correct. Once you have validated results, you can experiment with `ifft` for speed.

9.7.4 `probe` --- Observe Internal Voltages or Currents

Port matrices tell you what happens at the terminals. But often you need to see inside the circuit — the voltage at an internal node, the current through a particular branch. Probes let you instrument the circuit and extract those values without modifying the netlist.

```
# AC probe — frequency sweep of internal quantities
tdse adapter circuit probe \
  --domain ac --netlist-kind file --netlist my_circuit.cir \
  --observe "v(1);v(2,0)" --observe-source argument \
  --ac-excitation small_signal \
  --sweep-kind lin --fstart-hz 50 --fstop-hz 5000 --sweep-points 100 \
  --out-data probe.csv --json-out -

# Transient probe — time-domain waveform of internal quantities
tdse adapter circuit probe \
  --domain transient --netlist-kind file --netlist my_circuit.cir \
  --observe "v(1);i(r1)" --observe-source argument \
  --method transient --dt 1e-4 --steps 64 \
  --out-data probe.csv --json-out -
```

Probes work in both AC and transient domains. In the AC domain, you get frequency sweeps of complex voltages and currents. In the transient domain, you get time-domain waveforms. The probe expressions use a simple syntax:

Expression	Meaning
`v(n)`	Voltage at node `n` relative to ground
`v(p,n)`	Voltage between nodes `p` and `n`
`i(r1)`	Current through element named `r1`
`i(vsrc)`	Current through independent source `vsrc`

Flag	Meaning	Default
`--domain ac|transient`	AC small-signal or transient time-domain	(required)
`--observe "v(n);v(p,n);i(element);..."`	Probe expressions, semicolon-separated	—
`--observe-source argument|netlist|argument_or_netlist`	Probe definition source	`argument_or_netlist`
`--ac-excitation small_signal|legacy_tones`	AC excitation mode	`small_signal`
`--sweep-kind from_netlist|lin|dec|oct|list`	Frequency sweep type for AC	`from_netlist`

Flag	Meaning	Default
`--sweep-points <N>`	Number of points for `lin`/`dec`/`oct` sweeps	—
`--fstart-hz <f0>`	Start frequency in Hz	—
`--fstop-hz <f1>`	Stop frequency in Hz	—
`--freq-list-hz <f0,f1,...>`	Explicit frequency list for `list` sweep	—
`--method transient|ifft|tone`	Synthesis method (transient probes)	(required)
`--dt`, `--steps`	Time step and count (transient probes)	(required)
`--nfft <N>`	FFT size (transient)	—
`--t0 <t0>`	Start time offset (transient)	`0`

The `--observe-source` flag is worth understanding. When set to `argument`, the CLI uses the probes you specify with `--observe`. When set to `netlist`, it reads `.probe` directives from the netlist itself. The default `argument_or_netlist` checks both: if you pass `--observe` it uses those; otherwise it falls back to netlist probes.

9.7.5 When to Use Which Command

If you are new to Adapter Circuit, this table is your cheat sheet:

You want to…	Use
Check what solvers are available on this machine	`caps`
Get `Y` or `Z` over frequency for Builder handoff	`matrix`
Get `VOC` or `ISC` time-domain waveforms for a host solver	`series`
Observe internal voltages or branch currents for validation	`probe`

9.8 RAW Import

Not every circuit starts as a SPICE netlist. In power systems, the starting point is often a PSS/E RAW case file — a grid description with buses, generators, branches, loads, and transformers. RAW Import converts these files into circuit netlists that Adapter Circuit can compile.

The conversion process reads the RAW file, builds an equivalent circuit model for each power-system element, stitches them together, and writes out a SPICE netlist. Base- frequency validation runs automatically — the generated netlist is solved at the nominal system frequency and the resulting bus voltages and branch flows are compared against the original RAW data.

9.8.1 CLI

```
tdse adapter circuit raw-to-netlist \
  --raw-kind file --raw case.raw \
  --out-netlist converted.cir \
  --unit pu --transformer-model ideal_tap_series --load-model shunt \
  --include-generator-sources 1 --json-out -
```

The three model choices — transformer, load, and generator — control how faithfully the netlist reproduces the RAW case behavior:

Flag	Meaning	Default
`--raw-kind file|text|stdin`	How the RAW input is provided	(required)
`--raw <path_or_text>`	RAW source	(required)
`--out-netlist <path|->`	Output netlist path	—
`--out-report-json <path|->`	Output validation report	—
`--unit pu|si`	Per-unit or SI output	`pu`
`--transformer-model series_only|ideal_tap_series|tap_split_shunt`	Equivalent circuit model	`ideal_tap_series`
`--load-model shunt|series|zip_split`	Load equivalent model	`shunt`
`--include-generator-sources 0|1`	Include generator Norton equivalents	`1`
`--include-fallback-source 0|1`	Add fallback source for unexcited buses	`1`
`--include-tran 0|1`	Emit `.tran` directive	`1`
`--include-options 0|1`	Emit `.options` directive	`1`
`--dt <sec>`	Time step for `.tran` directive	`5e-5`
`--tstop <sec>`	Stop time for `.tran` directive	`0.4`
`--freq-hz|--nominal-frequency-hz <hz>`	Base frequency in Hz	`60`
`--global-bus-shunt-c-f <farad>`	Global bus shunt capacitance	`0`
`--include-inactive-bus 0|1`	Include inactive buses in validation	`0`
`--ac-adjust-from-sine 0|1`	Adjust AC phasors from sine reference	`1`
`--mag-tol-pu <x>`	Magnitude tolerance for validation	`0.03`
`--ang-tol-deg <x>`	Angle tolerance in degrees	`3.0`
`--complex-tol-pu <x>`	Complex tolerance for validation	`0.08`
`--report-top-k <N>`	Top-K worst-case buses in the report	`20`

9.8.2 C API

The C API for RAW import uses the same pattern as the rest of Adapter Circuit: configure a request struct, call the function, and check the result. It uses the same `tdse_adapter_circuit_err_t` error type, so `tdse_adapter_circuit_status_message()` works for both adapter and RAW errors.

```c
#include <tdse_adapter_circuit/raw.h>

tdse_adapter_circuit_raw_import_options_t import_opt;
memset(&import_opt, 0, sizeof(import_opt));
import_opt.struct_size = sizeof(import_opt);
import_opt.unit_mode = TDSE_ADAPTER_CIRCUIT_RAW_UNIT_PU;
import_opt.transformer_model =
    TDSE_ADAPTER_CIRCUIT_RAW_TRANSFORMER_IDEAL_TAP_SERIES;
import_opt.load_model = TDSE_ADAPTER_CIRCUIT_RAW_LOAD_SHUNT;
import_opt.include_generator_sources = 1;

tdse_adapter_circuit_raw_options_t opt;
memset(&opt, 0, sizeof(opt));
opt.struct_size = sizeof(opt);
```

```
opt.import_options = import_opt;
opt.validation_options.struct_size =
    sizeof(tdse_adapter_circuit_raw_validation_options_t);

tdse_adapter_circuit_raw_request_t req;
memset(&req, 0, sizeof(req));
req.struct_size = sizeof(req);
req.source_kind = TDSE_ADAPTER_CIRCUIT_RAW_SOURCE_FILE;
req.output_kind = TDSE_ADAPTER_CIRCUIT_RAW_OUTPUT_FILE;
req.raw_source = "case.raw";
req.out_netlist_path = "converted.cir";
req.options = &opt;

tdse_adapter_circuit_raw_result_t result;
memset(&result, 0, sizeof(result));
result.struct_size = sizeof(result);

int rc = tdse_adapter_circuit_raw_to_netlist(&req, &result);
```

There are two output modes. When `output_kind` is `RAW_OUTPUT_FILE`, the netlist is written to `out_netlist_path`. When `output_kind` is `RAW_OUTPUT_BUFFER`, the adapter allocates the output and you release it with `tdse_adapter_circuit_raw_free_owned_output()`. Use the buffer mode when you want to pipe the netlist directly into compilation without touching disk.

The legacy function `tdse_adapter_circuit_raw_status_message()` still works and delegates to the unified status message, but new code should use `tdse_adapter_circuit_status_message()`.

For runnable examples, see `raw_import_api` (C API) and `raw_import_cpp` (C++ wrapper) in the Examples Guide.

If you want the full RAW-to-pack pipeline in a single call, use the workflow `tdse_workflow_raw_to_pack()` which chains RAW import, compilation, frequency sweep (or adaptive planning), and Builder handoff into one operation.

9.8.3 Transformer Models

Transformers in PSS/E RAW files have tap ratios and phase shifts. The model you choose determines how these are represented in the circuit netlist:

Model	Enum constant	Description
`series_only`	`RAW_TRANSFORMER_SERIES_ONLY`	Only series impedance, ignores tap ratio. Simplest
`ideal_tap_series`	`RAW_TRANSFORMER_IDEAL_TAP_SERIES`	Ideal tap in series with leakage impedance
`tap_split_shunt`	`RAW_TRANSFORMER_TAP_SPLIT_SHUNT`	Tap split into series + shunt. Most physical, more nodes

9.8.4 Load Models

Loads (constant power, constant current, constant impedance) are converted to circuit elements. The model controls the equivalent:

Model	Enum constant	Description
`shunt`	`RAW_LOAD_SHUNT`	Constant admittance to ground. Works well for most studies
`series`	`RAW_LOAD_SERIES`	Series impedance. Use when you need to preserve the branch topology
`zip_split`	`RAW_LOAD_ZIP_SPLIT`	Separates constant-Z, I, P components. Most accurate for load-flow

9.8.5 Validation Output

After conversion, the import result includes a validation summary. This tells you whether the generated netlist reproduces the original RAW case behavior:

Field	Meaning
`bus_count`	Total buses in the RAW case
`active_bus_count`	Buses with defined voltage (excludes isolated/de-energized buses)
`generator_count`	Number of generators converted to Norton sources
`branch_count`	Number of branches (lines + transformers)
`warning_count`	Number of non-fatal warnings during conversion
`netlist_required_len`	Size of the generated netlist in bytes

9.9 Core C API Tasks

The C API is what you use when integrating Adapter Circuit into a host application. It is organized around a compile-then-compute pattern: compile a netlist once to get a handle, then run as many compute operations as you want against that handle.

Most integrations only need four C API tasks:

1. compile a netlist
2. compute a matrix, series, or probe output
3. inspect diagnostics on failure or during tuning
4. destroy the handle cleanly when the work is done

All functions return a `tdse_adapter_circuit_err_t` error code. Call `tdse_adapter_circuit_status_message(code)` for a human-readable description and `tdse_adapter_circuit_get_last_error_text()` for thread-local diagnostic detail.

If you prefer C++, the header `tdse_adapter_circuit.hpp` provides a thin wrapper with RAII handle management, exception-based error handling, and convenience constructors. The underlying solver is the same — the C++ wrapper is pure syntactic convenience.

9.9.1 Init Functions

Most structs in Adapter Circuit have a corresponding `_init()` function. These functions zero-initialize the struct and set `struct_size` correctly. Always use them — they protect you from ABI issues when structs grow new fields in future SDK versions.

Init function	Returns
`tdse_adapter_circuit_options_init()`	`tdse_adapter_circuit_options_t`

Init function	Returns
`tdse_adapter_circuit_region_spec_init()`	`tdse_adapter_circuit_region_spec_t`
`tdse_adapter_circuit_time_options_init()`	`tdse_adapter_circuit_time_options_t`
`tdse_adapter_circuit_probe_options_init()`	`tdse_adapter_circuit_probe_options_t`
`tdse_adapter_circuit_ac_probe_options_init()`	`tdse_adapter_circuit_ac_probe_options_t`
`tdse_adapter_circuit_ac_sweep_init()`	`tdse_adapter_circuit_ac_sweep_t`
`tdse_adapter_circuit_node_voltage_probe_init(p, n)`	`tdse_adapter_circuit_probe_def_t`
`tdse_adapter_circuit_branch_current_probe_init (name)`	`tdse_adapter_circuit_probe_def_t`
`tdse_adapter_circuit_named_port_def_init(p, n)`	`tdse_adapter_circuit_named_port_def_t`
`tdse_adapter_circuit_compile_request_init()`	`tdse_adapter_circuit_compile_request_t`
`tdse_adapter_circuit_compile_result_init()`	`tdse_adapter_circuit_compile_result_t`
`tdse_adapter_circuit_port_fsweep_request_init()`	`tdse_adapter_circuit_port_fsweep_request_t`
`tdse_adapter_circuit_port_fsweep_result_init()`	`tdse_adapter_circuit_port_fsweep_result_t`
`tdse_adapter_circuit_port_series_request_init()`	`tdse_adapter_circuit_port_series_request_t`
`tdse_adapter_circuit_port_series_result_init()`	`tdse_adapter_circuit_port_series_result_t`
`tdse_adapter_circuit_probe_compute_request_init()`	`tdse_adapter_circuit_probe_compute_request_t`
`tdse_adapter_circuit_probe_compute_result_init()`	`tdse_adapter_circuit_probe_compute_result_t`
`tdse_adapter_circuit_diagnostics_summary_init()`	`tdse_adapter_circuit_diagnostics_summary_t`
`tdse_adapter_circuit_policy_trace_init()`	`tdse_adapter_circuit_policy_trace_t`
`tdse_adapter_circuit_prepare_region_request_init()`	`tdse_adapter_circuit_prepare_region_request_t`
`tdse_adapter_circuit_prepare_region_result_init()`	`tdse_adapter_circuit_prepare_region_result_t`
`tdse_adapter_circuit_seq_options_init()`	`tdse_adapter_circuit_seq_options_t`
`tdse_adapter_circuit_seq_network_compile_request_ init()`	`tdse_adapter_circuit_seq_network_compile_request_t`
`tdse_adapter_circuit_seq_network_compile_result_ init()`	`tdse_adapter_circuit_seq_network_compile_result_t`
`tdse_adapter_circuit_seq_fault_request_init()`	`tdse_adapter_circuit_seq_fault_request_t`
`tdse_adapter_circuit_seq_fault_result_init()`	`tdse_adapter_circuit_seq_fault_result_t`

Related helper families are grouped by header:

- `planning.h`: `_adaptive_sweep_config_init`, `_adaptive_sweep_request_init`, `_adaptive_sweep_result_init`, `_tail_vs_nfreq_request_init`, `_tail_vs_nfreq_result_init`
- `raw.h`: `_raw_request_init`, `_raw_result_init`, `_raw_options_init`
- `seq.h`: the five `_seq_*_init` functions listed above

Use this table as a lookup aid. You do not need to memorize the full list before calling the core compile and compute APIs.

9.9.1.1 Configuring Newton Solver Tolerances

`tdse_adapter_circuit_time_options_init()` also sets these appended fields (gated on `struct_size`; safe defaults for standard circuits):

Field	Default	Description
`newton_abs_tol`	`1e-8`	Absolute convergence tolerance for Newton iterations
`newton_rel_tol`	`1e-6`	Relative convergence tolerance
`newton_residual_tol`	`1e-6`	Residual norm tolerance

Field	Default	Description
`newton_max_iterations`	`32`	Maximum Newton iterations per timestep (0 = use default)

Adjust these when circuits exhibit slow convergence (try relaxing tolerances) or require higher precision (tighten tolerances).

9.9.2 Compile a Netlist

Compilation is always the first step. You provide a netlist (as a file path or as an in-memory string) and receive a handle:

```
#include <tdse_adapter_circuit.h>

tdse_adapter_circuit_options_t opt = tdse_adapter_circuit_options_init();
opt.policy = TDSE_ADAPTER_CIRCUIT_W0_EXACT_DC_THEN_FALLBACK;
opt.inductor_gbig = 1e12;

tdse_adapter_circuit_compile_request_t req =
    tdse_adapter_circuit_compile_request_init();
req.source_kind = TDSE_ADAPTER_CIRCUIT_NETLIST_SOURCE_FILE;
req.netlist_source = "my_circuit.cir";
req.options = opt;

tdse_adapter_circuit_compile_result_t result =
    tdse_adapter_circuit_compile_result_init();
int rc = tdse_adapter_circuit_compile_from_netlist(&req, &result);
// result.handle is now ready for compute calls
```

The `options` struct controls DC policy and inductor modeling. If you are using the default `EXACT_DC_THEN_FALLBACK` policy, `inductor_gbig` sets the conductance used to regularize inductors when the exact DC solve fails. The default of `1e12` works for most circuits.

9.9.3 Query a Compiled Handle

Once you have a compiled handle, you can ask it about the circuit it contains. The most common queries are node count, MNA matrix size, and node names:

```
tdse_adapter_circuit_compiled_info_t info;
memset(&info, 0, sizeof(info));
info.struct_size = sizeof(info);
tdse_adapter_circuit_get_compiled_info(result.handle, &info);
// info.node_count           — total public nodes
// info.internal_node_count  — nodes created by MNA stamping
// info.mna_matrix_size      — dimension of the MNA system

// Query a public node name by index (0 = ground when present):
size_t required_len;
```

```
tdse_adapter_circuit_get_node_name(result.handle, 0, NULL, &required_len);
char* name = malloc(required_len);
tdse_adapter_circuit_get_node_name(result.handle, 0, name, &required_len);
```

9.9.4 Options

`tdse_adapter_circuit_options_init()` provides safe defaults. The fields you are most likely to adjust:

Field	Default	Meaning
`policy`	`W0_EXACT_DC_THEN_FALLBACK`	How DC (zero frequency) is handled
`inductor_gbig`	`1e12`	Conductance used to regularize inductors at DC
`dc_extrapolate_points`	`4`	Number of positive-frequency points used when extrapolating DC

9.9.5 Compute a Y Matrix

This is the most common operation. Given a compiled handle and a list of ports, compute the admittance matrix at every frequency in a grid:

```
tdse_adapter_circuit_port_def_t ports[] = {{1, 0}};
tdse_adapter_circuit_grid_t grid = {.w0 = 0.0, .dw = 100.0, .nfreq = 5};

tdse_adapter_circuit_port_fsweep_request_t freq_req =
    tdse_adapter_circuit_port_fsweep_request_init();
freq_req.handle = result.handle;
freq_req.matrix_kind = TDSE_ADAPTER_CIRCUIT_MATRIX_Y;
freq_req.ports = ports;
freq_req.port_count = 1;
freq_req.grid = grid;

// Output buffer: 2 doubles per matrix element (real + imag),
//                nfreq frequency points,
//                port_count rows × port_count columns
size_t n_vals = 2 * grid.nfreq * port_count * port_count;
double* matrix_ri = malloc(n_vals * sizeof(double));
freq_req.out_matrix_ri = matrix_ri;
freq_req.out_matrix_ri_len = n_vals;

tdse_adapter_circuit_port_fsweep_result_t freq_result =
    tdse_adapter_circuit_port_fsweep_result_init();
int rc = tdse_adapter_circuit_compute_port_fsweep(&freq_req, &freq_result);
```

The output is stored in interleaved real-imaginary format, row-major by port index. For a 2-port Y matrix, element `Y[p][q]` at frequency `k` lives at:

```
matrix_ri[2 * (k * nports * nports + p * nports + q)]      = real(Y_pq[k])
matrix_ri[2 * (k * nports * nports + p * nports + q) + 1]  = imag(Y_pq[k])
```

9.9.6 Compute Port Series (VOC / ISC)

For time-domain excitation, you need waveforms not matrices. `compute_port_series` produces open-circuit voltage or short-circuit current versus time:

```
tdse_adapter_circuit_port_series_request_t series_req =
    tdse_adapter_circuit_port_series_request_init();
series_req.handle = result.handle;
series_req.response_kind = TDSE_ADAPTER_CIRCUIT_PORT_RESPONSE_VOC;
series_req.ports = ports;
series_req.port_count = 1;
series_req.dt = 1e-4;
series_req.steps = 64;

double* series_out = malloc(series_req.steps * port_count * sizeof(double));
series_req.out_values = series_out;
series_req.out_values_len = series_req.steps * port_count;

tdse_adapter_circuit_port_series_result_t series_result =
    tdse_adapter_circuit_port_series_result_init();
int rc = tdse_adapter_circuit_compute_port_series(&series_req, &series_result);
```

The output is step-major: `series_out[step * nports + port]` gives the value at that time step for that port.

9.9.7 Compute Probes

Probes observe internal circuit quantities without extracting full port matrices. Define what you want to measure, then call `compute_probes`:

```
tdse_adapter_circuit_probe_def_t probe =
    tdse_adapter_circuit_node_voltage_probe_init(1, 0);
// or: tdse_adapter_circuit_branch_current_probe_init("r1");

tdse_adapter_circuit_probe_compute_request_t probe_req =
    tdse_adapter_circuit_probe_compute_request_init();
probe_req.handle = result.handle;
probe_req.domain = TDSE_ADAPTER_CIRCUIT_PROBE_DOMAIN_AC;
probe_req.probes = &probe;
probe_req.probe_count = 1;
// For AC: set ac_sweep and ac_probe_options
// For transient: set dt, steps, and time_options

tdse_adapter_circuit_probe_compute_result_t probe_result =
    tdse_adapter_circuit_probe_compute_result_init();
int rc = tdse_adapter_circuit_compute_probes(&probe_req, &probe_result);
```

Each probe domain has different required parameters. To prevent mistakes — like passing transient time-step fields to an AC probe — two convenience wrappers expose only the relevant fields:

```
// AC probes — only frequency-sweep parameters are accepted
tdse_adapter_circuit_compute_ac_probes(
    handle, probes, probe_count, ac_sweep, ac_probe_options, &result);

// Transient probes — only time-domain parameters are accepted
tdse_adapter_circuit_compute_transient_probes(
    handle, probes, probe_count, dt, steps, time_options, &result);
```

Use these wrappers in new code. They make domain-mismatch mistakes a compile-time error instead of a runtime surprise.

9.9.8 Prepare a Circuit Region

Some netlists define named regions — groups of ports treated as a unit. Region preparation resolves a region name into an explicit port list and creates a prepared handle for it:

```
tdse_adapter_circuit_region_spec_t region =
    tdse_adapter_circuit_region_spec_init();
region.name = "main";

tdse_adapter_circuit_prepare_region_request_t prep_req =
    tdse_adapter_circuit_prepare_region_request_init();
prep_req.source_handle = result.handle;
prep_req.region = &region;

tdse_adapter_circuit_prepare_region_result_t prep_result =
    tdse_adapter_circuit_prepare_region_result_init();
int rc = tdse_adapter_circuit_prepare_region(&prep_req, &prep_result);
// prep_result.resolved_port_count tells how many ports were resolved
```

This is most useful when your netlist defines regions with `TDSE` directives and you want the adapter to enumerate the ports automatically instead of listing them by hand.

9.9.9 Named Ports

When you know node names but not their numeric indices, use named ports. The adapter resolves string names to indices during the compute call:

```
tdse_adapter_circuit_named_port_def_t ports[] = {
    tdse_adapter_circuit_named_port_def_init("1", "0"),
    tdse_adapter_circuit_named_port_def_init("OUT", "GND"),
};
freq_req.named_ports = ports;
```

```
freq_req.port_count = 2;
// Leave freq_req.ports = NULL — the adapter resolves names to indices.
```

Named ports work everywhere numeric ports work: `port_fsweep_request_t`, `port_series_request_t`, and adaptive sweep / tail analysis requests. You can even mix them — pass `named_ports` for resolution and leave `ports` NULL.

9.9.10 Handle Reuse and Thread Safety

A compiled handle is designed to be reused. Compile once, then run as many compute calls as you need:

```
tdse_adapter_circuit_compile_from_netlist(&req, &result);

for (int k = 0; k < num_sweeps; ++k) {
    freq_req.handle = result.handle;
    freq_req.grid = grids[k];   // different grid each iteration
    tdse_adapter_circuit_compute_port_fsweep(&freq_req, &freq_result);
}

tdse_adapter_circuit_destroy(result.handle);
```

Thread safety: A handle is not safe for concurrent use from multiple threads. Each thread that needs to compute against a circuit should compile its own handle. The rule is one handle per execution thread — do not enter any compute or lifecycle API on the same handle from two threads simultaneously. This mirrors the Runtime contract and keeps the internal solver state simple and fast.

9.9.11 Progress and Cancellation

Long-running time-domain operations — transient probes, port series with many steps — can take seconds or minutes. You can receive progress updates and cancel in-flight computations:

```
tdse_adapter_circuit_time_options_t time_opt =
    tdse_adapter_circuit_time_options_init();

// Progress callback — called periodically during the solve
time_opt.progress.enable = 1;
time_opt.progress.interval_sec = 0.5;  // callback at most every 0.5 seconds
time_opt.progress.callback = my_progress_callback;
time_opt.progress.user_ptr = &my_state;

// Cancel callback — polled by the solver;
// return non-zero to request cancellation
time_opt.cancel.callback = my_cancel_callback;
time_opt.cancel.user_ptr = &my_state;
```

```
probe_req.time_options = &time_opt;
```

The progress event (`tdse_adapter_circuit_progress_event_t`) carries the current simulation time, accepted and rejected step counts, nonlinear iteration counts, and the transient integration scheme in use. The `progress` field is a number between 0 and 1 indicating overall completion.

Note: progress and cancellation are currently supported for time-domain operations only. Frequency-domain operations (`matrix`, AC probes) complete quickly enough that they do not expose progress callbacks.

9.9.12 Error Handling

All functions return a `tdse_adapter_circuit_err_t`. Check it against `TDSE_ADAPTER_CIRCUIT_OK`. For error messages, use `tdse_adapter_circuit_status_message()` — it handles adapter, RAW import, and sequence-network error codes:

```
if (tdse_adapter_circuit_handle_is_valid(handle)) {
    int rc = tdse_adapter_circuit_compute_port_fsweep(&req, &result);
    if (rc != TDSE_ADAPTER_CIRCUIT_OK) {
        fprintf(stderr, "fsweep failed: %s\n",
                tdse_adapter_circuit_status_message(rc));
        const char* detail = tdse_adapter_circuit_get_last_error_text();
        if (detail && detail[0]) {
            fprintf(stderr, "detail: %s\n", detail);
        }
    }
}
```

Always check the return code. A non-zero code means the output buffers were not filled and the result struct's diagnostics may contain partial or stale data.

9.9.13 Diagnostics

Every result struct includes a `diagnostics` field. Three query functions extract structured information from it:

```
// Human-readable summary of what happened
tdse_adapter_circuit_diagnostics_summary_t summary =
    tdse_adapter_circuit_diagnostics_summary_init();
tdse_adapter_circuit_diagnostics_get_summary(&result.diagnostics, &summary);
// summary.call_kind, summary.matrix_size, summary.timing_total_ns, ...

// Solver statistics — backend used, matrix properties, conditioning
tdse_adapter_circuit_solver_diagnostics_t solver =
    tdse_adapter_circuit_solver_diagnostics_init();
tdse_adapter_circuit_diagnostics_get_solver_stats(&result.diagnostics, &solver);
// solver.solver_backend, solver.matrix_density, solver.singular, ...
```

```
// Policy trace — how the solver backend was selected for this call
tdse_adapter_circuit_policy_trace_t trace =
    tdse_adapter_circuit_policy_trace_init();
tdse_adapter_circuit_diagnostics_get_policy_trace(&result.diagnostics, &trace);
// trace.solver_policy_source, trace.solver_backend,
// trace.solver_policy_override_used, ...
```

Use diagnostics during development and for production monitoring. The summary tells you whether things worked. The solver stats tell you why a particular backend was chosen and how the matrix behaved. The policy trace is essential when debugging "why did it use this solver instead of that one?"

9.10 Solver Policy

When you run a frequency sweep, the SDK selects a solver backend for you — CPU dense, CPU sparse (KLU), or CUDA. The default policy picks the best available backend for your circuit size and structure, and decides whether to parallelize the sweep across frequency points.

For most users, the defaults are correct and you can skip this section. You only need to touch solver policy when:

- You want to force a specific backend (e.g., always use CUDA)
- You want to disable parallel sweeps for deterministic ordering
- You are benchmarking or debugging solver selection

```
// Query the current default policy
tdse_adapter_circuit_solver_policy_t policy =
    tdse_adapter_circuit_get_solver_policy();

// Set process-wide defaults
policy.sweep_parallel_mode = TDSE_ADAPTER_CIRCUIT_SWEEP_PARALLEL_FORCE;
policy.sweep_parallel_max_workers = 4;
tdse_adapter_circuit_set_solver_policy(&policy);

// Or override per-request (takes precedence over process-wide defaults)
req.solver_policy_override = &custom_policy;
```

Key policy fields:

Field	Default	Meaning
`sweep_parallel_mode`	`AUTO`	`AUTO` (SDK decides), `FORCE` (always), `DISABLE` (never)
`sweep_parallel_max_workers`	—	Maximum number of worker threads for parallel sweeps
`sweep_parallel_min_points`	—	Minimum frequency points to trigger parallel sweep
`sweep_parallel_chunk_size`	—	Points per chunk in dynamic scheduling
`sweep_parallel_allow_cuda`	`0`	Whether the CUDA backend may be selected

For a multi-threaded example using solver policy, see `adapter_cpp_threaded_lanes` in the Examples Guide.

9.11 Adaptive Sweep Planning

Choosing a frequency grid is one of the hardest parts of frequency-domain modeling. Too few points and you miss resonant features. Too many and you waste computation, or worse, produce an over-resolved spectrum that creates fitting artifacts in Builder.

The adaptive sweep planner automates this. You give it a circuit and a target time step, and it runs planning sweeps to determine the required frequency range and resolution. It evaluates impulse response convergence and recommends production parameters: `nfreq`, `nh`, and `dt`.

```
#include <tdse_adapter_circuit/planning.h>

tdse_adapter_circuit_adaptive_sweep_config_t plan_cfg =
    tdse_adapter_circuit_adaptive_sweep_config_init();
plan_cfg.tolerance_mode = TDSE_ADAPTER_CIRCUIT_ADAPTIVE_SWEEP_TOLERANCE_MEDIUM;
plan_cfg.dt_target = 1e-4;
plan_cfg.limit_f_max_hz = 1e4;

// Create a report handle to capture detailed planning output
tdse_adapter_circuit_adaptive_sweep_report_t* report = NULL;
tdse_adapter_circuit_adaptive_sweep_report_create(&report);

tdse_adapter_circuit_adaptive_sweep_request_t plan_req =
    tdse_adapter_circuit_adaptive_sweep_request_init();
plan_req.handle = result.handle;
plan_req.ports = ports;
plan_req.port_count = port_count;
plan_req.matrix_kind = TDSE_ADAPTER_CIRCUIT_MATRIX_Y;
plan_req.correction_method = TDSE_BUILDER_CORRECTION_RECONSTRUCT_FROM_REAL;
plan_req.adapter_options = opt;
plan_req.planning_config = &plan_cfg;
plan_req.out_report = report;

tdse_adapter_circuit_adaptive_sweep_result_t plan_result =
    tdse_adapter_circuit_adaptive_sweep_result_init();
int rc = tdse_adapter_circuit_plan_adaptive_sweep(&plan_req, &plan_result);
// plan_result.summary.production_dt_s  — recommended time step
// plan_result.summary.production_nh   — recommended history length
// plan_result.summary.production_nfreq — recommended frequency count
```

The report handle gives you detailed justification for each recommendation:

```
tdse_adapter_circuit_adaptive_sweep_report_summary_t report_summary;
memset(&report_summary, 0, sizeof(report_summary));
report_summary.struct_size = sizeof(report_summary);
```

```
tdse_adapter_circuit_adaptive_sweep_report_get_summary(report, &report_summary);

// Export the full report as JSON or text for archival
size_t required;
tdse_adapter_circuit_adaptive_sweep_report_json(
    report, buf, buf_size, &required);
tdse_adapter_circuit_adaptive_sweep_report_text(
    report, buf, buf_size, &required);

tdse_adapter_circuit_adaptive_sweep_report_destroy(report);
```

The tolerance mode (`LOW`, `MEDIUM`, `HIGH`) controls how aggressively the planner refines the grid. `MEDIUM` is the right starting point. Use `HIGH` for production runs where accuracy matters more than speed.

9.12 Tail vs. Nfreq Analysis

When preparing for Builder handoff, you need to choose `nh` — the number of history samples in the discrete-time impulse response. Too small and the tail is truncated, introducing error. Too large and you waste memory and computation in Runtime.

The tail-vs-nfreq analysis helps. It scans the impulse response as `nfreq` increases, tracking how the tail magnitude decays. When the tail settles below a threshold, you have found your `nh`.

```
tdse_adapter_circuit_tail_vs_nfreq_request_t tail_req =
    tdse_adapter_circuit_tail_vs_nfreq_request_init();
tail_req.handle = result.handle;
tail_req.ports = ports;
tail_req.port_count = port_count;
tail_req.adapter_options = opt;
// The tail-vs-nfreq scan evaluates both Y and Z internally and uses
// RECONSTRUCT_FROM_REAL, so no matrix_kind/correction_method fields exist
// on this request struct.

tdse_adapter_circuit_tail_vs_nfreq_result_t tail_result =
    tdse_adapter_circuit_tail_vs_nfreq_result_init();
int rc = tdse_adapter_circuit_scan_tail_vs_nfreq(&tail_req, &tail_result);
// tail_result.summary.recommended_nh  — suggested history length
// tail_result.summary.tail_settled   — whether the tail converged
```

As with adaptive sweep planning, you can create a report handle (`tdse_adapter_circuit_tail_vs_nfreq_report_t`) for per-point detail and JSON/text export.

9.13 NPORT

NPORT is a SPICE extension that imports frequency-dependent multiport data from Touchstone files directly into a circuit solve. If you have measured or simulated S-parameters, Y-parameters, or Z-parameters in a Touchstone file, you can reference it from a netlist without manually converting formats:

```
NPORT NP1 1 0 2 0 FILE=example.y2p TYPE=Y
```

When the adapter compiles a netlist containing NPORT elements, it reads the referenced Touchstone file, interpolates the data onto the solver's frequency grid, and stamps a frequency-dependent admittance block into the MNA matrix. All of this happens at compile time — no extra steps for you.

Supported Touchstone formats include `.ynp`, `.znp`, `.y2p`, `.z2p`, `.s2p`, and their multi-port variants. The file must be readable relative to the working directory at compile time.

NPORT elements work transparently with all Adapter Circuit commands: `matrix`, `series`, and `probe`. For a runnable example, see `nport_y2p_ac` in the Examples Guide.

9.14 Integration Patterns

9.14.1 Builder Handoff

This is the most common end-to-end pattern. Adapter Circuit produces a Y matrix, and Builder consumes it to produce a runtime pack:

```c
// 1. Compute Y from adapter
tdse_adapter_circuit_compute_port_fsweep(&freq_req, &freq_result);

// 2. Feed into Builder
tdse_builder_cplx_mat_view_t spec;
spec.ri = y_matrix_data;
spec.nfreq = nfreq;
spec.nq = np; spec.np = np;
spec.ld = np;
spec.stride_freq = np * np;

tdse_builder_h_from_spectrum(omega, spec, dt, nh,
    TDSE_BUILDER_CORRECTION_RECONSTRUCT_FROM_REAL,
    0.0, NULL, h_data, h_data_len);

// 3. Build pack
tdse_builder_configure_ex(builder, &bopt);
tdse_builder_apply_h(builder, &h_desc);
tdse_builder_write_pack(builder, "model.pack");
```

A few convenience helpers make this pattern even shorter:

```c
// Create a frequency grid in Hz instead of rad/s
tdse_adapter_circuit_grid_t g = tdse_adapter_circuit_grid_from_hz(0, 50, 101);

// 1-port shorthand
tdse_adapter_circuit_port_def_t port = TDSE_ADAPTER_CIRCUIT_PORT_1P(1, 0);

// Convert adapter output shape to Builder input view in one call
tdse_builder_cplx_mat_view_t spec =
    tdse_adapter_circuit_to_builder_view(&result.shape, matrix_ri);
```

Critical invariant: The port order in the adapter's `ports` array must match the port order in Builder's `nq` and `np` dimensions. If adapter computes Y with ports in order `[A, B, C]` and Builder expects `[C, A, B]`, the resulting pack will be wrong and the error may not be obvious until runtime. Archive the frequency grid parameters and port ordering alongside every output.

9.14.2 Qualification Checks Before You Ship The Pack

Do not stop at "Builder wrote a file." The pack is ready for handoff only when the conversion path has also cleared the quality checks that matter for your integration.

Use this checklist:

Check	Why it matters	Best place to read it
Port order and matrix family are intentional	wrong ordering can produce a numerically valid but physically wrong pack	Adapter request + Builder handoff record
Passivity result is acceptable for the exported spectrum	catches unstable or non-physical source data before Runtime	workflow result or `tdse_nport_check_passivity(...)`
Round-trip frequency-response error is acceptable	confirms the written pack still reproduces the source spectrum closely enough	workflow result or `tdse_model_verify_frequency_response(...)`
Adaptive planning or tail scan produced an acceptable `nfreq` / tail budget	avoids shipping a pack that is too short, too coarse, or too aggressively truncated	`tail_scan`, workflow result, planning reports
Runtime can create the pack cleanly on the target host	catches compatibility surprises before the pack leaves engineering	`tdse_pack_validate`, `tdse_model_create(...)`, create diagnostics

In practice there are two good qualification styles:

1. **Workflow-first qualification** Use the workflow result as the primary artifact. It already carries passivity and round-trip fields and records whether the SDK auto-derived the grid or sweep plan.
2. **Manual handoff qualification** Archive the matrix family, port list, frequency grid, Builder settings, pack validation output, and one Runtime create proof on the same host profile where the pack will be consumed.

For most teams, the quality bar is:

- the source circuit is understood
- the chosen representation (`Y` or `Z`) is intentional
- passivity and round-trip metrics are within project limits
- Runtime creates the pack without compatibility or diagnostics surprises

That is a much stronger release signal than “the CSV looked reasonable.”

9.14.3 Embedding in a Host Solver (MNA)

When embedding TDSE inside a larger circuit simulator, the Runtime step loop integrates with the host’s Modified Nodal Analysis formulation:

```
tdse_step_begin(model, t, dt);
tdse_step_op(model, &op);     // op.G = conductance matrix G_tdse
tdse_step_hr(model, hr);      // hr   = history current source

// Host assembles the full system:
//   (G_host + G_tdse) * v = i_src - hr
// ...solve for v...

tdse_step_commit(model, v);   // advance internal state
```

This pattern lets you treat the TDSE model as a black-box Norton equivalent that stamps into the host’s MNA matrix. The host owns the global solve; TDSE provides the conductance stamp and the history current source each time step.

For a runnable example, see `mna_block_embed_2p` in the Examples Guide.

9.15 Deep Reference: Adapter Lookup

Use this section as a lookup area after the main task sections above. It is not the best place to start if you are still deciding which adapter operation you need.

9.15.1 Parameter Cookbook

This table is a quick lookup for every CLI flag and its purpose. Bookmark it.

Parameter	Where	Notes
`--netlist-kind file|text|stdin`	all commands	`file` for scripts, `stdin` for pipes, `text` for inline
`--matrix y|z`	matrix	Y = admittance (Norton form), Z = impedance (Thevenin form)
`--ports "p,n;p,n"`	matrix, series	Order matters — it must match Builder input order
`--w0-radps, --dw-radps, --nfreq`	matrix	Archive these with your output; you will need them later
`--dc-policy`	matrix	Affects DC stability: see DC policy values
`--series voc|isc`	series	`voc` = open-circuit voltage, `isc` = short-circuit current
`--method transient|ifft|tone`	series, transient probe	`transient` = general; `ifft` = freq; `tone` = steady-state
`--dt, --steps`	series, transient probe	Must match Builder’s `dt` for consistent handoff
`--nfft`	series, probe	FFT size; must be ≥ `2 * (nfreq - 1)` for ifft method
`--observe "v(1);i(r1)"`	probe	Semicolon-separated probe expressions

Parameter	Where	Notes
`--observe-source`	probe	`argument` = CLI; `netlist` = `.probe` directives; `argument_or_netlist` = auto
`--sweep-kind`	AC probe	`from_netlist`, `lin`, `dec`, `oct`, `list`
`--ac-excitation`	AC probe	`small_signal` for new designs; `legacy_tones` for backward compatibility
`--json-out -`	all commands	Always use `-` (stdout) in automation

9.15.2 Enum Reference

Enum type	Values
`tdse_adapter_circuit_matrix_kind_t`	`MATRIX_Y`, `MATRIX_Z`
`tdse_adapter_circuit_port_response_kind_t`	`PORT_RESPONSE_VOC`, `PORT_RESPONSE_ISC`
`tdse_adapter_circuit_probe_domain_t`	`PROBE_DOMAIN_AC`, `PROBE_DOMAIN_TRANSIENT`
`tdse_adapter_circuit_netlist_source_t`	`NETLIST_SOURCE_TEXT`, `NETLIST_SOURCE_FILE`
`tdse_adapter_circuit_raw_source_t`	`RAW_SOURCE_TEXT`, `RAW_SOURCE_FILE`
`tdse_adapter_circuit_raw_output_kind_t`	`RAW_OUTPUT_BUFFER`, `RAW_OUTPUT_FILE`
`tdse_adapter_circuit_raw_unit_mode_t`	`RAW_UNIT_PU`, `RAW_UNIT_SI`
`tdse_adapter_circuit_sweep_parallel_mode_t`	`SWEEP_PARALLEL_AUTO`, `_FORCE`, `_DISABLE`
`tdse_builder_correction_method_t`	`CORRECTION_NONE`, `_RECONSTRUCT_FROM_REAL|_IMAG|_MAG|_PHASE`

9.15.3 Failure Modes

When something goes wrong, start here:

Symptom	Likely cause	Fix
Parse failure	Malformed netlist, missing include	Run with `--json-out -` on smallest deck
Ports resolve incorrectly	Wrong node names or polarity	Reduce to one port and verify the node pair
Singular or ill-conditioned solve	Floating nodes, invalid topology	Try `extrapolate_from_positive` DC policy or single-freq Y
VOC/ISC differs from expectation	Method mismatch, insufficient `nfft`	Compare `transient` vs `ifft`
NPORT import fails	Type mismatch, bad path, non-monotonic data	Check TYPE, dimensions, CWD for relative paths
RAW conversion fails “invalid argument”	Missing `struct_size` fields	Set all struct_size; check `output_kind` matches method
All backends show “no” in caps	Build configuration issue	Rebuild with required dependencies (KLU, CUDA, etc.)

9.15.4 Troubleshooting

When a command fails, collect these four things before asking for help:

1. The exact command line you ran
2. The full `--json-out -` output
3. The smallest netlist that reproduces the problem
4. `tdse adapter circuit caps` output from the same machine

To narrow down the problem yourself: verify the netlist parses → verify one port → verify one frequency → add complexity. Almost all circuit bugs become obvious when you reduce to the simplest failing case. For Runtime-side issues after pack creation, continue in Troubleshooting.

9.15.5 Validation Checklist

Use this before handing off results to Builder or committing output to a project:

☐ `tdse adapter circuit caps` shows expected backends
☐ Minimal `matrix` replay passes on a known-good deck
☐ Output dimensions match expectations (ports × frequency points)
☐ Builder handoff produces a valid `.pack` file
☐ Port order is consistent throughout the pipeline (adapter → Builder → Runtime)

9.16 Advanced Tuning And Performance

This section is for tuning and scale work after the basic adapter flow is already correct. It covers the key knobs that affect Adapter Circuit throughput and when it is worth touching them.

9.16.1 Choosing a Solver Backend

For most circuits the auto-selector picks the right backend. Here is when to override it:

Backend	Best for
DENSE_LU	Small circuits (n < 200). Lowest overhead per solve point
SPARSE_KLU	Medium-to-large circuits (n > 200, density < 0.18). Much faster factorisation for sparse matrices
DENSE_CUDA	Large circuits (n > 256) when a CUDA GPU is available. Best for batched frequency sweeps
SPARSE_CUDA	Very large sparse circuits (n > 20000). Only available with CUDA GPU
SPARSE_MKL_PARDISO	Large sparse circuits when Intel MKL is available. Alternative to KLU

9.16.2 Parallel Frequency Sweeps

Frequency sweeps are embarrassingly parallel — each frequency point is an independent linear solve. Enable parallel sweeps to use all CPU cores:

```
tdse adapter circuit matrix ... --sweep-parallel auto
```

Or via C API:

```
policy.sweep_parallel_mode = TDSE_ADAPTER_CIRCUIT_SWEEP_PARALLEL_AUTO;
policy.sweep_parallel_max_workers = 0;  // use all hardware threads
tdse_adapter_circuit_set_solver_policy(&policy);
```

Parallel sweeps are most effective when:

- **nfreq ≥ 8**: Overhead of thread creation amortized over many points

- **n > 100**: Each solve point is expensive enough to justify threading
- **CUDA backend**: The CUDA solver can batch multiple frequency points into a single GPU kernel launch (see Dense CUDA Batched Fast Path below)

When NOT to use parallel sweeps:

- Debugging: serial execution gives deterministic ordering
- Very small circuits (n < 50): thread overhead exceeds benefit
- Already running many independent adapter calls in parallel (nesting overhead)

9.16.3 Dense CUDA Batched Fast Path

When the CUDA dense backend is selected and the following conditions are met, frequency sweeps take a highly optimised batched path:

- Affine AC build template is valid (enabled by default for nfreq ≥ 4)
- Frequency grid starts at w0 > 0 (non-DC)
- nfreq ≥ 2
- The selected backend is DENSE_CUDA
- `dense_cuda_batched_fsweep_fast_path_enabled()` returns true (default)

On this path, the solver batches all frequency points into one GPU kernel, dramatically reducing kernel launch overhead. For large circuits (n > 1000) with many frequency points (nfreq > 64), the speedup can be 5-20× versus per-point GPU solves.

The fast path is controlled by the environment variable:

```
TDSE_ADAPTER_DENSE_CUDA_BATCHED_FSWEEP_FAST_PATH=1   # enable (default)
TDSE_ADAPTER_DENSE_CUDA_BATCHED_FSWEEP_FAST_PATH=0   # disable
```

9.16.4 DC Policy Performance

The DC policy choice affects the number of factorizations at ω=0:

Policy	Factorizations	Notes
`exact_dc_then_fallback`	1-2	Tries exact first. Two factorizations only when singular
`regularized_exact_dc`	1	Always regularises with `inductor_gbig`. Fastest correct path
`extrapolate_from_positive`	0	Skips DC entirely. Fastest but least accurate at low frequencies

9.16.5 Sparse Solver Factor Caching

KLU and SPARSE_CUDA solvers cache the symbolic factorization pattern. When computing multiple sweeps against the same circuit with different frequency grids, the symbolic analysis is reused. This is automatic — no configuration needed. The diagnostics field `factor_cache_enabled` and `factor_cache_hit` report whether caching was used.

9.16.6 Affine AC Build

When enabled, the adapter builds an affine template of the MNA system at the first frequency point. Subsequent frequency points materialise the template with O(nnz) work instead of re-building the full system from scratch. This is enabled by default for nfreq ≥ 4 and provides approximately 2-3x speedup for large circuits.

9.16.7 Compile Once, Sweep Many

A compiled handle is reusable across multiple compute calls. Always compile once and reuse the handle for multiple sweeps, rather than recompiling:

```
// Good — compile once
tdse_adapter_circuit_compile_from_netlist(&req, &result);
for (int k = 0; k < num_configs; ++k) {
    freq_req.handle = result.handle;
    freq_req.grid = grids[k];
    tdse_adapter_circuit_compute_port_fsweep(&freq_req, &freq_result);
}

// Bad — recompiles every iteration
for (int k = 0; k < num_configs; ++k) {
    tdse_adapter_circuit_compile_from_netlist(&req, &result);
    freq_req.handle = result.handle;
    tdse_adapter_circuit_compute_port_fsweep(&freq_req, &freq_result);
    tdse_adapter_circuit_destroy(result.handle);
}
```

Compilation involves parsing, MNA construction, and topology analysis — it is typically 10-100× more expensive than a single frequency-point solve.

CHAPTER 10

RAW Import

Use this chapter when your starting artifact is a PSS/E RAW case and your first question is not "how does the adapter work internally?" but "how do I get from this file to a usable netlist or pack?"

RAW import lives inside Adapter Circuit. Keep this page as the quick bridge: enough to get oriented, then move into the detailed chapter only when you need the full conversion behavior or API details.

Treat this chapter as the RAW-specific branch of Adapter Circuit, not as a standalone main path. Most readers should understand the overall Adapter Circuit workflow first and then return here only if RAW conversion is their real starting artifact.

10.1 Start Here

Use this table to pick the shortest workable route:

If you need to...	Start here	Then continue with
convert a RAW file into a circuit netlist	`tdse adapter circuit raw-to-netlist`	CLI Reference
convert RAW and then build a pack	Adapter Circuit	Builder and Data Contracts
convert RAW directly into a pack with one public workflow	Adapter Circuit	Examples Guide
call RAW import from C or C++	Adapter Circuit	Examples Guide
validate why RAW conversion failed	Troubleshooting	CLI Reference

If your real goal is a qualified `.pack` and you do not need to inspect the intermediate netlist, prefer the workflow path first. Drop to `raw-to-netlist` when you need to review the converted artifact, archive import reports, or isolate a mismatch before Builder.

10.2 Shortest CLI Path

For most users, the first successful path is:

```
tdse adapter circuit raw-to-netlist \
  --raw-kind file --raw case.raw \
  --out-netlist ./case_from_raw.cir \
```

```
    --out-report-json ./raw_report.json \
    --json-out -
```

What you should expect from this command:

- a circuit-compatible netlist file
- a JSON report with conversion counts and validation detail
- a CLI JSON envelope that scripts can archive

If that command succeeds, the usual next step is to run `tdse adapter circuit matrix` on the generated netlist and then hand the result to Builder.

10.3 What RAW Import Covers

RAW import is responsible for translating grid-domain source data into circuit-domain artifacts that the rest of Adapter Circuit can process. In practice that usually means:

- buses, branches, loads, generators, and transformers are mapped into a netlist form
- base-frequency and compatibility checks are recorded in the import report
- the generated netlist becomes the new handoff point for matrix, series, probe, and Builder workflows

The detailed conversion behavior, supported model notes, and API-level usage live in Adapter Circuit. Keep this chapter as the quick bridge, not the long reference.

10.4 Support Boundary And Validation Expectations

RAW import is broad, but it is still a conversion path, not a promise of perfect one-to-one replay of every original study environment.

Use this table to decide how much validation you need after import:

Area	What to expect	What to validate next
buses and branch topology	usually direct structural mapping into a circuit netlist	node naming, connectivity, and port selection
generators and source injections	converted into circuit-domain equivalents used by Adapter	source polarity, port response, and expected drive conditions
transformers	behavior depends on the selected transformer model	tap behavior, shunt/series interpretation, and DC/low-frequency response
loads	behavior depends on the chosen load model (`shunt`, `series`, `zip_split`)	whether the chosen approximation matches the study intent
base-frequency semantics	validated and recorded in the import report	that the report matches your expected system base
unsupported or unusual records	may appear as warnings, omitted features, or validation findings	import report first, then matrix/probe comparison before Builder

Practical rule: treat the generated netlist as a new engineering handoff point. Do not send it straight to production Builder flows until one of the following is true:

- a matrix or probe run looks physically right
- the conversion report shows only expected approximations

- the resulting pack clears the normal qualification checks from Adapter Circuit

10.5 Recommended Next Step

Once RAW conversion works:

1. inspect the generated report and netlist when you need conversion visibility
2. if your goal is a short supported pack path, move to the workflow API; otherwise run `matrix` on the converted netlist
3. move to production Builder or pack handoff only after the adapter-side output or workflow qualification looks correct

CHAPTER 11

CLI Reference

Use this chapter when you already know you need the `tdse` command-line tool and want the shortest path to the right command, the stable output fields for automation, or the exact syntax for a specific verb.

11.1 How To Use This Chapter

Most readers use this chapter in one of two ways:

- to confirm which command family to use: `sdk`, `adapter circuit`, or `profiler`
- to look up exact flags, output files, and machine-readable fields

If you are still getting your first model running, start with Getting Started and come back here when you need exact command syntax.

11.2 Before You Start

- A successful build that produces the `tdse` executable.
- A writable output directory for `--out-*` targets.
- Representative input data for the path you want to exercise.

```
cmake --build build --target tdse
```

11.3 First Confidence Check

> For the general SDK installation and first-run path, see Installation.

Use these three commands to confirm that the CLI routes correctly, that Adapter Circuit is available, and that the profiler can emit a report:

```
tdse sdk version --json-out -
tdse adapter circuit caps --out-caps /tmp/caps.json --json-out -
tdse profiler quick --out-json ./tdse_profile_report.json --json-out -
```

Expected output sample:

```
tdse sdk version: success
tdse adapter circuit caps: success
tdse profiler quick: success
```

11.4 Choose A Command Family

Use this table before dropping into the full reference:

If you need to...	Use...	Primary outputs
confirm binary and SDK identity	`tdse sdk version`	version metadata JSON
inspect or convert circuit-domain input	`tdse adapter circuit ...`	netlists, matrix CSV, probe CSV, JSON envelopes
benchmark runtime behavior or derive a runtime plan	`tdse profiler ...`	profiler report JSON, summary Markdown, matrix CSV

Typical starting points:

- PSS/E RAW case -> `adapter circuit raw-to-netlist`
- circuit netlist -> `adapter circuit matrix`, `series`, or `probe`
- backend or route uncertainty -> `profiler calibrate`, `sweep`, `validate`, `explain`, `diff`

11.5 Most Common Workflows

If you want the shortest path from intent to command, start here:

Goal	First command	Then usually do this
prove the binary and installation are sane	`tdse sdk version --json-out -`	move to Getting Started
convert a RAW case	`tdse adapter circuit raw-to-netlist`	run `matrix` on the generated netlist
generate Builder-facing matrix data	`tdse adapter circuit matrix`	move to Builder and Data Contracts
inspect source-side waveforms	`tdse adapter circuit series`	compare against expected circuit behavior
inspect internal nodes or branches	`tdse adapter circuit probe`	use Troubleshooting if values look wrong
capture a baseline performance report	`tdse profiler quick`	follow with `validate`, `explain`, `diff`, or `tables`

Use Adapter Circuit when you need the circuit-domain meaning behind a command. Use this chapter when you need exact flags, outputs, and stable fields.

11.6 Common Automation Rules

These rules make CLI usage much more predictable in scripts and CI:

- always set `--json-out` for machine parsing
- archive the exact input identity for adapter and profiler runs
- keep port ordering stable across repeated matrix and series generation

- archive the exact frequency grid when generating Builder-facing matrix data
- for profiler runs, prefer emitting JSON, Markdown, and CSV together

Parameter quick reference:

Parameter	Where it appears	Practical rule	Why it matters
`--json-out`	most public commands	always set in automation	gives stable JSON output that scripts can parse
`--out-data`	adapter matrix/series/probe	point to a writable file or `-`	captures primary numeric output
`--ports`	adapter matrix/series/probe	keep ordering stable across replays	output shape depends on port order
`--w0-radps`, `--dw-radps`, `--nfreq`	adapter matrix	archive the exact grid	frequency sweep identity must be reproducible
`--out-json`, `--out-md`, `--out-csv`	profiler commands	emit all three in release records	JSON is best for automation, Markdown and CSV are easier to review
`--min-coverage`	profiler validate	pin the threshold in CI	prevents silent report drift

11.7 When The CLI Fails

Start with these checks before reading the deeper reference material:

- run `tdse sdk version --json-out -` to confirm you are using the expected binary
- add `--json-out -` so failure details remain visible in shell-only logs
- reduce adapter failures to the smallest netlist, one port, and one frequency when possible
- compare current profiler output against a known-good report with `tdse profiler diff`

11.8 Production And CI Path

Use this short path when you are qualifying a build or capturing release evidence:

1. run `tdse sdk version --json-out -` and archive the version JSON
2. run one representative adapter command with `--json-out`
3. run one representative profiler command with `--out-json`, `--out-md`, and `--out-csv`
4. rerun `tdse profiler validate`, `explain`, `diff`, or `tables` on the saved report when needed
5. archive exit codes and artifact paths alongside the input identity

11.9 Related Docs

- Getting Started
- Adapter Circuit
- Profiler
- Troubleshooting
- C API Reference

11.10 Reference: Commands

11.10.1 Command Summary

Command	Purpose	Key options
`sdk version`	Report CLI/SDK/git/runtime version metadata	`--json-out`
`adapter circuit caps`	Emit backend capability Markdown and build-feature JSON	`--out-caps`, `--out-build-features`, `--json-out`
`adapter circuit raw-to-netlist`	Convert PSS/E RAW input into circuit-compatible netlist with base-frequency validation	`--raw-kind`, `--raw`, `--out-netlist`, `--out-report-json`, `--json-out`
`adapter circuit matrix`	Compute Y or Z matrix over a frequency grid and emit CSV	`--netlist-kind`, `--matrix`, `--ports`, `--w0-radps`, `--dw-radps`, `--nfreq`, `--out-data`, `--json-out`
`adapter circuit series`	Compute VOC or ISC source-side time series	`--netlist-kind`, `--series`, `--ports`, `--method`, `--dt`, `--steps`, `--out-data`, `--json-out`
`adapter circuit probe`	Compute AC or transient probe outputs (deprecated alias: `adapter circuit simulate`)	`--domain`, `--netlist-kind`, `--observe`, `--observe-source`, `--out-data`, `--json-out`
`profiler quick`	Quick benchmark pass	`--out-json`, `--out-md`, `--out-csv`, `--json-out`
`profiler calibrate`	Calibrated benchmark with route-policy derivation	`--out-json`, `--out-md`, `--out-csv`, `--json-out`
`profiler sweep`	Full parameter sweep benchmark	`--out-json`, `--out-md`, `--out-csv`, `--json-out`
`profiler validate`	Validate a profiler report against coverage thresholds	`--min-coverage`, `--json-out`
`profiler explain`	Explain one shape from a profiler report	`--np`, `--nh`, `--dtype`, `--json-out`
`profiler diff`	Diff two profiler reports	`--out-md`, `--json-out`
`profiler tables`	Generate route/case tables from a profiler report	`--out-route-md`, `--out-matrix-md`, `--json-out`
`profiler irc-scan`	Advanced IRC scan for specialized workflows	`--scan-prefix-counts`, `--scan-growth-values`, `--scan-input`, `--out-recommend-json`, `--json-out`

11.10.2 SDK Commands

`sdk version` — Report CLI, SDK, git, and runtime version metadata.

Minimal replay:

```
tdse sdk version --json-out -
```

Documented `data` keys:

- `cli_version`
- `sdk_version`
- `git_commit`
- `runtime_version`
- `runtime_version_major`
- `runtime_version_minor`
- `runtime_version_patch`

11.10.3 Adapter Circuit Commands

`adapter circuit caps` — Emit backend capability Markdown and optional build-feature JSON.

Minimal replay:

```
tdse adapter circuit caps \
  --out-caps ./adapter_caps.md \
  --out-build-features ./adapter_build_features.json \
  --json-out -
```

Documented `data` keys:

- `backend_count`
- `caps_target`
- `build_features_target`

Capability Markdown is operator-facing, while build-features JSON is machine-readable.

`adapter circuit raw-to-netlist` — Convert PSS/E RAW input into a circuit-compatible netlist and run base-frequency validation in the same command path.

Minimal replay:

```
tdse adapter circuit raw-to-netlist \
  --raw-kind file --raw case.raw \
  --out-netlist ./case_from_raw.cir \
  --out-report-json ./raw_report.json \
  --json-out -
```

Documented `data` keys:

- `source_kind`
- `strict`
- `output_target`
- `report_target`
- `netlist_bytes`
- `report_required_bytes`
- `validation_pass`
- `import_report`
- `summary`

`import_report` and `summary` are the contract anchors for conversion count diagnostics and base frequency validation outcomes.

Byte-size semantics: `netlist_bytes` and `report_required_bytes` are payload bytes, excluding trailing C-string null terminators.

`adapter circuit matrix` — Compute `Y` or `Z` matrix data over a requested frequency grid.

Minimal replay:

```
tdse adapter circuit matrix \
  --netlist-kind file --netlist case.cir \
  --matrix y --ports "1,0;2,0" \
  --w0-radps 377 --dw-radps 0 --nfreq 1 \
  --out-data ./matrix.csv \
  --json-out -
```

Required options:

- `--netlist-kind <file|text|stdin>`
- `--matrix <y|z>`
- `--ports <p0p,p0n;p1p,p1n;...>`
- `--w0-radps <w0>`
- `--dw-radps <dw>`
- `--nfreq <N>`

Common optional output:

- `--out-data <path|->`
- `--json-out <path|->`

DC endpoint options:

- `--dc-policy <mode>`
- `--dc-extrapolate-points <N>`

DC policy notes:

- `--w0-radps` may be `0` or any positive finite value
- CLI default is `--dc-policy exact_dc_then_fallback`
- `regularized_exact_dc` keeps the regularized `0Hz` solve behavior as the primary endpoint strategy
- `extrapolate_from_positive` replaces the `0Hz` sample with a surrogate endpoint fitted from the first positive-frequency samples; it is intended for plotting and Builder-facing smooth endpoints, not as an exact DC proof point
- `exact_dc_then_fallback` first attempts an exact `0Hz` solve with DC element semantics; if that endpoint is singular or otherwise unusable and positive-frequency samples exist, it falls back to the same surrogate endpoint strategy
- when `--w0-radps 0` and `--dc-policy extrapolate_from_positive`, use `--nfreq >= 2`

Documented `data` keys:

- `matrix_kind`
- `netlist_kind`
- `np`
- `nfreq`
- `output_target`

`adapter circuit series` — Compute `VOC` or `ISC` source-side time series.

Minimal replay:

```
tdse adapter circuit series \
  --netlist-kind file --netlist case.cir \
  --series voc --ports "1,0" \
  --method transient --dt 1e-4 --steps 64 \
  --out-data ./series.csv \
  --json-out -
```

Required options:

- `--netlist-kind <file|text|stdin>`
- `--series <voc|isc>`
- `--ports <p0p,p0n;...>`
- `--method <tone|ifft|transient>`
- `--dt <dt>`
- `--steps <N>`

Method-specific note:

- `--nfft` is required when `--method=ifft`

Documented `data` keys:

- `response_kind`
- `netlist_kind`
- `method`
- `np`
- `steps`
- `dt`
- `output_target`

`adapter circuit probe` — Compute AC or transient probe outputs. Compatibility note: `adapter circuit simulate` remains accepted as a deprecated alias.

Required options:

- `--domain <ac|transient>`
- `--netlist-kind <file|text|stdin>`

Minimal AC replay:

```
tdse adapter circuit probe \
  --domain ac --netlist-kind file --netlist case.cir \
  --observe "v(1,0)" --observe-source argument \
  --ac-excitation small_signal \
  --sweep-kind list --freq-list-hz 50,60 \
  --out-data ./probe_ac.csv \
  --json-out -
```

Minimal transient replay:

```
tdse adapter circuit probe \
  --domain transient --netlist-kind file --netlist case.cir \
  --observe "v(1,0)" --observe-source argument \
  --method transient --dt 1e-4 --steps 64 \
  --out-data ./probe_tr.csv \
  --json-out -
```

Transient-specific options:

- `--method <tone|ifft|transient>`
- `--dt <dt>`
- `--steps <N>`
- optional `--nfft <N>`
- optional `--t0 <t0>`

AC-specific options:

- optional `--ac-excitation <small_signal|legacy_tones>`
- `--sweep-kind <from_netlist|lin|dec|oct|list>`
- sweep arguments matching the selected sweep kind

Probe options:

- `--observe "v(1,0);i(r1)"`
- `--observe-source <argument|netlist|argument_or_netlist|auto>`

Documented `data` keys:

- `domain`
- `netlist_kind`
- `probe_source`
- `probe_count`
- `method` and `steps` for transient mode
- `sweep_kind`, `excitation_mode`, and `nfreq` for AC mode
- `output_target`

11.10.4 Profiler Benchmark Commands (`quick`, `calibrate`, `sweep`)

Benchmark runtime shapes and produce route-policy artifacts.

Common output options:

- `--out-json <report.json>`
- `--out-md <summary.md>`
- `--out-csv <matrix.csv>`
- `--json-out <envelope.json>` for unified CLI status envelope

Common workload controls:

- `--np <N>`
- `--nh <N>`
- `--dtype <32|64>`
- `--steps <N>`

- `--warmup <N>`
- `--repeats <N>`

11.10.5 Profiler Utility Commands (`validate`, `explain`, `diff`, `tables`, `irc-scan`)

Validation:

- `tdse profiler validate <report.json> [--min-coverage X]`

Explain one shape:

- `tdse profiler explain <report.json> --np <N> --nh <N> [--dtype 64]`

Diff two reports:

- `tdse profiler diff <old_report.json> <new_report.json> [--out-md <diff.md>]`

Generate route/case tables:

- `tdse profiler tables <report.json> --out-route-md <route.md> --out-matrix-md <matrix.md>`
- all profiler verbs also support `--json-out <envelope.json>` for machine-readable status

`profiler irc-scan` — Advanced IRC scan for specialized workflows:

- `tdse profiler irc-scan`
- `--scan-prefix-counts ...`
- `--scan-growth-values ...`
- `--scan-input random|lowpass|step_ring|trace`
- `--out-recommend-json <recommend.json>`

11.11 Deep Reference: Output Formats and Stable Fields

Use this section when you parse CLI artifacts directly in automation or tooling. If you only need to run commands interactively, the command reference above is usually enough.

11.11.1 Common JSON Envelope

When `--json-out` is enabled, the CLI writes a completion envelope with these stable top-level keys:

- `group`
- `action`
- `status`
- `exit_code`
- `message`
- `data`

Parsing guidance:

- use `group` and `action` to determine which command completed
- use `status`, `exit_code`, and `message` for command outcome handling

- parse `data` according to the command family and the corresponding output shape
- accept undocumented extra keys conservatively rather than failing hard on their presence
- do not treat undocumented nested keys inside `data` as a long-term contract unless another public guide section says so explicitly

For profiler commands, `--json-out` is a command-level envelope emitted by the unified `tdse` router (`group="profiler"`), so automation can consume success/failure status consistently across adapter and profiler families.

11.11.2 Output Artifacts

Adapter Circuit. Primary adapter-circuit outputs:

- converted netlist
- validation report JSON
- matrix or probe CSV
- command completion JSON envelope

Use these as integration records before handoff to Builder and Runtime.

Profiler. Primary profiler outputs:

- profiler report JSON (authoritative machine-readable surface)
- optional summary Markdown
- optional matrix CSV
- runtime plan section for `tdse_backend_apply_plan(...)`

Cross-Family Handoff. Treat TDSE CLI outputs as one handoff chain:

1. adapter-domain artifacts establish model and signal validity
2. profiler artifacts establish route and policy validity
3. runtime consumes stable plan information only after both are accepted

Authoritative vs Summary.

- authoritative:
  - adapter and profiler JSON outputs
  - profiler schema-backed report
- summary:
  - Markdown and human review tables
  - CSV convenience outputs for analysis

11.11.3 Command Data Shapes

`sdk version` fields:

- `cli_version`
- `sdk_version`
- `git_commit`
- `runtime_version`
- `runtime_version_major`
- `runtime_version_minor`

- `runtime_version_patch`

`adapter circuit caps` documented `data` fields:

- `backend_count`
- `caps_target`
- `build_features_target`

`adapter circuit raw-to-netlist` documented fields:

- `source_kind`
- `strict`
- `output_target`
- `report_target`
- `netlist_bytes`
- `report_required_bytes`
- `validation_pass`
- `import_report`
- `summary`

Matrix Output (`Y` / `Z`). CLI CSV header:

```
freq_index,freq_radps,row,col,re,im
```

Parsing contract:

- one row per frequency / matrix-row / matrix-column tuple
- `freq_index` is zero-based
- `freq_radps` is the physical angular frequency for that row
- `row` and `col` are zero-based port indices
- `re` and `im` store the complex value
- frequency-major ordering

In-memory C API layout:

- frequency-major row-major
- packed as `out[2*(k*np*np + row*np + col) + {0,1}]`

Builder handoff note:

- this is the same ordering expected when the matrix is exposed through `tdse_builder_cplx_mat_view_t`

`VOC` / `ISC` Time-Series Output. CLI CSV header:

```
step,time,port,value
```

Parsing contract:

- one row per time step and port
- `step` is zero-based
- `time = t0 + step * dt`
- `port` is the zero-based port index
- `value` is the scalar `VOC` or `ISC` sample

- step-major ordering

In-memory C API layout:

- step-major row-major
- `out_values[step*np + port]`

Integration note:

- CLI JSON is metadata only; the sequence payload contract is the CSV shape above or the in-memory layout above

Transient Probe Output. CLI CSV header:

```
step,time,probe,value
```

Parsing contract:

- one row per time step and resolved probe
- `probe` is the zero-based resolved probe index
- the mapping from probe index to expression depends on CLI `--observe` order or netlist probe order

In-memory C API layout:

- step-major row-major
- `out_values[step*nprobe + probe]`

AC Probe Output. CLI CSV header:

```
freq_index,freq_hz,probe,re,im
```

Parsing contract:

- one row per frequency sample and probe
- `freq_index` is zero-based
- `freq_hz` is the physical frequency in Hz
- `probe` is the zero-based resolved probe index
- `re` / `im` store the complex probe response
- frequency-major ordering

In-memory C API layout:

- frequency-major row-major with RI interleaving
- `out_values_ri[2*(k*nprobe + p) + 0] = Re(X_p(f_k))`
- `out_values_ri[2*(k*nprobe + p) + 1] = Im(X_p(f_k))`

Unit note:

- AC probe sweep files use Hz, not rad/s

Region Preparation and Region-Compute Report. `tdse_adapter_circuit_prepare_region(...)` can return a machine-readable JSON report through `out_report_json`.

For machine integration, the primary shape contract is no longer JSON-only:

- region preparation returns `required_ports_len` plus optional `out_ports`
- matrix calls return `result.shape` plus optional `out_resolved_ports`
- series calls return `result.shape` plus optional `out_resolved_ports`
- probe calls return `result.shape`

Treat the JSON report as audit/debug metadata, not the only source of shape information.

Stable top-level keys for this release:

- `report_version`
- `summary`
- `boundary`
- `shape`
- `resolved_ports`
- `issues`

Parsing guidance:

- require `report_version == 2`
- treat the key set above as the documented top-level contract for this release
- tolerate additional undocumented keys conservatively
- use `result.shape`, `result.required_*`, and optional `out_resolved_ports` as the primary in-memory sizing contract

Top-level shape:

```
{
  "report_version": 2,
  "summary": {},
  "boundary": {},
  "shape": {},
  "resolved_ports": [],
  "issues": []
}
```

`summary` — operation metadata. Common keys include: `operation`, `accepted`, `requested_node_count`, `region_node_count`, `region_element_count`, `component_count`, `boundary_node_count`, `port_count`, `topology_hash`, and `parameter_hash`. The `operation` field identifies which phase emitted the report (for example, `prepare_region`).

`boundary` — region boundary structure. Contains `components`, where each component holds `boundary_nodes`, `reference_node`, and `ports`.

`shape` — dimension and layout contract. Always includes `port_count` and `layout`. Matrix reports add `required_values_len`, `w0`, `dw`, `nfreq`. Series reports add `required_values_len`, `dt`, `steps`. Preparation reports include `required_ports_len`.

`resolved_ports` — array of `[node_p, node_n]` pairs. Each entry is a canonical SPICE node label string (`"0"` for ground, retained name forms when available, numeric token otherwise). Hosts must not interpret these strings as internal node indices.

Profiler Canonical JSON Report. Primary machine-readable artifact:

- `tdse_profile_report.json`

Runtime-facing expectations:

- `schema_version == "tdse_profile_report_v1"`
- non-placeholder fingerprint fields
- `policy.rules`
- `runtime_plan`

Profiler `--json-out` Envelope. Profiler envelope `data` is currently an empty object `{}` and is intended as a stable status contract:

- `group`: always `profiler`
- `action`: resolved profiler action (`quick`, `calibrate`, `validate`, `diff`, etc.)
- `status`: `ok` or `error`
- `exit_code`: process exit code
- `message`: short completion/failure summary

`runtime_plan` Contract. `runtime_plan` is the profiler output intended for TDSE Core consumption through `tdse_backend_apply_plan(...)`.

At minimum, it must provide:

- a default backend choice
- scenario-based selections for relevant shape signatures

11.11.4 Human-Review And Debug Outputs

- Markdown summary and route tables are human-review surfaces
- CSV matrices are analysis/audit surfaces
- runtime consumption should rely on JSON contracts only

NPORT Debug CSV / JSON. Debug CSV shape:

```
# type=Y,nports=2,z0=50
freq_hz,row,col,re,im
1000,0,0,1.0,0.0
1000,0,1,0.0,0.0
1000,1,0,0.0,0.0
1000,1,1,2.0,0.0
```

Rules:

- the file must contain the full `N x N` matrix at each frequency
- `row` and `col` are zero-based
- `type` should be `Y` or `Z`

Debug JSON shape:

```
{
  "nports": 2,
  "type": "Y",
  "z0": 50,
  "freq_hz": [1000, 2000],
```

```
  "mats_ri": [
    [[1, 0], [0, 0], [0, 0], [2, 0]],
    [[3, 0], [0, 0], [0, 0], [4, 0]]
  ]
}
```

Rules:

- `freq_hz[k]` and `mats_ri[k]` must have matching outer lengths
- each `mats_ri[k]` can be either:
  - a flat row-major list of `N*N` complex pairs, or
  - a nested `N x N` list of complex pairs

Integrator Guidance. Recommended parser behavior:

- branch on `group` and `action`
- validate documented keys and headers
- ignore unknown extra keys conservatively
- archive artifact paths named in `output_target`, `caps_target`, `build_features_target`, `report_target`, and profiler report output options

11.12 Exit Codes

The CLI maps outcomes into stable process exit categories:

- `0`: success
- `2`: CLI argument shape error
- `10`: invalid argument
- `11`: parse failure
- `12`: I/O failure
- `13`: unsupported operation
- `14`: singular solve
- `15`: internal failure
- `16`: backend unavailable
- `20`: strict validation failure

Both command families (adapter and profiler) remain automation-friendly:

- nonzero code on failure family
- machine-readable completion payload via `--json-out` when available
- command-scoped stable fields documented in the output contracts section above

CHAPTER 12

Advanced Runtime Integration

> **Audience:** Integration engineers embedding TDSE into simulators with adaptive or multi-rate time-stepping (SPICE-family, PSCAD/EMTP, Keysight ADS, etc.).

Use this chapter when your simulator does not step at exactly the same interval used to build the pack. It explains how runtime `dt` relates to build-time `model_dt`, what accuracy changes when they differ, and which integration patterns are usually safe.

This chapter starts with variable time-step integration, then continues into runtime concurrency, scaling, and multi-model deployment patterns in the sections that follow.

Use it only after the fixed-step step loop is already understood. For a first minimal integration, keep `dt_runtime = model_dt` and return here only when the host truly needs adaptive or multi-rate stepping.

The three most common host cases are:

Host case	Recommended first policy
fixed-step simulator	keep `dt_runtime = model_dt`
adaptive simulator	start with clamped adaptive `dt`, then measure interpolation error
multi-rate orchestration	use multiple model handles, one time base per handle

12.1 Core Semantics

The impulse-response tensor `h[k]` (shape `[nh, nq, np]`) is sampled at build time with a fixed step size `model_dt`:

```
tau_k = k * model_dt,   k = 0, 1, ..., nh - 1
```

`model_dt` is an intrinsic model property, queryable via `tdse_model_info_t.dt`.

The `dt` parameter passed to `tdse_step_begin(model, t, dt)` is the **simulator's current physical time span** for this step. It does **not** need to equal `model_dt`. The runtime convolution engine bridges the two automatically.

12.2 Two Internal Paths

The runtime selects one of two paths based on the time-line history.

12.2.1 Uniform Fast Path

Triggered when **all** of these conditions are satisfied simultaneously:

1. The committed history timeline is monotonically increasing and uniform, with step size equal to `model_dt` (relative tolerance 1e-9, absolute 1e-12).
2. The current step time `t` minus the latest committed time `t_newest` equals `model_dt` (same tolerance).

Under this path:

- History terms are read directly from the ring buffer by tap index, with no interpolation.
- This is the most accurate and fastest path, equivalent to fixed-step convolution.

12.2.2 Time-Driven Interpolation Fallback

Entered whenever any trigger condition fails (including but not limited to: `dt != model_dt`, `dt` varying between steps, a large time-step jump, or initial conditions where the accepted queue has not yet been uniformized).

For each delayed tap `k >= 1`:

```
sample_t = t - tau_k = t - k * model_dt
```

The runtime performs **linear interpolation** on the committed primary input timeline to obtain `v(sample_t)`, then weights and accumulates with `h[k]`.

- Boundary semantics: when `sample_t` precedes the earliest committed time, that tap's contribution is treated as zero (equivalent to assuming the model was at rest before time zero).
- Interpolation order: first-order (linear). There is no zero-order hold or higher-order interpolation path.

12.3 Accuracy Impact

The dominant error in the history term `hr[n]` under the time-driven path is governed by linear interpolation:

```
err(hr) = O(dt_runtime^2 * sup_t |d^2 v / dt^2|)
```

where `v` is the committed primary input sequence. Practical guidance:

Scenario	Recommended `dt_runtime / model_dt`	Notes
Smooth input (AC steady-state, low-freq transient)	0.5 – 10	Large steps only lose linear interpolation accuracy in history
General transient (step, pulse)	0.5 – 2	Excessive ratio causes visible "smearing" near step edges
Steep transient (switching, surge, impulse)	0.5 – 1	Recommend `dt_runtime ~ model_dt`

Key numerical conclusions:

- The instantaneous path (`tdse_step_op`, direct response in `tdse_step_commit`) is driven by `h[0]` and is **exact for any runtime step size.**
- The IR term (`tdse_step_ir`) queries the pack's internal sampling table by absolute time `t`; `dt_runtime` affects only the time quantization.

- Only the **history term `hr`** accumulates linear interpolation error as `dt_runtime / model_dt` deviates from `1`.

12.4 Stability

Variable time-stepping does **not** change the model's stability properties:

- Model stability is determined by the spectral radius of `h[k]` in the pack, which is fixed at build time.
- Variable stepping only affects the computational approximation of the history term; it cannot make a stable model unstable.
- Extreme dt jumps (e.g., from `model_dt` to `100 * model_dt` and back) may amplify interpolation error in transients but will not inject spurious energy.

12.5 Monitoring

When exploring aggressive dt strategies, poll runtime guard metrics via `tdse_ext_get_runtime_guard_metrics()`:

Metric	Meaning	Watch For
`max_abs_g0`	max amplitude of `h[0]` entries this step	model-dependent; use as trend
`pivot_min`	minimum pivot	significant drop means op near-singular
`pivot_ratio`	current `pivot_min` vs baseline	sustained < 0.1 warrants investigation
`growth_factor`	inter-step `|hr|` growth rate	sustained > 1 indicates potential divergence

12.6 Multi-Rate Models

TDSE multi-rate at the runtime level is achieved by running **multiple model handles concurrently**, not through internal multi-clock support:

```
tdse_model_t* fast_model;  /* model_dt = 1 ns */
tdse_model_t* slow_model;  /* model_dt = 10 ns */

tdse_model_create(fast_pack, fast_len, &diag, &fast_model);
tdse_model_create(slow_pack, slow_len, &diag, &slow_model);

for (step = 0; step < N; ++step) {
    /* fast model: step every 1 ns */
    tdse_step_begin(fast_model, t_fast, 1e-9);
    /* ... */ tdse_step_commit(fast_model, primary_fast);

    /* slow model: step every 10 ns */
    if (step % 10 == 0) {
        tdse_step_begin(slow_model, t_slow, 10e-9);
        /* ... */ tdse_step_commit(slow_model, primary_slow);
```

```
    }
}
```

Each handle maintains its own history ring buffer independently. Concurrency rules still apply: no concurrent API calls on the **same handle**; multiple independent handles may run in parallel on different threads.

12.7 Recommended Integration Patterns

Use these in order. Do not jump to adaptive or multi-rate operation before the fixed-step path is already validated.

12.7.1 Fixed dt (simple SPICE integration)

Always use `dt_runtime = model_dt` in the simulator main loop:

```
const double model_dt = info.dt;
for (...) {
    tdse_step_begin(model, t, model_dt);
    /* op / hr / ir */
    tdse_step_commit(model, primary);
    t += model_dt;
}
```

This always hits the uniform fast path, maximizing accuracy and performance.

12.7.2 Adaptive dt (LTE-controlled SPICE)

Let `dt_runtime` follow the solver's local truncation error control:

```
const double model_dt = info.dt;
for (...) {
    double dt = solver_proposed_dt();
    /* Clamp to [0.1 * model_dt, 10 * model_dt] for typical EDA scenarios */
    if (dt < 0.1 * model_dt) dt = 0.1 * model_dt;
    if (dt > 10.0 * model_dt) dt = 10.0 * model_dt;

    tdse_step_begin(model, t, dt);
    /* ... */
    tdse_step_commit(model, primary);
    t += dt;
}
```

Host/TDSE split here:

- the host proposes and clamps `dt`
- TDSE interprets the accepted `(t, dt)` against `model_dt`
- interpolation error belongs to the integration policy, not to Runtime

12.7.3 Trace Replay

When `primary` comes from an external trace file with `trace_dt != model_dt`:

- Option A: drive the runtime with `trace_dt` directly (time-driven interpolation).
- Option B: resample the trace to `model_dt` externally, then use the fast path.

Both are mathematically equivalent. Option B enables more optimized convolution backends (CPU_BLAS / GPU_PACK_BLAS paths perform best on the uniform fast path).

12.7.4 Decision Table

If your main goal is...	Start with...	Escalate to...
fastest bring-up	fixed `dt_runtime = model_dt`	nothing until basic loop is stable
adaptive solver fidelity	clamped adaptive `dt`	error measurement and tuning
multi-rate orchestration	one handle per rate	explicit scheduling and cross-model review

Related API

- `tdse_step_begin(model, t, dt)` in `tdse/tdse.h`: core entry point discussed here
- `tdse_model_info_t.dt` in `tdse/tdse.h`: query `model_dt`
- `tdse_ext_get_runtime_guard_metrics()` in `tdse_ext.h`: runtime stability monitoring
- `tdse_builder_compute_consistent_grid()` in `tdse_builder.h`: derive consistent `model_dt`, `nh`, `nfft`, and `dw` at build time

12.8 Concurrency and Shutdown

Use this section when TDSE Runtime is live in more than one thread, worker, or shutdown path. It explains the one-handle rule, which calls can overlap safely, and how to reason about contention and teardown races.

> **Related Chapters** For the base lifecycle semantics, see Runtime Lifecycle. For threading and memory scaling with many models, see Threading and Scaling.

Most production Runtime incidents in this area are not numerical defects. They are ownership defects: two threads think they own the same handle, shutdown starts while a step call is still live, or host wrappers treat `release` as a policy API instead of a finalizer.

Read this chapter together with Lifecycle and Ownership: lifecycle explains what local ownership means; this chapter explains how that ownership behaves under contention and teardown.

12.8.1 The One-Handle Rule

The core runtime concurrency rule is simple:

- one live `tdse_model_t*` handle must not be entered concurrently for same-handle runtime APIs

The protected same-handle step surface is:

- `tdse_step_begin(...)`
- `tdse_step_op(...)`
- `tdse_step_hr(...)`
- `tdse_step_ir(...)`
- `tdse_step_commit(...)`
- `tdse_step_dr(...)`
- `tdse_model_close(...)`
- `tdse_model_destroy(...)`
- `tdse_model_release(...)`

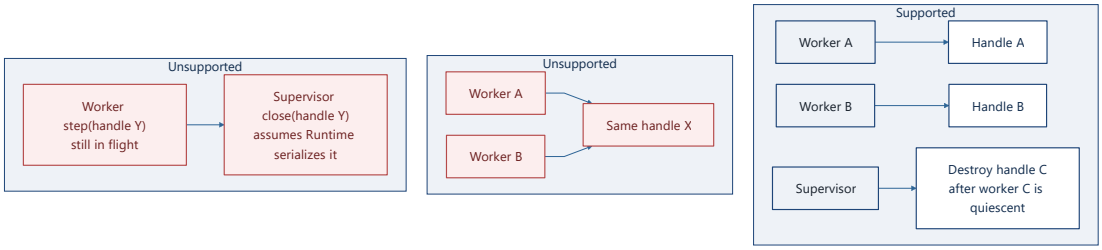


Figure 12.1 Supported versus unsupported host ownership

The runtime behavior is rejection, not automatic serialization. That is deliberate: silent serialization would hide ownership bugs and make timing behavior much harder to diagnose.

12.8.2 Which APIs Are Guarded Versus Snapshot-Style

Not every API participates in the same-handle execution guard. The most important distinction is:

- execution APIs and teardown APIs are guard-sensitive
- metadata and diagnostics snapshot APIs are live-handle reads

Snapshot-style query APIs are:

- `tdse_model_info(...)`
- `tdse_model_state_info(...)`
- `tdse_model_last_error_info(...)`

These queries do not enter the same-handle execution guard. On a live handle they are intended to remain readable even if step traffic is happening on another thread. They still stop being valid once close or destroy has already started on that handle, in which case they return `TDSE_ERR_INVALID_STATE`.

Operational consequence:

- do not use snapshot-query success as evidence that a handle is safe to step from another thread
- do use snapshot queries to capture diagnostics before teardown begins or after a step failure

12.8.3 Runtime Guarantees Under Conflict

When runtime detects same-handle overlap on the guarded surface:

- conflicting entrants are rejected rather than queued

- step conflicts return `TDSE_ERR_CONCURRENT_API_USE`
- lifecycle conflicts may return `TDSE_ERR_CONCURRENT_API_USE`, `TDSE_ERR_TIMEOUT`, or `TDSE_ERR_INVALID_STATE` depending on which API raced and whether ownership already moved
- runtime avoids deadlock as part of the supported behavior

Important nuance:

- forced overlap does not mean every entrant fails
- one caller may legitimately acquire the handle first and succeed
- conflicting entrants are rejected safely and observably

That distinction matters in stress tests. Under intentional race injection, “one winner plus one or more rejected entrants” is the expected shape.

12.8.4 Safe Parallelism Model

Supported host parallelism looks like this:

1. one simulation worker owns one runtime handle
2. different handles may run concurrently
3. optional internal execution strategy may parallelize work inside one step call
4. internal parallel execution does not make a single handle concurrently callable

Recommended host rule:

- assign both execution ownership and shutdown ownership to the same wrapper or thread controller

If those responsibilities are split across components, the ownership handoff protocol must be explicit rather than implied.

12.8.5 Teardown State Model

The shutdown APIs are easiest to reason about as an ownership state machine.

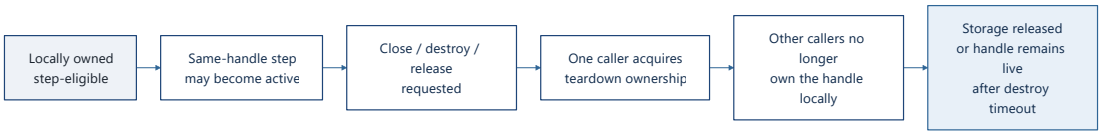


Figure 12.2 Teardown-oriented handle states

Two rules keep this understandable:

- once another thread acquires teardown ownership, the handle is no longer locally usable to you
- `TDSE_ERR_TIMEOUT` from destroy is the one status that means the handle is still live after return

12.8.6 Close, Destroy, And Release Under Contention

12.8.6.1 `tdse_model_close(...)`

`close` is the immediate-answer lifecycle API.

Use it when:

- you want a synchronous non-waiting lifecycle result
- you want to detect overlap instead of waiting through it

Under contention:

- if another guarded same-handle API is still in flight, close returns `TDSE_ERR_CONCURRENT_API_USE`
- if another thread already started close or destroy, close returns `TDSE_ERR_INVALID_STATE`

Operational reading:

- `close` is not a “fast destroy”
- it is the right API when overlap itself is the information you need

12.8.6.2 `tdse_model_destroy(...)`

`destroy` is the recommended business-logic shutdown API because it makes wait policy explicit.

Use it when:

- your host owns lifecycle policy
- you need bounded wait behavior
- you want structured wait telemetry

Destroy outcomes:

- `TDSE_OK`: teardown completed and storage is gone
- `TDSE_ERR_TIMEOUT`: destroy could not acquire teardown ownership within the budget; the handle remains valid
- `TDSE_ERR_INVALID_STATE`: another thread already owns close or destroy; local ownership is gone

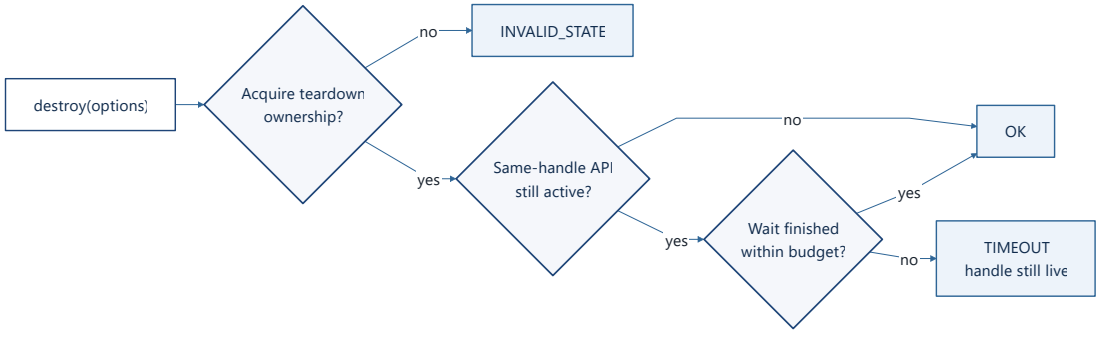


Figure 12.3 Destroy race interpretation

12.8.6.3 `tdse_model_release(...)`

`release` is for terminal cleanup, not shutdown policy.

Use it when:

- a destructor must not become a policy engine
- a `finally` or unwind path needs best-effort terminal cleanup

Under contention:

- release waits for an in-flight same-handle API to leave the guard once it owns terminal cleanup
- if another thread already started close, destroy, or release, release returns `TDSE_ERR_INVALID_STATE`

Operational reading:

- `release` is acceptable in finalizers because it is cleanup-oriented
- `release` is a poor choice for ordinary host shutdown because it does not carry a bounded-wait policy

12.8.7 Race Matrix

Use this matrix when a shutdown report is unclear about which thread acted first.

Situation	`close` sees	`destroy` sees	`release` sees	What the caller should assume
step call still in flight on same handle	`TDSE_ERR_CONCURRENT_API_USE`	`TDSE_OK` or `TDSE_ERR_TIMEOUT` depending on wait budget	waits until it can clean up	handle ownership is still local only if destroy timed out
another thread already started close/destroy	`TDSE_ERR_INVALID_STATE`	`TDSE_ERR_INVALID_STATE`	`TDSE_ERR_INVALID_STATE`	local ownership is gone
no same-handle activity, caller owns handle	`TDSE_OK`	`TDSE_OK`	`TDSE_OK`	storage is gone after return
destroy timed out while waiting	n/a	`TDSE_ERR_TIMEOUT`	n/a	handle is still live and policy must decide next step

The support-facing rule is:

- only `TDSE_ERR_TIMEOUT` from destroy preserves local ownership after return

12.8.8 Query Behavior During Shutdown

Support incidents often ask whether a host can still query state while shutdown is underway. Use the strict answer:

- before close or destroy starts, snapshot queries are allowed on a live handle
- after close or destroy has started, `tdse_model_info(...)`, `tdse_model_state_info(...)`, and `tdse_model_last_error_info(...)` may return `TDSE_ERR_INVALID_STATE`
- after successful close, destroy, or release, no further handle use is valid

That means the host should capture evidence before starting teardown whenever possible.

Recommended diagnostic order on a failing live handle:

1. `tdse_model_info(...)`

2. `tdse_model_state_info(...)`
3. `tdse_model_last_error_info(...)`
4. chosen shutdown API and timeout policy

12.8.9 Bounded Destroy Policy

`tdse_model_destroy_options_t.wait_timeout_ms` is part of the public behavior, not a tuning footnote.

Interpret the wait budget as:

- negative value: intentional infinite wait
- small bounded value: supervisory shutdown that prefers a fast answer
- medium bounded value: ordinary business-logic cleanup where some overlap is tolerated

Recommended policy questions:

1. Which thread is allowed to decide the wait budget?
2. What does the host do on `TDSE_ERR_TIMEOUT`?
3. At what point does the host stop retrying and defer to finalizer cleanup?
4. Which path records the observed `wait_ms` into logs or crash bundles?

Example bounded-destroy pattern:

```
tdse_model_destroy_options_t opt = tdse_model_destroy_options_init();
tdse_model_destroy_result_t result = tdse_model_destroy_result_init();
opt.wait_timeout_ms = 250.0;

tdse_status_t st = tdse_model_destroy(model, &opt, &result);
if (st == TDSE_OK) {
  model = NULL;
} else if (st == TDSE_ERR_TIMEOUT) {
  log_warn("destroy timeout wait_ms=%.3f timed_out=%d", result.wait_ms, result.timed_out);
  /* handle remains live; supervisor decides retry or escalation */
} else if (st == TDSE_ERR_INVALID_STATE) {
  model = NULL; /* ownership already moved elsewhere */
}
```

12.8.10 Worked Host Patterns

12.8.10.1 Pattern A. Worker-Owned Handle With Clean Shutdown

This is the preferred product integration pattern:

1. worker thread creates the handle
2. worker thread performs the step loop
3. worker thread or its owner wrapper initiates `destroy`
4. no other thread touches the handle after shutdown starts

Why it works:

- execution ownership and teardown ownership stay aligned

- there is no ambiguity about who records diagnostics or clears references

12.8.10.2 Pattern B. Supervisor Requests Stop, Worker Performs Destroy

This is often better than having the supervisor destroy directly:

1. supervisor sets a stop request in host code
2. worker exits its loop at a safe boundary
3. worker performs `tdse_model_destroy(...)`
4. supervisor observes the result through host telemetry

Why it works:

- it avoids same-handle races between a live step call and a remote destroy
- it keeps timeout policy near the code that already owns the handle

12.8.10.3 Pattern C. Destructor-Only Final Cleanup

Use this only when ordinary business shutdown has already failed or is unavailable:

```
class ModelGuard {
 public:
  ~ModelGuard() noexcept {
    if (handle_ != nullptr) {
      (void)tdse_model_release(handle_);
      handle_ = nullptr;
    }
  }
 private:
  tdse_model_t* handle_ = nullptr;
};
```

Why it is acceptable:

- destructors need a terminal cleanup target
- they should not be responsible for choosing timeout policy

12.8.11 Troubleshooting Shutdown Symptoms

12.8.11.1 Symptom: Destroy Times Out Repeatedly

Likely meaning:

- a same-handle step call is still live when destroy starts
- the host has no clear quiesce-before-destroy protocol

Collect:

- destroy `wait_timeout_ms`
- destroy `wait_ms`
- failing thread identities from host logs
- whether the worker loop had actually stopped before destroy

First corrective actions:

1. move destroy to the owning worker or wrapper
2. add an explicit stop-and-join phase before destroy
3. keep bounded destroy, but treat repeated timeout as an ownership bug

12.8.11.2 Symptom: `close` Returns `TDSE_ERR_CONCURRENT_API_USE`

Likely meaning:

- another same-handle API is still active

Correct interpretation:

- runtime is working as designed
- the host attempted an immediate-answer close during active execution

Corrective action:

- use `destroy` if bounded waiting is desired
- keep `close` only when fast overlap detection is the intent

12.8.11.3 Symptom: `destroy` Or `release` Returns `TDSE_ERR_INVALID_STATE`

Likely meaning:

- another thread already owns teardown

Corrective action:

- clear local references
- stop issuing further same-handle calls
- repair the host ownership model instead of retrying locally

12.8.12 Recommended Evidence for Concurrency Issues

When a concurrency issue is reported, collect the smallest bundle that explains ownership:

- handle identity in host logs
- failing API name
- thread or worker identity on both sides of the race
- `tdse_model_state_info(...)` captured before teardown when available
- `tdse_model_last_error_info(...)` captured before teardown when available
- chosen shutdown API: `close`, `destroy`, or `release`
- destroy wait budget and returned `wait_ms`
- whether the host had already requested worker stop
- whether the issue reproduced with one handle per thread

This bundle is usually more valuable than a large raw trace with no ownership annotations.

12.8.13 Review Checklist

During integration review, ask:

1. Which component owns each live handle?

2. Which component is allowed to start close or destroy?
3. Can a supervisor request stop without directly entering the handle?
4. Where is the timeout policy for destroy chosen and logged?
5. What happens to local references after `TDSE_ERR_INVALID_STATE`?
6. Which path captures diagnostics before teardown starts?

12.8.14 Anti-Patterns

Avoid these patterns even if they look harmless in local tests:

1. sharing one handle across workers and assuming Runtime will serialize it
2. using `release` as the default business-logic shutdown API
3. treating `close` as a faster version of `destroy`
4. retrying local teardown after `TDSE_ERR_INVALID_STATE`
5. treating `TDSE_ERR_TIMEOUT` as if the handle were already gone
6. starting teardown before the host has a stop or quiesce protocol

12.9 Threading and Memory Scaling

Use this section when the one-handle rule is already settled and the next question is scale: how much memory will N models use, when does NUMA matter, when is GPU sharing still reasonable, and which model sizes deserve extra caution.

> **Related Chapters** For concurrency rules governing individual handles, see Concurrency and Shutdown. For multi-model deployment patterns, see Multi-Model Patterns.

EDA integrations often run many TDSE models in parallel to maximize throughput. This chapter is about capacity planning and resource envelopes, not about same-handle ownership policy.

12.9.1 Capacity Planning For N Parallel Models

12.9.1.1 Per-Model Memory Footprint

Each TDSE model handle has a memory footprint that depends on:

- Port count (number of ports `np`)
- History depth (for IR models)
- Matrix size and sparsity
- Solver backend (dense vs sparse, CPU vs GPU)
- Internal workspace buffers

Think about the footprint in layers:

- **Model metadata**: ~1-10 KB (constant overhead)
- **Matrix storage**: scales with `np^2` for dense, `O(nnz)` for sparse
- **Solver workspace**: backend-dependent (LU factors, sparse symbolic structure)
- **IR convolution buffers**: scales with history depth × port count
- **Internal scratch buffers**: temporary workspace during step operations

12.9.1.2 Linear Scaling Model

For N parallel model handles, memory scales approximately linearly:

```
Total memory approximately N x (per-model footprint) + shared_runtime_overhead
```

The shared runtime overhead is typically small (< 10 MB) and does not scale with N.

12.9.1.3 Practical Planning Ranges

Based on typical workloads:

- **Small models** (np < 10, sparse): ~1-10 MB per model
- **Medium models** (np < 100, moderate density): ~10-100 MB per model
- **Large models** (np > 100, dense or IR-heavy): ~100 MB - 1 GB per model

These are rough estimates, not support limits. Actual memory usage depends on matrix sparsity, history depth, and backend selection.

12.9.1.4 Ways To Reduce The Footprint

To reduce memory footprint with many parallel models:

1. **Reuse one model sequentially when throughput is not the goal**: for repeated sweeps of the same pack, one model plus `tdse_model_reset(...)` can be cheaper than many parallel copies. See Multi-Model Patterns.
2. **Limit history depth**: IR convolution memory scales linearly with history depth
3. **Prefer sparse backends**: For sparse matrices, sparse backends use significantly less memory than dense
4. **Batch sweeps**: For AC sweeps with many frequency points, use sweep parallelism instead of separate models

12.9.1.5 What To Measure First

Use the following to monitor memory:

- `tdse_model_info(...)` returns model metadata including port count
- OS-level tools (Task Manager, `top`, `ps`) for process memory
- For GPU workloads, use CUDA tools (`nvidia-smi`) to monitor GPU memory

12.9.2 NUMA-Aware Allocation Strategy

12.9.2.1 Current NUMA Support

Runtime does not currently implement explicit NUMA-aware allocation. Memory allocation follows the OS default policy:

- On Linux: memory is allocated from the NUMA node where the allocating thread is running
- On Windows: memory allocation follows Windows NUMA policies

12.9.2.2 Best Practices For NUMA Systems

For optimal performance on multi-socket NUMA systems:

1. **Thread locality**: Keep the thread that creates a model handle on the same NUMA node as the thread that steps it

2. **Avoid remote ownership paths:** create and step a model on the same node whenever possible
3. **Bind threads to NUMA nodes:** Use OS tools (`numactl`, `SetProcessAffinityMask`) to pin threads to specific NUMA nodes
4. **Memory-first strategy:** Allocate models on NUMA nodes with sufficient memory

12.9.2.3 Recommended NUMA Pattern

```
// On Linux with numactl
// Run on NUMA node 0 with memory from node 0
numactl --cpunodebind=0 --membind=0 ./your_simulation

// Or programmatically:
numa_set_preferred(0);  // Prefer node 0
// Create model handles here
```

12.9.2.4 When NUMA Usually Matters

- TDSE does not provide explicit NUMA control APIs
- NUMA effects are most visible with very large models (> 1 GB) or high model counts (> 100)
- For typical workloads (< 50 models, < 100 MB each), OS default policies are often sufficient

12.9.3 GPU Multi-Stream Parallelism

12.9.3.1 Current GPU Parallelism Model

Think of GPU sharing in two layers:

- **Single-stream per model:** Each model handle uses one CUDA stream
- **AC sweep parallelism:** Multiple frequency points can be solved in parallel using CUDA batched operations
- **No explicit multi-stream API:** TDSE does not expose CUDA stream management to the host

12.9.3.2 AC Sweep Parallelism on GPU

When CUDA sweep parallelism is enabled:

- The adapter uses multiple worker threads, each with its own CUDA stream
- Frequency points are distributed across workers and streams
- Batched solves (2-4 points per chunk) improve GPU utilization

Diagnostics fields to monitor:

- `sweep_parallel_enabled`: Whether parallelism activated
- `sweep_parallel_workers`: Number of worker threads/streams
- `sweep_parallel_points`: Number of frequency points processed in parallel

12.9.3.3 Multi-Model GPU Parallelism

For multiple model handles on GPU:

- Each model handle uses its own CUDA context/stream
- Multiple models can run concurrently on the same GPU

- GPU memory is shared across all models on the same device

12.9.3.4 GPU Memory Management

GPU memory considerations:

- **Per-model GPU memory:** Matrix + solver workspace + convolution buffers
- **Shared GPU memory:** CUDA context overhead, driver overhead
- **Memory limits:** Use `nvidia-smi` to monitor GPU memory usage
- **Out-of-memory handling:** TDSE returns `TDSE_ERR_OUT_OF_MEMORY` if GPU allocation fails

12.9.3.5 Recommendations

1. **Monitor GPU memory:** Use `nvidia-smi` to ensure sufficient GPU memory for your model count
2. **Batch sweeps:** Use AC sweep parallelism instead of separate models when possible
3. **Prefer CPU for very small models:** Small models may not benefit from GPU overhead
4. **Limit concurrent GPU models:** If GPU memory is constrained, limit the number of concurrent GPU models

12.9.4 Practical Limits On Port Count And History Depth

12.9.4.1 Port Count Limits

There is no hard-coded maximum port count. Practical limits are determined by:

- **Memory:** Dense matrices scale as `O(np^2)`, sparse as `O(nnz)`
- **Performance:** Factorization cost grows super-linearly with matrix size
- **Numerical stability:** Very large matrices may have conditioning issues

Empirical guidance:

- **< 10 ports:** Fast, suitable for dense backends
- **10-100 ports:** Medium scale, sparse backends often beneficial
- **100-1000 ports:** Large scale, sparse backends recommended
- **> 1000 ports:** Very large, sparse backends essential; performance depends heavily on sparsity

12.9.4.2 History Depth Limits

For IR-based models, history depth affects:

- **Memory:** Linear scaling with history depth × port count
- **Performance:** Convolution cost scales with history depth
- **Accuracy:** Deeper history improves accuracy for long-time responses

Empirical guidance:

- **< 100 samples:** Fast, suitable for narrowband responses
- **100-1000 samples:** Medium depth, typical for many applications
- **1000-10000 samples:** Deep history, for wideband or long-time responses
- **> 10000 samples:** Very deep, may require significant memory and time

12.9.4.3 Solver Backend Interaction

Port count and history depth interact with solver backend selection:

- **Dense backends:** Port count dominates memory and performance
- **Sparse backends:** Sparsity pattern matters more than raw port count
- **IR convolution:** History depth dominates memory and performance
- **GPU backends:** Benefit from larger problems to amortize transfer overhead

12.9.4.4 When To Expect Issues

Watch for these warning signs:

- **Memory spikes:** Sudden large memory increases with small parameter changes
- **Performance degradation:** Non-linear performance drop with increasing size
- **Numerical warnings:** `near_singular` flags, large condition numbers
- **GPU out-of-memory:** CUDA allocation failures

Diagnostics to monitor:

- `solver_backend`: Which backend is selected
- `matrix_nnz`, `matrix_density`: Matrix characteristics
- `solver_rcond_estimate`, `solver_rgrowth`: Numerical health
- `near_singular`: Near-singularity warning

12.9.5 Summary Checklist

Use this checklist when you are sizing a deployment, not when you are debugging a same-handle race:

1. **Estimate per-model memory:** Use port count, history depth, and backend selection
2. **Plan for N x scaling:** Total memory approximately N x per-model footprint
3. **Consider NUMA:** On multi-socket systems, bind threads to NUMA nodes
4. **Monitor GPU memory:** Use `nvidia-smi` for GPU workloads
5. **Start with conservative limits:** Begin with moderate port counts and history depths
6. **Use diagnostics:** Monitor `solver_backend`, matrix metrics, and numerical health
7. **Profile scaling:** Test with your actual workload to verify scaling behavior

12.10 Multi-Model Deployment Patterns

Use this section when one model is no longer enough and you need to choose a deployment pattern: many independent models, one reused model for repeated sweeps, parallel copies for throughput, or separate models running at different rates.

> **Related Chapters** For one-handle ownership and shutdown rules, see Concurrency and Shutdown. For threading and memory scaling details, see Threading and Scaling. For variable time-stepping in multi-rate scenarios, see Variable Time-Step Integration.

Use these patterns when a deployment runs different TDSE models for different subsystems, or multiple copies of the same model for parameter sweeps.

12.10.1 Choose A Deployment Pattern

Situation	Recommended Pattern	Why
independent subsystems that can run side by side	N parallel independent models	simplest ownership story and highest operational clarity
same pack, repeated sweeps, memory is tight	batch sweep with one reused model	lowest memory footprint
same pack, repeated sweeps, throughput matters most	batch sweep with parallel copies	easiest way to scale wall-clock throughput
subsystems need different `dt` values	multi-rate coupling	keeps each model at the rate it actually needs

12.10.2 N Parallel Independent Models

This is the default production pattern. Each model has its own handle, its own state, and a clear owner.

```c
#define N_MODELS 8
tdse_model_t* models[N_MODELS];

for (int i = 0; i < N_MODELS; i++) {
    tdse_model_create_diagnostics_t diag = tdse_model_create_diagnostics_init();
    tdse_model_create(packs[i], pack_sizes[i], &diag, &models[i]);
}

/* Each worker thread owns one model */
#pragma omp parallel for
for (int i = 0; i < N_MODELS; i++) {
    for (uint64_t n = 0; n < nsteps; ++n) {
        tdse_step_begin(models[i], t[n], dt);
        tdse_step_op(models[i], &op);
        tdse_step_hr(models[i], hr);
        tdse_step_commit(models[i], primary[i]);
    }
}
```

Rules:

- Apply the one-handle rule from Concurrency and Shutdown: one worker owns one live handle at a time.
- Each handle has its own history buffer and state machine.
- Models may use different packs, different `dt`, or different backends.

12.10.3 Per-Model Resource Budget

Each model handle consumes its own resources. When planning a deployment:

Resource	Per Model	Shared
History ring buffer	yes (nh * nq * sizeof(double))	no
Operator workspace	yes (nq * np * sizeof(double))	no

Resource	Per Model	Shared
GPU stream/context	yes (when using CUDA backend)	GPU device memory is shared
CPU thread pool	configurable via `tdse_local_threads_set`	OS thread pool
Backend selection	per-model via `tdse_backend_set`	Backend registry

Use per-model resource controls only after ownership is already stable:

```
/* Assign CPU threads per model */
int threads_per_model = physical_cores / N_MODELS;
for (int i = 0; i < N_MODELS; i++) {
    tdse_local_threads_set(models[i], threads_per_model);
}
```

12.10.4 GPU Sharing Across Models

Multiple models can share the same GPU. Each gets its own CUDA stream, but device memory is shared.

Guidelines:

- Estimate total GPU memory as `sum(per_model_gpu_footprint)` + overhead (~50-100 MB).
- Monitor with `nvidia-smi` during initial deployment testing.
- If GPU allocation fails for any model, `tdse_model_create` returns `TDSE_ERR_OUT_OF_MEMORY`.
- Prefer the async pipeline mode for concurrent GPU models:

```
for (int i = 0; i < N_MODELS; i++) {
    tdse_cuda_backend_config_t cuda_cfg;
    tdse_cuda_backend_get_config(models[i], &cuda_cfg);
    cuda_cfg.pipeline_mode = TDSE_CUDA_PIPELINE_ASYNC;
    tdse_cuda_backend_set_config(models[i], &cuda_cfg);
}
```

12.10.5 Batch Sweep Pattern

Use this pattern when the mathematical model stays the same but inputs, operating points, or sweep values change.

For parameter sweeps where the same pack structure is reused with different inputs:

```
tdse_model_t* base_model;
tdse_model_create(pack, pack_size, &diag, &base_model);

/* Option A: Sequential reuse with reset between sweeps */
for (int sweep = 0; sweep < N_SWEEPS; sweep++) {
    for (uint64_t n = 0; n < nsteps; ++n) {
        tdse_step_begin(base_model, t[n], dt);
```

```
        /* ... solve with sweep-specific primary vectors ... */
        tdse_step_commit(base_model, primary_sweep[sweep]);
    }
    tdse_model_reset(base_model);  /* clear committed history for next sweep */
}
```

```
/* Option B: Parallel sweep with one model per sweep value */
tdse_model_t* sweep_models[N_SWEEPS];
for (int s = 0; s < N_SWEEPS; s++) {
    tdse_model_create(pack, pack_size, &diag, &sweep_models[s]);
}
```

Read the tradeoff plainly:

- Option A is memory-efficient and simpler to operate.
- Option B is throughput-efficient and easier to spread across workers or devices.
- If repeated sweeps are frequent but not latency-sensitive, start with Option A.

12.10.6 Multi-Rate Coupling

When different subsystems require different time resolutions, use separate models with different `model_dt` values and synchronize at coupling boundaries:

```
tdse_model_t* fast;  /* model_dt = 1 ns */
tdse_model_t* slow;  /* model_dt = 10 ns */

for (step = 0; step < TOTAL_STEPS; step++) {
    tdse_step_begin(fast, t_fast, 1e-9);
    /* ... step fast model ... */
    tdse_step_commit(fast, primary_fast);

    if (step % 10 == 0) {
        /* Extract coupling variables from fast model */
        /* ... */
        tdse_step_begin(slow, t_slow, 10e-9);
        /* ... step slow model with coupled inputs ... */
        tdse_step_commit(slow, primary_slow);
    }

    t_fast += 1e-9;
    t_slow = (step / 10) * 10e-9;
}
```

See Variable Time-Step Integration for more details on multi-rate patterns.

12.10.7 Deployment Checklist

☐ Estimate total memory: N * per-model footprint + shared overhead

☐ Assign thread resources: divide `local_threads` across models
☐ Select backend per model: CPU for small models, GPU for large ones
☐ Verify GPU memory budget if using CUDA backend
☐ Choose sweep strategy: sequential with reset vs. parallel copies
☐ Test scaling: run with 1, 2, 4, 8 models and measure throughput per model
☐ Monitor guard metrics on each model independently
☐ Keep shutdown ownership clear: destroy each model on its owning thread or wrapper path

CHAPTER 13

Profiler

> **Audience:** Integration engineers and performance engineers who need to benchmark convolution performance, explore IRC compression quality, or generate runtime plans.

Use this chapter when you want to measure TDSE performance on your hardware, compare execution choices, or generate a runtime plan you can apply in code.

Use this chapter to answer "how should I measure?" Then use Backend and Performance to answer "how should I interpret those results, choose a backend, and decide what evidence is strong enough for PoC or deployment?"

This split is intentional: Telemetry covers continuous runtime observability, while this chapter covers short, explicit measurement passes and runtime-plan generation.

13.0.1 What the Profiler Does

The TDSE Profiler is a CLI module (`tdse profiler`) that:

1. benchmarks convolution performance on your hardware
2. derives an optimal execution plan (backend selection, thread count, precision)
3. scans IRC compression parameters for the best speed/accuracy trade-off
4. produces a `runtime_plan` that can be passed directly to `tdse_backend_apply_plan()`

The profiler does not modify your pack. It measures and recommends; Runtime stays in charge of execution.

13.0.2 Quick Start

Minimal IRC scan with a synthetic lowpass input:

```
tdse profiler irc-scan \
  --np 8 --nh 257 \
  --scan-prefix-counts 16,32,64 \
  --scan-growth-values 2,4 \
  --scan-input lowpass \
  --scan-modes both
```

This scans all combinations of prefix counts `[16, 32, 64]` and growth values `[2, 4]` for both V1 and V2 IRC modes, using a built-in lowpass filter as the reference impulse response.

13.0.3 IRC Scan

13.0.3.1 What IRC Compression Does

Impulse Response Compression replaces a long impulse-response tail with a short prefix plus an exponential growth approximation. This reduces the per-step convolution cost from $O(P^2 \cdot nh)$ to $O(P^2 \cdot \text{prefix\_length})$ at the expense of a controllable accuracy loss.

When to use IRC:

- `nh` is large (hundreds or thousands of taps)
- the impulse response has a smooth exponential tail
- per-step latency matters more than absolute spectral accuracy

When not to use IRC:

- `nh` is small (under 50 taps)
- the impulse response has sharp features in the tail
- the application requires bit-exact convolution

13.0.3.2 Scan Parameters

Parameter	Meaning	Typical Values
`--np`	port count for the synthetic model	your model's `np`
`--nh`	history depth for the synthetic model	your model's `nh`
`--scan-prefix-counts`	comma-separated prefix lengths to test	`16,32,64,128`
`--scan-growth-values`	comma-separated growth factors to test	`2,4`
`--scan-input`	input source: `lowpass` or `trace`	`lowpass` for quick scan
`--scan-modes`	IRC mode(s): `v1`, `v2`, or `both`	`both`

13.0.3.3 IRC Modes

Mode	Description	Trade-off
V1	single-growth exponential tail	simpler, slightly less accurate
V2	piecewise growth with prefix	more accurate for complex tails

13.0.3.4 Output Formats

Flag	Output	Use When
`--out-md <path>`	Markdown summary report	human review
`--out-csv <path>`	CSV with per-combination metrics	spreadsheet analysis
`--out-recommend-json <path>`	machine-readable recommendation	feeding into build pipeline

13.0.3.5 Interpreting Results

The scan output includes per-combination metrics:

Metric	Meaning
`rel_rms`	relative RMS error of compressed vs full convolution
`peak_abs`	peak absolute error
`speedup`	speedup factor over uncompressed convolution

Metric	Meaning
`spec_rel_rms`	spectral-domain relative RMS (if spectral metrics enabled)
`spec_max_rel`	spectral-domain maximum relative error

A recommendation line appears if a feasible candidate exists:

```
RECOMMEND: mode=v2 prefix=64 growth=4 rel_rms=8.3e-4 speedup=3.2x
```

If no candidate meets all constraints:

```
NO_FEASIBLE: ... plus TOP1..TOP3 fallback rows
```

13.0.4 Trace Replay Scan

Instead of a synthetic lowpass, you can scan against a real simulation trace from your application. This produces more representative accuracy metrics.

13.0.4.1 CSV Input Format

Required columns: `t`, `x0`, `x1`, ..., `x{np-1}`

- One sample per row
- Time column must be strictly increasing
- Non-uniform intervals are allowed
- Missing or empty values are rejected

Example for `np=3`:

```
t,x0,x1,x2
0.000,1.0,0.0,0.5
0.001,0.95,0.05,0.48
0.002,0.90,0.10,0.46
```

13.0.4.2 Trace-Related Options

Option	Meaning	Default
`--scan-trace-csv <path>`	input CSV file (required with `--scan-input trace`)	-
`--scan-trace-time-col <name>`	time column name	`t`
`--scan-trace-port-prefix <prefix>`	port column prefix	`x` (matches `x0..x{np-1}`)
`--scan-trace-has-header 0|1`	CSV has header row	`1`
`--scan-trace-time-units s|ms|us|ns`	time unit conversion	`s`
`--scan-trace-use-all 0|1`	use full trace length	`1`
`--scan-trace-start-index <k>`	start from sample k	`0`
`--scan-trace-max-steps <n>`	cap number of steps	`0` (unbounded)

13.0.4.3 Example: Trace Replay with Autotune

```
tdse profiler irc-scan \
  --np 8 --nh 257 \
  --scan-prefix-counts 32,64,128 \
  --scan-growth-values 2,4 \
  --scan-input trace \
  --scan-trace-csv ./my_trace.csv \
  --scan-trace-time-col t \
  --scan-trace-port-prefix x \
  --scan-trace-has-header 1 \
  --scan-trace-time-units s \
  --scan-trace-use-all 1 \
  --scan-autotune 1 \
  --scan-target-rel-rms 1e-3 \
  --scan-objective speedup \
  --scan-prefer-mode either \
  --out-md ./irc_scan_report.md \
  --out-csv ./irc_scan_results.csv \
  --out-recommend-json ./irc_recommend.json
```

13.0.5 Autotune

Autotune automatically selects the best IRC parameters subject to accuracy constraints.

13.0.5.1 Options

Option	Meaning	Default
`--scan-autotune 0|1`	enable autotune	`0` (off)
`--scan-target-rel-rms <float>`	maximum acceptable `rel_rms`	`1e-3`
`--scan-target-peak-abs <float>`	maximum acceptable `peak_abs`	`0` (disabled)
`--scan-target-spec-rel-rms <float>`	maximum acceptable `spec_rel_rms`	`0` (disabled)
`--scan-target-spec-max-rel <float>`	maximum acceptable `spec_max_rel`	`0` (disabled)
`--scan-objective <type>`	optimization goal (see below)	`speedup`
`--scan-prefer-mode v1|v2|either`	preferred IRC mode	`either`

13.0.5.2 Objective Functions

Objective	Selects The Combination That…
`speedup`	maximizes speedup subject to accuracy constraints
`maxspeed`	maximizes raw speed regardless of nh reduction
`minnh`	minimizes effective nh subject to accuracy constraints
`minerror`	minimizes error subject to minimum speedup threshold

13.0.5.3 Output

When autotune succeeds:

```
RECOMMEND: mode=v2 prefix=64 growth=4 rel_rms=8.3e-4 speedup=3.2x
```

When no combination meets all constraints:

```
NO_FEASIBLE: target_rel_rms=1e-3, best_rel_rms=2.1e-3
TOP1: mode=v2 prefix=128 growth=4 rel_rms=2.1e-3 speedup=1.8x
TOP2: ...
TOP3: ...
```

Use the fallback rows to decide whether to relax constraints or accept the best available option.

13.0.6 Spectral Metrics

Enable frequency-domain consistency checking during scan:

```
tdse profiler irc-scan \
  --np 8 --nh 257 \
  --scan-prefix-counts 32,64,128 \
  --scan-growth-values 2,4 \
  --scan-input lowpass \
  --scan-lowpass-alpha 0.95 \
  --scan-spectral 1 \
  --scan-spectral-nfreq 128 \
  --scan-spectral-fmin 0 \
  --scan-spectral-fmax 0 \
  --scan-spectral-weighting lowfreq
```

13.0.6.1 Spectral Options

Option	Meaning	Default
`--scan-spectral 0|1`	enable spectral metrics	`0`
`--scan-spectral-nfreq <int>`	number of frequency sample points	`128`
`--scan-spectral-fmin <Hz>`	lower frequency bound	`0`
`--scan-spectral-fmax <Hz>`	upper frequency bound (auto-derives from tau if <=fmin)	`0`
`--scan-spectral-weighting flat|lowfreq|band`	weighting for `spec_rel_rms`	`flat`
`--scan-spectral-band f1,f2`	active frequency band when weighting is `band`	-

13.0.6.2 Metric Definitions

The spectral metrics compare the original impulse response `H(w)` against the compressed version `H'(w)`:

- `H(w) = sum_k h[k] * exp(-j*w*tau[k])` (NUDFT/DTFT on the configured grid)
- `rel_f = ||H(w) - H'(w)||_F / (||H(w)||_F + eps)`
- `spec_rel_rms = sqrt(mean_w(rel_f^2))`
- `spec_max_rel = max_w(rel_f)`

The weighting parameter controls how `spec_rel_rms` aggregates across frequency:

- `flat`: equal weight at all frequencies
- `lowfreq`: heavier weight at low frequencies (typical for power systems)
- `band f1,f2`: only frequencies in `[f1, f2]` contribute

13.0.7 Applying Profiler Output to Runtime

The profiler produces a `runtime_plan` block compatible with `tdse_backend_apply_plan()`.

13.0.7.1 Using the Recommendation JSON

1. Run the profiler with `--out-recommend-json plan.json`
2. Load the JSON in your build pipeline
3. At runtime, apply the plan:

```
tdse_backend_apply_plan(model, plan_json, plan_len);
```

The plan JSON contains backend selection, thread count, and precision settings derived from the profiler's benchmarks.

13.0.7.2 Manual Application

If you prefer to apply individual settings:

```
tdse_backend_set(model, backend_id);
tdse_local_threads_set(model, thread_count);
tdse_compute_precision_set(model, precision);
```

See Backend and Performance and Runtime API Summary for the full backend and performance API.

13.0.8 CLI Smoke Test

Verify the profiler CLI is working:

```
tdse profiler irc-scan \
  --np 2 --nh 16 \
  --scan-prefix-counts 4,8 \
  --scan-growth-values 2 \
  --scan-input lowpass \
  --out-md /dev/stdout
```

This should complete in under a second and print a Markdown table of scan results.

CHAPTER 14

Backend Selection and Performance

> **Audience:** Integration engineers choosing execution backends for production deployments, and anyone tuning GPU or CPU performance for large models.

Use this chapter when you are choosing an execution backend for deployment or trying to improve throughput on existing hardware. It shows how to discover available backends, pick the right one for a model, and tune the settings that matter most.

For best results, read this chapter after Profiler. The Profiler tells you how to measure your real workload; this chapter tells you how to interpret the measurements, pick a backend policy, and separate sizing guidance from stronger deployment evidence.

14.1 Backend Overview

TDSE runtime backends control how convolution and linear algebra operations are executed. The backend is selected **per model** before the first step.

14.1.1 Discovering Available Backends

```
uint32_t count = tdse_backend_registry_count();
for (uint32_t i = 0; i < count; ++i) {
    tdse_backend_capability_t cap;
    tdse_backend_registry_get(i, &cap);
    printf("backend %u: %s (np_max=%zu, cuda=%d)\n",
           i, tdse_backend_id_name(cap.id), (size_t)cap.max_np, cap.has_cuda);
}
```

14.1.2 Backend Identifier Reference

Backend ID	Description	Best For
`CPU_GENERIC`	portable CPU backend	small models, any hardware
`CPU_BLAS`	BLAS-accelerated CPU	medium models with OpenBLAS/MKL
`CPU_BLAS_SPARSE`	sparse BLAS CPU	large sparse models

Backend ID	Description	Best For
`CUDA`	NVIDIA GPU backend	large models, high throughput

Not all backends are available in every build. Use the registry API to check what is compiled in.

14.1.3 Setting the Backend

```
tdse_backend_id_t id = tdse_backend_id_from_name("CPU_BLAS");
tdse_status_t st = tdse_backend_set(model, id);
```

Important: `tdse_backend_set()` must be called before the first successful `tdse_step_begin()`. After the first step begins, configuration is frozen and `tdse_backend_set()` returns `TDSE_EXT_STATUS_UNSUPPORTED`.

Query the active backend at any time:

```
tdse_backend_id_t active = tdse_backend_get_active(model);
```

14.2 Runtime Plans

For repeatable deployments, prefer `tdse_backend_apply_plan()` over manual backend selection. A runtime plan is a JSON document that captures backend selection and optional per-scenario overrides in one place.

14.2.1 Applying a Plan

```
tdse_backend_apply_plan(model, plan_json, plan_json_len);
```

The plan JSON structure:

```
{
  "default": {
    "backend": "CPU_BLAS",
    "local_threads": 4,
    "compute_precision": "fp64"
  },
  "scenarios": {
    "large_np": {
      "backend": "CUDA",
      "cuda_config": {
        "pipeline_mode": "async"
      }
    }
  }
}
```

The `default` section applies to all models. The `scenarios` section provides named overrides that the host can activate when a model or workload needs a different policy.

Plans are typically generated by the TDSE Profiler. See Profiler for details.

14.3 CUDA Configuration

When the CUDA backend is available, configure per-model options:

```
tdse_cuda_backend_config_t cuda_cfg;
tdse_cuda_backend_get_config(model, &cuda_cfg);
cuda_cfg.pipeline_mode = TDSE_CUDA_PIPELINE_ASYNC;
tdse_cuda_backend_set_config(model, &cuda_cfg);
```

14.3.1 Pipeline Modes

Mode	Description	When to Use
sync	synchronous host-device transfer	debugging, small models
async	overlapping compute and transfer	production, large models

14.3.2 GPU Memory Management

If GPU allocation fails, `tdse_model_create()` or step APIs return `TDSE_ERR_OUT_OF_MEMORY`. Monitor GPU memory with `nvidia-smi`.

For multi-model GPU deployment, resource sharing, and memory planning, see the multi-model deployment patterns in Multi-Model Deployment.

14.3.3 GPU Recommendations

- Prefer GPU for models with `np > 10` and `nh > 100`
- Small models may not amortize the CPU-GPU transfer overhead
- Limit concurrent GPU models if memory is constrained

14.4 Compute Precision

Control the floating-point precision used for history convolution:

```
tdse_compute_precision_set(model, TDSE_COMPUTE_PRECISION_FP32);
```

Precision	Description	Impact
FP64	double precision (default)	highest accuracy, moderate performance
FP32	single precision	faster, reduced accuracy in history term

When to use FP32: large `nh` where the history term dominates step cost and the application tolerates reduced mantissa precision in the history accumulation.

When to stay with FP64: stiff systems, high-frequency dynamics, or when bit-exact reproducibility is required across hardware.

Query current setting:

```
tdse_compute_precision_t prec = tdse_compute_precision_get(model);
```

14.5 Thread Control

Override the number of CPU threads used internally by the runtime:

```
tdse_local_threads_set(model, 4);
```

By default, the runtime uses the number of logical CPU cores reported by `tdse_ext_runtime_logical_cores()`.

Guidelines:

- For single-model workflows, set `local_threads` to the number of physical cores
- For N-model parallel workflows, divide available cores across models
- Setting `local_threads` higher than available cores provides no benefit

14.6 Backend Selection Guide

```
How large is your model (np)?
├─ np < 10
│  └─ CPU_GENERIC or CPU_BLAS
│     └─ GPU overhead exceeds benefit
├─ np 10-100
│  └─ CPU_BLAS (with OpenBLAS/MKL)
│     └─ Consider CUDA if throughput matters
└─ np > 100
   └─ CUDA (if available) or CPU_BLAS_SPARSE
      └─ GPU transfer overhead is amortized
```

Additional factors:

Factor	Backend Impact
Sparse matrix structure	CPU_BLAS_SPARSE may outperform dense even at moderate np
Multiple models in parallel	Each gets its own handle; divide CPU threads or GPU memory
Variable dt	Fast-path backends (CPU_BLAS, CUDA) optimize for uniform stepping
Pack size	Large nh increases convolution cost; GPU benefits more

14.7 Build Features

Check what features the current build was compiled with:

```c
/* Query required buffer size */
size_t json_len = 0;
tdse_perf_get_build_features_json(NULL, &json_len);

/* Allocate and query */
char* json = malloc(json_len);
tdse_perf_get_build_features_json(json, &json_len);
printf("Features: %s\n", json);
free(json);
```

This returns a JSON object listing compiled-in features such as CUDA support, BLAS backend, telemetry, and other optional components.

14.8 Performance Benchmarks

The numbers below are sizing guidance, not guarantees. They help you estimate whether TDSE is in the right range for your deployment before you run your own measurements. Unless noted otherwise, all data uses the default CPU backend (`CPU_GENERIC`) on a representative x86_64 workstation.

Treat this section as early sizing guidance, not release evidence. Procurement, PoC, and real-time sign-off decisions should be based on measurements from your target machine, target pack shape, and target host-loop policy.

Use the numbers below for triage questions such as “is CPU enough?”, “is CUDA worth testing?”, or “is this pack likely to fit in memory?”. Do not use them as substitutes for customer acceptance criteria, target-machine qualification, or published product guarantees.

14.8.1 Representative Step Latency

Measurements with `nh = 256`, `dt = 1e-6`, and 10,000 warm-up steps followed by 10,000 measured steps:

np	nq	Backend	Step Latency (µs)	Throughput (steps/s)
1	1	CPU_GENERIC	0.3-0.8	1.2M-3.3M
3	3	CPU_GENERIC	0.5-1.5	670K-2.0M
3	3	CPU_BLAS	0.4-1.2	830K-2.5M
10	10	CPU_GENERIC	2-8	125K-500K
10	10	CPU_BLAS	1-4	250K-1.0M
10	10	CUDA	5-15*	67K-200K
50	50	CPU_BLAS	20-80	12K-50K
50	50	CUDA	8-25	40K-125K
100	100	CPU_BLAS	100-400	2.5K-10K
100	100	CUDA	15-50	20K-67K

*CUDA numbers include host-device transfer overhead. Small models may not benefit from GPU due to transfer latency.

14.8.2 Memory Footprint

Approximate per-model memory usage:

np	nh	Dense Memory	Notes
3	256	~50 KB	Small model, typical transmission line
10	1024	~2 MB	Medium model, multi-port subsystem
50	2048	~80 MB	Large model, distribution network
100	4096	~600 MB	Very large, dense subsystem

Memory scales approximately as `nh * nq * np * 8 bytes` for the H tensor plus `nq * np * 8 bytes` for workspace.

14.8.3 Scaling Behavior

- **Linear in nh:** doubling history depth approximately doubles per-step time
- **Quadratic in np:** port count has the strongest impact; minimize ports where possible
- **Linear in model count:** N independent models consume approximately N times the memory and can run in parallel on separate threads

14.8.4 Benchmarking Your Workload

To measure performance for your specific model:

```
tdse profiler calibrate --np <your_np> --nh <your_nh> --dtype 64 \
  --out-json ./profile.json --out-md ./profile.md
```

The profiler report includes:

- per-step latency for each available backend
- optimal backend recommendation
- generated runtime plan for `tdse_backend_apply_plan()`

For real-time deployments, also measure WCET with deterministic mode enabled (see Platform Notes).

For procurement, PoC, or customer-facing reporting, keep three evidence classes separate:

- sizing guidance from this chapter
- profiler output for your exact pack and hardware
- target-machine timing or field qualification records from your deployment program

When sharing performance numbers outside the immediate engineering team, record at least:

- CPU and GPU model
- operating system and compiler/toolchain
- build type and enabled backend/features
- model shape (`np`, `nq`, `nh`, `dt`)
- whether the host used fixed-step, variable-step, single-model, or multi-model execution

- whether telemetry, deterministic mode, or additional tracing was enabled during measurement

14.9 Performance Monitoring Checklist

☐ Verify backend selection with `tdse_backend_get_active()` after set
☐ Compare step latency between backends for your model
☐ Set `local_threads` intentionally rather than relying on default
☐ Monitor guard metrics when tuning precision or dt strategy
☐ Use Profiler to derive an optimal plan rather than manual tuning
☐ Archive the runtime plan alongside pack artifacts for release evidence

14.10 Before Deployment Sign-Off

Use the next chapter based on the kind of risk you are trying to close:

If the open question is...	Go next
target platform support or qualification boundary	Platform Notes
plugin deployment, manifest, or ABI compatibility	Plugin System
circuit-input subset or netlist support risk	Element Reference
long-running observability rather than benchmark measurement	Telemetry

CHAPTER 15

Telemetry

> **Applies to:** TDSE SDK 1.0.0-rc1 with `TDSE_ENABLE_TELEMETRY=ON`.

Use this chapter when you need runtime observability: enable telemetry, attach it to models, export events, and decide when continuous event collection is a better fit than a short measurement pass.

For RC evaluation, decide three things before you wire telemetry into a host:

- whether the delivered evaluation package actually includes telemetry support
- whether you need continuous observability or a short profiler-driven sizing pass
- whether the exported data is engineering evidence for debugging, or part of a formal qualification record you manage in your own host program

Profiler coverage is intentionally split into the next chapter, Profiler. Use this chapter for always-on or long-running observability. Use the Profiler chapter when the question is measurement methodology, backend comparison, or runtime-plan generation.

Need	Start Here	Why
capture a steady operational trace	this chapter	telemetry emits ongoing runtime events
benchmark one model or compare backend choices	Profiler	profiler is the measurement tool
generate a runtime plan to apply in code	Profiler	runtime plans are profiler outputs
keep debugging evidence from long runs	this chapter	telemetry is built for persistent observability

15.1 What Telemetry Provides

TDSE telemetry adds observability to runtime step execution without changing numerical behavior. When enabled, the runtime automatically emits step-level events from `tdse_step_begin`, `tdse_step_op`, `tdse_step_hr`, `tdse_step_ir`, and `tdse_step_commit`.

Use telemetry for long-running or production-like runs where you want a steady record of step activity. Use the Profiler when you want a focused performance investigation, backend comparison, or runtime-plan session.

Typical uses:

- measure per-step latency and throughput in production simulations
- detect performance regressions across SDK upgrades
- feed step-level metrics into existing monitoring infrastructure
- record resource usage alongside simulation progress

Telemetry is compiled out by default and has zero runtime cost when disabled.

15.2 Package Availability

Telemetry is package-variant-specific in the RC line. Before wiring the API, confirm that your delivered evaluation package or delivery notes explicitly say telemetry support is enabled (`TDSE_ENABLE_TELEMETRY=ON`).

If telemetry is not enabled in the delivered package, treat that as a package variant choice, not as a runtime misconfiguration in your integration.

15.3 Two-Step Lifecycle

Telemetry follows a process-global service + per-model attachment pattern.

15.3.1 Step 1: Initialize the Service (once per process)

```
#include <tdse/tdse_telemetry.h>

tdse_telemetry_service_config_t service_cfg;
tdse_telemetry_service_config_init(&service_cfg);
service_cfg.json_output_path = "tdse_telemetry.json";
service_cfg.worker_sleep_ms = 10;
tdse_telemetry_service_init(&service_cfg);
```

`tdse_telemetry_service_config_t` fields:

Field	Meaning	Recommended Default
`json_output_path`	file path for JSON lines export	`"tdse_telemetry.json"`
`worker_sleep_ms`	background worker poll interval in milliseconds	`10`

Call `tdse_telemetry_service_init()` once at process startup, before creating any models. Call `tdse_telemetry_service_is_initialized()` to check status.

15.3.2 Step 2: Attach Each Model

```
tdse_telemetry_model_config_t model_cfg;
tdse_telemetry_model_config_init(&model_cfg);
model_cfg.sampling_interval_steps = 1;
model_cfg.ring_capacity = 1024;
tdse_model_telemetry_attach(model, &model_cfg);
```

`tdse_telemetry_model_config_t` fields:

Field	Meaning	Recommended Default
`sampling_interval_steps`	record every N-th step	1 (every step)
`ring_capacity`	number of events retained in the ring buffer	`1024`

Set `sampling_interval_steps` to a larger value (e.g., 10 or 100) to reduce overhead in long-running simulations where per-step detail is not required.

Telemetry is designed for observability, not for zero-overhead measurement. When you are trying to prove peak throughput, strict WCET, or backend selection, use the Profiler first and add telemetry only if you also need a persistent operational trace.

15.3.3 Detach and Shutdown

```
/* per model */
tdse_model_telemetry_detach(model);

/* per process */
tdse_telemetry_service_shutdown();
```

Detach before destroying the model. Shutdown after all models are detached.

15.4 Event Reference

15.4.1 Telemetry Levels

Set per-model via `tdse_model_telemetry_set_level()`:

Level	Behavior
minimal	lifecycle events only (create, destroy)
standard	adds step-level timing
verbose	adds per-query breakdown (op, hr, ir, commit, dr)

15.4.2 Event Kinds

Events are typed via `tdse_telemetry_event_kind_t`. Key kinds emitted automatically by the runtime:

Kind	When Emitted	Payload
step begin	`tdse_step_begin()`	`t`, `dt`
step commit	`tdse_step_commit()`	`committed_steps`
model lifecycle	create / destroy	handle metadata

15.4.3 Extended Statistics

Query accumulated statistics at any time:

```
tdse_telemetry_model_stats_t stats;
tdse_model_telemetry_get_stats(model, &stats);

tdse_telemetry_model_extended_stats_t ext;
tdse_model_telemetry_get_extended_stats(model, &ext);
```

15.4.4 Custom Tags

Attach key-value tags for correlation in multi-model deployments:

```
tdse_model_telemetry_set_custom_tags(model, "circuit=my_netlist,run=42");
```

15.4.5 Instance IDs

Each attached model receives a unique instance ID for log correlation:

```
uint64_t id = tdse_model_telemetry_get_instance_id(model);
```

15.5 Exporters

15.5.1 JSON Lines (built-in)

Always active when the service is initialized. Events are written as JSON lines to `json_output_path`. Each line is a self-contained JSON object.

15.5.2 OpenTelemetry

Export to an OTLP-compatible endpoint:

```
tdse_telemetry_export_opentelemetry("http://127.0.0.1:4318");
```

Security note: the OpenTelemetry exporter only accepts loopback or private targets by default (e.g., `http://127.0.0.1:4318`). This is intentional to prevent accidental data exposure.

HTTPS export on non-Windows builds requires OpenSSL to be found at configure time.

15.5.3 Prometheus

Export to a Prometheus push gateway:

```
tdse_telemetry_export_prometheus("http://127.0.0.1:9091");
```

Same loopback/private restriction applies.

15.5.4 What Telemetry Does Not Prove

Telemetry can help you preserve evidence, correlate incidents, and compare runs under the same host policy. It does not by itself prove:

- release qualification on a new host platform
- WCET or target-machine timing acceptance for RT/HIL
- correctness of a runtime-plan or backend recommendation
- support for optional accelerators that were not included in the delivered package

15.5.5 Health Status

Check service health at any time:

```
tdse_telemetry_health_status_t health;
tdse_telemetry_get_health_status(&health);
```

15.5.6 System Metrics

Record or query system-level metrics:

```
tdse_telemetry_system_metrics_t sys;
tdse_telemetry_get_system_metrics(&sys);
tdse_telemetry_record_system_metrics();
tdse_telemetry_record_memory_usage();
tdse_telemetry_record_gpu_usage();
```

15.5.7 Performance Alerts

Record custom performance alerts:

```
tdse_telemetry_record_performance_alert(model, TDSE_TELEMETRY_ALERT_SEVERITY_WARNING,
    "step_latency_exceeded", 150.0);
```

Alert severities:

Severity	Use When
info	informational note
warning	performance degraded but tolerable
error	performance issue requiring attention
critical	simulation may be invalid

15.5.8 Backend Switch Events

The runtime automatically records backend switch events. You can also record custom events:

```
tdse_telemetry_record_backend_switch(model, old_backend_id, new_backend_id);
```

15.5.9 Flush

Force-flush pending events:

```
tdse_telemetry_flush_events();
```

15.6 Complete Example

```
#include <tdse/tdse.h>
#include <tdse/tdse_telemetry.h>

void run_with_telemetry(tdse_model_t* model, size_t nsteps) {
    /* Service is assumed to be initialized before this function. */

    tdse_telemetry_model_config_t model_cfg;
    tdse_telemetry_model_config_init(&model_cfg);
    model_cfg.sampling_interval_steps = 1;
    model_cfg.ring_capacity = 2048;
    tdse_model_telemetry_attach(model, &model_cfg);

    tdse_model_telemetry_set_custom_tags(model, "scenario=baseline");
    tdse_model_telemetry_set_level(model, TDSE_TELEMETRY_LEVEL_VERBOSE);

    /* Normal step loop */
    for (size_t n = 0; n < nsteps; ++n) {
        tdse_step_begin(model, n * 0.001, 0.001);
        /* op, hr, ir, solve, commit */
        tdse_step_commit(model, primary);
    }

    tdse_model_telemetry_detach(model);
}
```

15.7 Production Deployment Checklist

☐ Confirm the delivered RC package enables telemetry (`TDSE_ENABLE_TELEMETRY=ON`)
☐ Call `tdse_telemetry_service_init()` once at process startup
☐ Attach each model before stepping
☐ Set `sampling_interval_steps` intentionally (1 for dev, higher for prod)
☐ Configure exporter endpoints (JSON always active; OTLP/Prom optional)
☐ Ensure exporter targets are loopback/private or explicitly authorized
☐ On Linux, OpenSSL is available if HTTPS export is needed
☐ Detach before destroy, shutdown after all models are detached
☐ Verify `tdse_telemetry_service_is_initialized()` returns true before attaching
☐ Archive telemetry JSON alongside simulation results for post-hoc analysis

CHAPTER 16

Platform Notes

Use this appendix when you need to answer platform questions: what Linux configurations are qualified, how installed-package validation is established, and what is planned for ARM64.

This is not part of the shortest bring-up path. Read it when the real question becomes support boundary, qualification boundary, deployment risk, or target platform scope.

Read it in two passes: support boundary first, then the smallest validation or acceptance lane that matches the claim you need to make.

For integration work, this appendix usually answers one of these questions:

- is this host/platform combination inside the current support boundary
- what installed-package shape has already been validated
- what additional qualification is still the customer's responsibility, especially for RT/HIL

Read the status words in this appendix literally:

- `supported` means part of the current RC delivery and support boundary
- `qualified` means TDSE release validation has exercised that slice, often as an optional feature or lane
- `roadmap` means planned or actively explored, but not part of the current RC commitment
- `customer qualification` means the evaluation package may be usable there, but the final proof still belongs to the host program, target machine, or deployment process

Use the next two boundary chapters together with this one:

- use Plugin System when deployment risk becomes ABI, manifest, routing, or plugin-health evidence
- use Element Reference when deployment risk becomes source-deck coverage, unsupported directives, or netlist subset drift

16.1 Linux Support Scope

16.1.1 What Is Covered

Current Linux qualification covers the full SDK:

- Runtime library, pack consumption, model create, lifecycle, step execution, and diagnostics
- Builder transforms and pack write (`libtdse_builder`)
- Adapter Circuit, the circuit-domain adapter module (CPU dense + sparse KLU via SuiteSparse)

- CLI (`tdse` binary) and Profiler
- BLAS acceleration via OpenBLAS and LAPACK
- Telemetry with optional OpenSSL HTTPS export
- `pkg-config` and CMake package metadata
- Linux tarball, DEB, and RPM package forms for the qualified host profile

Optional accelerator stacks that require separate qualification:

- CUDA Adapter and CUDA Runtime backend (qualified on CUDA 12/13)
- Intel oneMKL BLAS backend and MKL Pardiso sparse solver

16.1.2 Platform Support Matrix

Platform slice	Status	Notes
Windows	supported	primary supported SDK platform and release flow
Linux / WSL baseline	supported	full SDK parity: Runtime, Builder, Adapter Circuit, CLI, Profiler
Linux optional accelerators	qualified	BLAS (OpenBLAS), LAPACK, KLU are validated; CUDA qualified on 12/13; MKL qualified separately

16.1.3 Supported Host Configuration

Current standard Linux support target is:

Dimension	Supported Baseline
CPU / OS class	`x86_64` GNU/Linux
libc	glibc-based userland
tested on	`ubuntu-24.04`
qualification baseline	installed SDK package consumed on the qualified host profile
build configuration	release-candidate package contents
dependencies	default dependencies only; optional BLAS/LAPACK/KLU/CUDA stacks not required
toolchain family	GCC-based baseline; broader compiler confidence may come from additional qualification
package form	Full SDK tarball, RuntimeCore tarball, DEB/RPM system packages, and installed `pkg-config`/CMake exports produced by the SDK release process

The following are not covered by the standard support scope:

- musl-based distributions
- `arm64` or `aarch64` (on roadmap, see ARM64 section below)
- static-link redistribution
- Intel oneMKL BLAS backend (requires separate MKL installation and qualification)

Configurations outside this table may still work but are not covered by the standard support scope unless documented in a release note.

16.1.4 Validation Coverage

Each TDSE Linux RC is validated on the supported host profile through installed-package consumption, release-build checks, threading stress tests, long-duration soak tests, and install/package smoke checks. Additional release-engineering lanes may use broader internal coverage, but the customer-visible claim is anchored to the delivered package on the qualified host profile.

16.1.5 Supported Usage

A supported Linux deployment means the application consumes the shipped SDK package or bundle through `pkg-config` or CMake exports, validates packs before model create, uses the standard create-diagnostics flow, runs CLI and Profiler commands cleanly on the qualified host profile, and passes the included tests for the current release.

16.1.6 Runtime Requirements

The Linux runtime expects one handle per execution thread, no same-handle concurrent step entry, and validated pack bytes as input. Adapter Circuit (circuit-domain adapter), Builder, and CLI are fully supported on Linux. CUDA GPU acceleration requires a compatible NVIDIA driver and CUDA toolkit.

16.1.7 Reporting Issues

When reporting a Linux RuntimeCore issue, please collect:

- package version
- package format used
- `tdse_pack_validate` output
- `tdse_model_create_diagnostics_t`
- `tdse_model_info(...)`
- `tdse_model_state_info(...)`
- `tdse_model_last_error_info(...)`
- compiler, distro, and glibc baseline

16.1.8 Issue Categories

The following are treated as product defects: a qualified configuration regresses on unchanged inputs, install or export smoke fails on the supported host profile, or `tdse_model_create(...)` fails with a previously qualified pack.

The following are outside the standard support scope and may require a separate qualification agreement: unsupported distros or host classes (musl, arm64), optional dependency stacks outside the current scope, or same-handle concurrent stepping.

16.2 Linux/WSL Validation Guide

This section summarizes the Linux validation lanes used to establish RC confidence. It is release-validation context, not the primary path for a closed-source SDK evaluation package.

16.2.1 Supported Workflow

- For normal customer evaluation, stay on the installed-package path from Installation and Getting Started.
- RC qualification evidence is organized into `default`, `extended`, and `release` validation lanes.
- Escalate from `default` to `extended` when the question becomes install-tree or package consumption.
- Escalate to `release` when the question becomes release-candidate confidence or support evidence.

16.2.2 CTest Tiers

The Linux release process treats validation as lanes with different purposes. Use the smallest lane that answers your current question.

Tier	When to use it	What it emphasizes
`default`	day-to-day development	core unit/integration correctness without install/package burden
`extended`	install-tree or package validation	install smoke, RuntimeCore/full-SDK package consumption, Linux package artifact checks
`release`	release-candidate confidence	randomized CLI smoke, clean-host consumption, ABI/provenance/release evidence gates
`all`	pre-merge or final qualification on a prepared machine	runs the three lanes in sequence

Use `default` unless you are explicitly validating an install tree, a package artifact, or a release candidate.

16.2.3 Prerequisites

These lanes depend on the TDSE release-validation environment. Closed-source RC consumers do not need to reproduce them just to evaluate the installed SDK package.

16.2.4 Install And Package Layout

Linux installs are intentionally prefix-relative.

In a standard Linux install under `/opt/tdse-sdk`, the important paths are:

- `/opt/tdse-sdk/bin/tdse`
- `/opt/tdse-sdk/lib/libtdse.so`
- `/opt/tdse-sdk/lib/libtdse_builder.a`
- `/opt/tdse-sdk/lib/libtdse_adapter_circuit.a`
- `/opt/tdse-sdk/lib/plugins/sim/libtdse_sim_cpu_dense.so`
- `/opt/tdse-sdk/lib/plugins/sim/libtdse_sim_cpu_sparse.so`
- `/opt/tdse-sdk/lib/plugins/sim/libtdse_sim_cuda.so`
- `/opt/tdse-sdk/lib/plugins/sim/plugin_manifest.json`
- `/opt/tdse-sdk/lib/pkgconfig/tdse.pc`
- `/opt/tdse-sdk/lib/pkgconfig/tdse-runtimecore.pc`
- `/opt/tdse-sdk/lib/cmake/tdse/`

- /opt/tdse-sdk/lib/cmake/tdseRuntimeCore/
- /opt/tdse-sdk/share/tdse/tdse_sdk_variant.json
- /opt/tdse-sdk/share/tdse/tdse_runtimecore_variant.json

Use the installed CMake package:

```
cmake -S <consumer-source> \
  -B build/package-consumer-cpu \
  -DCMAKE_PREFIX_PATH=/opt/tdse-sdk \
  -DTDSE_ADAPTER_TARGETS=AUTO
cmake --build build/package-consumer-cpu
LD_LIBRARY_PATH=/opt/tdse-sdk/lib:${LD_LIBRARY_PATH} \
  build/package-consumer-cpu/<consumer-binary>
```

Use the installed `pkg-config` package:

```
export PKG_CONFIG_PATH=/opt/tdse-sdk/lib/pkgconfig:${PKG_CONFIG_PATH}
c++ -std=c++20 app.cpp -o app $(pkg-config --cflags --libs tdse)
```

Optional feature slices:

- sparse CPU: append `$(pkg-config --variable=sparse_libs tdse)`
- CUDA: append `$(pkg-config --variable=cuda_libs tdse)`

16.2.5 Linux Package Targets

The SDK release process produces tarball, DEB, and RPM packages for Linux deployment.

For customers, the important point is not the internal artifact staging path but the validation outcome: tarballs are checked for relocation, installed `pkg-config`/CMake metadata is checked for downstream discovery, and `.deb`/`.rpm` payloads are checked for completeness.

16.2.6 Acceptance Shapes

The release process validates more than one package-consumption shape. From a customer-handbook perspective, the important ones are:

Shape	What it proves
RuntimeCore install	a Runtime-only consumer can link and run against `tdse::tdse`
Full SDK install	Runtime, Builder, Adapter, CLI, and package metadata work together from the installed tree
Clean-host CPU-only consumer	a downstream machine without optional accelerators can still consume the supported CPU slice
Clean-host full-feature consumer	installed-package discovery and optional feature wiring behave as shipped

Use the smallest acceptance shape that matches the support claim you need to make. Do not infer clean-host or full-feature release confidence from a same-host default build alone.

This is why the handbook points you to installed-package examples instead of relying on same-host validation alone: the install tree itself is part of the supported surface.

16.2.7 Notes

- The `default` configuration is the supported day-to-day Linux baseline.
- RC consumers should treat the installed package as the supported surface. Engineering-only validation lanes may explain how confidence was established, but they are not required reading for first evaluation.

16.2.8 Troubleshooting

- Missing compiler or linker tools: Install `build-essential`.
- `pkg-config` consumer cannot find TDSE: Add `<prefix>/lib/pkgconfig` to `PKG_CONFIG_PATH` before building the downstream app.
- CMake configure succeeds but runtime install smoke cannot find shared libraries: Re-run the release team's `extended` Linux validation lane; that lane already sets the needed runtime environment for its smoke checks.
- Linux package targets fail to produce `.rpm`: Install the host `rpm` toolchain so `rpmbuild` is available.
- Stale build state after large branch changes: Clean the local build directory and configure again.
- You need a plan for ARM64 or Apple Silicon: Start with the ARM64/AArch64 roadmap section below.

16.3 Linux ARM64 / AArch64 Roadmap

16.3.1 Current Status

TDSE Linux support is currently qualified for `x86_64` on `ubuntu-24.04`. That means `arm64` / `aarch64` is not part of the standard supported host profile today.

At the same time, ARM64 matters for:

- Apple Silicon developer laptops running Linux VMs or containerized validation environments
- AWS Graviton build and simulation fleets
- on-prem Linux clusters that are gradually adding ARM64 capacity

Slice	Status	Notes
`x86_64` GNU/Linux	supported	current Linux qualification baseline
Apple Silicon macOS host running Linux VM/container	roadmap	useful as a developer and validation host, not yet a supported product target
native `arm64` / `aarch64` GNU/Linux	roadmap	no validation, package, or release qualification yet

16.3.2 Phase Plan

16.3.2.1 Phase 1: Toolchain Bring-Up

Exit criteria:

- configure + build succeed on native `aarch64` GNU/Linux
- qualification tests pass with default dependencies

- RuntimeCore install + relocation smoke pass
- `pkg-config` and CMake package consumers both compile and run

Target hosts:

- AWS Graviton runner or equivalent native `aarch64` Linux host
- Apple Silicon developer host using an Ubuntu ARM64 VM or container

16.3.2.2 Phase 2: Linux Package Parity

Exit criteria:

- ARM64 tarballs are generated in the same layout as `x86_64`
- ARM64 `.deb` and `.rpm` packages are generated in the validation pipeline
- package artifact smoke validates tarball, `pkg-config`, and CMake consumption on ARM64

Target hosts:

- `ubuntu-24.04` ARM64 runner
- at least one non-Ubuntu validation host for package-install sanity

16.3.2.3 Phase 3: Performance And Support Qualification

Exit criteria:

- threading stress, sanitizer, and soak tests are green on ARM64
- package notes document any architecture-specific performance or backend limits
- Linux support scope is expanded to name the qualified ARM64 host profile explicitly

16.3.3 Known Technical Risks

- SIMD paths are currently tuned around x86 feature detection and may need NEON-aware follow-up work
- optional CUDA paths are not part of the initial ARM64 support claim
- external dependency availability can differ across distros, especially SuiteSparse/KLU packaging
- Apple Silicon developer convenience does not automatically imply macOS product support

16.3.4 Current ARM64 Position

Until Phase 3 is complete, the current ARM64 position is:

- ARM64/AArch64 enablement is on the public roadmap
- Linux support today is qualified on `x86_64` GNU/Linux only
- Apple Silicon is a development-host convenience story, not a separate supported runtime platform

16.3.5 What Completion Looks Like

This roadmap will be considered complete when all of the following are true:

- ARM64 validation infrastructure exists
- install/package smoke exists for ARM64

- Linux support scope names the ARM64 host profile directly
- the handbook and package-consumer paths treat ARM64 Linux as an ordinary Linux package consumer path

---

16.4 Real-Time and HIL Deployment

Use this section when TDSE must meet a real-time or hardware-in-the-loop timing budget, not just run correctly offline.

Keep the host/TDSE boundary simple in real-time deployments:

- the host owns the wall-clock scheduler, solver sequencing, and any external I/O or device synchronization
- TDSE exchanges model data only at step boundaries through `tdse_step_begin(...)`, `tdse_step_op(...)`, `tdse_step_hr(...)`, `tdse_step_ir(...)`, `tdse_step_commit(...)`, and optional `tdse_step_dr(...)`
- a practical HIL integration usually treats `primary` as the host-to-TDSE input vector and `op` / `hr` / `ir` / `dr` as TDSE-to-host outputs used inside the host's accepted-step policy

For most teams, RT/HIL readiness is a three-lane progression:

Lane	What it proves
offline fixed-step proof	step order and committed-state discipline are correct
target-machine timing proof	the host and TDSE fit within the wall-clock budget
field qualification proof	scheduler, I/O, and platform variance are acceptable on the real target

16.4.1 Step-Loop Timing Budget

In a real-time simulation, each time step must complete within a fixed wall-clock budget. For TDSE, the per-step cost depends on:

Factor	Impact on Step Time	Mitigation
`np` (port count)	quadratic: `O(np^2)` in operator multiply	minimize port count
`nh` (history depth)	linear: `O(nh)` in convolution	use IRC compression for large `nh`
`nq > np` (rectangular view)	additional rows in operator	only use when measurements require it
backend choice	GPU has higher per-step overhead but better throughput for large `np`	CPU for `np < 10`, GPU for `np > 50`
variable dt (non-uniform path)	interpolation adds overhead per history tap	prefer fixed `dt = model_dt` for real-time

16.4.2 Worst-Case Execution Time (WCET)

For real-time certification, measure WCET under these conditions:

1. Use deterministic mode (`tdse_ext_set_deterministic_mode(1)`) to eliminate scheduling jitter
2. Run the worst-case model configuration (largest `np`, `nh` in the deployment)
3. Measure over at least 10,000 consecutive steps
4. Record the maximum observed step time plus a safety margin (recommended: 20% margin)

The step-loop APIs with the highest individual latency are:

- `tdse_step_hr()`: proportional to `nh * nq` (convolution over all history taps)
- `tdse_step_op()`: proportional to `nq * np` (dense matrix factorization for the operator)
- `tdse_step_commit()`: low constant cost (state advancement only)

16.4.3 Multi-Rate Coupling Time Budget

When coupling TDSE with an EMT solver at a different time step:

```
EMT step (e.g., 50 us):
├─ TDSE sub-steps: EMT_dt / TDSE_dt steps
│   ├─ each TDSE step: begin + op + hr + ir + commit
│   └─ per-step budget: EMT_dt / N_substeps
└─ remaining budget: host EMT solve + overhead
```

Example: EMT step = 50 μs, TDSE model_dt = 1 μs → 50 TDSE sub-steps per EMT step. Each TDSE sub-step must complete in under 50 μs / 50 = 1 μs (minus host overhead).

16.4.4 Deterministic Mode for Real-Time

Enable deterministic mode for all real-time deployments:

```
tdse_ext_set_deterministic_mode(1);
```

This ensures:

- bit-identical results across runs
- reduced runtime-side variability from non-deterministic execution choices
- more reproducible timing behavior during measurement

Deterministic mode improves measurement discipline, but it does not replace host-level WCET measurement, scheduler tuning, or target-machine qualification.

The throughput reduction from deterministic mode is acceptable in most real-time scenarios because the time budget is fixed regardless.

16.4.5 HIL Integration Patterns

For hardware-in-the-loop deployments:

1. **Dedicated core allocation.** Pin the TDSE worker thread to dedicated CPU cores. Use OS affinity APIs (`pthread_setaffinity_np` on Linux, `SetThreadAffinityMask` on Windows).
2. **Pre-allocate all resources.** Create all model handles before the real-time loop starts. No dynamic allocation during stepping.
3. **Avoid OS interference.** Isolate CPU cores from the OS scheduler (Linux `isolcpus` boot parameter) for the lowest jitter.
4. **Monitor guard metrics.** In real-time, continuous monitoring is essential. Poll `tdse_ext_get_runtime_guard_metrics()` periodically (e.g., every 1000 steps) and log any threshold crossing.

5. **Plan for graceful degradation.** If a step exceeds its budget, the host must decide whether to skip the commit (reject the trial) or accept the latency overrun. TDSE's trial/commit separation supports both strategies.

CHAPTER 17

Plugin System

Use this chapter when you need to deploy, validate, or troubleshoot TDSE solver plugins in a packaged environment. Each plugin is a shared library (`.so` on Linux, `.dll` on Windows) that the host loads at runtime to execute transient and AC simulations.

If your PoC stays on the default CPU path and you do not need manifest, deployment, or ABI decisions yet, you can defer this chapter until deployment planning. Come here when the question becomes compatibility, routing, signing, or plugin-health evidence.

Use the surrounding boundary chapters with it:

- use Platform Notes for the broader host, package, and qualification boundary
- use Element Reference when the deployment risk is really netlist coverage rather than plugin loading

17.0.1 Architecture

Three simulation engine plugins are provided:

Plugin	Binary name	Solver family
CPU Dense	`libtdse_sim_cpu_dense.so`	LAPACK, LU
CPU Sparse	`libtdse_sim_cpu_sparse.so`	KLU, MKL Pardiso
CUDA	`libtdse_sim_cuda.so`	CUDA dense, CUDA sparse

The host selects the correct plugin automatically based on the requested solver backend. Each plugin exports a standard entry point `tdse_plugin_get_info()` that returns ABI metadata and a function table.

17.0.2 ABI Version

The plugin ABI is versioned. The current version is **1.1**.

The host accepts plugins with the same major version and the same or older minor version. A v1.0 plugin loads on a v1.1 host; a v1.2 plugin is rejected by a v1.1 host.

The ABI contract and compatibility policy are documented in the SDK headers and the Plugin ABI Evolution Guide shipped with the evaluation package.

17.0.3 Deployment Layout

```
/opt/tdse/
  lib/
    plugins/
      sim/
        libtdse_sim_cpu_dense.so
        libtdse_sim_cpu_sparse.so
        libtdse_sim_cuda.so
        plugin_manifest.json
```

Plugins and their manifest must live together in the `plugins/sim/` directory. Do not rearrange or flatten this layout.

17.0.4 Plugin Manifest

The manifest (`plugin_manifest.json`) maps logical plugin names to on-disk binaries and carries integrity metadata:

```
{
  "schema_version": 1,
  "plugins": [
    {
      "name": "sim_cpu_dense",
      "kind": "sim",
      "abi_major": 1,
      "abi_minor": 1,
      "path": "libtdse_sim_cpu_dense.so",
      "sha256": "a1b2c3d4..."
    }
  ],
  "signature": "<base64 Ed25519 signature>"
}
```

- `path` is resolved relative to the manifest directory.
- `sha256` is mandatory. The loader computes the hash of the on-disk binary before loading and rejects mismatches.
- `signature` is an optional Ed25519 signature over the manifest payload. When a public key is configured, the signature is verified at load time.

17.0.5 Environment Variables

Variable	Purpose
`TDSE_PLUGIN_MANIFEST`	Path to `plugin_manifest.json`. Required in production.
`TDSE_PLUGIN_STRICT_MODE`	Set to 1 for production. Requires manifest and disables fallback directory probing.
`TDSE_PLUGIN_DIR`	Directory containing plugin `.so` files. Optional; use only for controlled evaluation or non-manifest deployments.

Variable	Purpose
TDSE_PLUGIN_PUBLIC_KEY_HEX	64-char hex Ed25519 public key for manifest signature verification.

17.0.6 Manifest Signing

The manifest can carry an Ed25519 signature. Signing and verification use the signing utility shipped with the SDK evaluation package:

```
# Generate a key pair
python3 tools/tdse_sign_manifest.py --generate-key-pair

# Sign the manifest in place
python3 tools/tdse_sign_manifest.py --sign manifest.json --private-key key.priv --in-place

# Verify
python3 tools/tdse_sign_manifest.py --verify manifest.json --public-key <hex-key>
```

When a public key is configured (via TDSE_PLUGIN_PUBLIC_KEY_HEX or the build-time TDSE_PLUGIN_PUBLIC_KEY), the loader verifies the signature before consuming manifest entries. Unsigned or tampered manifests are rejected.

See the Plugin Deployment Guide shipped with the evaluation package for the detailed signing workflow and key management expectations.

17.0.7 Production Configuration

For a packaged deployment, start with this configuration:

```
export TDSE_PLUGIN_MANIFEST=/opt/tdse/lib/plugins/sim/plugin_manifest.json
export TDSE_PLUGIN_STRICT_MODE=1
export TDSE_PLUGIN_PUBLIC_KEY_HEX=<64-hex-chars>
```

In strict mode:

- The host loads plugins **only** from the manifest.
- Fallback directory probing and relative directory scans are rejected.
- Manifest integrity (sha256) and authenticity (signature) are enforced.

This is the recommended baseline whenever you are shipping TDSE outside a local evaluation sandbox or controlled validation environment.

17.0.8 Health Check

Each plugin exports a health check function. Use `tdse_plugin_doctor` to inspect all installed plugins:

```
TDSE_PLUGIN_MANIFEST=/opt/tdse/lib/plugins/sim/plugin_manifest.json \
  tdse_plugin_doctor
```

The doctor reports for each plugin:

- ABI version and host compatibility
- Health status (`ok`, `cuda_partial`, `cuda_unavailable`)
- Manifest entry match and sha256 verification
- Struct size and vtable presence

Treat these health values as operational readiness signals, not performance claims. For example, `ok` means the plugin loaded compatibly under the current policy; it does not by itself prove throughput, WCET, or target-machine qualification.

Expected health status values:

- `ok` — plugin is fully operational
- `cuda_unavailable` — no CUDA runtime or driver detected
- `cuda_partial` — only some CUDA backends available
- `dense_unavailable` — dense CPU solver backend not available
- `klu_unavailable` — KLU sparse solver library not available
- `mkl_unavailable` — MKL Pardiso sparse solver not available
- `sparse_unavailable` — no sparse solver backends available

17.0.9 Monitoring

Plugin execution metrics are available through the host API (`snapshot_metrics()`). Tracked counters include:

- Load success and failure counts, with failure category
- Per-call-kind simulation counts (transient, AC matrix, AC probe)
- Total steps completed and frequency points processed
- Error count and last error code
- Backend switch count
- Session duration

Load failures are classified into stable categories (`[file_not_found]`, `[abi_mismatch]`, `[hash_mismatch]`, `[manifest_invalid]`) for machine parsing from logs.

17.0.10 Compatibility

The plugin ABI uses `struct_size` versioning for forward-compatible field additions:

- New fields are appended to the end of config structs.
- The host checks `struct_size` before accessing new fields.
- Defaults for missing fields are safe (zero, null, off).

This means a MINOR ABI bump does not break existing plugins. Plugin authors should recompile to advertise the new version, but old binaries continue to work.

17.0.11 External Compatibility Contract

For packaged deployments, the practical compatibility promises are:

- plugin load compatibility follows the major/minor ABI rule described above

- extensible plugin-facing config, info, and diagnostics structs grow through `struct_size`; intentionally frozen payloads are documented as exceptions
- manifest integrity checks and documented health / failure classifications are stable operational behavior, not optional debugging extras
- install-tree plugin consumption is exercised through smoke and compatibility tests, not only ad hoc local loading

That means customers should treat manifest validation, ABI compatibility, documented health states, and documented failure classes as part of the supported contract surface. Use `tdse_plugin_doctor` as the first field diagnostic for those signals, while its human-readable presentation may evolve.

17.0.12 Troubleshooting

17.0.13 ``No simulation engine plugin loaded''

Check `TDSE_PLUGIN_MANIFEST` or `TDSE_PLUGIN_DIR`. Run `tdse_plugin_doctor` to see detailed diagnostic output. Common causes:

- **ABI version mismatch**: the plugin was compiled against a different SDK version. The error message includes the plugin and host ABI versions.
- **Manifest sha256 mismatch**: the on-disk binary does not match the manifest. Rebuild or update the manifest.
- **Missing entry point**: the `.so` file does not export `tdse_plugin_get_info`. Check with `nm -D`.

17.0.14 Plugin loads but simulation fails with UNSUPPORTED

The loaded plugin does not support the requested solver backend. The host automatically loads the correct plugin for a given backend; check that the manifest includes the required plugin entry.

17.0.15 CUDA plugin reports `cuda_unavailable`

The CUDA toolkit or compatible driver is not installed, or the CUDA runtime libraries are not on the library path. Verify with `tdse_plugin_doctor`.

CHAPTER 18

Element Reference

Use this reference when you want to confirm whether a circuit element is supported, check its syntax, or compare model limitations before you prepare a netlist. Elements not listed here are unsupported and will produce a parse error.

Before you commit an Adapter Circuit workflow to a PoC or customer-facing demonstration, compare your actual netlist subset against this chapter first. The fastest way to lose time in evaluation is to assume broader SPICE coverage than the documented subset actually provides.

Use this reference as a support-boundary document, not just a syntax sheet. If your source deck depends on an element, directive, device level, or simulator behavior not listed here, treat that dependency as outside the current RC scope unless delivery notes or a support addendum say otherwise.

Most readers should not start here. Use it as a boundary check before you commit serious PoC effort to a netlist-based path, or when Adapter Circuit behavior needs to be compared against the documented subset.

Use the adjacent boundary chapters with it:

- use Platform Notes when the question is host or package support scope
- use Plugin System when the question is manifest, ABI, or solver-plugin deployment behavior

18.0.1 Passive Elements

18.0.1.1 R -- Resistor

```
Rname n1 n2 value
```

Linear resistor between nodes `n1` and `n2`. Resistance in ohms. No temperature coefficient or nonlinear resistor model.

18.0.1.2 C -- Capacitor

```
Cname n1 n2 value
```

Linear capacitor. Capacitance in farads. No initial condition (`.ic` is not supported).

18.0.1.3 L -- Inductor

```
Lname n1 n2 value
```

Linear inductor. Inductance in henries. No initial current.

18.0.1.4 K -- Mutual Inductor Coupling

```
Kname L1 L2 coefficient
```

Mutual coupling between two named inductors `L1` and `L2`. Coupling coefficient `|k| <= 1`. The coupled inductors share a matrix stamp.

18.0.2 Independent Sources

18.0.2.1 V -- Voltage Source

```
Vname n+ n- [DC dc_value] [AC ac_mag [ac_phase]] [waveform]
```

18.0.2.2 I -- Current Source

```
Iname n+ n- [DC dc_value] [AC ac_mag [ac_phase]] [waveform]
```

Positive current flows from `n+` to `n-` through the source.

18.0.2.3 Waveform Functions

Appended to a V or I source declaration:

Waveform	Syntax	Parameters
DC only	`DC value`	Constant value (volts or amps)
SIN/SINE	`SIN(vo va freq [td [theta [phase]]])`	Offset, amplitude, frequency (Hz), delay (s), damping, phase (deg)
PULSE	`PULSE(v1 v2 td tr tf pw [per])`	Low/high levels, delay, rise/fall times, pulse width, optional period
PWL	`PWL(t1 v1 t2 v2 ...)`	Piecewise-linear time-value pairs; times must be nondecreasing
EXP	`EXP(v1 v2 td1 tau1 td2 tau2)`	Dual-exponential: starts at v1, rises to v2
SFFM/FM	`SFFM(vo va fm_freq mdi fc [td [phasem [phasec]]])`	Single-frequency FM
AM	`AM(vo vmo vma fm_freq fc [td [phasem [phasec]]])`	Amplitude modulation

Default waveform when none is specified: DC 0.

18.0.3 Controlled Sources

18.0.3.1 E -- VCVS (Voltage-Controlled Voltage Source)

```
Ename n+ n- nc+ nc- gain
```

`V(n+,n-) = gain * V(nc+,nc-)`. Voltage gain (dimensionless).

18.0.3.2 G -- VCCS (Voltage-Controlled Current Source)

```
Gname n+ n- nc+ nc- transconductance
```

`I(n+→n-) = transconductance * V(nc+,nc-)`. Transconductance in siemens.

18.0.3.3 F -- CCCS (Current-Controlled Current Source)

```
Fname n+ n- vsource_name gain
```

`I(n+→n-) = gain * I(vsource_name)`. The controlling element must be a voltage source (its current is sensed). Current gain (dimensionless).

18.0.3.4 H -- CCVS (Current-Controlled Voltage Source)

```
Hname n+ n- vsource_name transresistance
```

`V(n+,n-) = transresistance * I(vsource_name)`. The controlling element must be a voltage source. Transresistance in ohms.

18.0.4 Semiconductor Devices

18.0.4.1 D -- Diode

```
Dname anode cathode [AREA area]
```

Shockley equation: `Id = IS * (exp(Vd / (N * VT)) - 1)`. Parameters: IS (saturation current, A), N (ideality factor), VT (thermal voltage, V). All must be positive (defaults: IS=1e-14, N=1, VT=0.02585).

18.0.4.2 Q -- BJT (Bipolar Junction Transistor)

```
Qname C B E [S] model_name [AREA area]
```

Gummel-Poon DC model (NPN or PNP). Four terminals: collector, base, emitter, optional substrate. References a `.model` card.

`.model` parameters: IS (saturation current), BF (forward beta), BR (reverse beta), VT (thermal voltage). Instance parameter: AREA.

18.0.4.3 M -- MOSFET

```
Mname D G S B model_name [W w] [L l] [M multiplier]
```

LEVEL=1 only (Shichman-Hodges model). NMOS or PMOS. Four terminals: drain, gate, source, bulk. References a `.model` card.

`.model` parameters: KP (transconductance, A/V^2), VTO (threshold voltage, V), LAMBDA (channel-length modulation, 1/V). Instance parameters: W, L, M (all default to 1.0; must be > 0).

18.0.5 Switches

18.0.5.1 S -- Voltage-Controlled Switch

```
Sname n+ n- nc+ nc- model_name [ON|OFF]
```

Controlled by voltage `V(nc+,nc-)`. `.model` type `SW`.

`.model` `SW` parameters: RON, ROFF (on/off resistance, ohms), VON, VOFF (on/off threshold voltages), or VT, VH (threshold, hysteresis).

18.0.5.2 W -- Current-Controlled Switch

```
Wname n+ n- vsource_name model_name [ON|OFF]
```

Controlled by current `I(vsource_name)`. `.model` type `CSW`.

`.model CSW` parameters: RON, ROFF (on/off resistance), ION, IOFF (on/off threshold currents), or IT, IH (threshold, hysteresis).

18.0.6 Frequency-Domain Blocks

18.0.7 NPORT -- N-port from Touchstone Data

```
NPORT_name n1 n2 ... n2N FILE="path/to/file.sNp" [TYPE=Y|Z] [INTERP=linear|log]
```

Imports frequency-dependent N-port from a Touchstone file (`.sNp`, `.yNp`, `.zNp`). **AC analysis only** – transient analysis with NPORT elements returns an error.

Parameters: FILE (path), TYPE (Y or Z, default: from file extension), FORCE_TYPE (override file type), INTERP (linear or log), EXTRAP (hold or linear).

18.0.8 S -- S-parameter Block

```
Sname n1 n2 ... n2N FILE="path/to/file.sNp" [Z0=50]
```

Loads Touchstone S-parameter data. Supports Z0/ZREF override for reference impedance.

18.0.9 Power System Elements

18.0.10 XTAPZ -- Tapped Series Impedance

```
XTAPZ_name n1 n2 tap r x [phase deg] [freq Hz]
```

Tapped series impedance for power system modelling. Required: tap ratio, r (resistance, ohms), x (reactance, ohms). Optional: phase offset (degrees), frequency base (Hz).

18.0.11 Subcircuits

18.0.12 .SUBCKT / .ENDS / X

```
.subckt subname n1 n2 ...
... elements ...
.ends

Xinst n1 n2 ... subname
```

Subcircuits are expanded (flattened) at parse time. Parameters passed via `.param` are supported numerically. `.global` nodes are silently ignored – all node connections are local to the subcircuit instance.

18.0.13 Unsupported SPICE Features

The following standard SPICE directives are recognised but **silently ignored** (a warning is emitted during parse):

`.temp`, `.ic`, `.nodeset`, `.option`, `.options`, `.dc`, `.op`, `.tf`, `.meas`, `.measure`, `.plot`, `.func`, `.global`, `.if`/`.else`/`.endif`, `.title`, `.csparam`

Do not treat these warnings as cosmetic during evaluation. If your source deck depends on one of the directives above for operating point, initialization, or post-processing intent, the imported behavior may differ from your original simulator.

For PoC planning, treat every ignored directive as a required review item. A deck may parse successfully and still fall outside the intended acceptance scope if those directives carry essential setup or validation meaning in the source simulator.

The following SPICE elements are **not supported** and will cause a parse error:

- J (JFET)
- T (transmission line, lossless)
- O (transmission line, lossy)
- U (uniform RC line)
- Z (MESFET)
- MOSFET LEVEL > 1 (BSIM, etc.)
- Nonlinear magnetics / saturable cores

CHAPTER A

Appendix

Use this appendix for final integration checks and one compact worked example. For terminology and abbreviations, see the Preface.

A.1 Integration Checklist

A.1.1 Functional Bring-Up Checklist

- ☐ Pack bytes validate before runtime create.
- ☐ `tdse_model_create(..., &create_diag, ...)` succeeds on the target host.
- ☐ `tdse_model_info(...)` matches expected `np`, `nq`, `nh`, and `dt`.
- ☐ The prime-step pattern (`t0 - dt`) is implemented intentionally.
- ☐ `op`, `hr`, `ir`, and `commit` wiring are step-aligned.
- ☐ `tdse_step_dr(...)` is only used after commit and outside an active trial step.

A.1.2 Lifecycle Checklist

- ☐ Ordinary host shutdown uses `tdse_model_destroy(...)`, not `tdse_model_release(...)`.
- ☐ Destroy wait policy is explicit and documented.
- ☐ The host handles `TDSE_ERR_TIMEOUT` intentionally.
- ☐ The host treats `TDSE_ERR_INVALID_STATE` during close or destroy as ownership handoff, not as local retryable success.
- ☐ RAII or destructor paths use `tdse_model_release(...)` only as terminal cleanup.

A.1.3 Diagnostics Checklist

- ☐ `tdse_model_create_diagnostics_t` is logged or archived on non-OK create paths.
- ☐ Runtime failure capture includes both `tdse_model_state_info(...)` and `tdse_model_last_error_info(...)`.
- ☐ One known-failure path such as `TDSE_ERR_IR_STEP_OUT_OF_RANGE` is exercised and logged.
- ☐ Failure logs include API name, status, `t`, `dt`, and step index.

A.1.4 Concurrency Checklist

- ☐ One live runtime handle is owned by exactly one execution thread at a time.
- ☐ Same-handle concurrent step entry is prevented by host design.

☐ Threading stress remains green on the target build.
☐ The integration wrapper does not silently share a runtime handle across worker threads.

A.2 Worked Example (np=3, nh=4)

Use this example when you want one concrete end-to-end flow after reading the main manual.

A.2.1 What This Example Demonstrates

The example shows:

1. configure Builder for `np = nq = 3`, `nh = 4`
2. apply explicit time-domain `H`
3. apply optional `IR`
4. write a pack
5. create a runtime model with request-scoped diagnostics
6. run `begin -> hr -> ir -> commit`
7. query committed-step `dr`

A.2.2 Compact C Example

Builder half:

```c
#include <tdse/tdse.h>
#include <tdse_builder.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

int main(void) {
    const size_t np = 3, nh = 4;
    const double dt = 1e-3;

    /* ---- Builder: create pack ---- */
    double h[4 * 3 * 3] = {
        1,0,0, 0,1,0, 0,0,1,
        0.2,0,0, 0,0.2,0, 0,0,0.2,
        0.1,0,0, 0,0.1,0, 0,0,0.1,
        0.05,0,0, 0,0.05,0, 0,0,0.05
    };

    tdse_builder_t* b = NULL;
    tdse_builder_create(&b);

    tdse_builder_options_t opt = tdse_builder_options_init();
    opt.dt = dt; opt.nh = nh; opt.np = np; opt.nq = np;
    tdse_builder_configure_ex(b, &opt);
```

```
    tdse_h_desc_t hd = {0};
    hd.struct_size = sizeof(hd);
    hd.np = (int32_t)np; hd.nq = (int32_t)np; hd.nh = nh;
    hd.data = h;
    tdse_builder_apply_h(b, &hd);
    tdse_builder_write_pack(b, "appendix_example.pack");
    tdse_builder_destroy(b);
```

Runtime half:

```
    /* ---- Runtime: load and step ---- */
    FILE* f = fopen("appendix_example.pack", "rb");
    fseek(f, 0, SEEK_END); long sz = ftell(f); fseek(f, 0, SEEK_SET);
    unsigned char* pack = (unsigned char*)malloc((size_t)sz);
    fread(pack, 1, (size_t)sz, f); fclose(f);

    tdse_model_t* m = NULL;
    tdse_model_create_diagnostics_t diag = tdse_model_create_diagnostics_init();
    tdse_status_t st = tdse_model_create(pack, (size_t)sz, &diag, &m);
    free(pack);
    if (st != TDSE_OK) { printf("create failed\n"); return 1; }

    /* Prime step at n = -1 */
    double primary0[3] = {0,0,0};
    tdse_step_begin(m, -dt, dt);
    tdse_step_commit(m, primary0);

    /* Ordinary steps */
    double hr[3], ir[3], dr[3];
    for (size_t n = 0; n < 5; ++n) {
        double primary[3] = {sin(0.1*n), cos(0.1*n), 0.5};
        tdse_step_begin(m, n * dt, dt);
        tdse_step_hr(m, hr);
        tdse_step_ir(m, ir);
        tdse_step_commit(m, primary);
        tdse_step_dr(m, dr);
        printf("step=%zu hr=[%+.4f %+.4f %+.4f] dr=[%+.4f %+.4f %+.4f]\n",
               n, hr[0], hr[1], hr[2], dr[0], dr[1], dr[2]);
    }

    /* Bounded shutdown */
    tdse_model_destroy_options_t dopt = tdse_model_destroy_options_init();
    tdse_model_destroy_result_t dres = tdse_model_destroy_result_init();
    dopt.wait_timeout_ms = 250.0;
    tdse_model_destroy(m, &dopt, &dres);
    return 0;
}
```

A.2.3 What To Notice

Three details are worth carrying into production code:

- create uses request-scoped diagnostics
- the runtime loop keeps `commit` as the only state-advancing call
- shutdown uses bounded destroy rather than release

CHAPTER A

Index