
xstar-tools 0.6.90.5.8 Documentation

Release 0.6.90.5.8

Ashkbiz Danekar

Oct 07, 2026

USER GUIDE

1	User guide	3
1.1	Installation	3
1.2	Windows native installation and usage	6
1.3	Command-line interfaces	11
1.4	Atomic-data setup	12
1.5	First run	15
1.6	Configuration	17
1.7	Choosing a backend	17
1.8	Output products	18
1.9	Reproducibility and provenance	19
1.10	Performance expectations	19
1.11	Troubleshooting	20
2	Python guide	23
2.1	Running XSTAR from Python	23
2.2	Atomic database API	25
2.3	Reading XSTAR output products	27
2.4	XSTAR2XSPEC from Python	27
3	Python API reference	29
3.1	Stable execution API	29
3.2	Atomic-data API	39
3.3	Output-reader API	41
3.4	XSTAR2XSPEC Python API	44
4	C++ and native CLI reference	47
4.1	Native build	47
4.2	xstar-cpp	49
4.3	xstar-xspec-initable	50
4.4	xstar-xspec-table	51
4.5	xstar-xspec	52
4.6	xstar-xspec-mpi	53
4.7	C ABI	54
5	Developer guide	57
5.1	XSTAR tools architecture at the parity-freeze boundary	57
5.2	Execution modes	66
5.3	Cross-platform portability status	66
5.4	Qualification policy	68
5.5	Performance rules	69

5.6	Versioning and science freeze	69
5.7	Fortran source map	70
5.8	Python / C++ / Fortran concordance	75
5.9	Python / FORTRAN source correspondence comments	81
5.10	C++ / FORTRAN source correspondence comments	81
5.11	Function-level source comments	82
5.12	Stable public Python API	83
5.13	Unified CLI contract	84
5.14	xstar-cpp native executable contract — 0.6.69	85
5.15	Native C ABI contract	86
5.16	Adding or changing atomic-data/rate support	87
5.17	Source-name hygiene and ABI naming	88
5.18	Production diagnostics policy for Python execution modes	88
5.19	XSTAR2XSPEC / xstar2table source concordance — 0.6.81	90
5.20	Reference data and release candidates	94
5.21	CI and release engineering	94
5.22	Packaging and native-wheel policy	95
5.23	Conda packaging	97
5.24	Documentation authoring and build policy	98
5.25	Building the documentation	98
6	Science guide	103
6.1	Algorithm overview	103
6.2	Data-type and rate-type semantics	103
6.3	XSTAR scientific references and implementation correspondence	104
6.4	Equations and source correspondence	110
7	Project history	113
7.1	0.6.90.5.2 — LATEX_DVI_PS_PDF_PIPELINE_CLOSURE	113
7.2	0.6.90.5.1 - LATEXPDF_MAKE_MODE_EPS_GRAPHICS_AND_READTHEDOCS_CLOSURE	113
7.3	0.6.90.5 — DOCUMENTATION_BUILD_OUTPUTS	114
7.4	0.6.90.3 — REPOSITORY_HISTORY_AND_QUALIFICATION_CLEANUP	114
7.5	0.6.90.4 — DOCUMENTATION_COVERAGE_CLOSURE	116
	Python Module Index	117
	Index	119

`xstar-tools` provides Python, command-line, shared-C++, and native interfaces for XSTAR atomic data and source-faithful XSTAR execution. Distribution/productization versions evolve independently from the accepted scientific revision.

Accepted native non-MPI portability baseline: Linux GCC x86_64, macOS arm64/x86_64 with Apple Clang, and Windows MSYS2 UCRT64/MinGW-w64. Windows conda packages are not supported; MSVC and Windows MPI are not planned. Local Windows parallelism is available through `xstar-xspec --processes N`.

USER GUIDE

Start here to install `xstar-tools`, configure XSTAR atomic data, run a model, build XSPEC tables, choose an execution backend, or inspect outputs.

1.1 Installation

`pyproject.toml` is the authoritative Python package metadata/build configuration. The small `setup.py` file exists only to register the native `setuptools` build hook.

The accepted native-wheel baselines cover CPython 3.9–3.14 on Linux `x86_64` (`manylinux_2_28_x86_64`), macOS `arm64` and `x86_64` (`macosx_11_0_*`), and Windows `AMD64` (`win_amd64`). `0.6.89.5 - PYPI_CROSS_PLATFORM_RELEASE_CLOSURE` rebuilds those accepted profiles under one release version so PyPI can select the appropriate wheel automatically for `pip install xstar-tools`. The frozen science revision and public ABIs are unchanged.

The PyPI Linux, macOS, and Windows binary profiles omit the optional standalone Python-embedding plugin so distributable wheels do not acquire a Python-runtime library dependency. The default source/native installation profile remains full. Python-driven `zone-cpp`, the public C API, `xstar-cpp`, and `XSTAR2XSPEC` executables remain present. MPI remains opt-in and `atdb.fits` remains external. Package metadata uses the SPDX expression `GPL-3.0-only` and ships the GNU GPL Version 3 LICENSE file. Native macOS and Windows release wheels are built/tested on GitHub-hosted runners; local access to those operating systems is not required.

1.1.1 Python package installation

For a released package:

```
python -m pip install xstar-tools
```

For a source archive or checkout:

```
python -m pip install .
```

A Python-only installation is explicit:

```
XSTAR_TOOLS_NATIVE=off python -m pip install .
```

To require native compilation instead of allowing fallback:

```
XSTAR_TOOLS_NATIVE=required python -m pip install .
```

The native packaging hook stages `xstar-cpp`, `xstar-xspec-initable`, `xstar-xspec-table`, and `xstar-xspec` plus their platform-native shared libraries. It does not stage `xstar-xspec-mpi`, and it does not bundle `atdb.fits`.

`XSTAR_TOOLS_NATIVE_JOBS=N` controls Make parallelism used by the Python build hook.

The wheel/packaging policy is separate from the direct native source-build qualification described below; do not infer source-build support from an older wheel-policy milestone.

1.1.2 Conda packaging

0.6.90.1 - CONDA_NATIVE_HOST_QUALIFICATION qualifies the refreshed conda-forge v1 recipe natively on Linux x86_64, macOS arm64, and macOS Intel x86_64. CFITSIO is resolved from the conda environment rather than vendored. MPI, the standalone Python-embedding plugin, and atdb.fits are not part of the ordinary conda package. Windows conda packages are not supported. Windows native support remains on the accepted PyPI/MSYS2 UCRT64/MinGW-w64 path; MSVC and Windows MPI are not planned.

The public conda-forge package is available now. Install with:

```
conda install -c conda-forge xstar-tools
```

The current feedstock release is based on the published PyPI 0.6.89.5 source; 0.6.90.1 is the host-qualification/source milestone and is not assumed to exist on PyPI.

1.1.3 Native source builds

The current accepted source tree is qualified on four non-MPI hosts:

Platform	Toolchain	Status
Linux x86_64	GCC / GNU Make	ACCEPT
macOS arm64	Apple Clang / GNU Make	ACCEPT
macOS x86_64	Apple Clang / GNU Make	ACCEPT
Windows x86_64	MSYS2 UCRT64 / MinGW-w64 GCC / GNU Make	ACCEPT

The accepted portability baseline is 0.6.88.6.1.2.1.

Linux

Typical prerequisites are:

- a C++17 compiler;
- GNU Make;
- Python development support / python3-config;
- CFITSIO development headers/libraries;
- pkg-config.

On Debian/Ubuntu, CFITSIO development files are normally provided by libcfitsio-dev.

Build:

```
make -C src/xstar_tools/xstar/cpp -j4 PLATFORM=linux
```

macOS

Install:

```
brew install cfitsio pkg-config
```

Build:

```
make -C src/xstar_tools/xstar/cpp -j4 PLATFORM=macos
```

The qualified macOS build uses Apple Clang, .dylib libraries, @rpath install names, and @loader_path sibling discovery.

Windows

Use an **MSYS2 UCRT64** terminal. The accepted dependency set and detailed build/run instructions are in *Windows native installation and usage*.

The normal build is:

```
make -C src/xstar_tools/xstar/cpp -j4 PLATFORM=windows
```

Windows MPI is not part of the accepted Windows contract.

1.1.4 Optional MPI build

MPI is not part of the normal build. On an MPI-capable Linux/HPC environment:

```
make -C src/xstar_tools/xstar/cpp mpi
```

or:

```
make -C src/xstar_tools/xstar/cpp xstar-xspec-mpi
```

The wrapper can be overridden with `MPICXX=/path/to/mpic++`.

1.1.5 Editable development checkout

```
python -m pip install -e '[dev]'
```

For documentation dependencies:

```
python -m pip install -e '[docs]'
```

1.1.6 Verify the installation

```
xstar-tools version
xstar-tools backends
xstar-tools doctor
```

For native binaries:

```
src/xstar_tools/xstar/cpp/xstar-cpp --version
src/xstar_tools/xstar/cpp/xstar-cpp --abi
```

1.1.7 Atomic data is separate

`atdb.fits` and `coheat.dat` are not silently downloaded as part of a normal model run. See *Atomic-data setup* for explicit setup and discovery precedence.

1.2 Windows native installation and usage

xstar-tools supports a native Windows build through **MSYS2 UCRT64**, MinGW-w64 GCC, GNU Make, CFITSIO, and pkg-config. The current accepted portability baseline, 0.6.88.6.1.2.1, qualifies this toolchain as part of the full four-host matrix. Windows passes warning-free native compilation, PE DLL/import-library/export checks, shared-library discovery, the Win32 process layer, local-process XSTAR2XSPEC scheduling, fixed-state same-host determinism and cross-platform reference equivalence, the embedded Python backend, the Python bridge, and the existing regression suite.

Windows MPI is not part of the accepted Windows contract. Use `xstar-xspec --processes N` for local parallel XSTAR2XSPEC work on Windows. Do not expect `xstar-xspec-mpi` to be produced by the Windows build.

The accepted Windows fixed-state run reproduces the canonical FITS payload byte-exactly. Its text products may use CRLF line endings; the qualified cross-platform comparison normalizes line endings and requires exact discrete state plus strict numerical equivalence.

1.2.1 1. Install MSYS2 and open UCRT64

Install MSYS2 from its normal Windows installer, then open the **MSYS2 UCRT64** terminal. The shell must report:

```
echo "$MSYSTEM"
```

with:

```
UCRT64
```

Do not use the plain MSYS shell for the native XSTAR build.

Update the package database and base installation as needed:

```
pacman -Syu
```

If MSYS2 asks you to close and reopen the terminal during a full update, do that and rerun the update command until it completes normally.

1.2.2 2. Install native build dependencies

For `xstar-cpp` and the native C++ runtime:

```
pacman -S --needed \  
  make \  
  mingw-w64-ucrt-x86_64-gcc \  
  mingw-w64-ucrt-x86_64-binutils \  
  mingw-w64-ucrt-x86_64-cfitsio \  
  mingw-w64-ucrt-x86_64-pkgconf
```

For the complete build, embedded Python backend, and qualification tests, also install:

```
pacman -S --needed \  
  mingw-w64-ucrt-x86_64-python \  
  mingw-w64-ucrt-x86_64-python-pytest
```

Verify the environment:

```
echo "$MSYSTEM"  
g++ --version
```

(continues on next page)

(continued from previous page)

```
make --version
python --version
pkg-config --modversion cfitsio
pkg-config --cflags --libs cfitsio
```

1.2.3 3. Extract a source release

Windows drive C: is visible in MSYS2 as /c. For example, if the source archive is in C:\Users\you\Downloads:

```
cd /c/Users/you/Downloads
tar -xzf xstar_tools-<version>.tar.gz
cd xstar_tools-<version>/src/xstar_tools/xstar/cpp
```

For a Git checkout, enter the same src/xstar_tools/xstar/cpp directory.

1.2.4 4. Inspect the Windows build contract

```
make PLATFORM=windows print-config
```

The important values include:

```
PLATFORM=windows
SHLIB_EXT=.dll
EXEEXT=.exe
V0682401_PRODUCTION_COMPILER=MINGW64
```

CFITSIO should normally report CFITSIO_DISCOVERY=pkg-config in the qualified MSYS2 UCRT64 environment.

1.2.5 5. Build xstar-cpp

To build only the standalone native XSTAR frontend and its required DLLs:

```
make -j4 PLATFORM=windows xstar-cpp.exe
```

To build the complete non-MPI Windows native target set:

```
make -j4 PLATFORM=windows
```

The build produces executables such as:

```
xstar-cpp.exe
xstar_cpp.exe
xstar-xspec-initable.exe
xstar-xspec-table.exe
xstar-xspec.exe
```

and XSTAR DLLs such as:

```
libxstar_api.dll
libxstar_production_zone.dll
libxstar_local_zone.dll
libxstar_engine.dll
libxstar_solver.dll
```

(continues on next page)

(continued from previous page)

```
libxstar_emissivity.dll  
libxstar_opacity.dll  
libxstar_thermal.dll
```

MinGW import libraries use the `.dll.a` suffix.

Check the frontend version:

```
./xstar-cpp.exe --version  
./xstar-cpp.exe --abi
```

For an initial source-tree run, keep `xstar-cpp.exe` in the same build directory as the generated `libxstar_*.dll` files.

1.2.6 6. Configure XSTAR atomic data

A real XSTAR calculation requires `atdb.fits` and `coheat.dat`. They are external data and are not bundled into the source archive.

For example, if the files are stored in:

```
C:\xstar\data\atdb.fits  
C:\xstar\data\coheat.dat
```

then the MSYS2 path is:

```
/c/xstar/data
```

Verify them:

```
ls -lh /c/xstar/data/atdb.fits  
ls -lh /c/xstar/data/coheat.dat
```

The clearest first run uses an explicit directory:

```
./xstar-cpp.exe --data-dir /c/xstar/data ...
```

You can instead set:

```
export XSTAR_DATA=/c/xstar/data
```

Native discovery also honors the established explicit-file and environment-variable precedence, including `XSTAR_ATOMIC_DB`, `XSTAR_ATDB_FITS`, `XSTAR_COHEAT`, `XSTAR_DATA`, and `HEADAS/refdata` when those locations are configured.

1.2.7 7. Run a small included example

From `src/xstar_tools/xstar/cpp`, a useful first real calculation is the included C5 parameter file:

```
./xstar-cpp.exe \  
--input ../../../../qualification/c5_terminal_step_0_6_82_6/params/c5_ne1e8.par \  
--data-dir /c/xstar/data \  
--output-dir run_c5_windows
```

Inspect the output directory:

```
find run_c5_windows -maxdepth 1 -type f -print
```

Depending on the input output-control parameters, products can include files such as `xout_step.log`, `xout_abund1.fits`, `xout_lines1.fits`, `xout_rrc1.fits`, and `xout_spect1.fits`.

The broader example file can be run in the same way:

```
./xstar-cpp.exe \
--input ../../../../examples/xstar_example.par \
--data-dir /c/xstar/data \
--output-dir run_example_windows
```

That example is a more substantial multi-element calculation and is not intended as the fastest smoke test.

1.2.8 8. Run without a parameter file

`xstar-cpp` also accepts `name=value` parameters directly. For example:

```
./xstar-cpp.exe \
--data-dir /c/xstar/data \
--output-dir run_direct_windows \
spectrum=pow spectun=0 trad=-1 \
temperature=100 pressure=0.03 density=1e12 \
column=1e20 rlradi38=1e6 rlogxi=1 cfrac=0.4 \
habund=1 heabund=1 cabund=1 \
nabund=0 oabund=0 neabund=0 mgabund=0 \
siabund=0 sabund=0 arabund=0 caabund=0 \
crabund=0 feabund=0 niabund=0 \
modelname=windows_test abundtbl=xdef \
nsteps=10 niter=99 lwrite=1 lprint=1 lstep=0 \
emult=0.5 taumax=5 xeemin=0.1 critf=1e-6 \
vturbi=100 radexp=0 npass=1 ncn2=9999
```

For reproducible science runs, keeping the parameters in a file is usually easier to audit.

1.2.9 9. Run local-process XSTAR2XSPEC

Build the complete native target set first:

```
make -j4 PLATFORM=windows
```

Then run a serial/local grid with `xstar-xspec.exe`:

```
./xstar-xspec.exe \
--input /c/path/to/xstinitable.par \
--data-dir /c/xstar/data \
--output-dir run_xspec_windows \
--processes 2
```

`--processes 2` means two independent `xstar-cpp` **OS processes** may run at the same time. It does not create two XSTAR threads. The Windows process backend uses the qualified Win32 process layer.

Windows MPI remains out of scope. Do not run `mpirun xstar-xspec.exe`; `xstar-xspec-mpi` is the separate POSIX/MPI frontend and is opt-in on supported MPI platforms.

1.2.10 10. Clean and rebuild

```
make clean PLATFORM=windows
make -j4 PLATFORM=windows
```

A strict Windows qualification build is expected to be warning-free.

1.2.11 11. Common Windows problems

pkg-config cannot find CFITSIO

Confirm you are in the UCRT64 shell and that the UCRT64 packages are installed:

```
echo "$MSYSTEM"
pacman -Q mingw-w64-ucrt-x86_64-cfitsio
pkg-config --modversion cfitsio
```

A XSTAR DLL cannot be found

For source-tree use, run the executable with the generated sibling `libxstar_*.dll` files still present in the C++ build directory. `XSTAR_PLUGIN_PATH` may also point to that directory when explicitly testing plugin discovery.

Atomic data cannot be found

Use an explicit path first:

```
./xstar-cpp.exe --data-dir /c/xstar/data ...
```

Then verify both required files exist in that directory.

Embedded Python cannot import ctypes

The accepted Windows embedder registers the initialized Python runtime DLL directory before importing the `xstar_tools` Python package. Use the UCRT64 Python package and keep the Python installation consistent with the compiler environment.

Path syntax

Inside MSYS2 commands, prefer POSIX-style paths such as `/c/xstar/data`. The native code contains explicit Windows path-boundary handling for existing narrow CFITSIO/XSTAR C APIs, but shell commands are easier to reproduce when written in MSYS2 form.

1.2.12 12. Current qualification checks

The historical 0.6.88.6.1.2.1 host runner established the accepted Windows portability baseline, but that closed milestone runner is no longer shipped in the lean release tree. Current source releases keep compact active gates instead:

```
python tools/qualification/check_parity_freeze.py
python tools/qualification/check_source_concordance.py
python tools/release/check_release_candidate.py
```

Run the ordinary pytest suite in the same UCRT64 environment for platform/runtime regression coverage. The accepted policy remains: Windows native support uses MSYS2 UCRT64/MinGW-w64; Windows conda packages are unsupported; MSVC and Windows MPI are not planned. Local parallel XSTAR2XSPEC execution remains available through `xstar-xspec --processes N`.

1.3 Command-line interfaces

The preferred general entry point is:

```
xstar-tools <command>
```

The preferred native XSTAR/XSTAR2XSPEC entry points are `xstar-cpp`, `xstar-xspec-initable`, `xstar-xspec-table`, and `xstar-xspec`. True MPI is built explicitly as `xstar-xspec-mpi` and is not part of the ordinary install contract.

1.3.1 Unified xstar-tools commands

```
xstar-tools run
xstar-tools inspect
xstar-tools data
xstar-tools backends
xstar-tools compare
xstar-tools doctor
xstar-tools version
```

Run one model

```
xstar-tools run xstar.par \
  --mode zone-cpp \
  --data-dir /path/to/xstar/data \
  --output-dir run1
```

`run` accepts these options:

Option	Meaning
INPUT	XSTAR .par, run_xstar.sh, or .cmd input
--mode	pure-python, zone-python, zone-cpp, zone-all, or xstar-cpp
--data-dir DIR	directory containing atdb.fits and coheat.dat
--output-dir DIR	output directory, default run1
--threads N	run-scoped thread count
--overwrite	replace a non-empty output directory
--no-progress	suppress human-readable science progress
--log-level LEVEL	DEBUG, INFO, WARNING, ERROR, or CRITICAL
--json-log FILE	write JSONL run events/provenance
--summary-json FILE	write the final XStarResult as JSON
--cache-dir DIR	explicit cache location
--no-cache	disable cache use
--rebuild-cache	rebuild cache data
--no-reproducible	disable reproducible-run intent

Runtime/status commands

```
xstar-tools backends
xstar-tools doctor --require zone-cpp --data-dir /path/to/xstar/data
xstar-tools version --json
```

`xstar-tools data` delegates to the explicit atomic-data configurator:

```
xstar-tools data --show
xstar-tools data --set-path /path/to/atdb.fits
```

Normal model execution never downloads scientific data implicitly. `xstar-tools data --destination ...` `--yes` is an explicit download action.

1.3.2 Installed console scripts

The distribution installs 30 console scripts. Use the primary commands above unless a specialist or compatibility interface is specifically needed.

Primary public commands

Command	Purpose
<code>xstar-tools</code>	unified user/developer command tree
<code>xstar-cpp</code>	one standalone native XSTAR calculation
<code>xstar-xspec-initable</code>	plan an XSTAR2XSPEC grid
<code>xstar-xspec-table</code>	assemble final XSPEC tables from spectra
<code>xstar-xspec</code>	complete serial/local-process XSTAR2XSPEC workflow

Specialist atomic-data/output utilities

These are retained direct tools for focused work: `xstar-tools-inspect`, `xstar-tools-hierarchy`, `xstar-tools-lines`, `xstar-tools-photoionization`, `xstar-tools-collisions`, `xstar-tools-recombination`, `xstar-tools-emissivity`, `xstar-tools-solver`, `xstar-tools-xstar-outputs`, and `xstar-tools-download-data`.

Compatibility/older convenience frontends

The following remain installed for compatibility with earlier `xstar-tools` workflows: `xstar-tools-run-xstar`, `xstar-tools-inspect-atdb`, `xstar-tools-xstinitable`, `xstar-tools-xstar2table`, `xstar-tools-xstar2xspec`, `xstar-tools-mpixstar`, and `xstar-tools-plan-xstar-run`. New XSTAR2XSPEC commands should prefer the native names. `xstar-tools-mpixstar --np N` is a local-process compatibility wrapper; true MPI uses `mpirun -np N xstar-xspec-mpi`.

Development/qualification-oriented commands

`xstar-tools-benchmark`, `xstar-tools-run-python`, `xstar-tools-original-parity-gate`, `xstar-tools-cpp-parity-gate`, `xstar-tools-qualification`, `xstar-tools-validate-collisions`, `xstar-tools-export`, and `xstar-tools-export-superwind` are retained for benchmark, qualification, validation, export, or advanced development workflows rather than as the default user front door.

The unified equivalents for active development/qualification work are:

```
xstar-tools dev ...
xstar-tools qualify ...
```

1.4 Atomic-data setup

Native XSTAR execution requires `atdb.fits` and `coheat.dat`. Normal scientific execution never downloads these files implicitly.

1.4.1 Recommended explicit setup

For one native model:

```
xstar-cpp \
  --input xstar.par \
  --data-dir /path/to/xstar/data \
  --output run1
```

For XSTAR2XSPEC:

```
xstar-xspec \
  --input xstinitable.par \
  --data-dir /path/to/xstar/data \
  --output-dir run_table \
  --processes 2
```

--data-dir DIR means that native execution should use:

```
DIR/atdb.fits
DIR/coheat.dat
```

For xstar-cpp the two files can also be selected independently:

```
xstar-cpp \
  --input xstar.par \
  --atomic-db /path/to/atdb.fits \
  --coheat /path/to/coheat.dat \
  --output run1
```

1.4.2 Native discovery when explicit paths are omitted

The runtime tries candidates in order and selects the first existing regular file.

atdb.fits

1. parameter-envelope atomic_database, atomic_db, or atdb;
2. atdb.fits beside the parameter envelope;
3. \$XSTAR_ATOMIC_DB;
4. \$XSTAR_ATDB_FITS;
5. \$XSTAR_DATA/atdb.fits;
6. \$HEADAS/refdata/atdb.fits;
7. \$XSTAR_HOME/data/atdb.fits;
8. executable-relative ../data/atdb.fits;
9. executable-relative ../../data/atdb.fits;
10. src/xstar_tools/xstar/data/atdb.fits;
11. ./atdb.fits.

The first item includes explicit paths generated by public xstar-cpp --atomic-db and --data-dir handling.

coheat.dat

1. parameter-envelope coheat_file or coheat;
2. coheat.dat beside the parameter envelope;
3. \$XSTAR_COHEAT;
4. \$XSTAR_DATA/coheat.dat;
5. \$HEADAS/refdata/coheat.dat;
6. \$XSTAR_HOME/data/coheat.dat;
7. executable-relative ../data/coheat.dat;
8. executable-relative ../../data/coheat.dat;
9. src/xstar_tools/xstar/data/coheat.dat;
10. ./coheat.dat.

The first item includes explicit paths generated by public `xstar-cpp --coheat` and `--data-dir` handling.

1.4.3 HEASoft HEADAS/refdata

The accepted native runtime supports a HEASoft installation as a fallback. For example:

```
heainit
unset XSTAR_DATA
unset XSTAR_ATOMIC_DB
unset XSTAR_ATDB_FITS
unset XSTAR_COHEAT
unset XSTAR_HOME

xstar-cpp --input xstar.par --output run_headas
```

If the files exist, the runtime can select:

```
$HEADAS/refdata/atdb.fits
$HEADAS/refdata/coheat.dat
```

The selected paths are reported through native diagnostics/provenance and are canonicalized when possible.

1.4.4 XSTAR2XSPEC ownership

`xstar-xspec-initable` does not independently search `XSTAR_DATA`, `HEADAS`, or `XSTAR_HOME`. If `xstar-xspec --data-dir DIR` is supplied, the planner emits that explicit data directory into every `xstar-cpp` job. If `--data-dir` is omitted, every child `xstar-cpp` owns its normal runtime data discovery.

For true MPI execution across multiple nodes, prefer an explicit shared path:

```
mpirun -np 4 xstar-xspec-mpi \
--input xstinitable.par \
--data-dir /shared/xstar/data \
--output-dir /shared/run_mpi
```

This avoids depending on how a particular MPI launcher exports shell environment variables to remote nodes.

1.4.5 Python data helper

Python-side validation remains available:

```
from xstar_tools.data import XStarData

data = XStarData.from_directory("/path/to/xstar/data")
print(data.validate().as_dict())
print(data.identity())
```

Installed wheels do not bundle the large atomic database. Configured paths and explicit downloads remain outside site-packages.

1.5 First run

1.5.1 Native xstar-cpp

Commands below use the Unix executable spelling. In an MSYS2 UCRT64 Windows build, use `xstar-cpp.exe` / `xstar-xspec.exe`.

Using an XSTAR parameter file:

```
xstar-cpp \
  --input xstar.par \
  --data-dir /path/to/xstar/data \
  --output run1
```

Without a parameter file:

```
xstar-cpp \
  --data-dir /path/to/xstar/data \
  --output run_direct \
  spectrum=pow temperature=100 density=1e8 column=1e20 rlogxi=1 \
  cfrac=1 niter=0 ncn2=9999 modelname=direct_example
```

Values after `--input` override the file:

```
xstar-cpp --input xstar.par --output run_override density=1e10 rlogxi=2
```

For a realistic direct single-model parameter set including abundances, pressure, convergence, and output controls, see *xstar-cpp*.

If `--data-dir` is omitted, native atomic-data discovery is used; see *Atomic-data setup*.

1.5.2 Native XSTAR2XSPEC

Serial:

```
xstar-xspec \
  --input xstinitable.par \
  --data-dir /path/to/xstar/data \
  --output-dir run_serial \
  --processes 1
```

Two simultaneous local processes:

```
xstar-xspec \
--input xstinitable.par \
--data-dir /path/to/xstar/data \
--output-dir run_local2 \
--processes 2
```

--processes counts **OS processes**, not threads. Each process slot launches a distinct `xstar-cpp`. The operating system schedules those processes on available logical CPUs; physical-core pinning is external to `xstar-xspec`.

A direct four-job 2 x 2 grid can be created without `xstinitable.par`:

```
xstar-xspec \
--data-dir /path/to/xstar/data \
--output-dir run_grid \
--processes 2 \
column=1e21 columntyp=2 columnsof=1e20 columnnst=2 columnint=1 \
rlogxi=2 rlogxityp=2 rlogxisof=1 rlogxinst=2 rlogxiint=0
```

For a realistic direct XSTAR2XSPEC grid with many physical and grid-control parameters, see *xstar-xspec*.

Restart:

```
xstar-xspec --input xstinitable.par --output-dir run_local2 --processes 2 --restart
```

The visible `xstar2xspect-work/` tree is preserved by default. Failed XSTAR products are retained for inspection and are never automatically deleted.

1.5.3 True MPI XSTAR2XSPEC (0.6.86 accepted)

MPI is opt-in at build time:

```
make -C src/xstar_tools/xstar/cpp mpi
```

Run one grid over four MPI ranks:

```
mpirun -np 4 src/xstar_tools/xstar/cpp/xstar-xspect-mpi \
--input xstinitable.par \
--data-dir /shared/xstar/data \
--output-dir /shared/run_mpi
```

There is no local --processes option in `xstar-xspect-mpi`; `mpirun/mpiexec -np N` supplies the MPI rank count. The initial implementation requires a shared filesystem for executable, atomic-data, work, and output paths.

1.5.4 Python API

The established Python interface remains available:

```
from xstar_tools import XStarConfig, BackendMode, run_xstar

config = XStarConfig(
    input_file="xstar.par",
    data_dir="/path/to/xstar/data",
    output_dir="run_python",
    mode=BackendMode.PURE_PYTHON,
)
```

(continues on next page)

(continued from previous page)

```
result = run_xstar(config)

if not result.success:
    raise RuntimeError(result.status)
```

1.6 Configuration

XStarConfig is the stable configuration object for public Python execution.

Supported input sources are:

```
XStarConfig.from_par_file(...)
XStarConfig.from_mapping(...)
XStarConfig.from_fortran_run_directory(...)
```

Exactly one parameter source is allowed per configuration. Textual numerical literals read from HEASoft/IRAF-style parameter files or command-style sources are preserved as text until the qualified execution path consumes them.

Important public controls include:

- `data_dir` — XSTAR data directory;
- `output_dir` — run directory;
- `mode` — one of the five stable execution modes;
- `threads` — run-scoped thread count;
- `progress` and `log_level`;
- `reproducible` — stable-run reproducibility intent;
- `overwrite` — explicit output replacement;
- cache controls for development/reference workflows.

Advanced backend overrides are accepted only for Python-controller modes. Monolithic `zone-cpp`, `zone-all`, and `xstar-cpp` reject them so their public names remain exact contracts.

A configuration can be exported back to a deterministic HEASoft/IRAF-style parameter file:

```
config.to_par_file("canonical-xstar.par")
```

1.7 Choosing a backend

Mode	Python troller	con-	C++ science	Stan- dalone	Recommended role
pure-python	yes		no	no	source-faithful reference/debugging
zone-python	yes		modular kernels	no	accelerated Python
zone-cpp	invocation only		shared zone production	no	production shared-zone runs
zone-all	invocation only		shared all-zone produc- tion	no	production shared-all runs
xstar-cpp	no for direct run		native production	yes	Python-free native execution

Use:

```
xstar-tools backends
```

to see host availability and native identities.

The old internal backend flags remain advanced compatibility aliases for the current deprecation period. New scripts should use the stable mode names.

A public mode name is intentionally stronger than a convenience alias: `zone-cpp`, `zone-all`, and `xstar-cpp` cannot be mixed with modular backend overrides.

1.7.1 Native platform scope

The native non-MPI path is qualified on Linux GCC x86_64, macOS Apple Clang arm64/x86_64, and Windows MSYS2 UCRT64/AMD64. `xstar-xspec --processes N` is the accepted local parallel table path on all of those hosts. True MPI is an explicit Linux/HPC-oriented build and Windows MPI is not planned; Windows native parallel execution uses the accepted non-MPI local-process path.

1.8 Output products

XStarProducts provides typed paths for the principal products:

Accessor	File
abundances	xout_abund1.fits
lines	xout_lines1.fits
rrc	xout_rrc1.fits
continuum	xout_cont1.fits
spectrum	xout_spect1.fits
detail	xo01_detail.fits
detail_lines	xo01_detal2.fits
detail_rates	xo01_detal3.fits
detail_continuum	xo01_detal4.fits
step_log	xout_step.log

Example:

```
result = run_xstar(config)
print(result.products.lines)
print(result.products.produced_fits)
print(result.step_log)
```

XStarResult also reports status, return code, runtime, timings, diagnostics, warnings, and provenance.

1.8.1 XSTAR2XSPEC products

A successful native `xstar-xspec` or `xstar-xspec-mpi` run publishes:

```
xout_ain.fits
xout_aout.fits
xout_mtable.fits
xout_etable.fits
xout_step.log
```

(continues on next page)

(continued from previous page)

```
xstinitable.lis
xstinitable.fits
```

Per-job products remain under `xstar2xspec-work/jobs/NNNNNN/` by default. Each successfully completed job includes `xstar-cpp.success`; failed/partial products are deliberately retained without that success marker.

Local-process orchestration writes `xstar2xspec.log` and `xstar2xspec_scheduler.log`. True MPI writes `xstar2xspec-mpi.log`, `xstar2xspec-mpi-scheduler.log`, and one `xstar2xspec-mpi-rank-NNNNNN.log` per rank.

1.9 Reproducibility and provenance

The distribution version and scientific revision are intentionally independent. A productization-only release can change documentation or interfaces without changing accepted science.

Record both when publishing or comparing a run:

```
xstar-tools version
```

Stable provenance can include:

- requested and actual execution mode;
- package version and science revision;
- C API and production-zone ABI;
- C++ library/executable identity;
- CPU feature/dispatch information;
- fallback events;
- atomic database, `coheat.dat`, and package-constant paths/hashes.

CLI execution can persist the stable result object and event stream:

```
xstar-tools run xstar.par \
--mode zone-python \
--data-dir /path/to/xstar/data \
--output-dir run1 \
--summary-json run1-summary.json \
--json-log run1-events.jsonl
```

Run-scoped thread/reproducibility environment settings are restored after each stable Python API run.

1.10 Performance expectations

Performance depends strongly on element, ionization regime, density, spectrum, host CPU, compiler, and which scientific paths are active. The project therefore avoids promising one universal speedup.

General guidance:

- `pure-python` prioritizes source-faithful transparency and is normally the slowest public mode;
- `zone-python` keeps Python orchestration but uses qualified modular C++ kernels;
- `zone-cpp` and `zone-all` use the shared native production implementation;

- direct `xstar-cpp` removes the Python runtime from normal standalone execution.

For reproducible performance work, record the mode, compiler/native build, CPU features, thread count, science revision, atomic-data hashes, and input model. Scientific acceptance and performance are separate gates.

Historical optimization measurements remain available through version-control history and release tags; they are development evidence, not performance guarantees for arbitrary systems.

1.10.1 XSTAR2XSPEC concurrency

`xstar-xspec --processes N` is process parallelism, not thread parallelism. N independent `xstar-cpp` processes may be runnable at the same time, which can consume approximately N times the per-model memory in the worst case. The OS chooses logical CPUs unless the user or batch system applies affinity.

`xstar-xspec-mpi` uses one XSTAR child per MPI rank. `mpirun -np N` therefore permits up to N concurrent XSTAR calculations. On clusters, choose rank count from both CPU and memory limits; do not infer a safe rank count from CPU count alone.

1.11 Troubleshooting

1.11.1 Data validation fails

Confirm the selected directory contains readable `atdb.fits` and `coheat.dat`:

```
xstar-tools doctor --data-dir /path/to/xstar/data
```

1.11.2 A C++ mode is unavailable

Inspect the installed native-build metadata and capabilities:

```
xstar-tools backends
xstar-tools doctor --require zone-cpp
```

For a source checkout, build with `make -C src/xstar_tools/xstar/cpp -j2`. For a Linux source/wheel installation that must contain native support, reinstall with `XSTAR_TOOLS_NATIVE=required`. Python-only installations intentionally expose only `pure-python`.

1.11.3 Output directory already exists

Stable runs reject non-empty output directories by default. Choose a new directory or explicitly request replacement with Python `overwrite=True` / CLI `--overwrite`.

1.11.4 Package version and science revision differ

That is expected during productization. Use `xstar-tools version`; do not infer the scientific baseline from the distribution version.

1.11.5 A comparison reports inventory differences

The qualification framework intentionally separates material numerical science from identity/order/inventory diagnostics. Known accepted structural exceptions are encoded in the parity-freeze metadata rather than hidden by broad tolerances.

1.11.6 Need development diagnostics

Use `xstar-tools dev ...` or `xstar-tools qualify ....` Normal user workflows should not depend on qualification marker strings.

1.11.7 `xstar-xspec --processes 2` does not create MPI ranks

That option creates two independent local `xstar-cpp` child **processes**. It is not a thread count and it does not call `mpirun`. Use `xstar-xspec-mpi` for true MPI.

1.11.8 `make mpi` says `mpic++` is missing

MPI is optional and is intentionally excluded from the default build. Install/load an MPI implementation that provides a C++ wrapper, or point the build at the correct wrapper:

```
make -C src/xstar_tools/xstar/cpp mpi MPICXX=/path/to/mpic++
```

1.11.9 MPI ranks cannot see the plan, data, or job directories

The first `xstar-xspec-mpi` implementation requires a shared filesystem. Use paths visible from every node and prefer an explicit shared atomic-data directory:

```
mpirun -np 4 xstar-xspec-mpi \
  --input /shared/model/xstinitable.par \
  --data-dir /shared/xstar/data \
  --output-dir /shared/results/run1
```

For Python applications, continue with the *Python guide*. For direct native executables, see *C++ and native CLI reference*.

PYTHON GUIDE

The Python interfaces cover two complementary tasks:

1. stable orchestration of XSTAR runs through `XStarConfig` and `run_xstar`;
2. direct inspection and analysis of the packed XSTAR atomic database through `XSTARAtomic` and specialist modules.

They also include readers for XSTAR FITS products and Python wrappers around the native XSTAR2XSPEC planner/table pipeline.

2.1 Running XSTAR from Python

The stable application API is exported from `xstar_tools`:

```
from xstar_tools import BackendMode, XStarConfig, XStarData, run_xstar
```

2.1.1 From `xstar.par`

```
from xstar_tools import XStarConfig, run_xstar

config = XStarConfig.from_par_file(
    "xstar.par",
    data_dir="/path/to/xstar/data",
    output_dir="run1",
    mode="zone-cpp",
    threads=1,
)
config.validate()
result = run_xstar(config)

print(result.success)
print(result.status)
print(result.runtime_seconds)
print(result.products.spectrum)
```

2.1.2 From a parameter mapping

```
config = XStarConfig.from_mapping(
    {
        "spectrum": "pow",
```

(continues on next page)

(continued from previous page)

```
"density": "1e12",
"column": "1e20",
"rlogxi": "1",
"npass": "1",
"modelname": "python_example",
},
data_dir="/path/to/xstar/data",
output_dir="run_mapping",
mode="pure-python",
)
```

Textual numeric values can be retained as strings when preserving source input spelling matters.

2.1.3 From an existing FORTRAN run directory

```
config = XStarConfig.from_fortran_run_directory(
    "/path/to/fortran/run",
    data_dir="/path/to/xstar/data",
    output_dir="run_from_fortran_inputs",
    mode="zone-cpp",
)
```

The directory must contain `run_xstar.sh`.

2.1.4 Inspect or serialize the configuration

```
mapping = config.parameter_mapping()
command = config.to_xstar_command()
config.to_par_file("normalized_xstar.par")
```

`validate()` checks the input source, selected execution mode, and local XSTAR data without running the scientific calculation.

2.1.5 Results

`run_xstar(XStarConfig(...))` returns `XStarResult`. Useful fields include:

```
result.success
result.status
result.return_code
result.output_dir
result.runtime_seconds
result.provenance
result.diagnostics
result.warnings
result.products.expected_fits()
result.products.produced_fits()
result.as_dict()
```

`XStarProducts` provides deterministic paths for the standard XSTAR products, including abundance, line, RRC, continuum, spectrum, detail FITS products, and `xout_step.log`.

2.1.6 Output directory policy

A non-empty output directory is rejected by default. Use `overwrite=True` when deliberate replacement is wanted:

```
config = XStarConfig.from_par_file(
    "xstar.par",
    data_dir="/path/to/xstar/data",
    output_dir="run1",
    overwrite=True,
)
```

2.1.7 Stable modes

BackendMode exposes PURE_PYTHON, ZONE_PYTHON, ZONE_CPP, ZONE_ALL, and XSTAR_CPP. ExecutionMode is also exported for the underlying execution classification; applications normally choose BackendMode/its string values through XStarConfig.

The older keyword-style call remains available for compatibility:

```
run_xstar(mode="zone-cpp", run_script="run_xstar.sh", atdb_path="/path/to/atdb.fits")
```

New applications should prefer the XStarConfig form because it returns the typed XStarResult contract.

2.2 Atomic database API

XSTARAtomic is the high-level Python interface to XSTAR's packed atdb.fits database:

```
from xstar_tools.api import XSTARAtomic

with XSTARAtomic("/path/to/atdb.fits") as atomic:
    print(atomic.summary())
```

By default the record/element/ion index is built at construction time. For repeated work, an NPZ index cache can be enabled:

```
atomic = XSTARAtomic(
    "/path/to/atdb.fits",
    index_cache=True,
    index_cache_format="npz",
)
```

2.2.1 Ion selection

Ion selectors accept normal element/ion notation supported by `parse_ion`. Typical calls are:

```
levels = atomic.levels("O VII")
lines = atomic.lines("O VIII", wavelength=(18.8, 19.1), slim=True)
wavelengths = atomic.get_wavelengths("Ne IX")
line = atomic.match_line("O VIII", wavelength=18.97)
```

2.2.2 Photoionization

```
photo = atomic.photoionization(  
    "O VII",  
    threshold_ev=(500.0, 1000.0),  
    include_grid=True,  
)
```

With `include_grid=True`, the result contains `summary` and matching raw grid rows.

2.2.3 Collisions

```
collision = atomic.collisions(  
    "O VII",  
    temperatures=[1.0e5, 1.0e6, 1.0e7],  
    include_grid=True,  
)  
print(collision["summary"])  
print(collision["evaluated"])
```

The returned mapping also retains the earlier aliases `matches` and `evaluated_rates`.

2.2.4 Recombination and emissivity

```
recomb = atomic.recombination("O VIII", temperatures=[1.0e5, 1.0e6])  
emis = atomic.emissivity("O VII", temperature=1.0e6, electron_density=1.0e10)
```

For convenience, the high-level object also exposes `get_recombination`, `calc_emissivity`, and `calc_triplet` compatibility helpers.

2.2.5 Population/matrix tools

```
matrix = atomic.build_matrix("O VII", temperature=1.0e6, electron_density=1.0e10)  
pop = atomic.solve_populations("O VII", temperature=1.0e6, electron_density=1.0e10)
```

These methods delegate to the maintained matrix/solver modules; they do not replace the source-faithful XSTAR run controller.

2.2.6 Type-50 and XSTAR-run diagnostics

Advanced APIs include:

```
atomic.type50_rate(...)  
atomic.audit_type50_line_pumping(...)  
atomic.reproduce_xstar_run("/path/to/run", ...)  
atomic.build_xstar_local_target("/path/to/run", ...)
```

These are intended for detailed atomic/rate validation and comparison work. Use the stable execution API in *Running XSTAR from Python* for ordinary full XSTAR calculations.

2.3 Reading XSTAR output products

`xstar_tools.xstar_outputs` provides lightweight readers for standard XSTAR FITS products. Where possible it uses Astropy; selected XSTAR table products also have a small internal FITS fallback for minimal diagnostic environments.

```
from xstar_tools.xstar_outputs import (
    list_fits_hdus,
    read_xout_abundances,
    read_xout_lines,
    read_xout_parameters,
    read_xout_spectra,
)

lines = read_xout_lines("run1/xout_lines1.fits")
spectra = read_xout_spectra("run1/xout_spect1.fits")
params = read_xout_parameters("run1/xout_spect1.fits")
abund = read_xout_abundances("run1/xout_abund1.fits")
hdus = list_fits_hdus("run1/xout_spect1.fits")
```

2.3.1 Generic table access

```
from xstar_tools.xstar_outputs import read_fits_table, read_fits_table_hdus

rows = read_fits_table("run1/xout_lines1.fits", "XSTAR_LINES")
```

2.3.2 Filtering and CSV conversion

```
from xstar_tools.xstar_outputs import filter_rows, write_csv, write_xout_spectra_csv

selected = filter_rows(lines, ion="O VIII")
write_csv(selected, "o8_lines.csv")
write_xout_spectra_csv(spectra, "spectrum.csv")
```

The line-emission columns in XSTAR outputs are model-dependent radiative-transfer quantities. Do not equate them directly with local atomic emissivity coefficients unless the ion fractions, geometry, density, and radiative-transfer assumptions are matched.

A command-line interface is also installed as `xstar-tools-xstar-outputs` for conversion/inspection workflows.

2.4 XSTAR2XSPEC from Python

The `xstar_tools.tables` package wraps the native XSTAR2XSPEC executables without reimplementing their scientific/table semantics in Python.

```
from xstar_tools.tables import (
    XSpecTableProducts,
    build_xspec_tables,
    build_xstinitable,
    run_xstar2xspec,
)
```

2.4.1 Build the grid plan

```
build_xstinitable(  
    input_file="xstinitable.par",  
    data_dir="/path/to/xstar/data",  
    output_dir="grid_plan",  
    xstar="cpp",  
)
```

This creates `xstinitable.lis` and `xstinitable.fits`. `xstar="fortran"` emits the canonical historical `xstar` `key=value` command contract instead of `xstar-cpp` commands.

2.4.2 Run the complete pipeline

```
run_xstar2xspec(  
    input_file="xstinitable.par",  
    data_dir="/path/to/xstar/data",  
    output_dir="table_run",  
    processes=4,  
    restart=True,  
)
```

`processes` is local OS-process concurrency, not a thread count and not MPI. The older `workers=` keyword remains a compatibility alias.

2.4.3 Build tables from existing spectra

```
build_xspec_tables(  
    "grid_plan/xstinitable.fits",  
    ["job1/xout_spect1.fits", "job2/xout_spect1.fits"],  
    "tables",  
    initable=True,  
)  
  
products = XSpecTableProducts.from_directory("tables")  
print(products.ain)  
print(products.aout)  
print(products.mtable)  
print(products.etable)
```

The input spectra must be supplied in the planner's loopcontrol order. For new production workflows, use the native/canonical `xstinitable.fits` contract (`initable=True`) rather than the older characterization metadata text format.

The generated API reference is under *Python API reference*.

PYTHON API REFERENCE

Generated reference pages complement the task-oriented *Python guide* guide.

3.1 Stable execution API

The stable application execution interface is generated from source type hints and docstrings.

<i>BackendMode</i>	alias of <i>ExecutionMode</i>
<i>ExecutionMode</i>	
<i>XStarConfig</i>	Validated configuration for one stable public XSTAR run.
<i>XStarData</i>	Stable locator for local XSTAR atomic/scientific data.
<i>XStarProducts</i>	Deterministic typed paths for principal XSTAR output products.
<i>XStarResult</i>	Stable public result returned by <i>xstar_tools.run_xstar()</i> .
<i>run_xstar</i>	Run XSTAR through the stable public API.
<i>backends.available</i>	
<i>backends.describe</i>	

3.1.1 *xstar_tools.BackendMode*

xstar_tools.BackendMode
alias of *ExecutionMode*

3.1.2 *xstar_tools.ExecutionMode*

class *xstar_tools.ExecutionMode*(*values)

__init__(*args, **kws)

Methods

encode ([encoding, errors])	Encode the string using the codec registered for encoding.
------------------------------------	--

continues on next page

Table 2 – continued from previous page

<code>replace(old, new, /[, count])</code>	Return a copy with all occurrences of substring <code>old</code> replaced by <code>new</code> .
<code>split([sep, maxsplit])</code>	Return a list of the substrings in the string, using <code>sep</code> as the separator string.
<code>rsplit([sep, maxsplit])</code>	Return a list of the substrings in the string, using <code>sep</code> as the separator string.
<code>join(iterable, /)</code>	Concatenate any number of strings.
<code>capitalize()</code>	Return a capitalized version of the string.
<code>casefold()</code>	Return a version of the string suitable for caseless comparisons.
<code>title()</code>	Return a version of the string where each word is titlecased.
<code>center(width[, fillchar])</code>	Return a centered string of length <code>width</code> .
<code>count</code>	Return the number of non-overlapping occurrences of substring <code>sub</code> in string <code>S[start:end]</code> .
<code>expandtabs([tabsize])</code>	Return a copy where all tab characters are expanded using spaces.
<code>find</code>	Return the lowest index in <code>S</code> where substring <code>sub</code> is found, such that <code>sub</code> is contained within <code>S[start:end]</code> .
<code>partition(sep, /)</code>	Partition the string into three parts using the given separator.
<code>index</code>	Return the lowest index in <code>S</code> where substring <code>sub</code> is found, such that <code>sub</code> is contained within <code>S[start:end]</code> .
<code>ljust(width[, fillchar])</code>	Return a left-justified string of length <code>width</code> .
<code>lower()</code>	Return a copy of the string converted to lowercase.
<code>lstrip([chars])</code>	Return a copy of the string with leading whitespace removed.
<code>rfind</code>	Return the highest index in <code>S</code> where substring <code>sub</code> is found, such that <code>sub</code> is contained within <code>S[start:end]</code> .
<code>rindex</code>	Return the highest index in <code>S</code> where substring <code>sub</code> is found, such that <code>sub</code> is contained within <code>S[start:end]</code> .
<code>rjust(width[, fillchar])</code>	Return a right-justified string of length <code>width</code> .
<code>rstrip([chars])</code>	Return a copy of the string with trailing whitespace removed.
<code>rpartition(sep, /)</code>	Partition the string into three parts using the given separator.
<code>splitlines([keepends])</code>	Return a list of the lines in the string, breaking at line boundaries.
<code>strip([chars])</code>	Return a copy of the string with leading and trailing whitespace removed.
<code>swapcase()</code>	Convert uppercase characters to lowercase and lowercase characters to uppercase.
<code>translate(table, /)</code>	Replace each character in the string using the given translation table.
<code>upper()</code>	Return a copy of the string converted to uppercase.
<code>startswith</code>	Return True if the string starts with the specified prefix, False otherwise.
<code>endswith</code>	Return True if the string ends with the specified suffix, False otherwise.
<code>removeprefix(prefix, /)</code>	Return a str with the given prefix string removed if present.

continues on next page

Table 2 – continued from previous page

<code>removesuffix(suffix, /)</code>	Return a str with the given suffix string removed if present.
<code>isascii()</code>	Return True if all characters in the string are ASCII, False otherwise.
<code>islower()</code>	Return True if the string is a lowercase string, False otherwise.
<code>isupper()</code>	Return True if the string is an uppercase string, False otherwise.
<code>istitle()</code>	Return True if the string is a title-cased string, False otherwise.
<code>isspace()</code>	Return True if the string is a whitespace string, False otherwise.
<code>isdecimal()</code>	Return True if the string is a decimal string, False otherwise.
<code>isdigit()</code>	Return True if the string is a digit string, False otherwise.
<code>isnumeric()</code>	Return True if the string is a numeric string, False otherwise.
<code>isalpha()</code>	Return True if the string is an alphabetic string, False otherwise.
<code>isalnum()</code>	Return True if the string is an alpha-numeric string, False otherwise.
<code>isidentifier()</code>	Return True if the string is a valid Python identifier, False otherwise.
<code>isprintable()</code>	Return True if all characters in the string are printable, False otherwise.
<code>zfill(width, /)</code>	Pad a numeric string with zeros on the left, to fill a field of the given width.
<code>format(*args, **kwargs)</code>	Return a formatted version of the string, using substitutions from args and kwargs.
<code>format_map(mapping, /)</code>	Return a formatted version of the string, using substitutions from mapping.
<code>maketrans</code> <code>__init__(*args, **kws)</code>	Return a translation table usable for <code>str.translate()</code> .

Attributes

<code>PURE_PYTHON</code>
<code>ZONE_PYTHON</code>
<code>ZONE_CPP</code>
<code>ZONE_ALL</code>
<code>XSTAR_CPP</code>

3.1.3 xstar_tools.XStarConfig

```
class xstar_tools.XStarConfig(input_file=None, data_dir=None, output_dir='run1',
                              mode=ExecutionMode.PURE_PYTHON, threads=1, progress=True,
                              log_level='INFO', reproducible=True, overwrite=False, cache_dir=None,
                              use_cache=True, rebuild_cache=False,
                              advanced_backend_overrides=<factory>, parameters=None,
                              fortran_run_directory=None)
```

Validated configuration for one stable public XSTAR run.

Exactly one input source is required: `input_file`, `parameters` or `fortran_run_directory`. Use the named constructors for the clearest API.

`overwrite=False` gives the public API a deterministic output-directory policy: a non-empty existing directory is rejected before science executes. With `overwrite=True` the directory is replaced before the run.

Parameters

- `input_file` (*Path* | *str* | *None*)
- `data_dir` (*Path* | *str* | *Any* | *None*)
- `output_dir` (*Path* | *str*)
- `mode` (*ExecutionMode* | *str*)
- `threads` (*int*)
- `progress` (*bool*)
- `log_level` (*str*)
- `reproducible` (*bool*)
- `overwrite` (*bool*)
- `cache_dir` (*Path* | *str* | *None*)
- `use_cache` (*bool*)
- `rebuild_cache` (*bool*)
- `advanced_backend_overrides` (*Mapping*[*str*, *str*])
- `parameters` (*Mapping*[*str*, *Any*] | *None*)
- `fortran_run_directory` (*Path* | *str* | *None*)

```
__init__(input_file=None, data_dir=None, output_dir='run1', mode=ExecutionMode.PURE_PYTHON,
          threads=1, progress=True, log_level='INFO', reproducible=True, overwrite=False,
          cache_dir=None, use_cache=True, rebuild_cache=False,
          advanced_backend_overrides=<factory>, parameters=None, fortran_run_directory=None)
```

Parameters

- `input_file` (*Path* | *str* | *None*)
- `data_dir` (*Path* | *str* | *Any* | *None*)
- `output_dir` (*Path* | *str*)
- `mode` (*ExecutionMode* | *str*)
- `threads` (*int*)
- `progress` (*bool*)

- `log_level` (*str*)
- `reproducible` (*bool*)
- `overwrite` (*bool*)
- `cache_dir` (*Path* | *str* | *None*)
- `use_cache` (*bool*)
- `rebuild_cache` (*bool*)
- `advanced_backend_overrides` (*Mapping*[*str*, *str*])
- `parameters` (*Mapping*[*str*, *Any*] | *None*)
- `fortran_run_directory` (*Path* | *str* | *None*)

Return type

None

Methods

<code>__init__</code> (<i>[input_file, data_dir, output_dir, ...]</i>)	
<code>from_fortran_run_directory</code> (<i>directory</i> , <i>**kwargs</i>)	
<code>from_mapping</code> (<i>parameters</i> , <i>**kwargs</i>)	
<code>from_par_file</code> (<i>path</i> , <i>**kwargs</i>)	
<code>parameter_mapping</code> ()	
<code>source_run_script</code> ()	Return an existing source-faithful run script when one is available.
<code>to_par_file</code> (<i>path</i>)	
<code>to_xstar_command</code> ()	
<code>validate</code> ()	Validate input/data/backend combinations without executing science.

Attributes

<code>cache_dir</code>
<code>data_dir</code>
<code>fortran_run_directory</code>
<code>input_file</code>
<code>log_level</code>

continues on next page

Table 5 – continued from previous page

mode
mode_name
output_dir
overwrite
parameters
progress
rebuild_cache
reproducible
source_kind
threads
use_cache
advanced_backend_overrides

3.1.4 xstar_tools.XStarData

class xstar_tools.XStarData(directory, cache_dirs=())

Stable locator for local XSTAR atomic/scientific data.

Validation is intentionally local-only. It never downloads `atdb.fits` or any other large scientific data as a side effect of a run.

Parameters

- **directory** (*Path*)
- **cache_dirs** (*tuple*[*Path*, ...])

__init__(directory, cache_dirs=())

Parameters

- **directory** (*Path*)
- **cache_dirs** (*tuple*[*Path*, ...])

Return type

None

Methods

__init__(directory[, cache_dirs])

continues on next page

Table 6 – continued from previous page

<code>from_directory(directory, *[, cache_dirs])</code>
<code>identity()</code>
<code>validate()</code>

Attributes

<code>atdb</code>
<code>cache_dirs</code>
<code>coheat</code>
<code>constants</code>
<code>directory</code>

3.1.5 xstar_tools.XStarProducts

class `xstar_tools.XStarProducts`(*output_dir*)

Deterministic typed paths for principal XSTAR output products.

Parameters

output_dir (*Path*)

__init__(*output_dir*)

Parameters

output_dir (*Path*)

Return type

None

Methods

<code>__init__(output_dir)</code>
<code>as_dict()</code>

Attributes

<code>abundances</code>
<code>continuum</code>

continues on next page

Table 9 – continued from previous page

detail
detail_continuum
detail_lines
detail_rates
expected_fits
lines
produced_fits
rrc
spectrum
step_log
output_dir

3.1.6 xstar_tools.XStarResult

class xstar_tools.XStarResult(*success, status, return_code, output_dir, products, step_log, runtime_seconds, timings=<factory>, provenance=<factory>, diagnostics=(), warnings=(), raw_summary=<factory>*)

Stable public result returned by `xstar_tools.run_xstar()`.

Parameters

- **success** (*bool*)
- **status** (*str*)
- **return_code** (*int*)
- **output_dir** (*Path*)
- **products** (*XStarProducts*)
- **step_log** (*Path*)
- **runtime_seconds** (*float*)
- **timings** (*Mapping[str, Any]*)
- **provenance** (*Mapping[str, Any]*)
- **diagnostics** (*tuple[str, ...]*)
- **warnings** (*tuple[str, ...]*)
- **raw_summary** (*Mapping[str, Any]*)

__init__(*success, status, return_code, output_dir, products, step_log, runtime_seconds, timings=<factory>, provenance=<factory>, diagnostics=(), warnings=(), raw_summary=<factory>*)

Parameters

- **success** (*bool*)
- **status** (*str*)
- **return_code** (*int*)
- **output_dir** (*Path*)
- **products** (*XStarProducts*)
- **step_log** (*Path*)
- **runtime_seconds** (*float*)
- **timings** (*Mapping[str, Any]*)
- **provenance** (*Mapping[str, Any]*)
- **diagnostics** (*tuple[str, ...]*)
- **warnings** (*tuple[str, ...]*)
- **raw_summary** (*Mapping[str, Any]*)

Return type

None

Methods

<code>__init__(success, status, return_code, ...)</code>
<code>as_dict()</code>

Attributes

<code>diagnostics</code>	
<code>mode</code>	
<code>produced_fits</code>	
<code>ready</code>	Compatibility alias for the Milestone-3 <code>PublicRunResult.ready</code> .
<code>warnings</code>	
<code>success</code>	
<code>status</code>	
<code>return_code</code>	
<code>output_dir</code>	

continues on next page

Table 11 – continued from previous page

products
step_log
runtime_seconds
timings
provenance
raw_summary

3.1.7 xstar_tools.run_xstar

```
xstar_tools.run_xstar(config=None, *(Keyword-only parameters separator (PEP 3102)),  
                      mode=ExecutionMode.PURE_PYTHON, run_script=None, command=None,  
                      atdb_path=None, coheat_path=None, output_dir='.', **kwargs)
```

Run XSTAR through the stable public API.

Recommended usage is `run_xstar(XStarConfig(...))`. The Milestone-3 keyword-style call remains a compatibility alias and returns `PublicRunResult`.

Parameters

- **config** (*Any*)
- **mode** (*str* / `ExecutionMode`)
- **run_script** (*str* / *Path* / `None`)
- **command** (*str* / `None`)
- **atdb_path** (*str* / *Path* / `None`)
- **coheat_path** (*str* / *Path* / `None`)
- **output_dir** (*str* / *Path*)
- **kwargs** (*Any*)

3.1.8 xstar_tools.backends.available

```
xstar_tools.backends.available()
```

Return type

`dict[str, bool]`

3.1.9 xstar_tools.backends.describe

```
xstar_tools.backends.describe()
```

Return type

`dict[str, Any]`

3.1.10 Public module documentation

xstar-tools: source-faithful Python/C++ tools for XSTAR workflows.

Top-level imports are intentionally lightweight. Use subpackages such as `xstar_tools.atomic` and `xstar_tools.xstar` for functionality. Stable public run configuration for `xstar_tools`.

This module is a productization layer. It normalizes user-facing input sources without implementing XSTAR science; execution remains delegated to the frozen qualified paths in `xstar_tools.execution`. Stable public result and product objects for `xstar_tools`. Data-location and download helpers for XSTAR `atdb.fits`.

The package does not bundle the large XSTAR atomic database. These helpers record a local data directory in the project-level `datapath` file and optionally retrieve `atdb.fits` from the public HEASARC/LHEASOFT distribution. Stable public execution modes for xstar-tools.

This module is intentionally a thin productization layer over the qualified execution paths. It does not implement scientific calculations itself. Capability and provenance API for stable xstar-tools execution modes.

3.2 Atomic-data API

<code>XSTARAtomic</code>	High-level interface to XSTAR's packed <code>atdb.fits</code> atomic database.
<code>parse_ion</code>	Normalize ion input into (Z, ion_stage, element_symbol).

3.2.1 xstar_tools.api.XSTARAtomic

```
class xstar_tools.api.XSTARAtomic(fitsfile=None, *, load_reals=True, build_index=True,
                                  index_cache=False, rebuild_index_cache=False,
                                  index_cache_format='npz')
```

High-level interface to XSTAR's packed `atdb.fits` atomic database.

Parameters

- **fitsfile** (`Union[str, Path, None]`) – Path to XSTAR's released packed atomic database, usually `xstar/data/atdb.fits`. If omitted, xstar-atomic resolves the path from `XSTAR_ATDB_FITS`, the persistent `datapath` file, or by prompting with `xstar_tools.download_data()`.
- **load_reals** (`bool`) – Whether to load/map the REALS array immediately. Most physical decoders need real values, so the default is `True`.
- **build_index** (`bool`) – If `True`, build the record/element/ion index at initialization.
- **index_cache** (`Union[bool, str, Path]`)
- **rebuild_index_cache** (`bool`)
- **index_cache_format** (`str`)

```
__init__(fitsfile=None, *, load_reals=True, build_index=True, index_cache=False,
          rebuild_index_cache=False, index_cache_format='npz')
```

Parameters

- **fitsfile** (`str | Path | None`)
- **load_reals** (`bool`)
- **build_index** (`bool`)

- `index_cache` (*bool* | *str* | *Path*)
- `rebuild_index_cache` (*bool*)
- `index_cache_format` (*str*)

Methods

<code>__init__</code> ([fitsfile, load_reals, ...])	
<code>audit_type50_line_pumping</code> (cases_csv, **kwargs)	Run the reusable He-like type-50 line-pumping audit workflow.
<code>build_index</code> (*[, index_cache, ...])	Build and cache (records, elements, ions) from the packed ATDB.
<code>build_matrix</code> (ion, **kwargs)	Workflow-style alias for <code>db.matrix.build_ion(...)</code> .
<code>build_xstar_local_target</code> (run_dir, *[, ion, ...])	Extract exact local-state/triplet target values from XSTAR outputs.
<code>calc_emissivity</code> (*args, **kwargs)	Alias for <code>emissivity()</code> .
<code>calc_triplet</code> ([ion])	Calculate a He-like triplet summary from rows or an XSTAR context.
<code>close</code> ()	Close the underlying FITS file handle.
<code>collisions</code> ([ion, element, ion_stage, ...])	Return collision summaries and evaluated rates.
<code>emissivity</code> ([ion, element, ion_stage, ...])	Build a direct-excitation emissivity table for selected lines.
<code>get_collisions</code> (*args, **kwargs)	Alias for <code>collisions()</code> .
<code>get_levels</code> (*args, **kwargs)	Alias for <code>levels()</code> for workflow-style code.
<code>get_lines</code> (*args, **kwargs)	Alias for <code>lines()</code> for workflow-style code.
<code>get_photoionization</code> (*args, **kwargs)	Alias for <code>photoionization()</code> .
<code>get_recombination</code> (*args, **kwargs)	Alias for <code>recombination()</code> .
<code>get_wavelengths</code> ([ion])	Return sorted wavelengths in Angstrom for selected line records.
<code>levels</code> ([ion, element, ion_stage])	Decode level records for an element/ion selection.
<code>lines</code> ([ion, element, ion_stage, wavelength, ...])	Return decoded radiative line records, optionally filtered.
<code>match_line</code> ([ion])	Return the closest selected line by wavelength or energy.
<code>photoionization</code> ([ion, element, ion_stage, ...])	Return photoionization/bound-free summaries, optionally with grid rows.
<code>recombination</code> ([ion, element, ion_stage, ...])	Return recombination inventory/evaluations and optional source terms.
<code>reproduce_xstar_run</code> (run_dir, *[, ion, ...])	Extract exact same-run XSTAR local targets and optionally compare the solver.
<code>select</code> ([ion, element, ion_stage])	
<code>solve_populations</code> (ion, **kwargs)	Workflow-style alias for <code>db.solve.ion(...)</code> .
<code>summary</code> ()	Return a compact hierarchy summary for the database.
<code>type50_rate</code> (*kwargs)	Evaluate an audit-only XSTAR type-50 bound-bound radiative rate.

Attributes

elements
ions
records

3.2.2 xstar_tools.api.parse_ion

`xstar_tools.api.parse_ion(ion=None, *, element=None, ion_stage=None)`

Normalize ion input into (Z, ion_stage, element_symbol).

Examples

"O VIII" -> (8, 8, "O") "o viii" -> (8, 8, "O") "o_viii" -> (8, 8, "O") "OVIII" -> (8, 8, "O") ("O", 8) -> (8, 8, "O") (8, 8) -> (8, 8, "O")

Parameters

- **ion** (*str* | *Tuple*[*str*, *int*] | *Tuple*[*int*, *int*] | *None*)
- **element** (*str* | *int* | *None*)
- **ion_stage** (*int* | *None*)

Return type

Tuple[*int* | *None*, *int* | *None*, *str* | *None*]

3.3 Output-reader API

<code>list_fits_hdus</code>	Return a compact inventory of HDUs and table columns in a FITS file.
<code>read_fits_table</code>	Read a named FITS table HDU into normalized row dictionaries.
<code>read_fits_table_hdus</code>	Read all HDUs with a given name and return per-HDU row lists.
<code>read_xout_abundances</code>	Read XSTAR <code>xout_abund1.fits</code> local-state tables.
<code>read_xout_lines</code>	Read an XSTAR <code>xout_lines*.fits</code> line table.
<code>read_xout_parameters</code>	Read the PARAMETERS table from an XSTAR output FITS file if present.
<code>read_xout_spectra</code>	Read an XSTAR continuum/spectrum table such as <code>xout_cont1.fits</code> .
<code>write_csv</code>	Write a list of dictionaries to CSV.
<code>write_xout_spectra_csv</code>	Write XSTAR spectrum rows to CSV for lower-level solver routines.

3.3.1 xstar_tools.xstar_outputs.list_fits_hdus

`xstar_tools.xstar_outputs.list_fits_hdus(path)`

Return a compact inventory of HDUs and table columns in a FITS file.

This is useful for XSTAR local-state products whose exact HDU names can vary between run modes, such as `xout_abund1.fits` and `xout_detail.fits`. If `astropy` is unavailable, a lightweight FITS ASCII-table fallback is used.

Parameters

path (*str* | *Path*)

Return type

List[Dict[str, Any]]

3.3.2 xstar_tools.xstar_outputs.read_fits_table

`xstar_tools.xstar_outputs.read_fits_table(path, hdu_name)`

Read a named FITS table HDU into normalized row dictionaries.

Unlike `read_xout_lines()`, this routine is deliberately generic and is used by diagnostics for `xout_abund1.fits` and related local-state files. If `astropy` is unavailable, XSTAR ASCII TABLE and BINTABLE extensions are parsed by a lightweight built-in reader.

Parameters

- **path** (*str* | *Path*)
- **hdu_name** (*str*)

Return type

List[Dict[str, Any]]

3.3.3 xstar_tools.xstar_outputs.read_fits_table_hdus

`xstar_tools.xstar_outputs.read_fits_table_hdus(path, hdu_name)`

Read all HDUs with a given name and return per-HDU row lists.

XSTAR detail products often contain several XSTAR_RADIAL extensions, one per printed radial zone. This helper preserves the HDU index so callers can map zone state back to the original detail files.

Parameters

- **path** (*str* | *Path*)
- **hdu_name** (*str*)

Return type

List[Dict[str, Any]]

3.3.4 xstar_tools.xstar_outputs.read_xout_abundances

`xstar_tools.xstar_outputs.read_xout_abundances(path)`

Read XSTAR `xout_abund1.fits` local-state tables.

XSTAR writes this file in `pprint.f90`. The ABUNDANCES extension carries one row per radial zone with columns such as `radius`, `delta_r`, `ion_parameter`, `x_e`, `n_p`, `pressure`, `temperature` (in units of 10^4 K), `frac_heat_error` and one column per ion. The COLUMNS extension stores integrated ionic columns, while HEATING and COOLING contain thermal-balance terms.

Parameters**path** (*str* | *Path*)**Return type***Dict[str, List[Dict[str, Any]]]*

3.3.5 xstar_tools.xstar_outputs.read_xout_lines

`xstar_tools.xstar_outputs.read_xout_lines(path, hdu_name='XSTAR_LINES')`

Read an XSTAR xout_lines*.fits line table.

Parameters

- **path** (*str* | *Path*) – Path to an XSTAR line-output FITS file, commonly xout_lines1.fits.
- **hdu_name** (*str*) – Name of the line-table HDU. Defaults to "XSTAR_LINES".

ReturnsLine rows with columns such as `ion`, `lower_level`, `upper_level`, `wavelength`, `emit_inward` and `emit_outward`.**Return type***list of dict*

3.3.6 xstar_tools.xstar_outputs.read_xout_parameters

`xstar_tools.xstar_outputs.read_xout_parameters(path, hdu_name='PARAMETERS')`

Read the PARAMETERS table from an XSTAR output FITS file if present.

Parameters

- **path** (*str* | *Path*)
- **hdu_name** (*str*)

Return type*List[Dict[str, Any]]*

3.3.7 xstar_tools.xstar_outputs.read_xout_spectra

`xstar_tools.xstar_outputs.read_xout_spectra(path, hdu_name='XSTAR_SPECTRA')`

Read an XSTAR continuum/spectrum table such as xout_cont1.fits.

The XSTAR writer `writespectra3.f90` stores an ASCII table named `XSTAR_SPECTRA` with columns normally called `energy`, `incident`, `transmitted`, `emit_inward` and `emit_outward`. This lightweight reader returns normalized lower-case column names and works with the built-in ASCII-table fallback when `astropy` is unavailable.

Parameters

- **path** (*str* | *Path*)
- **hdu_name** (*str*)

Return type*List[Dict[str, Any]]*

3.3.8 xstar_tools.xstar_outputs.write_csv

`xstar_tools.xstar_outputs.write_csv(rows, path)`

Write a list of dictionaries to CSV.

Parameters

- **rows** (*Sequence*[*Dict*[*str*, *Any*]] | *str* | *Path*) – Preferred order is `write_csv(rows, path)`. For compatibility with early test/example code, `write_csv(path, rows)` is also accepted.
- **path** (*str* | *Path* | *Sequence*[*Dict*[*str*, *Any*]]) – Preferred order is `write_csv(rows, path)`. For compatibility with early test/example code, `write_csv(path, rows)` is also accepted.

Return type

None

3.3.9 xstar_tools.xstar_outputs.write_xout_spectra_csv

`xstar_tools.xstar_outputs.write_xout_spectra_csv(rows, path)`

Write XSTAR spectrum rows to CSV for lower-level solver routines.

Parameters

- **rows** (*Sequence*[*Mapping*[*str*, *Any*]])
- **path** (*str* | *Path*)

Return type

Path

3.4 XSTAR2XSPEC Python API

<i>XSpecTableProducts</i>	Paths of the four table products written by native xstar2table.
<i>build_xspec_tables</i>	Build the four canonical-compatible XSPEC table products.
<i>build_xstinitable</i>	Create <code>xstinitable.lis</code> and <code>xstinitable.fits</code> .
<i>run_xstar2xspec</i>	Run the complete native XSTAR2XSPEC pipeline with bounded process parallelism.

3.4.1 xstar_tools.tables.XSpecTableProducts

`class xstar_tools.tables.XSpecTableProducts(ain, aout, mtable, etable)`

Paths of the four table products written by native xstar2table.

Parameters

- **ain** (*Path*)
- **aout** (*Path*)
- **mtable** (*Path*)
- **etable** (*Path*)

`__init__(ain, aout, mtable, etable)`

Parameters

- **ain** (*Path*)
- **aout** (*Path*)
- **mtable** (*Path*)
- **etable** (*Path*)

Return type

None

Methods

`__init__(ain, aout, mtable, etable)`

`from_directory(directory)`

Attributes

`ain`

`aout`

`mtable`

`etable`

3.4.2 xstar_tools.tables.build_xspec_tables

`xstar_tools.tables.build_xspec_tables(config, spectra, output_dir, *, initable=False, native_executable=None)`

Build the four canonical-compatible XSPEC table products.

By default config is the historical 0.6.81 metadata text format. With `initable=True`, config is the native/canonical `xstinitable.fits` contract used by the 0.6.84 production pipeline.

Parameters

- **config** (*str* | *PathLike[str]*)
- **spectra** (*Iterable[str | PathLike[str]]*)
- **output_dir** (*str* | *PathLike[str]*)
- **initable** (*bool*)
- **native_executable** (*str | PathLike[str] | None*)

Return type

CompletedProcess[str]

3.4.3 xstar_tools.tables.build_xstinitable

```
xstar_tools.tables.build_xstinitable(parameters=(), output_dir='.', *, input_file=None, xstar='cpp',
                                     data_dir=None, native_executable=None)
```

Create `xstinitable.lis` and `xstinitable.fits`.

`xstar='cpp'` is the default production contract and emits `xstar-cpp` commands. `xstar='fortran'` preserves the canonical historical `xstar` `key=value` command contract. A HEASoft/IRAF-style `xstinitable.par` may be supplied via `input_file` and trailing parameters override values loaded from that file.

Parameters

- **parameters** (*Iterable*[*str*])
- **output_dir** (*str* | *PathLike*[*str*])
- **input_file** (*str* | *PathLike*[*str*] | *None*)
- **xstar** (*str*)
- **data_dir** (*str* | *PathLike*[*str*] | *None*)
- **native_executable** (*str* | *PathLike*[*str*] | *None*)

Return type

CompletedProcess[*str*]

3.4.4 xstar_tools.tables.run_xstar2xspect

```
xstar_tools.tables.run_xstar2xspect(parameters=(), output_dir='.', *, input_file=None, data_dir=None,
                                     processes=1, workers=None, save=False, restart=False,
                                     verbose=False, native_executable=None)
```

Run the complete native XSTAR2XSPEC pipeline with bounded process parallelism.

Parameters

- **parameters** (*Iterable*[*str*])
- **output_dir** (*str* | *PathLike*[*str*])
- **input_file** (*str* | *PathLike*[*str*] | *None*)
- **data_dir** (*str* | *PathLike*[*str*] | *None*)
- **processes** (*int*)
- **workers** (*int* | *None*)
- **save** (*bool*)
- **restart** (*bool*)
- **verbose** (*bool*)
- **native_executable** (*str* | *PathLike*[*str*] | *None*)

Return type

CompletedProcess[*str*]

C++ AND NATIVE CLI REFERENCE

The non-MPI native CLI is qualified on Linux GCC, macOS Apple Clang (arm64 and x86_64), and Windows MSYS2 UCRT64/MinGW-w64. True MPI remains an explicit Linux/HPC-oriented build. Windows MPI and MSVC are not planned; Windows local parallel execution uses `xstar-xspec --processes N`.

4.1 Native build

The accepted native build continues to use the retained GNU Make build rather than translating the scientific tree to a new build system.

4.1.1 Platform matrix

Platform	Toolchain	Non-MPI native status
Linux x86_64	GCC / GNU Make	ACCEPT
macOS arm64	Apple Clang / GNU Make	ACCEPT
macOS x86_64	Apple Clang / GNU Make	ACCEPT
Windows x86_64	MSYS2 UCRT64 / MinGW-w64 GCC / GNU Make	ACCEPT

Current accepted portability closure: 0.6.88.6.1.2.1.

4.1.2 Source checkout build

Normal non-MPI build:

```
make -C src/xstar_tools/xstar/cpp -j4
```

Inspect resolved platform values:

```
make -C src/xstar_tools/xstar/cpp print-config
```

Important build-abstraction variables include:

```
PLATFORM
SHLIB_EXT
EXEEXT
SHLIB_LDFLAGS
PIC_FLAGS
DL_LIBS
THREAD_LIBS
```

(continues on next page)

(continued from previous page)

```
RPATH_ORIGIN  
FILESYSTEM_LIBS
```

Linux

```
make -C src/xstar_tools/xstar/cpp -j4 PLATFORM=linux
```

Requirements include a C++17 compiler, CFITSIO development files, pkg-config, and Python development support for the embedded Python backend.

macOS

```
brew install cfitsio pkg-config  
make -C src/xstar_tools/xstar/cpp -j4 PLATFORM=macos
```

The accepted build uses .dylib, @rpath, and @loader_path conventions.

Windows

From an MSYS2 UCRT64 terminal:

```
make -C src/xstar_tools/xstar/cpp -j4 PLATFORM=windows
```

Windows builds .dll, .dll.a, and .exe artifacts and uses the qualified Win32 process backend. See [Windows native installation and usage](#) for packages and examples.

Windows MPI is intentionally out of scope.

4.1.3 Optional MPI build

xstar-xspec-mpi is opt-in and not part of all:

```
make -C src/xstar_tools/xstar/cpp mpi
```

or:

```
make -C src/xstar_tools/xstar/cpp xstar-xspec-mpi
```

Override the compiler wrapper when needed:

```
make -C src/xstar_tools/xstar/cpp mpi MPICXX=/path/to/mpic++
```

The accepted production use case is Linux/HPC. A shared filesystem is required by the current file-backed MPI implementation.

4.1.4 Python/wheel build controls

The Python packaging hook can request native compilation without changing the qualified Makefile/compiler policy:

```
XSTAR_TOOLS_NATIVE=auto|required|off  
XSTAR_TOOLS_NATIVE_JOBS=N
```

required converts missing native prerequisites into a build failure; off creates a Python-only installation. Packaging policy and direct source-build portability are deliberately treated as separate contracts.

4.2 xstar-cpp

xstar-cpp runs one native XSTAR calculation. It accepts an XSTAR parameter file or direct key=value parameters.

4.2.1 With xstar.par

```
xstar-cpp \
  --input xstar.par \
  --data-dir /path/to/xstar/data \
  --output-dir run_xstar
```

The parameter file may also be positional:

```
xstar-cpp xstar.par --data-dir /path/to/xstar/data --output run_xstar
```

Trailing key=value values override values loaded from the file.

4.2.2 Without a parameter file

```
xstar-cpp \
  --data-dir /path/to/xstar/data \
  --output-dir run_direct \
  spectrum=pow density=1e12 column=1e20 rlogxi=1 \
  temperature=100 pressure=0.03 npass=1 modelname=direct_example
```

Supply the full parameter contract required by the selected model when using direct values.

4.2.3 Options

Option	Meaning
--input PATH	HEASoft/IRAF-style .par file
--data-dir DIR	directory containing atdb.fits and coheat.dat
--input-dir DIR	source directory for fixed-name inputs such as density.dat
--output DIR, --output-dir DIR	output directory
--atomic-db PATH	explicit atdb.fits
--coheat PATH	explicit coheat.dat
--json-summary FILE	machine-readable run summary
--provenance FILE	additional provenance JSON
\-progress none	text
--debug	show internal version-tagged development/qualification diagnostics
--threads N	set OMP_NUM_THREADS for this model
--profile FILE	frontend wall-time/profile JSON
--deterministic	record reproducible-run intent in provenance
--print-option N	print the requested completed xout_step.log section
--parameters-out PATH	retain/write the generated parameter envelope
--abi	print compiled/runtime ABI identities
--version	print package/science/ABI versions
--help, -h	help

run-production --parameters parameters.json --output-dir DIR is the low-level production-envelope entry point used internally by the frontend; ordinary users should prefer the parameter-file/direct-value forms above.

4.2.4 Terminal output

With the default `--progress text`, `xstar-cpp` prints the FORTRAN-style pass header and accepted radial-zone rows, followed by `final print`, the spectral-publication messages, and total runtime. Internal version-tagged lines such as `V064895_...` are suppressed in ordinary runs.

Use `--debug` to restore the full diagnostic stream, or set `XSTAR_CPP_DEBUG=1` when invoking either `xstar-cpp` or the low-level `xstar_cpp run-production` executable. `--progress json` is an explicit machine-readable frontend mode and continues to emit JSON lifecycle events.

Normal mode also hides legacy first-evaluation and profiling output such as `V0648117_...`, `V064896_TYPE50_MODE`, `DETAIL_*_SECONDS`, and `PUBLIC_*_SECONDS`. These remain available with `--debug` or `XSTAR_CPP_DEBUG=1`.

4.2.5 Atomic-data discovery

Explicit paths have priority. If omitted, native discovery considers direct parameter/envelope paths, `XSTAR_ATOMIC_DB/XSTAR_ATDB_FITS`, `XSTAR_COHEAT`, `XSTAR_DATA`, `$HEADAS/refdata`, `$XSTAR_HOME/data`, executable/package-relative fallbacks, and the current directory as documented in [Atomic-data setup](#).

4.2.6 Threads versus processes

`xstar-cpp --threads N` controls thread-related environment for one model. `xstar-xspec --processes N` instead launches up to `N` independent `xstar-cpp` OS processes.

4.3 xstar-xspec-initable

`xstar-xspec-initable` is the native XSTAR2XSPEC grid planner. It creates:

```
xstinitable.lis
xstinitable.fits
```

The default command contract targets `xstar-cpp`.

4.3.1 From xstinitable.par

```
xstar-xspec-initable \
--input xstinitable.par \
--data-dir /path/to/xstar/data \
--output-dir plan
```

4.3.2 From direct values

```
xstar-xspec-initable \
--output-dir plan \
rlogxityp=2 rlogxisof=0 rlogxi=5 rlogxinst=6 rlogxiint=0 \
columnstyp=2 columnsof=1e20 column=1e24 columnnst=9 columnint=1
```

Trailing `key=value` arguments override values loaded from `--input`.

4.3.3 Options

Option	Meaning
<code>`-xstar cpp</code>	fortran, <code>-xstar`</code>
<code>--input PATH</code>	HEASoft/IRAF-style <code>xstinitable.par</code>
<code>--data-dir DIR, -data-dir DIR</code>	data directory embedded in generated <code>xstar-cpp</code> lines
<code>--output-dir DIR</code>	destination for <code>.lis</code> and <code>.fits</code> planner products
<code>--help, -h</code>	help

`--xstar fortran` preserves the canonical historical `xstar key=value` command contract. It does not change the grid definition itself.

The generated `xstinitable.fits` is the canonical metadata input for `xstar-xspec-table` and the complete `xstar-xspec/xstar-xspec-mpi` pipelines.

4.4 xstar-xspec-table

`xstar-xspec-table` converts loopcontrol-ordered `xout_spect1.fits` files into the four canonical-compatible XSPEC table products:

```
xout_ain.fits
xout_aout.fits
xout_mtable.fits
xout_etable.fits
```

4.4.1 Canonical native metadata

```
xstar-xspec-table \
  --initable plan/xstinitable.fits \
  --output-dir tables \
  jobs/000001/xout_spect1.fits \
  jobs/000002/xout_spect1.fits
```

The spectra must be listed in planner `loopcontrol` order.

4.4.2 Legacy characterization metadata

The older metadata-text contract remains available for regression/compatibility work:

```
xstar-xspec-table \
  --metadata CONFIG \
  --output-dir tables \
  xout_spect1_1.fits xout_spect1_2.fits
```

Exactly one of `--initable` or `--metadata` is required.

4.4.3 Options

Option	Meaning
<code>--initable PATH</code>	native/canonical <code>xstinitable.fits</code> metadata
<code>--metadata PATH</code>	legacy characterization metadata text
<code>--output-dir DIR</code>	destination for the four table products
positional spectra	one or more <code>xout_spect1.fits</code> files in loopcontrol order
<code>--help, -h</code>	help

For ordinary production use, prefer `--initable`. The complete `xstar-xspec` driver invokes this stage automatically after all grid jobs succeed.

4.5 xstar-xspec

`xstar-xspec` is the complete native serial/local-process XSTAR2XSPEC driver:

```
xstar-xspec-initable -> bounded xstar-cpp grid -> loopcontrol gather -> xstar-xspec-table
```

4.5.1 From `xstinitable.par`

Serial:

```
xstar-xspec --input xstinitable.par --output-dir run_serial --processes 1
```

Two simultaneous XSTAR processes:

```
xstar-xspec \  
  --input xstinitable.par \  
  --data-dir /path/to/xstar/data \  
  --output-dir run_local2 \  
  --processes 2
```

`--processes N` means independent **OS processes**, not C++ threads. `--workers N` and `-j N` are compatibility aliases. `-np` is reserved for MPI launchers.

4.5.2 Direct grid parameters

`--input` is optional when the full `xstinitable` parameter set is supplied as trailing `key=value` values:

```
xstar-xspec --output-dir run_direct --processes 2 \  
  rlogxityp=2 rlogxisof=0 rlogxi=5 rlogxinst=6 rlogxiint=0 \  
  columntyp=2 columnsof=1e20 column=1e24 columnnst=9 columnint=1
```

Trailing values override `--input` values.

4.5.3 Options

Option	Meaning
<code>--input PATH</code>	<code>xstinitable.par</code>
<code>--data-dir DIR, -data-dir DIR</code>	explicit atomic-data directory
<code>--output-dir DIR, --output DIR</code>	final XSTAR2XSPEC output directory
<code>--processes N</code>	maximum simultaneous <code>xstar-cpp</code> OS processes
<code>--workers N, -j N</code>	compatibility aliases for <code>--processes</code>
<code>--save</code>	compatibility flag; work/products are already preserved by default
<code>--cleanup-work</code>	remove <code>xstar2xspec-work/</code> only after full success
<code>--restart</code>	reuse completed jobs with required success products
<code>--verbose</code>	replay child output while assembling deterministic root log
<code>--initable-bin PATH</code>	override <code>xstar-xspec-initable</code> executable
<code>--xstar-cpp PATH</code>	override <code>xstar-cpp</code> executable
<code>--table-bin PATH</code>	override <code>xstar-xspec-table</code> executable
<code>--version</code>	package version
<code>--help, -h</code>	help

If `--data-dir` is omitted, each `xstar-cpp` child applies normal native data discovery.

4.5.4 Restart and work directory

With `--restart`, a job is reusable only when `xout_spect1.fits`, `xout_step.log`, and `xstar-cpp.success` exist. Partial/failed products remain available for inspection.

The work tree is:

```
xstar2xspec-work/jobs/NNNNNN/
```

Final root products are `xout_ain.fits`, `xout_aout.fits`, `xout_mtable.fits`, `xout_etable.fits`, and the canonical concatenated `xout_step.log`.

4.6 xstar-xspec-mpi

`xstar-xspec-mpi` is the accepted true-MPI XSTAR2XSPEC executable. It is separate from local-process `xstar-xspec`. The production use case is Linux/HPC; **Windows MPI is not planned**.

4.6.1 Build

MPI is not part of the default build:

```
make -C src/xstar_tools/xstar/cpp
```

Build it explicitly:

```
make -C src/xstar_tools/xstar/cpp mpi
```

or:

```
make -C src/xstar_tools/xstar/cpp xstar-xspec-mpi
```

Override the MPI compiler wrapper when needed:

```
make -C src/xstar_tools/xstar/cpp mpi MPICXX=/path/to/mpic++
```

4.6.2 Run

```
mpirun -np 4 xstar-xspec-mpi \  
  --input xstinitable.par \  
  --data-dir /shared/xstar/data \  
  --output-dir /shared/run_mpi
```

Direct `key=value` grid input is also accepted.

4.6.3 Options

`xstar-xspec-mpi` accepts `--input`, `--data-dir`, `--output-dir/--output`, `--save`, `--cleanup-work`, `--restart`, `--verbose`, `--initable-bin`, `--xstar-cpp`, `--table-bin`, `--version`, and `--help` with the same meanings as the corresponding local-process driver.

There is deliberately **no** `--processes`, `--workers`, or `-j` process-count option. Concurrency is selected by `mpirun/mpirun -np N`; attempts to use local process-count options are rejected.

4.6.4 Rank model

Every rank, including rank 0, participates in XSTAR job execution. Jobs are claimed dynamically through an MPI-3 RMA queue, with at most one `xstar-cpp` child per rank.

Rank 0 additionally owns planning, final success checking, canonical STEP concatenation, loopcontrol-ordered spectrum gathering, and the final `xstar-xspec-table` invocation. Rank identity/completion order never determines scientific placement; loopcontrol does.

4.6.5 Filesystem requirement

The current MPI implementation uses job directories as the inter-stage scientific boundary, so all ranks must see the same executables, output directory, atomic-data directory, and any referenced spectrum/density input files. A shared filesystem and an explicit shared `--data-dir` are recommended for multi-node use.

4.6.6 Failure and restart

A failing rank marks the distributed queue failed; other ranks stop claiming work and active children are terminated when peer failure is observed. Final XSPEC tables are not published for a failed MPI run, while partial products remain for inspection.

`--restart` reuses a job only when `xout_spect1.fits`, `xout_step.log`, and `xstar-cpp.success` all exist.

4.7 C ABI

The current public C API ABI is 60487; the production-zone ABI is 6048110. ABI versions are independent of both the Python distribution version and the frozen science revision.

`xstar-cpp` queries the linked ABI identities before scientific execution and fails clearly if the packaged frontend and linked production libraries do not agree.

Compatibility rules:

1. do not change ABI numbers for documentation/API-only productization;
2. change an ABI only when its exported binary contract changes;

3. keep the science revision independent unless scientific behavior is intentionally reopened;
4. fail before science when a required runtime ABI is incompatible.

The exported-function inventory and implementation details are documented in `docs/developer/c_abi.md`.

DEVELOPER GUIDE

The active developer documentation is organized by current architectural and release concerns rather than by every historical milestone. Closed milestone notes and one-off qualification narratives are preserved in version-control history and release tags.

5.1 XSTAR tools architecture at the parity-freeze boundary

Productization version: 0.6.56

Qualified science revision: 0.6.90.5.5

Canonical executable authority: XSTAR Fortran 2.59g

Frozen C++ production baseline: 0.6.48.12.3.44

Frozen production-zone ABI: 6048110

This document reconstructs the execution architecture from the canonical Fortran source and then maps that architecture onto the Python and C++ implementations. It is a productization document, not a new scientific specification.

5.1.1 Authority and change policy

When sources disagree, use this order:

1. accepted qualification evidence for behavior already closed by the parity campaign;
2. executable semantics of the canonical XSTAR 2.59g Fortran source;
3. XSTAR papers and manuals for scientific explanation and context.

The parity freeze remains controlling. Documentation, comments, APIs, packaging, and orchestration may evolve without changing qualified scientific output. Any refactor of a scientific module must identify a concordance entry from `python_cpp_fortran_concordance.md` and the characterization tests protecting that entry.

5.1.2 Canonical execution layers

XSTAR is easiest to reason about as six ownership layers rather than as a flat collection of rate routines.

1. Process initialization and parameter semantics

`xstar.f90` owns the top-level executable lifecycle. `rread1.f90` reads user parameters and performs source-defined conversions/default handling. `ener.f90`, `xstarsetup.f90`, and `init.f90` construct the energy grids, atomic-data pointers, and initial physical/radiation state.

A crucial rule is that **literal kind and parameter-reader kind are part of the executable semantics**. The accepted radius repair is the clearest example: `r = r19*(1.e+19)` uses a default-REAL literal before promotion into a REAL(8) expression. The source-faithful ports therefore do not replace all constants with idealized binary64 values.

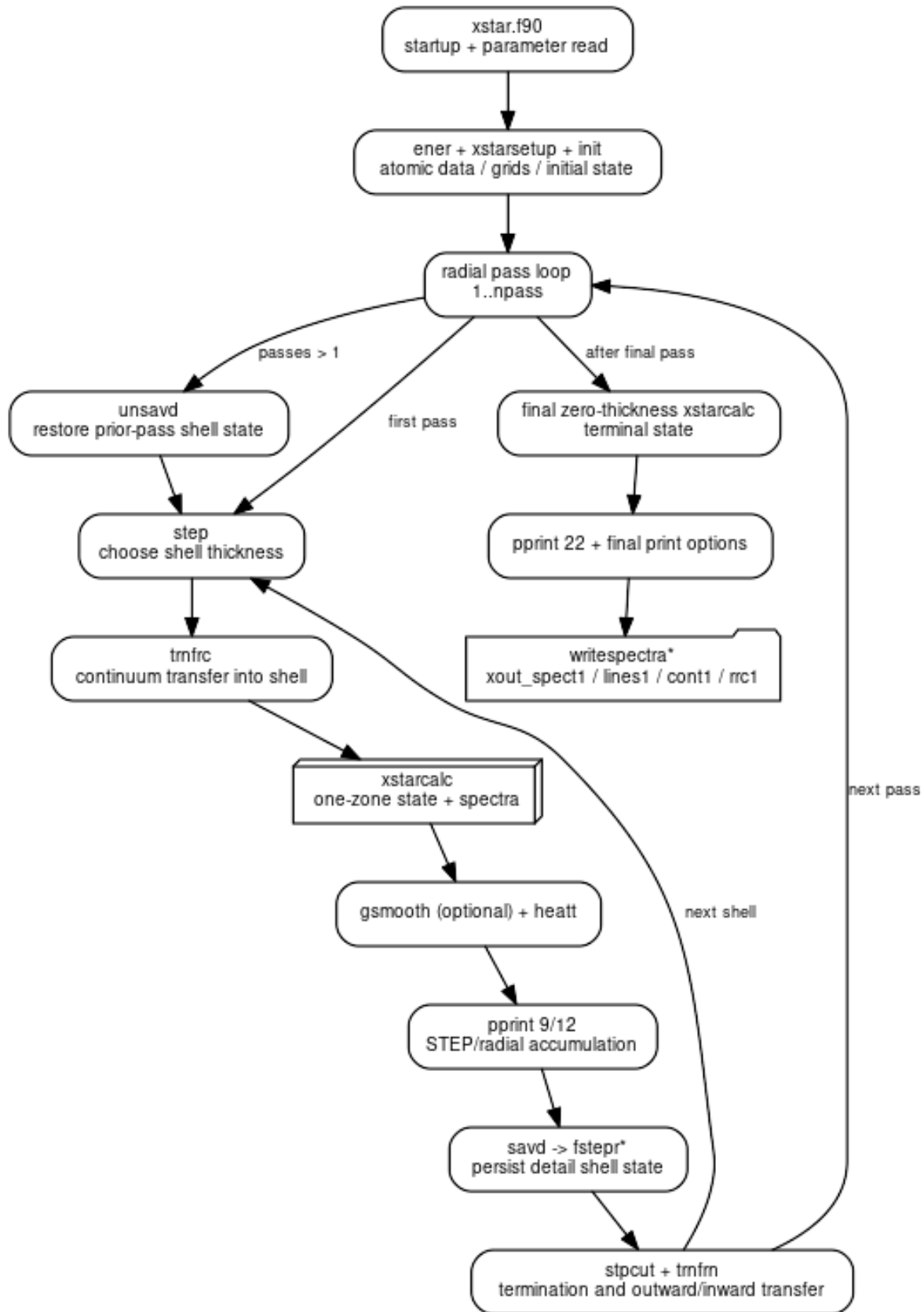
2. Radial controller and transfer ownership

The radial controller lives primarily in `xstar.f90`, with the shell primitives in `step.f90`, `trnfrc.f90`, `stpcut.f90`, and `trnfrn.f90`.

The source order is significant:

```
step -> trnfrc -> xstarcalc -> [gsmooth] -> heatt -> pprint -> savd -> stpcut -> trnfrn
```

On later passes, `unsavd` restores shell state before the shell is reevaluated. The pass count is source-controlled; it is not an invented convergence loop in the Python port.



3. One-zone scientific solve

`xstarcalc.f90` is the canonical one-zone orchestration boundary. Its scientific order is:

1. map the incident radiation to the working grid (`bremsmap`);
2. run `dsec` when thermal balance must be solved;
3. run a final `calc_hmc_all` at the committed temperature/electron state;
4. run `calc_emisab_all` to construct integrated line/RRC quantities;
5. run `calc_emis_all` to construct full-grid emission and opacity.

This ordering explains why a local workspace may be numerically correct yet a publication product can still be wrong: publication consumes the state *after* distinct reduced-grid and full-grid producers have run.

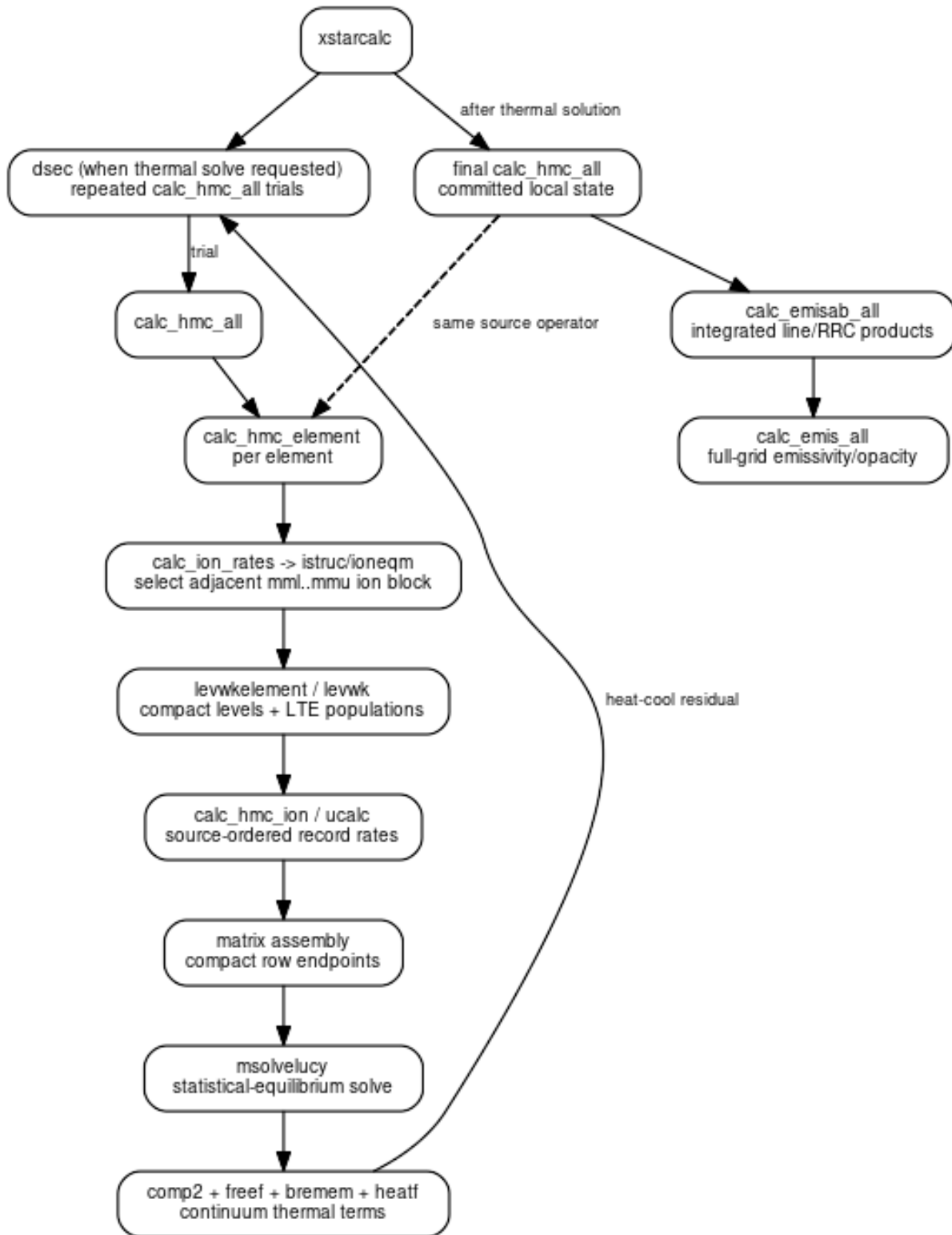
4. Fixed-state ion/level solve

`calc_hmc_all` loops over elements. For each element, `calc_hmc_element` first computes a cheap total-rate ion balance, selects an adjacent active ion block, constructs compact level/LTE state, assembles the multilevel statistical-equilibrium matrix from source-order atomic records, and solves it with `msolve Lucy`.

The two-step population strategy is explicitly described in the XSTAR manual:

- first solve ion fractions from total ionization/recombination rates and select materially populated adjacent stages;
- then solve the full level-population kinetic matrix for the selected stages.

The accepted C++ implementation preserves the source compact-row endpoint behavior, including the terminal clamp required by `msolve Lucy` semantics.



5. Spectral construction and Type50

The source deliberately separates integrated/reduced-grid products from full-grid products:

- `calc_emisab_all` constructs integrated line emissivity/opacity and RRC quantities used by detail/publication paths;
- `calc_emis_all` constructs the emergent full-grid continuum/line opacity/emissivity state;
- `linopac` is the line-profile accumulation primitive used by the Type50 path.

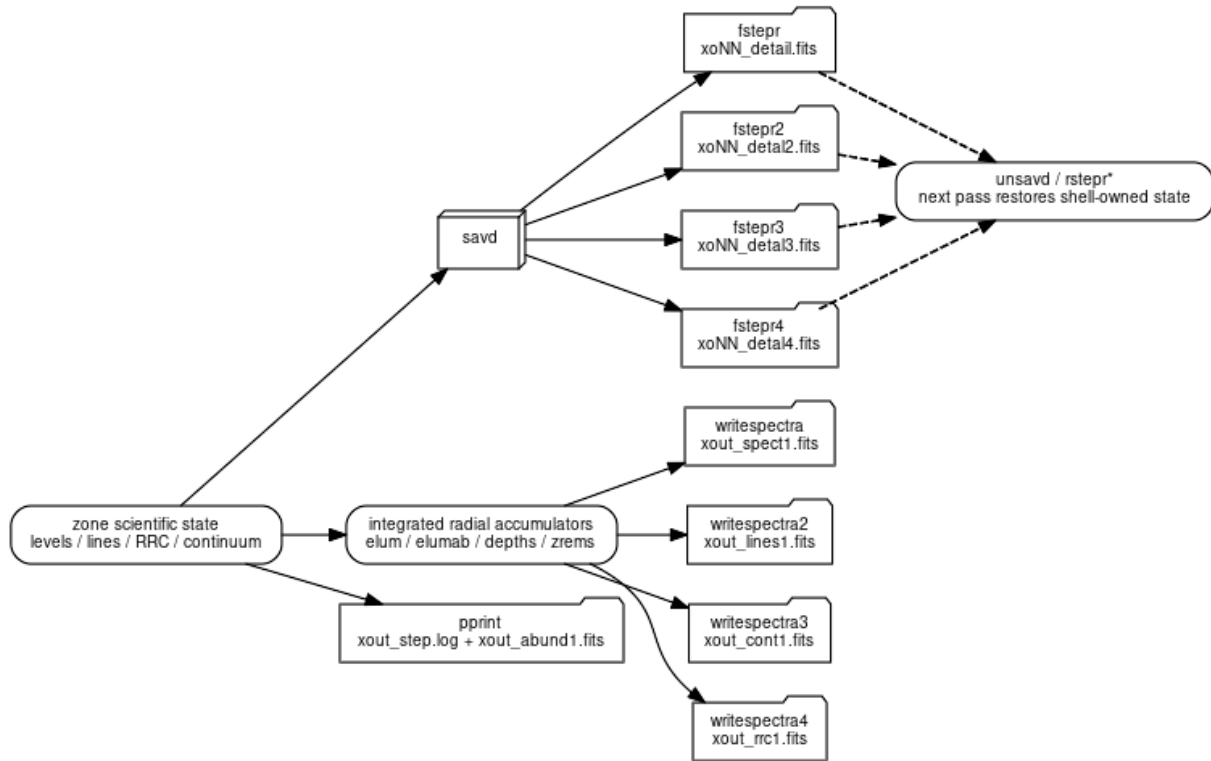
The frozen optimized C++ Type50 implementation changes traversal mechanics, not science: monotone cursors and source-compatible SIMD/scalar helpers reduce repeated range searches while preserving source ordering and the accepted numerical update semantics. In 0.6.53 these files received standardized source-correspondence blocks. In 0.6.54 the comments remain documentation-only while four oracle/coheat headers are given stable names and `constants.def` is colocated with the C++ sources. The non-science refactor gate canonicalizes comments/whitespace and reverses only those approved path/identifier relocations; canonical C++ content must remain identical to 0.6.53.

6. Publication and terminal-state ownership

Output is not a passive serialization of one universal state. The Fortran source has multiple publication owners with different lifetimes:

- `savd` and `fstepr*` publish per-shell detail state and also provide the persistence substrate used by `unsavd` on later passes;
- `pprint` owns STEP text surfaces and `xout_abund1.fits` accumulation/finalization;
- `writespectra`, `writespectra2`, `writespectra3`, and `writespectra4` own the final spectrum, line, continuum, and RRC products respectively.

The final zero-thickness evaluation and the lifetime of local producer workspaces are therefore scientifically observable at publication surfaces. The accepted C5 709/762 repair and the accepted Ca/O structural exceptions are consequences of this ownership model, not changes to matrix physics.



5.1.3 Python architecture

The Python implementation has two roles:

- a source-faithful executable reference/debug implementation;
- the controller/envelope for qualified C++ kernels and the shared production-zone path.

Major source-correspondence modules include:

Python module	Primary source role
atomic_database.py	readtbl/setptrs packed database and one-based pointers
ion_balance.py	pre-matrix calc_ion_rates -> istruc/ioneqm
element_equilibrium.py	levwk*, calc_hmc_ion, matrix assembly, msolverlucy
local_zone.py	calc_hmc_all fixed-state element/thermal aggregation
dsec.py	source-ordered thermal/charge nonlinear controller
emissivity.py	calc_emisab_* integrated emissivity/opacity path
emergent_emissivity.py	calc_emis_*, linopac, full-grid spectral path
radial_transfer.py	step, trnfrn, stpcut, trnfrn, savd/unsavd orchestration
radial_control.py	inline xstar.f90 pass/density control predicates
saved_radial_state.py	persistence semantics of savd/unsavd
pprint_legacy.py	STEP print-option semantics
output_writers.py	detail/final FITS publication ownership
physical_runner.py	executable-style parameter/state orchestration

Some of these are frozen source files. Milestone 2 does not alter their bytes; their source correspondence is recorded in documentation and the machine-readable concordance.

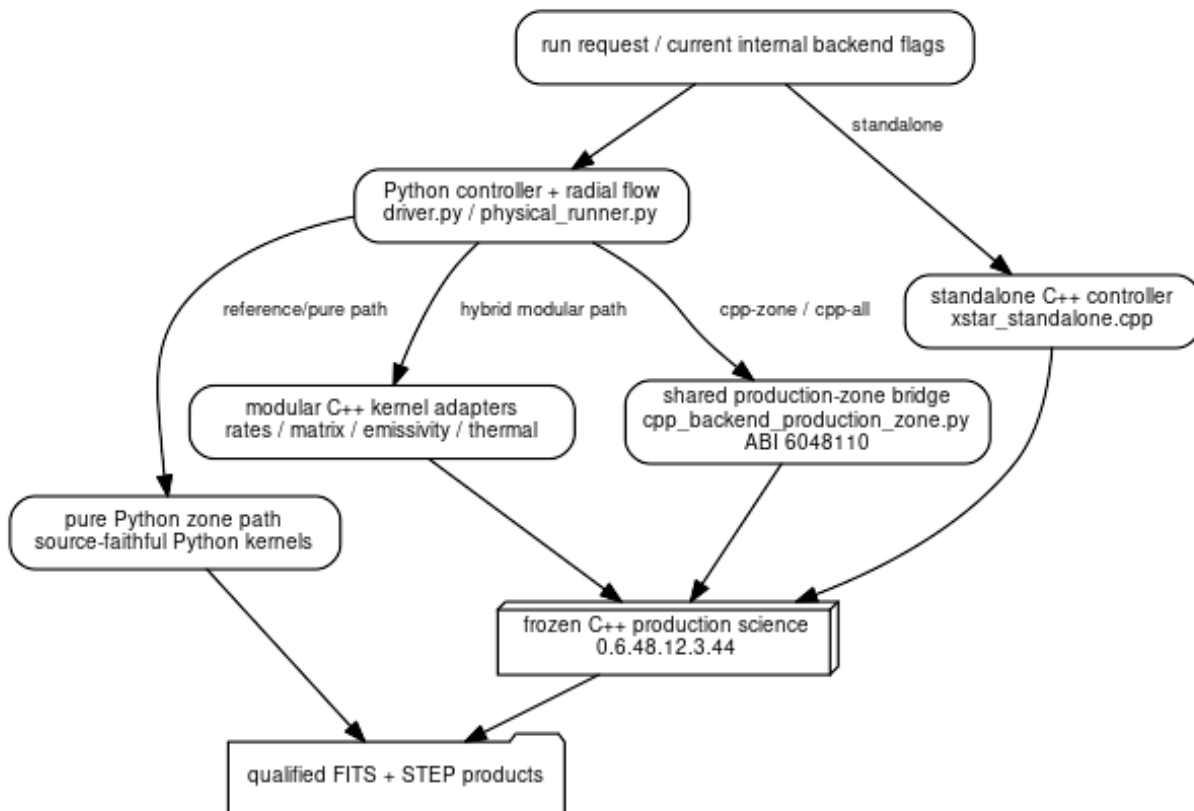
5.1.4 C++ architecture

The accepted C++ path is not a separate scientific model. It is a source-equivalent/optimized implementation of qualified operations plus standalone orchestration.

Important ownership areas are:

- `xstar_atdb_runtime.cpp`: native ATDB traversal, pointer/layout construction, parameter/default-real compatibility;
- `local_zone_engine.cpp`, `element_engine.cpp`, `matrix_kernels.cpp`, `level_population.cpp`, `rate_kernels.cpp`: local rate/matrix/population solve;
- `thermal_kernels.cpp`: source thermal leaves/reduction;
- `line_emissivity.cpp`, `opacity_kernels.cpp`: line/RRC/full-grid spectral kernels including Type50;
- `xstar_engine.cpp`, `xstar_run_state.*`: production state orchestration;
- `xstar_science_fits.cpp`: final/detail FITS publication;
- `xstar_step_log.cpp`: STEP/public text publication;
- `xstar_standalone.cpp`: standalone native controller;
- `cpp_backend_production_zone.py`: Python-to-native production-zone ABI (6048110).

All production C++ science remains pinned to the accepted C++44 baseline. Starting in 0.6.53, concise Fortran-source comments are carried in marked XSTAR-SOURCE-CORRESPONDENCE blocks. Version 0.6.54 additionally permits only the manifest-listed header/namespace/include relocation; a dedicated canonical non-comment gate must reduce every C++ file back to the 0.6.53 canonical content. Any other token-level change remains a freeze violation. Optimization correspondence and invariants are recorded in `python_cpp_fortran_concordance.md`.



5.1.5 Public execution envelopes

The stable public execution modes are `pure-python`, `zone-python`, `zone-cpp`, `zone-all`, and `xstar-cpp`. They expose progressively more native execution while preserving the accepted source-faithful science contract. See *Execution modes* for the current mapping.

5.1.6 Scientific invariants that architecture work must preserve

1. **Indexing:** canonical atomic-data and compact-level identity remains source/one-based where the source uses one-based indexing.
2. **Ordering:** source record and producer order is part of accepted behavior when floating-point accumulation or rank publication depends on it.
3. **Kinds:** default-REAL/default-INTEGER conversion semantics are preserved where they affect executable results.
4. **Workspace lifetime:** producer-local arrays are not assumed to survive unless the source explicitly persists or republishes them.
5. **Two-stage population solve:** preliminary ion selection and full statistical-equilibrium solve remain separate.
6. **Thermal iteration:** `dsec` remains the source nonlinear controller; it is not replaced by a generic numerical root solver.
7. **Publication separation:** numerical science, rank/order/identity diagnostics, and inventory diagnostics remain separate acceptance dimensions.
8. **Type50:** optimized traversal must retain source-equivalent profile arithmetic and update ordering under accepted qualification gates.
9. **ABI:** production-zone ABI 6048110 remains frozen until deliberately versioned.
10. **Accepted exceptions:** the frozen `Ca/O xo01_delta13` membership exceptions remain explicit metadata and must not be hidden by broader tolerances or model-specific physics.

5.1.7 Refactor gate

Before merging a refactor that touches scientific behavior or a module listed by `qualification/source_concordance.json`:

1. identify the concordance ID(s) in the change description;
2. establish and run at least one current runnable characterization test for each affected behavior; preserve the listed historical characterization/qualification lineage as evidence;
3. run `python tools/qualification/check_parity_freeze.py`;
4. run `python tools/qualification/check_source_concordance.py`;
5. if a frozen/pinned source must change, treat it as a dedicated science/refactor qualification rather than ordinary productization.

See `fortran_source_map.md` for the routine-level map and `python_cpp_fortran_concordance.md` for the implementation/qualification matrix.

5.1.8 Stable public execution modes

The productization-facing execution boundary is defined in *execution_modes.md*. Public mode names are stable; legacy backend flags remain advanced aliases. Any refactor that changes a mode-to-internal-path mapping must update the Milestone 3 characterization contract before implementation.

5.2 Execution modes

The stable single-model public modes are:

Mode	Controller	Scientific execution
pure-python	Python	source-faithful Python implementation
zone-python	Python	Python controller with qualified modular native kernels
zone-cpp	Python	shared production-zone native implementation
zone-all	Python	all-native zone path under the public Python controller
xstar-cpp	native C++	standalone native controller

Use the same names from Python:

```
config = XStarConfig.from_par_file(
    "xstar.par",
    data_dir="/path/to/xstar/data",
    output_dir="run1",
    mode="zone-cpp",
)
result = run_xstar(config)
```

or from the unified CLI:

```
xstar-tools run xstar.par \
  --mode zone-cpp \
  --data-dir /path/to/xstar/data \
  --output-dir run1
```

Advanced per-kernel backend overrides are limited to the Python-controller modes and are not accepted for the monolithic native modes.

5.2.1 XSTAR2XSPEC concurrency is separate

`xstar-xspec --processes N` runs a bounded pool of independent `xstar-cpp` OS processes; it does not create `N` threads or MPI ranks. `xstar-xspec-mpi` uses MPI ranks selected by `mpirun/mpiexec -np N` and deliberately does not accept local process-count options.

Any implementation refactor that changes a stable mode-to-runtime mapping must update characterization/regression coverage before changing behavior.

5.3 Cross-platform portability status

This page summarizes the **current accepted native portability contract**. Detailed milestone investigations and historical host rejections remain in the individual 0.6.88.* developer notes and in `CHANGELOG.md`.

5.3.1 Current accepted release

0.6.88.6.1.2.1 - `WINDOWS_GIT_PREFLIGHT_SHELL_CLOSURE` is formally accepted across the full four-host GitHub Actions matrix.

Host	Toolchain	Native non-MPI status
Linux x86_64	GCC / GNU Make	ACCEPT
macOS arm64	Apple Clang / GNU Make	ACCEPT
macOS x86_64	Apple Clang / GNU Make	ACCEPT
Windows x86_64	MSYS2 UCRT64 / MinGW-w64 GCC / GNU Make	ACCEPT

Windows MPI remains intentionally out of scope. The accepted Windows concurrency path is `xstar-xspec --processes N`.

5.3.2 Accepted fixed-state cross-platform contract

The portability qualification deliberately separates two properties:

1. **same-host determinism** — repeat the fixed-state calculation on the same host and require raw-byte equality;
2. **cross-platform scientific equivalence** — compare discrete state, FITS structure, and numerical payload against the canonical reference without requiring unlike architectures to emit identical floating-point bytes.

Cross-platform rules are:

- text line endings are normalized from CRLF/CR to LF for semantic comparison;
- visited-record identity/order is exact;
- non-floating step-log state is exact;
- FITS HDU/schema/row/column structure is exact;
- zero/nonzero, NaN, and infinity behavior is exact;
- every differing finite floating value must satisfy both relative difference $\leq 1e-13$ and ULP distance ≤ 64 .

Observed accepted results:

Host	Result
Linux x86_64	canonical fixed-state payloads byte-exact
macOS x86_64	canonical fixed-state payloads byte-exact
Windows UCRT64	FITS byte-exact; text differs only by CRLF serialization and normalizes exactly
macOS arm64	same-host deterministic; maximum observed relative difference $4.5635339780748665e-15$, maximum 39 ULP

5.3.3 Portability boundaries

Cross-platform behavior is intentionally concentrated in infrastructure layers:

- Makefile platform variables own library/executable extensions and linker policy;
- the dynamic-library wrapper owns `dlopen/dlsym` vs. Win32 library loading;
- `xstar_process.hpp` owns POSIX process/environment calls vs. `CreateProcessW` and Win32 wait/termination;
- `xstar_path_compat.hpp` adapts Windows `std::filesystem::path` objects at existing narrow CFIT-SIO/XSTAR C interfaces;
- Windows embedded Python registers the UCRT64 DLL directory through the qualified `sys.base_prefix` path.

The scientific kernels, controller/traversal order, publication semantics, output schema, and public ABI values remain outside these portability adaptations.

5.3.4 Historical portability sequence

The accepted path through the 0.6.88 series was:

- 0.6.88.1 — platform build abstraction;
- 0.6.88.2 / .2.1 / .2.2 — portable dynamic-library layer and fixed-state regression scaffold closure;
- 0.6.88.3 / .3.1 / .3.2 — macOS compile/link closure;
- 0.6.88.4 / .4.1 / .4.2 — portable process layer and Apple-Clang warning cleanup;
- 0.6.88.5 through .5.7 — Windows UCRT64 build, path, local-zone, XSTAR2XSPEC, embedded-Python, and warning-free closure;
- 0.6.88.6 — first four-host unified qualification; historically rejected by an over-strict cross-host raw-hash rule;
- 0.6.88.6.1 — fixed-state equivalence contract; historically rejected only by hard-coded workflow lookup;
- 0.6.88.6.1.1 — workflow discovery closure; Windows then exposed CRLF conversion of checked-in text fixtures;
- 0.6.88.6.1.2 — reference-payload line-ending closure; Windows workflow stopped before the host runner because git was not visible inside the MSYS2 preflight shell;
- 0.6.88.6.1.2.1 — native-PowerShell Git preflight closure; all four hosts accepted.

Historical REJECT labels are intentionally preserved. Later closures do not rewrite the result of an earlier real host run.

5.3.5 Build policy

The normal build is non-MPI:

```
make -C src/xstar_tools/xstar/cpp -j4
```

True MPI is explicit:

```
make -C src/xstar_tools/xstar/cpp mpi
```

The accepted MPI implementation is a Linux/macOS/HPC path. Windows MPI is not planned; Windows native support remains the standalone/local-process MinGW path, while production HPC deployment uses POSIX platforms.

5.4 Qualification policy

Qualification follows this authority order:

1. accepted qualification evidence for already-closed behavior;
2. canonical XSTAR Fortran 2.59g executable semantics;
3. papers/manuals for explanation and scientific context.

The parity boundary is machine-readable in `qualification/parity_freeze.json` and enforced by `tools/qualification/check_parity_freeze.py`.

Normal productization changes should run the relevant targeted characterization gates. Scientific/orchestration changes use representative fast fixtures first. All-62 qualification is reserved for deliberate science changes, release candidates when required, or explicit parity campaigns rather than every documentation/API change.

Material numerical science, inventory/membership, rank/order, and attachment diagnostics remain distinct surfaces. Known structural exceptions belong in qualification metadata rather than being hidden by broad tolerances.

5.4.1 Cross-platform portability qualification

The accepted 0.6.88.6.1.2.1 portability contract distinguishes same-host reproducibility from cross-platform numerical equivalence. A fixed-state run is repeated on each host and must be raw-byte deterministic against itself. Cross-host comparison then requires exact discrete state and FITS structure, normalizes text line endings, and applies the strict dual limit of relative difference $\leq 1e-13$ and ULP distance ≤ 64 only to differing finite floating values.

This avoids treating CRLF serialization or legitimate last-bit architecture differences as scientific regressions while retaining exact checks for traversal/inventory/order/schema and strong per-host determinism. See *Cross-platform portability status*.

5.5 Performance rules

Performance optimization must preserve the accepted scientific implementation.

Rules:

- measure before optimizing;
- keep science acceptance separate from runtime acceptance;
- use the same model/input/data/compiler when comparing timings;
- record CPU features and thread settings;
- prefer generic source-equivalent optimizations over element/model-specific shortcuts;
- do not reopen frozen Type50, trajectory, matrix, or transport semantics merely for speed;
- after a behavior-affecting optimization, qualify one fast model before broadening coverage.

Historical performance reports remain available through version-control history and release tags. They are useful attribution evidence, not current user-facing performance guarantees.

5.6 Versioning and science freeze

xstar-tools tracks several independent identities:

- **package version** — product/API/docs/packaging evolution (0.6.90.5.8 here);
- **science revision** — accepted Python scientific behavior (0.6.90.5.5);
- **frozen C++ science baseline** — accepted science-changing native milestone (0.6.90.5.5);
- **C API ABI** — 60487;
- **production-zone ABI** — 6048110.

A documentation, CLI, API, or packaging release must not change the science revision just because its distribution version changes.

PARITY_FREEZE.md and qualification/parity-freeze.json define the frozen scientific invariants, accepted comparator policy, and structural exceptions. Any intentional scientific reopening requires explicit qualification and a new science revision.

The 0.6.90.5.5 science revision was explicitly requalified and re-frozen in package 0.6.90.5.8 using an all-elements mnabund=1 FORTRAN-vs-C++ host comparison. Package 0.6.90.5.6 and 0.6.90.5.7 changed terminal diagnostics only and therefore do not advance the science revision.

5.7 Fortran source map

This map is based on the supplied XSTAR 2.59g source archive (xstar_source.tar.gz, SHA-256 f64c8c394e05f5a056206f1143259a8830f58ddf950c1efc15a03a00572db37a). It records the canonical executable ownership that productization must preserve.

The table intentionally distinguishes **scientific computation**, **caller-owned workspace**, and **publication lifetime**. Many parity defects arose when those were treated as interchangeable.

5.7.1 Routine-level concordance

Fortran routine/file	Scientific role	Important local/caller workspace	Python implementation	C++ implementation	Qualification evidence	Intentional deviations / notes
src/xstar/xstar.f90	Top-level executable, parameter/setup sequence, radial pass/shell controller, final-output sequence	radial counters kk/jk/jkp, direction ldir, radius/column, radiation/depth arrays, global population and product arrays	physical_rupy, driver.py, radial_transpy, radial_controlpy	xstar_stancpp, xstar_enginecpp, xstar_run_*	all-62 STEP 12.3.42; all-62 FITS 12.3.43.3; three-mode 12.3.44	Python decomposes inline controller blocks into testable functions but preserves source predicates/order.
rread1.f90	Read parameters; calculate initial pressure/density/ r_z and source-control values	r19, r, xpx, p, zeta, abundance vector, ncn2, npass, critf, hidden switches	physical_runormalize_xsource-real helpers in runner/radial path	xstar_atdbcpp contains source reader/default REAL compatibility for native production	12.3.36_FORTTRAN_F	uclgsr8 source values and default-REAL literals are reproduced deliberately; 1.e+19 is not replaced by an ideal binary64 literal.
ener.f90, bremsmap.f90, init.f90	Energy-grid initialization, radiation remapping, initial global state	epi, epim, bremsa, bremsam, bremsint, depth/emissivity arrays	physical_ruener_grid, radiation helpers, initial-state builders	source-realenergy-grid helpers and native run-state construction	source-port 0.4.60/0.4.63; C++ three-mode 12.3.44	Python/C++ use explicit arrays rather than COMMON/global storage, but retain source grid values/order.
xstarsetup.f90	Load ATDB, call setptrs, initialize atomic/LTE support data and abundance/data tables	packed record vectors, pointer table, level temperature data	atomic_data_readtbl, atomic_data_setptrs, populate_at	xstar_atdbcpp	atomic source-port tests; all-element fixed-state 12.3.x; C++ 12.3.44	Memory-mapped Python FITS storage avoids a second GiB-scale copy; logical record semantics remain source-one-based.
setptrs.f90	Construct derived database topology and publication attachments	nparr, npfirst, npfi/npfe, npnxt, npilev/npilevi, npcon/npconi/npconi2, npplin/npplini, nlevs	atomic_data_setptrs, XSTARDerive	native build_poincode in xstar_atdbcpp	generic npilev/public; repairs through 12.3.20-22 and 12.3.43; 45.x publication qualification	Cache files may persist derived pointers, but the pointer identities are checked against the same source construction.
calc_ion.f90, istruc.f90, ioneqm.f90	First-pass total-rate ionization balance and active adjacent ion-stage	total ionization/recombination arrays, preliminary ion fractions, mml/mmu	ion_balancecalc_ion_rationeqm, istruc, select_ion	rate/element construction in local_zonecpp / element_enccpp	source-port pre-matrix qualification; all-62 Chapter 5. Developer guide 12.3.42/43	This pass is intentionally separate from population-diagnostics.

5.7.2 Top-level startup and initialization

The canonical `xstar.f90` startup is approximately:

```
parameter/default setup
-> rread1
-> ener (primary and working energy grids)
-> xstarsetup
    -> readtbl
    -> setptrs
    -> LTE/atomic support initialization
-> pprint startup parameter/header surfaces
-> init
-> radial passes
```

The source itself prints `xstar version 2.59g`. The source map in this document is tied to that supplied tree rather than to an inferred generic XSTAR version.

5.7.3 Parameter and default-REAL semantics

`rread1` uses the XPI/UCL parameter interface (`uclgsr8`, `uclgsi`, `uclgst`) and then performs source calculations. Two kinds of semantics matter:

1. **reader conversion semantics** - the parameter system may store a lower-kind value which is promoted by a higher-kind reader;
2. **literal semantics** - a literal such as `1.e+19` is a default REAL literal even when used in a `REAL(8)` expression.

The accepted 12.3.36 repair reproduces both where material. Future refactors must not normalize these values merely because a mathematically equivalent decimal can be written as Python/C++ `binary64`.

5.7.4 Atomic-database pointer ownership

`setptrs` is not just an index accelerator. Its derived relationships participate in physics and publication identity. In particular:

- `npilev/npilevi` bind atomic records to level topology;
- `npcon/npconi/npconi2` bind continuum/RRC records and their source identities;
- `nplin/nplini` bind line records;
- `parent/next-record` chains establish traversal and attachment order.

Therefore pointer caches are acceptable only when they reproduce the canonical `setptrs` result for the same ATDB.

5.7.5 Two-stage ion and level population solve

The Fortran and the XSTAR manual agree on the conceptual split:

1. use total ionization/recombination rates to estimate ion fractions and choose an adjacent active stage block;
2. assemble and solve the full multilevel kinetic operator for the selected ions.

This distinction is preserved in Python by `ion_balance.py` versus `element_equilibrium.py` and in the qualified native local-zone engine.

5.7.6 Thermal construction and dsec

`calc_hmc_all` is a fixed-state scientific operator: given a trial temperature/electron state and current radiation/escape state, it computes level populations, rates, heating, cooling, and charge-balance information. `dsec` is the nonlinear controller that repeatedly calls that operator.

Do not merge those responsibilities in a way that resets source-owned state between trials or changes the source branch order without dedicated characterization.

5.7.7 Reduced-grid versus full-grid spectral paths

The current source explicitly places:

```
calc_hmc_all
calc_emisab_all
calc_emis_all
```

at the end of `xstarcalc`. `calc_emisab_all` owns integrated line/RRC quantities; `calc_emis_all` owns the subsequent full-grid spectral construction. This split is required to understand detail-publication lifetime and Type50 ownership.

5.7.8 STEP print-option map

`pprint.f90` documents and implements a broad set of print options. The qualification campaign most heavily exercises these:

Op- tion	Source role	Qualification interpretation
1	strongest emission lines sorted by strength	numerical ranked-array comparison separated from identity/order diagnostics
5	energy sums / error	<code>err</code> is compared relatively after 12.3.34 comparator repair
9	short radial-zone summary	trajectory/zone characterization
12	append radial abundance data to <code>xout_abund1</code>	final abundance publication
15	line luminosities/depth material surface	normalized-L1 material science is primary; inventory is separate
17	column headings	exact source-style header semantics used by STEP qualification
19	recombination-continuum luminosities	RRC numerical/product publication
22	final ionization/summary quantities	terminal source state
23	strongest absorption lines sorted by strength	numerical rank science separated from identity/order/membership
24	absorption edges	inventory and numerical common-row science separated
27	ion column densities in the current port	format/default-REAL threshold fixes accepted in 45.1

The exact full option list remains defined by `pprint.f90`; the table above records the options with explicit frozen comparator policy.

5.7.9 Terminal/final-output lifetime

After the radial passes, `xstar.f90` performs another local calculation at the final boundary, calls final STEP options, and then writes final FITS products. Per-shell detail products have already been written by `savd -> fstrepr*` during the radial loop.

This means three states must not be conflated:

1. the physical local state used during a shell solve;
2. the state retained/persisted for a detail shell or later pass;
3. the terminal state consumed by final public products.

The accepted Python publication bridge at science revision 45.3.3.8 is deliberately **output-only**: it reconstructs source publication lifetime where necessary without modifying rates, matrices, populations, thermal balance, transport, or trajectory.

5.7.10 Related documents

- `architecture.md` - execution and ownership overview.
- `python_cpp_fortran_concordance.md` - implementation and qualification map with stable concordance IDs.
- `../science/xstar_references.md` - literature/manual correspondence.

5.8 Python / C++ / Fortran concordance

This document assigns stable **concordance IDs** to scientific ownership boundaries. Refactors should cite these IDs rather than relying on historical version names alone.

Science revision: 0.6.90.5.5

C++ baseline: 0.6.48.12.3.44

Production-zone ABI: 6048110

5.8.1 How to read the map

Implementation status uses three terms:

- **source-exact** - control flow, ordering, data-kind behavior, and observable source semantics are deliberately re-produced;
- **mathematically equivalent** - representation differs, but the same source equations/operator are evaluated and qualified;
- **optimized-equivalent** - the source operation is transformed for performance, with explicit invariants and qualification showing accepted products are preserved.

The terms do not supersede qualification evidence. The parity freeze remains the release gate.

5.8.2 Concordance matrix

ID	Fortran author- ity	Python imple- mentation	C++ imple- mentation	Status	Characterization / qualifica- tion evidence
ARCH	xstar.f90, xstarcalc. f90	driver.py, physical_runne py, radial_transfe py	xstar_engine. cpp, xstar_standalo cpp, xstar_run_sta *	source-exact orchestration contract / equivalent storage	tests/ test_source_characterization_current. py::test_arch_001_controller_one_zone_bo all-62 12.3.42-44
INPU	rread1.f90	parameter nor- malization and source- real helpers in physical_runne py / radial setup	xstar_atdb_run cpp reader/default- real helpers	source-exact where exe- cutable kinds matter	tests/ test_source_characterization_current. py::test_input_001_fortran_input_semanti 12.3.36 rread1 radius repair; all-62 STEP
DB-0	readtbl.f90, setptrs.f90, xstarsetup. f90	atomic_databas py (readtbl, setptrs, pointer/cache validation)	xstar_atdb_run cpp native reader/pointer layout	source-exact identities with equiva- lent storage	tests/ test_source_characterization_current. py::test_db_001_atomic_database_pointer_ all-element qualification; 12.3.44
ION-	calc_ion_rates f90, istruc. f90, ioneqm. f90, first pass of calc_hmc_eleme f90	ion_balance. py	fixed- state/element engine total- rate pass	source-exact stage- selection logic	tests/ test_source_characterization_current. py::test_ion_001_pre_matrix_ion_balance_ pre-matrix ion-balance qualifi- cation; all-62
LEVE	levwkelement. f90, levwk. f90	element_equilib py compact basis and LTE tables	level_populat cpp, fixed-state layout	mathe- matically equivalent representa- tion with source- one-based topology	tests/ test_source_characterization_current. py::test_level_001_compact_level_topolog 12.3.25 level/matrix closure
MATR	calc_hmc_ion. f90, calc_hmc_eleme f90, msolverlucy. f90	element_equilib py, linear_algebra py	local_zone_eng cpp, element_engine cpp, matrix_kernel cpp	source- equivalent opera- tor/solve	tests/ test_source_characterization_current. py::test_matrix_001_matrix_assembly_and_ 12.3.25 terminal clamp; 12.3.44
THER	calc_hmc_all. f90, comp2. f90, freef. f90, bremem. f90, heatf. f90	local_zone. py, compton.py, free_free.py, bremsstrahlung py, thermal_balance py	thermal_kerne cpp, fixed-state thermal reduc- tion	source- equivalent, selected native re- ductions optimized- equivalent	tests/ test_source_characterization_current. py::test_therm_001_thermal_construction_ thermal parity; 12.3.44
DSEC	dsec.f90	dsec.py	fixed- state/standalone DSEC contr- oller support	source-exact nonlinear branch/order contract	tests/ test_source_characterization_current. py::test_dsec_001_nonlinear_controller_a DSEC ownership/trajectory clo- sure; 12.3.42
EMIS	calc_emisab_al f90	emissivity.py	line_emissivi cpp, fixed-state product state, xstar_science. cpp	source-exact producer order; output lifetime bridge	tests/ test_source_characterization_77 py::test_emisab_001_reduced_grid_publica 12.3.43.x; 45.3.3.8

5.8. Python/C++/Fortran concordance

5.8.3 Detailed equivalence notes

INPUT-001 - default-REAL and parameter-reader semantics

The accepted port treats Fortran kinds as observable when the source does. The radius calculation is the canonical example:

```
r = r19*(1.e+19)
```

The decimal expression is not interpreted as an abstract real number; the default-REAL literal is rounded at the source kind and then promoted. The native implementation similarly models the relevant `uclgsr8` conversion path. This is **source-exact numeric semantics**, not a mathematical simplification.

DB-001 - database pointer construction

Python uses memory-mapped FITS packed vectors and explicit dataclasses rather than COMMON blocks. C++ uses native reader/layout objects. Those are storage transformations only. `npar`, `level`, `continuum`, and line attachment identities remain governed by `setptrs` traversal and are validated as scientific/publication state.

MATRIX-001 - compact matrix endpoints and Lucy solve

The accepted matrix repair established that the source solver consumes compact endpoint ranges using clamped endpoints. A translated record whose local endpoint is at/after the terminal compact row must therefore be clamped to the source solver's `ipmat` domain rather than dropped as invalid. This is frozen by 12.3.25 and is part of the source operator definition.

The C++ solver may store dense/compact intermediates differently from Fortran. Qualification is based on the resulting source operator, populations, rates, thermal totals, and downstream products.

THERM-001 - source-order thermal reductions

The C++ thermal path groups and vectorizes selected computations, but the production reduction maintains the accepted source ownership and ordered accumulation semantics where roundoff affects qualified products. Performance transformations must not silently reorder a surface already shown to be order-sensitive.

DSEC-001 - nonlinear controller

`dsec` is not simply “find the root of heating minus cooling.” It is a stateful Fortran controller whose trial sequence, bracketing/secant logic, branch order, and population ownership can alter the next evaluation. Python therefore follows the source labels/branches rather than substituting `scipy.optimize` or another generic solver.

EMISAB-001 and DETAIL-001 - producer versus publication lifetime

`calc_emisab_all` produces physical integrated line/RRC values. `fstepr3` publishes a particular source-visible continuum/RRC inventory at shell-save time. The accepted 45.3.3.8 Python bridge retains publication values without writing them back into physical `cemab`, `cabab`, or `opakab`. This is a deliberate boundary:

Publication repair may restore source lifetime/ownership; it must not change the physical state merely to force row inventory.

The accepted Ca and O7 `xo01_detail3` exceptions remain structural membership exceptions under the frozen material criterion.

TYPE50-001 - optimized C++ line-profile path

Canonical operation: `linopac` walks the relevant energy range for each source line and accumulates the line-profile opacity into the continuum opacity array.

Accepted production transformations include monotone cursor advancement and specialized vector/scalar profile generation. The frozen invariants are:

1. line/record production order remains source-compatible;
2. active range and boundary decisions are source-equivalent;
3. per-bin update uses the accepted source arithmetic/update semantics;
4. optimization fallbacks do not change qualified numerical products;
5. production and reference paths passed the accepted Type50 decomposition and downstream FITS/STEP parity gates.

The production science remains frozen. Version 0.6.53 introduced reversible top-of-file C++ correspondence comments. Version 0.6.54 adds only an approved native-layout/name relocation plus source-derived atomic-data comments: four version-labeled runtime headers are renamed, `constants.def` is colocated with the C++ sources, and no C++ file may include a parent-directory header. The 0.6.54 refactor gate removes comments/formatting and reverses only the explicitly listed path/identifier aliases; the resulting canonical C++ content must match 0.6.53.

RADIAL-001 - radial controller

The Python port factors inline blocks from `xstar.f90` into functions for testing, but source order remains part of the contract. In particular, `stpcut` termination and the exact initial radius feed the shell trajectory; changing either can shift all downstream zone states even if the local-zone solver is unchanged.

STATE-001 and TERMINAL-001 - state lifetime

The source has multiple lifetimes:

- local workspaces inside a zone evaluation;
- shell values persisted by `savd/fstepr*` and restored by `unsavd`;
- final/terminal state consumed by `final pprint` and `writespectra*`.

A refactor must state which lifetime it owns. “Last value in memory” is not a sufficient publication contract.

STEP-001 - comparator policy is part of qualification, not the physics

The current comparator deliberately separates:

- ranked numerical science by rank position;
- line identity/order/membership diagnostics;
- common-row numerical science;
- inventory/membership differences;
- normalized-L1 material surfaces.

This prevents sort artifacts or tiny tails from being misclassified as numerical physics failures while still retaining structural diagnostics.

5.8.4 Pinned-source comment policy

Milestone 2 adds concise Source correspondence comments to important **unpinned** Python/C ABI files. It does **not** edit:

- any C++ production source pinned by `qualification/parity_freeze_science_hashes.json`;
- frozen Python production files such as `emissivity.py`, `element_equilibrium.py`, `output_writers.py`, `pprint_legacy.py`, `driver.py`, or `physical_runner.py`.

For pinned sources, this document is the source-origin comment layer until a deliberate refactor qualification permits changing their frozen hashes.

5.8.5 Refactor checklist

A scientific refactor PR should include a statement like:

```
Concordance: MATRIX-001, THERM-001
Characterization:
  pytest -q tests/test_source_characterization_current.py::test_matrix_001_matrix_
↪assembly_and_solve_anchor
  pytest -q tests/test_source_characterization_current.py::test_therm_001_thermal_
↪construction_anchor
Freeze:
  python tools/qualification/check_parity_freeze.py
  python tools/qualification/check_source_concordance.py
```

The current characterization anchor is `tests/test_source_characterization_current.py`; the `historical_characterization_tests` entries in `qualification/source_concordance.json` preserve earlier qualification/characterization lineage. Some old tests still depend on legacy import names or optional scientific dependencies and are therefore not a claim that every listed test runs in every productization environment. Before changing a scientific implementation, the affected concordance ID must have at least one **current runnable characterization test** in the target development environment. If the listed lineage is stale, unavailable, or too expensive, add or modernize a small characterization fixture **before** changing the implementation; do not simply drop the concordance requirement.

5.8.6 C++ source correspondence comments and 0.6.54 native-layout refactor

All 47 `.cpp`, `.h`, and `.hpp` files under `src/xstar_tools/xstar/cpp/` now begin with a concise marked block containing: Fortran authority, C++ role, implementation relation, concordance IDs, and qualification boundary. Files that are C++ infrastructure or historical oracle data explicitly say that they have **no direct Fortran routine**, while identifying the scientific boundary or source behavior they support.

The comments do not redefine the scientific baseline. Current exact science-critical source bytes are protected by `tools/qualification/check_parity_freeze.py`, while routine/ownership relationships are checked by `tools/qualification/check_source_concordance.py`. Closed overlay/refactor manifests remain available through version-control history and release tags.

5.8.7 0.6.55 Python source-comment overlay

The production Python source carries Fortran/source-correspondence comments parallel to the C++ comments. The comments also document the atomic-database distinction between **data type** (record formula/interpretation in `ucalc`) and **rate type** (downstream use of returned rates). Current freeze enforcement uses the compact parity/source-concordance gates; see `python_fortran_source_comments.md`.

5.8.8 0.6.56 active-namespace note

The source concordance maps production behavior, not the historical parity-campaign tooling surface. 0.6.56 archives one-off Python audit/attribution/closure modules that have no live runtime caller. The Type53 diagnostic formerly named `type53_semantics.py` remains provenance only; active Type53 science remains mapped through the `UCalc/ionization/matrix` implementation entries. Historical characterization lineage does not replace the current runnable characterization-test requirement for future scientific refactors.

5.9 Python / FORTRAN source correspondence comments

Production Python modules in the source-faithful XSTAR layer carry concise comments that identify the corresponding FORTRAN XSTAR routines and important representation constraints. These comments are explanatory; they do not establish an independent scientific authority.

The canonical scientific oracle remains FORTRAN XSTAR 2.59g. The active scientific freeze is enforced by `tools/qualification/check_parity_freeze.py`, which verifies the accepted science revision and exact hashes for the current science-critical source set.

5.9.1 Atomic-data terminology

Keep the distinction between **data type** and **rate type** explicit. Data type identifies the packed record formula/interpretation; rate type describes how the resulting data/rate is used downstream. They must not be treated as interchangeable identifiers.

5.9.2 Maintenance rule

When editing correspondence comments:

- do not change source arithmetic or operation ordering under a documentation-only change;
- keep routine/file names factual and tied to the actual implementation;
- run the parity-freeze and source-concordance gates after any change near science-critical modules;
- if executable science must change, create a dedicated science qualification milestone rather than rebasing the documentation closure.

Closed comment-overlay manifests/checkers from earlier milestones are retained by version-control history/tags rather than shipped in active releases.

5.10 C++ / FORTRAN source correspondence comments

The C++ scientific implementation carries marked source-correspondence comments describing the FORTRAN XSTAR routine(s) that own the corresponding behavior. These comments help maintainers navigate a source-faithful port; they do not authorize changing accepted formulas, indexing, accumulation order, or publication semantics.

The canonical scientific oracle remains FORTRAN XSTAR 2.59g. Current exact science-critical source bytes are protected by `tools/qualification/check_parity_freeze.py`; structural/routine ownership is described by `qualification/source_concordance.json` and checked by `tools/qualification/check_source_concordance.py`.

5.10.1 Important interpretation rules

- preserve one-based/source identities where the source uses them;
- preserve default-REAL/default-INTEGER conversion semantics when they affect results;
- keep numerical science distinct from output inventory/order diagnostics;
- do not treat an ABI wrapper or portability layer as an independent source of physics;
- keep optimization comments tied to qualified source-equivalent behavior rather than performance claims.

5.10.2 Maintenance rule

Comment-only maintenance should not alter science-critical code bytes. If a required documentation edit touches a frozen source file, it must be handled under an explicit qualified source/documentation exception or a new science/refactor milestone rather than silently rebasing the freeze.

Earlier version-specific comment-overlay evidence is retained in repository history/tags rather than copied into every release archive.

5.11 Function-level source comments

`xstar_tools 0.6.73` adds a reversible documentation overlay to the active Python and C++ XSTAR implementation. The goal is that a maintainer can open a function and understand its local responsibility, physical role, inputs/outputs in context, and source basis without first reconstructing the parity campaign or opening the literature.

5.11.1 Source hierarchy

Comments follow the same authority hierarchy as the parity freeze:

1. accepted qualification evidence for behavior already closed;
2. canonical XSTAR Fortran executable semantics;
3. the XSTAR manual and papers for physical explanation and context.

The literature therefore explains *why* an operation exists and what physical process it represents; it does not override a qualified Fortran behavior.

5.11.2 Reference set reviewed for this pass

Every page of the supplied reference bundle was rendered/reviewed and its text indexed before the source pass. The bundle contains:

- *XSTAR Manual* (`xstarmanual.pdf`), especially sections 11.4-11.7, chapter 12, and chapter 14;
- Kallman et al. (1996), *Photoionization Equilibrium Modeling of Iron L Line Emission*;
- Kallman & Bautista (2001), *Photoionization and High-Density Gas*;
- Bautista & Kallman (2001), *The XSTAR Atomic Database*;
- Kallman et al. (2004), *Photoionization Modeling and the K Lines of Iron*;
- Mendoza et al. (2021), *The XSTAR Atomic Database*.

The reviewed archive SHA-256 is `a66718cf3b3d831906ed640c3bb9a2d8e81ac3bcdbfd4878d09fc680e6616e84`.

5.11.3 What the comments explain

Scientific functions identify the relevant process and where it sits in the XSTAR algorithm. In particular, comments distinguish:

- preliminary ionization/recombination balance from the detailed multilevel statistical-equilibrium solve;
- level/superlevel/LTE and detailed-balance handling;
- heating/cooling terms and the thermal-equilibrium update;
- radiative/collisional rate evaluation from matrix consumption;
- bound-bound line emission/opacity and Type-50 profile/rebin work;
- bound-free photoionization/recombination-continuum work and Milne-related inverse rates;

- radial transfer, DSEC, state retention, and terminal publication;
- ATDB **data type** (how a record is interpreted/evaluated) versus **rate type** (how the evaluated quantity is consumed by XSTAR);
- K-vacancy/Auger/fluorescence paths where the 2004 Fe-K paper is applicable;
- final FITS/STEP publication versus the science that produced the values.

Implementation-only helpers are labeled as such. ABI, parsing, serialization, statistics, and diagnostic functions do not receive invented physical citations.

5.11.4 Reversible overlay

Function comments are bracketed by:

```
XSTAR-FUNCTION-COMMENT-BEGIN
...
XSTAR-FUNCTION-COMMENT-END
```

The qualification checker removes only these blocks and requires exact 0.6.72 source bytes underneath. Python additionally requires AST equality; C++ translation units must remain warning-clean under the project compile settings. This makes the comments useful documentation without silently changing the accepted scientific implementation.

5.11.5 Maintenance rule

New scientific functions should include a concise explanation of:

- the physical or algorithmic quantity being computed;
- how the function fits into the XSTAR workflow;
- the units/normalization or ordering invariant when it is not obvious;
- the relevant Fortran routine/data type/rate type when known;
- the manual/paper context when it genuinely applies.

Do not paste long derivations or debugging history into production code. Link such material from the developer/science guides instead.

5.12 Stable public Python API

The supported application-facing execution surface is:

```
from xstar_tools import (
    BackendMode,
    ExecutionMode,
    XStarConfig,
    XStarData,
    XStarProducts,
    XStarResult,
    run_xstar,
)
```

The task-oriented user documentation is in *Running XSTAR from Python*; generated signatures/docstrings are in *Stable execution API*.

`xstar_tools.xstar.*` remains the source-faithful scientific/runtime implementation layer. Normal applications should not depend on internal qualification modules.

5.12.1 Configuration contract

Exactly one input source is required: `input_file`, `parameters`, or `fortran_run_directory`. The named constructors are preferred:

```
XStarConfig.from_par_file(...)
XStarConfig.from_mapping(...)
XStarConfig.from_fortran_run_directory(...)
```

The stable API rejects non-empty output directories unless `overwrite=True`, validates local data explicitly, and keeps package version and scientific revision distinct.

5.12.2 Result contract

`run_xstar(XStarConfig(...))` returns `XStarResult`; `XStarProducts` provides deterministic product paths and `produced_fits()` reports files actually present. Provenance records requested/actual mode, package/science identities, ABI values, native/runtime identity, atomic-data identity, thread settings, and fallback events where applicable.

5.12.3 Compatibility boundary

The earlier keyword-style `run_xstar(mode=..., run_script=..., ...)` call is retained for compatibility with benchmark/legacy integrations. New applications should use the typed configuration/result API.

Changes to this public layer must keep the current behavioral regression tests green and must pass `tools/qualification/check_parity_freeze.py` plus `tools/qualification/check_source_concordance.py` when the change could touch source-faithful behavior.

5.13 Unified CLI contract

The primary application CLI is:

```
xstar-tools <command>
```

It is an orchestration/productization surface; it does not redefine XSTAR science.

5.13.1 User commands

```
xstar-tools run
xstar-tools inspect
xstar-tools data
xstar-tools backends
xstar-tools compare
xstar-tools doctor
xstar-tools version
```

`xstar-tools run INPUT --mode MODE --data-dir DATA --output-dir OUT` constructs `XStarConfig` and calls the same `run_xstar(config)` public API used by Python applications. CLI and Python therefore share input validation, backend mapping, output policy, environment isolation, result paths, and provenance.

The stable mode names are `pure-python`, `zone-python`, `zone-cpp`, `zone-all`, and `xstar-cpp`.

5.13.2 Development and qualification namespaces

```
xstar-tools dev legacy-run ...
xstar-tools dev benchmark ...

xstar-tools qualify benchmark ...
xstar-tools qualify original-parity ...
xstar-tools qualify cpp-parity ...
xstar-tools qualify reference ...
```

The older top-level `xstar-tools benchmark ...` alias is retained as a deprecated compatibility route.

5.13.3 Logging

A normal run can expose human-readable progress, Python logging, JSONL events via `--json-log`, and a final result document via `--summary-json`. Qualification marker strings are not required for normal application use.

The complete user-facing command reference, including all installed compatibility/specialist console scripts, is *Command-line interfaces*.

5.14 xstar-cpp native executable contract — 0.6.69

`xstar-cpp` is the first-class, Python-free command-line frontend for the frozen native XSTAR production implementation. It is a productization layer only; it does not implement scientific algorithms.

5.14.1 Normal use

```
xstar-cpp --input xstar.par --data-dir /path/to/xstar/data --output run1
```

A HEASoft/IRAF-style parameter file is read as `name,type,mode,value,min,max,prompt`. The value field is preserved as text and written to the same JSON parameter envelope consumed by the qualified `xstar_cpp run-production` path. XSTAR-style `name=value` tokens can be used instead of a `.par` file and command-line tokens override values read from the file.

`--data-dir DIR` resolves `DIR/atdb.fits` and `DIR/coheat.dat`; explicit `--atomic-db` and `--coheat` paths override those defaults. The frontend does not download scientific data.

5.14.2 xstar-cpp extensions

The following are orchestration extensions, not canonical XSTAR parameters:

- `--json-summary FILE` — final machine-readable status/products summary;
- `--provenance FILE` — additional provenance JSON path;
- `--progress {none,text,json}` — frontend orchestration events;
- `--threads N` — run-scoped `OMP_NUM_THREADS`;
- `--profile FILE` — frontend wall-time/profile JSON;
- `--deterministic` — record reproducible-run intent in provenance;
- `--print-option N` — read-only extraction from the completed `xout_step.log`;
- `--parameters-out FILE` — retain the generated JSON parameter envelope. Without this option, the envelope is an ephemeral system-temporary orchestration file and is removed after native execution; it is never placed in the science output directory;

- `--abi` — print expected and runtime ABI identities.

`--print-option` never changes STEP generation. It only reads a completed log.

5.14.3 Scientific-control architecture

The frontend delegates to the sibling compatibility executable:

```
xstar-cpp
-> xstar_cpp run-production
    -> command_run_standalone_production()
```

The two Python-facing shared production modes use the same frozen operator:

```
zone-all
-> xstar_production_zone_run_all_v0648110()
    -> command_run_standalone_production()

zone-cpp
-> xstar_production_zone_context_create_v0648110()
    -> worker -> command_run_standalone_production()
```

Thus the frontend, `zone-all`, and `zone-cpp` do not carry independent scientific control implementations.

5.14.4 ABI behavior

`xstar-cpp` links to `libxstar_api.so` and `libxstar_production_zone.so` for runtime identity queries. Before a scientific run it requires:

C API ABI	60487
production-zone ABI	6048110
science revision	0.6.90.5.5

A mismatch fails before science with exit status 70 and a diagnostic naming the expected and observed ABI.

Package version, science revision, C API ABI, and production-zone ABI are separate identifiers. A productization release does not change science or ABI.

5.14.5 Qualification

The first-class native frontend was established during the earlier productization campaign. Current releases no longer ship its closed milestone checker; the active release boundary uses the compact parity-freeze/source-concordance gates plus the retained behavioral regression tests.

```
python tools/qualification/check_parity_freeze.py
python tools/qualification/check_source_concordance.py
```

The scientific cross-mode authority remains the accepted frozen qualification campaign; productization-only changes must not silently alter the standalone/production-zone scientific implementation.

5.15 Native C ABI contract

The Python bindings and native productization layers use two separately versioned C interfaces. Their versions are intentionally independent of the Python distribution version and the frozen science revision.

5.15.1 Public XSTAR API

Header: `src/xstar_tools/xstar/cpp/xstar_api.h`

Current ABI:

```
XSTAR_API_ABI_VERSION = 60487
XSTAR_API_VERSION_STRING = 0.6.90.5.5
```

Runtime identity functions:

```
uint32_t xstar_api_abi_version(void);
const char* xstar_api_version_string(void);
```

The header also exports the context/configuration, backend-discovery, zone-evaluation, component-information, and statistics interfaces. Structure sizes and `abi_version` fields are part of the compatibility contract.

5.15.2 Production-zone ABI

Header: `src/xstar_tools/xstar/cpp/xstar_production_zone_bridge.h`

Current ABI:

```
XSTAR_PRODUCTION_ZONE_ABI_V0648110 = 6048110
```

Runtime identity functions:

```
int32_t xstar_production_zone_abi_version_v0648110(void);
const char* xstar_production_zone_backend_name_v0648110(void);
```

The production-zone surface exports:

- one-shot `xstar_production_zone_run_all_v0648110()` (zone-all);
- persistent context `create/run-next/done/finalize/destroy` functions (zone-cpp).

Both call the same `command_run_standalone_production()` scientific implementation used by standalone `xstar_cpp run-production`.

5.15.3 Compatibility rules

1. Package/API productization releases may advance without changing either ABI.
2. A C ABI change requires an intentional ABI revision, documentation, tests, and qualification; do not reuse 60487 or 6048110 for incompatible layouts or semantics.
3. The runtime ABI query must agree with the frontend's compiled expectation before `xstar-cpp` starts science.
4. Science revision 0.6.90.5.5 remains separately reported.
5. Historical exported identifiers containing version suffixes are preserved for compatibility; source-architecture naming changes do not rewrite them.

5.16 Adding or changing atomic-data/rate support

Scientific changes are outside ordinary productization. Before modifying a data/rate path:

1. identify the canonical Fortran `setptrs/ucllc/caller` behavior;
2. identify the data type and rate type separately;

3. update the Fortran/Python/C++ concordance entry;
4. add characterization evidence before changing frozen code;
5. test one fast affected model first;
6. widen qualification only after that model passes;
7. reopen the science revision only when the change is intentionally scientific.

The key semantic rule is: **data type determines how record constants are interpreted/calculated; rate type determines how the resulting quantity is consumed by XSTAR.**

See `docs/science/rate_data_semantics.md`, `docs/developer/fortran_source_map.md`, and `docs/developer/python_cpp_fortran_concordance.md`.

5.17 Source-name hygiene and ABI naming

`xstar_tools` separates **private implementation names** from **exported ABI symbol names**.

Private Python/C++ helpers should use stable semantic names. Development-version labels such as `_v0648...` or `_patch...` should not be introduced into private function definitions. If a historical helper is unreachable, remove it rather than preserving a campaign-era name indefinitely.

Exported C-linkage symbols are different: their spelling is part of the ABI. Existing `_v1` and production-zone `_v0648110` exports remain versioned because external binaries, plugins, ctypes, or dynamic-library clients may bind those exact names. Renaming internal callers does not make an ABI rename safe.

If a cleaner semantic C API is introduced later, add wrapper/alias symbols while retaining the old exports for a documented ABI transition. Remove old exported names only under an intentional ABI revision with qualification.

5.17.1 Warning-only annotations

Compiler-warning cleanup may use local standard-language annotations such as `[[maybe_unused]]` where appropriate. Do not hide broad warning classes with global `-Wno-*` flags simply to make a release appear clean.

Current science-critical naming/ABI changes are governed by the compact parity-freeze and source-concordance gates rather than by closed version-specific source-name manifests.

5.18 Production diagnostics policy for Python execution modes

5.18.1 Scope

`0.6.80` separates normal public output from the historical diagnostics used to qualify and debug the source-faithful Python implementation.

The scientific boundary is unchanged:

- accepted Python science revision: `0.6.90.5.5`;
- frozen C++ baseline: `0.6.48.12.3.44`;
- C API ABI: `60487`;
- production-zone ABI: `6048110`.

5.18.2 Public production behavior

Stable `run_xstar(...)`, `XStarConfig`, and `xstar-tools run` executions in `pure-python` and `zone-python` keep the accepted internal `diagnostics_mode="full"` computational/orchestration path, but diagnostic file capture/emission is disabled by default.

Normal production runs therefore do not create:

- `xstar_tools_continuum_diagnostics_summary.json`;
- `xstar_tools_phase_snapshots.csv`;
- `xstar_tools_phase_snapshots.jsonl`;
- `xstar_tools_ucalc_continuum_side_effects.csv`;
- `xstar_tools_ucalc_continuum_side_effects.jsonl`;
- `radial_spectrum_diagnostics_v0500/` (or the older internal `v0499` spelling).

`runner_summary.json` contains stable execution/data/backend provenance rather than serializing the historical `Mg/DSEC/shadow/forensic` attribution maps.

This is intentionally **not** implemented by switching to `diagnostics_mode="none"`. That expert mode has independent retention/ownership semantics inside the source-faithful solver and requires separate scientific qualification before it can become a production optimization.

5.18.3 Explicit debugging

Development/qualification diagnostics remain available through the existing advanced interface:

```
xstar-tools dev legacy-run ... --diagnostics full
```

The compatibility Python interface can also explicitly request both output surfaces:

```
run_xstar(
    mode="pure-python",
    ...,
    write_diagnostic_files=True,
    include_debug_provenance=True,
)
```

These controls are deliberately not promoted into `XStarConfig`; ordinary users should receive production outputs by default.

5.18.4 0.6.75 three-model benchmark review

The user-supplied `benchmark_0675_python` archive contains successful `pure-python` and `zone-python` runs for:

- `helike_type69/ca19_xi2_ne1`;
- `helike_type69/mg11_ne1e8`;
- `helike_type69/o7_ne1e10`.

The six benchmark subprocesses completed successfully. Direct comparison of the two Python modes gives:

Model		STEP sci- ence	FITS material result
Ca	XIX	ACCEPT	8 products exact; xout_spect1 material NL1 about $4.03\text{e-}17$
ca19_xi2_ne1			
Mg	XI	ACCEPT	all product schemas/identities match; worst material NL1 about $2.45616\text{e-}4$ (0.0246%) in xout_abund1
mg11_ne1e8			
O VII o7_ne1e10		ACCEPT	all 9 FITS table payloads numerically/structurally identical

These results show no material pure-python versus zone-python scientific problem under the frozen <1% policy. The uploaded archive does not contain the canonical `original_xstar.tar.gz` Fortran product reference, so this review does not claim a new independent Fortran comparison.

5.19 XSTAR2XSPEC / xstar2table source concordance — 0.6.81

This document records the canonical XSTAR source semantics used by the `xstar_tools 0.6.81` native characterization layer. It is a product-layer port only; it does not change XSTAR atomic or plasma science.

5.19.1 Canonical sources reviewed

The supplied XSTAR Release 2.5x source archive was mapped against the manual, especially Chapter 6 and Chapter 14.3.

Canonical source	0.6.81 responsibility	SHA-256 of supplied source
xstar/src/ xstar2table/ xstar2table.c	reads xout_spect1.fits, slices energy, computes AIN/AOUT/MTABLE/ETABLE, maps loopcontrol to table row/column	5a69fba776f8644f55718f49a75743b1e88c44fd64
xstar/ xstarlib/src/ xstartablelib. c	writes ENERGIES and SPECTRA extensions, PARAMVAL, INTPSPEC, ADDSPnnn, LASTSPEC, OGIP keywords	0f99aea6f73f7f13a4e85f4957e4979714b96e4666
xstar/src/ xstinitable/ xstinitable.c	writes PRIMARY/PARAMETERS table metadata; interpolated rows precede additive rows	eece9d4ce052eca976710dd2f45a4ab3353e423bfc
xstar/scripts/ xstar2xspect	orchestration order and one xstar2table call per XSTAR model	fb4593724ed10b9e0eb671cfcb5c530d501b4cdcb5

5.19.2 Canonical pipeline

```
xstinitable
-> PRIMARY/PARAMETERS table template
-> xstinitable.lis XSTAR command list

xstar2xspect
-> run XSTAR for each grid point/additive case
-> call xstar2table after each run

xstar2table
-> read xout_spect1.fits
-> fill xout_ain.fits
```

(continues on next page)

(continued from previous page)

```
-> fill xout_aout.fits
-> fill xout_mtable.fits
-> fill xout_etable.fits
```

The manual describes the same three-component division. `xstinitable` creates the parameter grid/table skeleton, while `xstar2table` post-processes each `xout_spect1.fits` into the XSPEC table products.

5.19.3 FITS input contract

The historical `Read_XSTAR_Spectra` routine moves to `XSTAR_SPECTRA` and reads all five columns with CFITSIO TFLOAT:

```
energy
incident
transmitted
emit_inward
emit_outward
```

Therefore the 0.6.81 compatibility reader intentionally stores these as 32-bit float arrays. The compatibility transform also uses float intermediate/output storage to preserve historical rounding behavior.

`rlrad38`, `loopcontrol`, and interpolated parameter values are read from the ordinary XSTAR PARAMETERS extension.

5.19.4 Energy slicing

The historical `SliceEnergySpectra` behavior is unusual and is preserved for normal interior ranges:

- the low side includes the interval that straddles ELOW;
- the high side keeps the last interval whose upper edge is not above EHIGH;
- table edges are written in keV, while XSTAR input spectral energies are eV.

For example, with source energies

```
50, 100, 200, 400, 800, 1600 eV
```

and ELOW=150, EHIGH=900, the historical selection is

```
100-200
200-400
400-800 eV
```

The original C function can read `energy[j+1]` at the final array element and leaves some state undefined if a boundary is exactly the first grid value. 0.6.81 makes only those endpoint cases bounds-safe; the canonical interior selection semantics are unchanged.

5.19.5 Spectral transforms

For selected source bin `i`, the historical additive normalization is:

```
enorm = 8.356e-7 * (energy[i+1] - energy[i])
        / (rlrad38 * energy[i])

AIN[i]  = enorm * emit_inward[i]
AOUT[i] = enorm * emit_outward[i]
```

AIN and AOUT are written as XSPEC additive spectra in photons/cm²/s per table bin.

The multiplicative model is:

```
if incident[i] == 0:
    MTABLE[i] = 0
else:
    MTABLE[i] = transmitted[i] / incident[i]
```

The historical code diagnoses ratios outside [0, 1] but does not clamp them. 0.6.81 preserves the numerical value.

The exponential model reuses the multiplicative spectrum and applies only the historical floor used for the logarithm:

```
transmission = max(MTABLE[i], 1e-32)
ETABLE[i] = -ln(transmission)
```

Consequences deliberately covered by the 0.6.81 fixture:

- exact zero incident flux -> MTABLE=0;
- tiny positive transmission -> ETABLE=-ln(1e-32);
- transmission greater than one remains greater than one in MTABLE and gives a negative ETABLE optical depth, matching the historical arithmetic.

5.19.6 XSPEC table schema

The 0.6.81 writer reproduces the legacy table organization:

```
PRIMARY
PARAMETERS
ENERGIES
SPECTRA
```

PARAMETERS columns:

NAME	12A
METHOD	J
INITIAL	E
DELTA	E
MINIMUM	E
BOTTOM	E
TOP	E
MAXIMUM	E
NUMBVALS	J
VALUE	nE

Interpolated parameters are written before additive parameters. An interpolated row has positive NUMBVALS; an additive row has zero NUMBVALS.

ENERGIES uses 32-bit float columns:

ENERG_LO	E	keV
ENERG_HI	E	keV

SPECTRA contains:

```
PARAMVAL
INTPSPEC
ADDSP001
ADDSP002
...
```

with 32-bit float vector columns. `ADDMODEL` is true for `xout_ain.fits` and `xout_aout.fits`, false for `xout_mtable.fits` and `xout_etable.fits`. `REDSHIFT` is retained as the primary-header logical metadata flag.

5.19.7 Parameter and job ordering

Legacy `xstar2table` maps:

```
row    = 1 + (loopcontrol - 1) / (NADDPARM + 1)
column = (loopcontrol - 1) % (NADDPARM + 1)
```

column zero is `INTPSPEC`; positive columns are `ADDSPnnn`.

0.6.81 deliberately keeps the sequential `LASTSPEC/positive-loopcontrol` contract because this release is characterization, not the parallel-grid redesign. For an ordinary standalone XSTAR file that records `loopcontrol=0`, explicitly supplied file order is accepted as the compatibility job order.

The later parallel milestone will replace completion-order dependence with a stable job-ID destination mapping.

5.19.8 Energy-grid consistency improvement

The canonical source contains a placeholder asking whether later spectra match the first `ENERGIES` extension; it does not actually perform the check. 0.6.81 makes exact selected-energy-grid equality a hard error. This does not change a valid canonical table but prevents silently combining incompatible spectra.

5.19.9 Scope boundary

0.6.81 does **not** implement the full native `xstinitable` planner. A compact metadata file supplies the table definition while ordinary `xout_spect1.fits` files supply the spectra and parameter values. Full constant/additive/interpolated grid generation is scheduled for 0.6.83.

0.6.81 also does not yet consume `ProductWritingState` directly; the in-memory bridge is the 0.6.82 milestone.

5.19.10 0.6.81.1 canonical 2x3-grid closure

A real XSTAR 2.59g / MPI_XSTAR grid with `column=(1e20,1e21)` and `rlogxi=(1,2,3)` exposed two characterization defects in 0.6.81 that the synthetic fixture did not catch.

1. Historical `SliceEnergySpectra` reports `eBinHigh` as an **energy-edge index** while `nEnergyBins` counts intervals. For the canonical (4452–7866) edge range the correct number of XSPEC bins is therefore $7866 - 4452 = 3414$, not 3415.
2. The source literal `8.356e-7` is a C double literal. Canonical `AIN/AOUT` arithmetic evaluates that constant in double precision and stores the final `enorm` in `float`; changing the literal itself to `8.356e-7f` produces small but real last-bit differences.

0.6.81.1 corrects only those two table-conversion details. The permanent compressed regression fixture contains the six canonical `xout_spect1.fits` inputs, `xstinitable.fits/.lis`, the four canonical table outputs, and the characterized metadata. Qualification requires bit-exact `ENERG_LO`, `ENERG_HI`, `PARAMVAL`, and all four `INTPSPEC` arrays. XSTAR scientific kernels and XSPEC-table ABI 1 are unchanged.

5.20 Reference data and release candidates

Current release candidates use a compact reference boundary.

- `references/current/` catalogs the small active local/external reference policy.
- `qualification/frozen/` contains frozen comparator/reference code required to enforce accepted science.
- `qualification/parity_freeze_current_source_hashes.json` pins exact active science-critical source bytes.
- `xstar_test_run/` retains small reference CSVs still consumed by runtime/regression code.
- `atdb.fits` remains external and must not be bundled.

Closed milestone reports, forensic scripts, and predecessor replay manifests are not active release inputs after the repository cleanup; version-control history and tags preserve those records.

Before producing a release candidate run:

```
python tools/ci/check_tree_clean.py
python tools/ci/check_package_metadata.py
python tools/qualification/check_parity_freeze.py
python tools/qualification/check_source_concordance.py
python tools/release/check_release_candidate.py
```

5.21 CI and release engineering

The active CI policy is intentionally compact. Closed version-specific host closure workflows are preserved in repository history rather than copied into each release.

5.21.1 Current always-on gates

`.github/workflows/ci.yml` installs the package with development dependencies, runs generic repository/metadata policy checks, validates the frozen scientific boundary and source concordance, then runs the current pytest regression suite.

The generic policy tools are:

- `tools/ci/check_tree_clean.py` — rejects generated build/cache artifacts;
- `tools/ci/check_package_metadata.py` — package-version/science/ABI consistency;
- `tools/ci/check_python_contract.py` — Python syntax/import/public API typing;
- `tools/ci/check_text_policy.py` — conservative text-format policy;
- `tools/qualification/check_parity_freeze.py` — exact scientific source/reference/ABI boundary;
- `tools/qualification/check_source_concordance.py` — current Fortran/Python/C++ correspondence map;
- `tools/release/check_release_candidate.py` — current release-tree boundary.

`.github/workflows/parity-freeze.yml` provides a lightweight standalone frozen boundary check. `.github/workflows/pypi-release.yml` is the generic wheel/sdist build workflow.

5.21.2 Optional deeper qualification/report helpers

`tools/ci/check_science_results.py`, `require_science_assets.py`, `report_benchmark_results.py`, and `performance_report.py` remain for controlled scientific/performance qualification where external reference assets or specialized runners are available. They are not historical replay gates.

5.22 Packaging and native-wheel policy

5.22.1 Authoritative configuration

`pyproject.toml` is authoritative for project metadata, dependencies, entry points, package discovery, and installed static package data.

`setup.py` is deliberately thin. It registers only `XStarBuildPy` and `XStarDistribution` from `build_support.py`; it contains no duplicated project metadata. `setup.cfg` was removed in 0.6.71. Setuptools may synthesize a minimal `[egg_info] setup.cfg` inside an sdist; that generated tag file contains no authoritative project metadata or build policy.

5.22.2 Why setuptools remains the build backend

A scikit-build-core/CMake migration would require translating a mature Makefile that carries qualification-sensitive compiler flags, per-translation-unit floating-point controls, RPATH behavior, CFITSIO linking, and the standalone shared-production architecture. Milestone 8 therefore chooses the lower-risk path: retain setuptools and formalize the existing Makefile build as an isolated wheel-build step.

A future CMake migration should be treated as its own science-neutral build refactor with exact native product qualification.

5.22.3 Native wheel construction

The build hook:

1. builds normal Python modules/package data;
2. evaluates `XSTAR_TOOLS_NATIVE`;
3. validates platform-native build prerequisites;
4. copies the self-contained `src/xstar_tools/xstar/cpp/` tree to a temporary staging directory;
5. runs `make clean` and then the retained `make all` there;
6. verifies every required runtime artifact exists;
7. copies only runtime binaries/libraries to the wheel build tree;
8. writes `native_build.json` with package/science/ABI/build provenance.

This design prevents old `.o/.so` files in a checkout from entering a wheel.

PyPI Linux release profile (0.6.89.2, closures through 0.6.89.2.2)

The 0.6.89.2.1 workflow first installs the declared PEP 517 build-backend requirements (`setuptools>=77` and `wheel`) into the qualification interpreter before importing `build_support.py`. 0.6.89.2.2 additionally maps the `pypi-linux` packaging profile to a dedicated `make pypi-linux` target so the standalone Python-embedding plugin is never built in the release-wheel path; ordinary `make all` remains unchanged. Public Linux release wheels are produced with `cibuildwheel` in a `manylinux_2_28_x86_64` container and repaired with `auditwheel`. CFITSIO 4.6.2 is built from a pinned official source archive inside the container and its runtime library is vendored into the repaired wheel.

`XSTAR_TOOLS_NATIVE_PROFILE=pypi-linux` changes only the staged binary payload: it omits `libxstar_backend_python.so`, whose standalone embedding contract links `libpythonX.Y` and therefore is not a valid portable manylinux dependency. The default full profile used by ordinary source/native installations is unchanged. The PyPI profile still contains the public C API, production-zone library, modular C++ libraries, `xstar-cpp`, and all non-MPI XSTAR2XSPEC executables.

The clean installed-wheel test exercises `zone-cpp`, so omission of the embedding plugin does not weaken the Python-to-native production path.

Cross-platform PyPI release closure (0.6.89.5)

0.6.89.5 - `PYPI_CROSS_PLATFORM_RELEASE_CLOSURE` does not introduce a new science or native-runtime implementation. It advances package/version metadata and rebuilds the already-qualified `pypi-linux`, `pypi-macos`, and `pypi-windows` profiles under one version. The release workflow uses GitHub-hosted runners and produces exactly 24 native wheels plus one sdist:

- 6 CPython 3.9–3.14 `manylinux_2_28_x86_64` wheels;
- 6 CPython 3.9–3.14 `macosx_11_0_arm64` wheels;
- 6 CPython 3.9–3.14 `macosx_11_0_x86_64` wheels;
- 6 CPython 3.9–3.14 `win_amd64` wheels;
- 1 `xstar_tools-0.6.89.5.tar.gz` source distribution.

The aggregate gate rejects extra or missing distributions and specifically rejects a `py3-none-any` control wheel from the publication set. Each native wheel is clean-install tested by `cibuildwheel` on its platform before aggregation. The final aggregate job runs `twine check` and writes SHA-256 checksums. Publication itself remains a separate deliberate action so the same qualified bundle can be sent first to TestPyPI and then to PyPI.

No local Mac or Windows machine is required for release construction: macOS arm64/x86_64 and Windows AMD64 are built and tested on the corresponding GitHub-hosted runners.

5.22.4 Wheel contents

Normal wheels include only installed Python modules plus required runtime data:

- compact benchmark/reference fixtures still consumed by current tests/runtime;
- `coheat.dat`;
- `constants.def`;
- public C/production-zone ABI headers;
- built native runtime artifacts on native wheels;
- `native_build.json`.

They exclude:

- generated/history-only material that is not part of the active release contract;
- documentation source;
- tests and qualification reports;
- C++ implementation `.cpp` files and Makefile;
- object files, caches, and source-tree build products;
- `atdb.fits`.

The sdists retains source/build/qualification material required to reproduce and validate a build, while still pruning the top-level historical archive.

5.22.5 Data policy

`atdb.fits` is external scientific data. It is not bundled and is never fetched silently during a run. Installed packages store configured data paths under the user configuration directory (normally `~/.config/xstar-tools/datapath`) and use a user data default (normally `~/.local/share/xstar-tools`) rather than writing into site-packages.

5.22.6 Release checks

Packaging CI must perform the equivalent of:

```
python -m build
python -m twine check dist/*
pip install dist/*.whl
xstar-tools doctor
```

Native Linux jobs additionally run:

```
xstar-cpp --abi
xstar-tools doctor --require zone-cpp
```

Clean-wheel characterization tests verify import/API/CLI behavior and wheel payload policy without relying on the source checkout.

Persistent release-build staging

For constrained release builders, `XSTAR_TOOLS_NATIVE_BUILD_DIR=/path` enables a persistent staging directory. The hook fingerprints the complete native source plus `pyproject.toml` and automatically discards the staging tree if that fingerprint changes. This allows long qualified translation units to be built incrementally without ever consuming unchecked object files from the source checkout. Normal builds do not set this variable and use a temporary clean stage.

5.22.7 Conda packaging

The conda-forge recipe reuses this same build hook and native capability policy; see [Conda and conda-forge packaging](#).

Windows delvewheel in-wheel dependency resolution

For the `pypi-windows` profile, XSTAR DLLs are already packaged under `xstar_tools/xstar/cpp`. The repair command intentionally uses `--ignore-existing` together with `--analyze-existing` `--analyze-existing-exes`: the first enables resolution of DLLs already inside the wheel and keeps those XSTAR DLLs in place; the analysis flags still discover external CFITSIO and MinGW runtime dependencies for vendoring. The external search path remains `XSTAR_TOOLS_WINDOWS_DLL_PATH`.

5.23 Conda packaging

The 0.6.90 conda campaign is closed. The accepted public conda-forge package is `xstar-tools 0.6.89.5 build 1` on `linux-64`, `osx-arm64`, and `osx-64`. Windows conda packages are not supported.

The repository copy under `conda/recipe/` is retained as the accepted recipe reference. `tools/packaging/prepare_local_conda_recipe.py` can prepare that `recipe.yaml` for local qualification work. The closed 0.6.90.1 and 0.6.90.2 host/release-closure workflows are no longer carried in the active workflow directory; their exact definitions remain available from the corresponding repository history/tag.

The scientific/runtime policy is unchanged: conda uses external conda-forge CFITSIO, `atdb.fits` stays external, ordinary packages are non-MPI, and native scientific behavior remains governed by the parity freeze.

5.24 Documentation authoring and build policy

Authored user/developer/science guides use **MyST Markdown**. reStructuredText is reserved mainly for Sphinx autodoc/autosummary API reference pages.

Sphinx provides cross-references, autodoc/autosummary, intersphinx, HTML generation, and link checking. The Read the Docs theme uses repository-owned branding assets under `docs/_static/`.

5.24.1 Release documentation gates

From `docs/sphinx/`:

```
make clean
make html
make linkcheck
```

Equivalent direct commands are:

```
sphinx-build -W --keep-going -b html docs docs/_build/html
sphinx-build -W --keep-going -b linkcheck docs docs/_build/linkcheck
```

Warnings are errors for release documentation. Every authored history page must be reachable from a toctree; API pages should be included exactly once; internal links must resolve.

5.24.2 Current versus historical documentation

Active documentation describes current behavior and current qualification commands. Closed milestone narratives, forensic reports, and removed checkers are preserved by Git history/release tags rather than referenced as if they were still shipped directories.

A productization/documentation-only release must not change the accepted science revision, ABI contracts, or frozen science-critical source bytes.

5.25 Building the documentation

The documentation source under `docs/` supports HTML, LaTeX source, and PDF output through Sphinx. The build entry point is `docs/sphinx/Makefile` on POSIX systems and `docs/sphinx/make.bat` on Windows.

5.25.1 Install the Sphinx dependencies

From the repository root, install the documentation extra into the Python environment used for the build:

```
python -m pip install -e ".[docs]"
```

For reproducible Read the Docs hosted builds, `docs/requirements.txt` pins the warning-clean Sphinx/MyST toolchain used by the accepted documentation closure and includes the RTD theme plus NumPy/Astropy needed by autodoc imports. The top-level `.readthedocs.yaml` explicitly installs that requirements file.

A hosted build therefore uses the equivalent dependency step:

```
python:
  install:
    - requirements: docs/requirements.txt
```

The current hosted documentation toolchain is Sphinx 8.2.3, MyST Parser 5.1.0, and `sphinx-rtd-theme` 3.1.0 on Python 3.13.

5.25.2 External tools for LaTeX/PDF

The HTML builder needs only the Python/Sphinx documentation environment. The LaTeX builder additionally needs ImageMagick for SVG-to-EPS3 conversion. The PDF target uses the classic DVI/PostScript pipeline and therefore requires `latex`, `dvips`, and `ps2pdf`.

Check the external tools with:

```
convert --version
latex --version
dvips --version
ps2pdf --version
```

The SVG converter prefers the ImageMagick 7 `magick` launcher and falls back to the ImageMagick 6 `convert` command on POSIX systems. The four architecture SVG diagrams are converted with the ImageMagick EPS3 coder, equivalent to:

```
convert input.svg eps3:output.eps
```

HTML continues to use the original SVG files.

5.25.3 HTML

```
cd docs/sphinx
make clean
make html
```

The HTML documentation is written to:

```
docs/_build/html/
```

The HTML target uses `-W --keep-going`, so Sphinx warnings fail the target after the remaining sources have been processed.

5.25.4 LaTeX source

Generate the Sphinx LaTeX document without compiling a PDF:

```
cd docs/sphinx
make latex
```

The primary output is:

```
docs/_build/latex/xstar-tools.tex
```

During this target Sphinx converts each architecture SVG to EPS3. Sphinx normally emits references such as:

```
\sphinxincludegraphics{{backend_dispatch}.eps}
```

The project then normalizes that generated spelling to the conventional filename form:

```
\sphinxincludegraphics{backend_dispatch.eps}
```

The same normalization applies to `local_zone_solve.eps`, `controller_radial_flow.eps`, and `publication_ownership.eps`. Only generated LaTeX is changed; HTML continues to use SVG.

The converted EPS assets are copied into `docs/_build/latex/` beside `xstar-tools.tex`.

5.25.5 PDF: LaTeX -> DVI -> PostScript -> PDF

The project intentionally uses the classic EPS-aware pipeline rather than Sphinx `-M latexpdf/latexmk`:

```
SVG -> ImageMagick EPS3 -> LaTeX/DVI -> dvips -> PostScript -> ps2pdf -> PDF
```

Run:

```
cd docs/sphinx
make latexpdf
```

The target is equivalent to:

```
make latex
cd ../_build/latex
latex -interaction=nonstopmode -halt-on-error xstar-tools.tex
latex -interaction=nonstopmode -halt-on-error xstar-tools.tex
dvips -o xstar-tools.ps xstar-tools.dvi
ps2pdf xstar-tools.ps xstar-tools.pdf
```

Two LaTeX passes are used so the table of contents, references, and page numbers can settle before the DVI is converted to PostScript.

The expected outputs are:

```
docs/_build/latex/xstar-tools.tex
docs/_build/latex/xstar-tools.dvi
docs/_build/latex/xstar-tools.ps
docs/_build/latex/xstar-tools.pdf
```

The LaTeX configuration does not load `epstopdf`; EPS is consumed natively by the `latex/dvips` path.

5.25.6 Link checking and release documentation

The link checker remains:

```
make linkcheck
```

The normal documentation release target remains:

```
make release
```

and builds HTML plus link checking only. PDF generation stays explicit because ImageMagick, TeX, dvips, and Ghostscript are optional host tools.

5.25.7 Windows helper

The Windows helper exposes matching commands:

```
cd docs\sphinx  
make.bat html  
make.bat latex  
make.bat latexpdf
```

The Windows `latexpdf` command uses the same two-pass `latex -> dvips -> ps2pdf` pipeline when those programs are available on `PATH`.

Release-history records are indexed under *[Project history](#)*.

SCIENCE GUIDE

The science documentation explains the algorithms and source correspondence without replacing the canonical executable semantics.

6.1 Algorithm overview

At a high level, a source-faithful XSTAR calculation performs:

1. parameter/data initialization and atomic-record pointer construction;
2. shell/radial controller setup;
3. local ionization/thermal/rate construction;
4. statistical-equilibrium matrix assembly and solve for selected ions/levels;
5. emissivity, opacity, and line/RRC/continuum construction;
6. radial transfer, convergence, and shell-state updates;
7. STEP and final FITS publication.

The exact numerical/discrete semantics are defined by the accepted qualification evidence and canonical XSTAR Fortran source. Literature equations explain the physical model but do not by themselves specify record traversal, numerical kinds, cutoffs, workspace lifetime, publication ordering, or accepted source quirks.

See `docs/developer/fortran_source_map.md` for the routine-level map and `docs/developer/python_cpp_fortran_concordance.md` for implementation correspondence.

6.2 Data-type and rate-type semantics

XSTAR atomic records carry two distinct classifications that must not be conflated:

- **data type** selects the record layout/formula used to interpret constants and calculate a rate, cross section, or related quantity;
- **rate type** selects how XSTAR consumes the resulting quantity in the physical calculation.

The XSTAR manual and atomic-database literature explain these families; the current `ucalc.f90` dispatcher remains the executable authority for the exact active branching used by the qualified implementation.

When adding support for a record family, document both dimensions, its caller context, pointer/level topology, units, and the characterization model used to prove equivalence.

See `docs/science/xstar_references.md` for the literature mapping.

6.3 XSTAR scientific references and implementation correspondence

This bibliography/concordance was built from the supplied reference bundle (`xstar_references.tar.gz`, SHA-256 `a66718cf3b3d831906ed640c3bb9a2d8e81ac3bcd9fd4878d09fc680e6616e84`) and the supplied XSTAR 2.59g Fortran source.

6.3.1 Evidence hierarchy

For this project, references are interpreted in the following order:

1. **accepted qualification evidence** for behavior already closed in `xstar_tools`;
2. **canonical XSTAR 2.59g executable semantics** for source behavior;
3. **papers/manuals** for equations, assumptions, physical interpretation, and provenance.

A paper is therefore not used to “correct” already qualified executable behavior merely because it presents an idealized or older formulation differently.

6.3.2 Core references

XSTAR Team, *XSTAR Manual*, Release 2.5x, 20 Dec 2025

Role: primary current explanatory manual supplied with the project references.

Important sections for the port:

Man- ual sec- tion	Topic / algorithm	Fortran correspon- dence	Python/C++ correspon- dence	Implementation rela- tion
4.1- 4.3	user parameters, hidden controls, density/pressure, ionization parameter, step/print controls	rread1.f90, top of xstar.f90	parameter normalization in physical_runner.py; native parameter/default-real helpers	source-exact for executable conversions/default-kind effects; UI representation may differ
5	output products and xout_step.log	pprint.f90, fstepr*, writespectra*	pprint_legacy.py, output_writers.py, xstar_step_log.cpp, xstar_science_fits.cpp	source-equivalent publication; comparator policy is qualification metadata
11.1	spherical point-source geometry, steady state, inward/outward diffuse radiation, local escape probabilities	radial/transfer caller in xstar.f90, escape factors in rate/emission routines	radial transfer and escape-state modules; native run state	mathematically/source equivalent within XSTAR's stated approximation
11.3	ionization parameter, including $\xi = L/(n R^2)$ in the optically thin limit	parameter/radius/radiation calculation in rread1 and radial state	parameter normalization/radiation state	source-exact numerical semantics where source kinds matter
11.4.1	two-step population algorithm: preliminary ion fractions then full multi-level kinetic matrix	calc_ion_rates, istruc/ioneqm, levwk*, calc_hmc_ion/element, msolve Lucy	ion_balance.py, element_equilibrium.py, fixed-state C++	source-equivalent; compact representation optimized/translated
11.4.4	thermal equilibrium from heating/cooling balance	calc_hmc_all, dsec, comp2/freef/bremem/heatf	local_zone.py, dsec.py, thermal modules/kernels	source-exact controller; equivalent/optimized leaves
11.5	recombination-continuum emission and escape	RRC branches in ucalc, calc_emisab_*, calc_emis_*, fstepr3, writespectra4	emissivity/publication modules and native FITS path	source-equivalent; terminal publication lifetime explicitly qualified
11.5.1	line emission and escape probabilities; upper-level population times net decay with trapping/destruction	line rate/emission paths, pesc1/pescv, Type50/line transfer	element_equilibrium, emergent_emissivity, C++ line/opacity kernels	source-equivalent, Type50 optimized-equivalent
11.6	continuum emission	freef, bremem, recombination/two-photon branches	free-free/brems/emission modules and C++ kernels	mathematically/source equivalent
11.6.1	photoabsorption and Thomson-scattering opacity	bound-free opacity/rate paths and continuum opacity assembly	opacity/emergent-emissivity/native opacity paths	source-equivalent
11.6.2 11.6.5	shell-by-shell two-stream transfer and iterative passes	step, trnfr, trnfrn, stpcut, radial loop in xstar.f90	radial_transfer.py, radial_control.py, native controller	source-exact control/predicate intent; object storage differs
11.7	atomic-process summary	ucalc, data/rate-type branches and thermal/rate consumers	ucalc.py, rate kernels	source-exact/equivalent per qualified data/rate type
6.3. XSTAR scientific references and implementation correspondence				
12.1-	atomic database records:	readtbl, setptrs,	atomic_database.py,	source-exact record

The manual’s population description is especially important: XSTAR first estimates ion fractions using total ionization/recombination rates, filters/selects an adjacent set of significant ions (controlled by `critf`), then solves the full kinetic matrix for levels in those ions. That directly corresponds to the split between `ion_balance.py` and `element_equilibrium.py`.

The manual also makes the publication/transfer distinction explicit: local radiation used for rates is not identical to every spectrum ultimately exposed to an observer. This supports preserving distinct local, persisted-shell, and final-output ownership in the port.

T. Kallman & M. Bautista (2001), “Photoionization and High-Density Gas,” *ApJS* 133, 221-253

Scientific topic: XSTAR v2 computational model, high-density physics, level populations, heating/cooling, recombination continuum and line escape, and LTE consistency.

Important correspondence:

Paper topic	Equation/algorithm content	Fortran routine(s)	Python/C++ implementation	Relation
basic model assumptions	steady-state photoionized gas with simplified diffuse transfer/escape treatment	radial caller, transfer and escape routines	radial/escape modules	source model implements the paper’s approximation; current source details prevail
multi-level populations	simultaneous excitation/ionization through explicit levels plus super-levels/continuum levels	levwk*, calc_hmc_ion/element, msolve Lucy	element_equilibrium.py, native local-zone engine	mathematically/source equivalent
Compton heating/cooling	paper equation (1) gives the nonrelativistic Compton energy exchange form	comp2.f90	compton.py, native thermal kernel	mathematically equivalent and qualified
bremsstrahlung cooling	paper equation (2) summarizes free-free cooling	freef.f90, bremem.f90	free_free.py, bremsstrahlung.py	mathematically/source equivalent
recombination/RRC	Milne-relation recombination rates and emissivity (paper equations 3-4)	photoionization/recombination ucalc branches, calc_emisab_*, calc_emis_*	ucalc.py, emissivity modules, C++ rate/spectral kernels	source-equivalent; current ATDB/data types govern details
continuum escape	escape suppression for optically thick recombination continua (paper equation 5)	escape-probability use in rate/emission routines	escape context and equilibrium/emissivity modules	mathematically/source equivalent
resonance-line escape/destruction	paper equations 6-11 and associated discussion	line escape/rate paths and linopac context	pesc1/pescv, Type50/full-grid line path	source-equivalent; native Type50 traversal optimized-equivalent
thermal balance	heating/cooling terms assembled into a local equilibrium solution	calc_hmc_all, dsec	local_zone.py, dsec.py, thermal kernels	source-exact controller/equivalent kernels

This paper explains *why* the code is organized around level-specific atomic data and a coupled ionization/excitation solve. The 2.59g source is used for exact data-type/rate-type branching and current control flow.

M. A. Bautista & T. R. Kallman (2001), “The XSTAR Atomic Database,” *ApJS* 134, 139-149

Scientific topic: database organization, multilevel ion models, level-specific rates, detailed balance, and LTE convergence.

The paper describes a database containing level energies, wavelengths, radiative probabilities, collision rates, photoionization cross sections, recombination, collisional ionization, and fluorescence/Auger information. It emphasizes level-specific processes and multilevel models designed to approach LTE under appropriate conditions.

Correspondence:

Paper concept	Fortran	Python/C++	Relation
database record families and level-specific rates	packed ATDB + readtbl, setptrs, drd, ucalc	atomic_database.py, ucalc.py, native ATDB/rate kernels	source-exact identity/traversal; storage optimized
statistical-equilibrium balance over level transitions	calc_hmc_ion/element, msolverlucy	element_equilibrium.py, matrix/local-zone engine	mathematically/source equivalent
continuum level representing the adjacent ion/ionization channel	compact level topology and parent/continuum pointers	derived pointers + compact basis	source-exact identities are required; 12.3.25 endpoint repair is part of qualification
detailed-balance/LTE design	calc_rates_level_11 levwk*, inverse-rate branches	level/LTE/rate modules	source-equivalent
state-specific recombination/photoionization	ATDB data types and ucalc	ucalc.py, native rate kernels	current Fortran/data records are canonical

This paper is the principal scientific context for treating `setptrs`/record attachment and compact level topology as part of the scientific contract rather than as disposable implementation detail.

C. Mendoza et al. (2021), “The XSTAR Atomic Database,” *Atoms* 9, 12, DOI 10.3390/atoms9010012

Scientific topic: two decades of XSTAR atomic-database updates, K-line support through $Z \leq 30$, high-density extensions, and data provenance/curation.

Use in the concordance:

- confirms that the atomic database and code are designed for broad, systematically updated data coverage rather than a fixed small set of ions;
- motivates preserving atomic-data provenance and explicit database identity/hash reporting during productization;
- provides scientific context for high-density and inner-shell/K-line data now represented in the ATDB;
- does **not** override the supplied 2.59g `setptrs`/`ucalc` executable behavior.

Related implementation: `atomic_database.py`, ATDB fingerprint/cache metadata, native ATDB reader, rate kernels, future public data/provenance API.

Relation: database/provenance context; source-exact behavior comes from current Fortran + ATDB.

Atomic-data record semantics used by the native comments

The 0.6.54 C++ comments and 0.6.55 Python comments use two source-specific distinctions from **XSTAR Manual Chapter 12** and **Mendoza et al. (2021), Appendix A**. A record’s **data type** determines the formula/layout by which its constants are interpreted to calculate a rate or cross section; its **rate type** determines how XSTAR uses the returned quantity in the physical calculation. The ASCII database record header described by the manual contains six integers:

data type, rate type, continuation flag, number of real values, number of integer values, and number of character values. The current `uvalc.f90` remains the executable dispatcher and therefore the final authority for present branching.

The Appendix-A families called out directly in the native implementation are:

Data type	Source description used by the concordance	Native relevance
49	level-resolved partial photoionization cross section represented by energy/cross-section pairs	bound-free lowering and matrix/continuum ownership
50	bound-bound radiative line record containing wavelength/radiative information and lower/upper level identities	line rates, Type50 opacity/profile path, DSEC/manifold oracles
51	CHIANTI/Burgess-Tully effective collision strength for a bound-bound transition	collision-strength lowering/evaluation
53	resonance-averaged TOPbase partial photoionization cross section represented by energy/cross-section pairs	bound-free lowering and Type53 DSEC attribution
63	collisional transition probability reconstructed from quantum-defect/hydrogenic information	bound-bound collision path
70	superlevel recombination/photoionization data over density/temperature with a photoionization curve	superlevel bound-free path
71	radiative transitions from superlevels to spectroscopic levels	superlevel line path
72	satellite-level autoionization data	autoionization path
76	two-photon radiative decay	two-photon radiative branch
85	Fe K-edge photoionization parameterization	inner-shell bound-free path
86	Auger/radiative widths of a K-vacancy level	K-vacancy width/rate path
88	damped-excess photoionization cross section to a K-shell superlevel	K-shell bound-free path
91	APED radiative line data	routed by current <code>uvalc.f90</code> through Type-50 radiative handling
95	level collisional-ionization fit	collisional-ionization path
98	variable-length CHIANTI/Burgess-Tully effective collision-strength record	general collision-strength evaluator
99	newer superlevel recombination/photoionization table	superlevel bound-free/recombination path

Types **89**, **96**, and **97** are present in the supplied current `uvalc.f90` but are not enumerated in the requested Appendix-A/Chapter-12 data-type snapshots. Comments for those branches therefore identify the **current executable Fortran source**, not the two reference tables, as their semantic basis.

T. R. Kallman, D. Liedahl, A. Osterheld, W. Goldstein & S. Kahn (1996), “Photoionization Equilibrium Modeling of Iron L Line Emission,” *ApJ* 465, 994-1009

Scientific topic: detailed Fe L emission in photoionized gas, including the importance of recombination cascades and multilevel line formation.

Use in the concordance:

- supports the need for explicit excited-level populations and recombination cascades in photoionized spectra;
- provides scientific context for line/RRC emissivity products and for high-lying/superlevel treatment;
- is not used as the exact current control-flow specification because the modern XSTAR 2.59g implementation and 2001/2025 descriptions supersede its software details.

Related Fortran: level/rate solve, recombination and line emissivity paths.

Related Python/C++: `element_equilibrium.py`, emissivity/line kernels.

Relation: scientific context; current implementation is source-equivalent to modern Fortran, not asserted to be a line-by-line implementation of the 1996 model.

T. R. Kallman, P. Palmeri, M. A. Bautista, C. Mendoza & J. H. Krolik (2004), “Photoionization Modeling and the K Lines of Iron,” *ApJS* 155, 675-701

Scientific topic: Fe K-shell line emission/absorption using improved inner-shell atomic data, fluorescence/Auger processes, and damped photoionization resonances.

Use in the concordance:

- provides physical context for inner-shell photoionization/Auger branches and K-vacancy line formation;
- reinforces that line/edge observables depend on detailed level populations and source atomic data, not just total ion fractions;
- supports keeping database/data-type provenance visible in future APIs.

Related Fortran: inner-shell/fluorescence/Auger `ucalc` data types, level population solve, line/opacity paths.

Related Python/C++: `ucalc.py` typed rate families, matrix/rate kernels, line/full-grid spectral code.

Relation: scientific/data context; exact modern branching is governed by XSTAR 2.59g and the supplied ATDB.

6.3.3 Equations and algorithm ownership

The literature describes physics in continuous mathematical notation, while the source defines discrete grids, record traversal, numerical kinds, cutoffs, and publication ordering. The project therefore uses the following correspondence rules.

Ionization parameter

The manual states the optically thin scaling with $\xi = L/(n R^2)$, with L integrated over 1-1000 Ry. The implementation relation is **mathematically equivalent**, subject to source-specific units, spectrum integration, and default-REAL radius semantics.

Statistical equilibrium

The papers/database descriptions express each level equation as total population flow in equals total flow out, with a conservation equation replacing one row. `calc_hmc_ion/element` plus `msolve` are the executable realization. Python/C++ are considered **source-equivalent**, not merely equation-equivalent, because endpoint topology, record order, active ion selection, and normalization are part of qualification.

Recombination and RRC emission

Kallman & Bautista (2001) relate recombination rates/emissivities to photoionization cross sections through the Milne relation and describe escape suppression. In the implementation, the current ATDB data types and `ucalc` branches determine the exact quadratures/fits and inverse-rate construction. `calc_emisab_*` and `calc_emis_*` then expose integrated and full-grid products.

Line escape and Type50

The manual and 2001 paper describe complete-redistribution/escape-probability line treatment and continuum destruction. The source additionally specifies discrete line profile/bin accumulation (`linopac`) and source ordering. The C++ Type50 implementation is therefore classified **optimized-equivalent**, with equivalence established by the accepted Type50 and downstream product gates rather than by the analytic escape formula alone.

Continuum transfer

The manual describes shell-by-shell transfer with inward/outward diffuse radiation and iterative passes. The source implements the discrete realization in `xstar.f90`, `trnfrc`, `trnfrn`, `step`, and `stpcut`. Python factors these into explicit workspaces and functions but is classified **source-exact in control/predicate semantics**.

6.3.4 What the papers do not define for this port

The supplied papers/manuals do not, by themselves, define:

- exact Python/C++ object layout;
- exact floating-point kind/promotion for every source literal;
- fixed-capacity STEP rank identity behavior;
- the accepted comparator distinction between material numerical science and structural inventory;
- the exact 45.3.3.8 C5/Ca/O publication policy;
- the frozen C++ production ABI or optimization implementation.

Those are governed by accepted qualification evidence and the canonical source.

6.3.5 Reference bundle inventory

The Milestone 2 review used these supplied PDFs:

- XSTAR Team, *XSTAR Manual*, Release 2.5x, 20 Dec 2025.
- Kallman, T. & Bautista, M. (2001), *Photoionization and High-Density Gas*, ApJS 133, 221-253.
- Bautista, M. A. & Kallman, T. R. (2001), *The XSTAR Atomic Database*, ApJS 134, 139-149.
- Kallman, T. R., Liedahl, D., Osterheld, A., Goldstein, W. & Kahn, S. (1996), *Photoionization Equilibrium Modeling of Iron L Line Emission*, ApJ 465, 994-1009.
- Kallman, T. R., Palmeri, P., Bautista, M. A., Mendoza, C. & Krolik, J. H. (2004), *Photoionization Modeling and the K Lines of Iron*, ApJS 155, 675-701.
- Mendoza, C. et al. (2021), *The XSTAR Atomic Database*, Atoms 9, 12, DOI 10.3390/atoms9010012.

The PDFs are reference inputs; they are not copied into the normal `xstar_tools` distribution by this milestone.

6.3.6 Function-comment application (0.6.73)

The active Python/C++ function-level source comments were re-audited against the reference set above. The comments use Manual sections 11.4-11.7 for population, thermal, emission/opacity and transfer roles; chapter 12 for ATDB data/rate-type semantics; and chapter 14 for controller/radial-state flow. The 1996 Fe-L and 2004 Fe-K papers are cited only on functions that actually participate in those line/cascade or inner-shell/Auger contexts. See `docs/developer/function_commenting.md` for the reversible-overlay policy.

6.4 Equations and source correspondence

The project distinguishes three forms of correspondence:

- **source-exact/source-faithful** — control predicates, record traversal, numerical semantics, and publication behavior intentionally track Fortran;
- **mathematically equivalent** — the same physical relation is expressed differently while preserving qualified numerical behavior;

- **optimized-equivalent** — a transformed C++ implementation is accepted because dedicated science/product gates prove equivalence.

Important maps:

- docs/developer/fortran_source_map.md — canonical Fortran routines and workspaces;
- docs/developer/python_cpp_fortran_concordance.md — Python/C++ implementation mapping;
- docs/science/xstar_references.md — papers/manuals and their algorithmic context.

The authority hierarchy remains accepted qualification evidence, then canonical executable semantics, then literature for explanation/context.

PROJECT HISTORY

The active source tree carries current documentation plus the small frozen artifact set needed to enforce accepted scientific behavior. Closed version-by-version forensic/qualification material is retained by version-control history and release tags rather than copied into every new package.

7.1 0.6.90.5.2 — LATEX_DVI_PS_PDF_PIPELINE_CLOSURE

This documentation-build closure replaces the generic Sphinx `latexpdf` make-mode path with the project's explicitly qualified EPS-aware classic TeX pipeline:

```
SVG -> ImageMagick EPS3 -> LaTeX/DVI -> dvips -> PostScript -> ps2pdf -> PDF
```

`make latex` still uses the Sphinx LaTeX builder and the `svg_to_eps` extension to convert the four architecture SVG diagrams to EPS3. A small deterministic postprocessor then normalizes Sphinx image references from forms such as `{{backend_dispatch}.eps}` to the conventional `backend_dispatch.eps` filename spelling inside `\sphinxincludegraphics{...}`.

`make latexpdf` depends on `make latex`, executes two `latex` passes, then runs `dvips` and `ps2pdf`. It no longer calls Sphinx `-M latexpdf`, `latexmk`, `pdflatex`, or `epstopdf`. The normal documentation release target remains HTML plus link checking and therefore does not require TeX/Ghostscript tooling.

Read the Docs continues to build HTML with SVG assets and the pinned documentation dependency set established in 0.6.90.5.1. There are no science, ABI, production-kernel, or runtime behavior changes in this closure.

7.2 0.6.90.5.1 - LATEXPDF_MAKE_MODE_EPS_GRAPHICS_AND_READTHEDOCS

This documentation-build hotfix corrects the Sphinx make-mode invocation introduced in 0.6.90.5, establishes vector EPS graphics for the LaTeX documentation path, and closes the hosted Read the Docs dependency boundary.

The `latexpdf` target now places `-M latexpdf` first, as required by Sphinx make-mode, followed by the source/build directories and warning options. The `latex` and `latexpdf` builders use a local Sphinx image post-transform that converts SVG diagrams to EPS3 through ImageMagick. The generated TeX references `.eps` images, while HTML continues to use the original SVG files.

The four current architecture diagrams covered by this path are `backend_dispatch`, `local_zone_solve`, `controller_radial_flow`, and `publication_ownership`. PDF compilation remains an optional host-toolchain operation and requires ImageMagick plus the LaTeX/latexmk/epstopdf tooling.

The failed Read the Docs build 34879414 showed that the repository configuration left `python.install` commented out, so RTD installed only bare Sphinx 9.1.0 and then failed while importing `myst_parser`. The current configuration adds `docs/requirements.txt` and makes `.readthedocs.yaml` install it explicitly. The hosted requirements also include `sphinx_rtd_theme`, NumPy, and Astropy so the configured theme and autodoc imports are available before Sphinx reads the source tree.

There are no science, ABI, production-kernel, or runtime behavior changes in this closure.

7.3 0.6.90.5 — DOCUMENTATION_BUILD_OUTPUTS

0.6.90.5 extends the accepted 0.6.90.4 documentation baseline with explicit Sphinx LaTeX and PDF build targets. It is a documentation/build-metadata milestone only; the accepted XSTAR science revision, science-critical source hashes, and public ABI values are unchanged.

7.3.1 Changes

- Added `make latex` under `docs/sphinx/` to generate `docs/_build/latex/xstar-tools.tex` with warnings treated as errors.
- Added `make latexpdf` to invoke Sphinx `latexpdf` make-mode and produce `docs/_build/latex/xstar-tools.pdf` when a host LaTeX toolchain is installed.
- Added equivalent `latex` and `latexpdf` commands to `docs/sphinx/make.bat`.
- Added explicit `latex_documents` metadata so the generated source file is predictably named `xstar-tools.tex` and the compiled manual `xstar-tools.pdf`.
- Added restrained LaTeX defaults for letter paper, 10-point text, and figure placement.
- Added a developer documentation page describing HTML, TeX, PDF, link-check, dependency, and output-location workflows.
- Kept `make release` as HTML plus link checking so ordinary documentation/release validation does not require a system TeX installation.

7.3.2 Scientific boundary

The frozen science revision remains 0.6.48.12.3.45.3.3.8. C API ABI 60487, production-zone ABI 6048110, fixed-state ABIs 60486/60488, and XSPEC-table ABI 1 are unchanged. No science-critical source file is modified by this milestone.

7.4 0.6.90.3 — REPOSITORY_HISTORY_AND_QUALIFICATION_CLEANUP

0.6.90.3 is a structural repository cleanup following the formally closed 0.6.90 conda campaign. It changes no XSTAR science, science-critical source bytes, ABI values, atomic-data policy, or production runtime behavior.

The milestone removes closed one-off qualification machinery and historical reports from active release trees, consolidates the scientific freeze into a compact current gate, simplifies packaging/release utilities, removes generated cache/build artifacts, and keeps only current/frozen reference assets needed by active regression checks. Historical detail remains available through version-control history and release tags.

7.4.1 Cleanup result

Relative to the accepted 0.6.90.2 source archive:

Area	0.6.90.2	0.6.90.3
root Markdown files	757	5
top-level test Python files	224	61
qualification/ files	258	19
tools/ files	638	22
tools/qualification/ files	615	4
references/ files	14	11
docs/ files	201	93
total active source-tree files	2452	565

The remaining `tests/` files are current behavioral/regression tests rather than closed runner/manifests replay tests. Historical fixtures are retained only where active tests or runtime reference paths still consume them.

7.4.2 Frozen scientific boundary

The exact active science-critical bytes are pinned in `qualification/parity_freeze_current_source_hashes.json`, captured from the accepted 0.6.90.2 predecessor. The compact parity gate additionally pins the frozen comparator/reference artifacts, public ABI values, compiler floating-point policy, science revision, and external `atdb.fits` contract.

Preserved contracts include:

- science revision 0.6.48.12.3.45.3.3.8;
- C API ABI 60487;
- production-zone ABI 6048110;
- fixed-state program/engine ABIs 60486 / 60488;
- XSPEC-table ABI 1;
- no `-ffast-math` / `-Ofast`, with `-ffp-contract=off` retained;
- `atdb.fits` external;
- normal build non-MPI, with MPI explicit.

7.4.3 Acceptance

The cleaned tree passes:

```
python tools/ci/check_tree_clean.py
python tools/ci/check_package_metadata.py
python tools/ci/check_python_contract.py
python tools/ci/check_text_policy.py
python tools/qualification/check_parity_freeze.py
python tools/qualification/check_source_concordance.py
python tools/release/check_release_candidate.py
```

The compact parity gate verifies all 51 pinned active science-critical source files byte-for-byte. The source-concordance gate validates all 18 current Fortran/Python/C++ correspondence entries.

In the available build environment, the dependency-light pytest pass completed with 258 passed, 5 skipped. Two retained test modules (six tests total) require `astropy`, which was not installed in that environment; an attempted dependency installation could not reach PyPI because the environment had no network/name resolution. Those tests remain in the suite and are exercised by normal CI, which installs project dependencies.

7.5 0.6.90.4 — DOCUMENTATION_COVERAGE_CLOSURE

0.6.90.4 is a documentation/productization-only closure on top of the cleaned 0.6.90.3 repository. It does not change the accepted science revision, public ABI numbers, or science-critical source bytes.

The closure:

- integrates the repository-owned Sphinx `_static` branding bundle (logo, favicon, CSS, and corrected xstar-tools sidebar JavaScript);
- closes all Sphinx warnings reported against 0.6.90.3, including autosummary import cycles, heading-level consistency, history-toctree inclusion, and duplicate API-toctree references;
- adds task-oriented Python documentation for stable execution, the XSTARAtomic database API, XSTAR FITS output readers, and Python XSTAR2XSPEC wrappers;
- adds a unified CLI reference and classifies all installed console scripts as primary, specialist, compatibility, or development/qualification interfaces;
- adds dedicated native references for `xstar-xspec-initable` and `xstar-xspec-table`;
- completes the option reference for `xstar-cpp`, `xstar-xspec`, and `xstar-xspec-mpi`;
- makes the Windows guide version-neutral and removes instructions that depended on retired milestone runners;
- removes active-documentation references to historical directories/checkers that are no longer shipped after 0.6.90.3.

Release documentation is expected to pass Sphinx HTML and linkcheck builds with warnings treated as errors when the documentation dependencies and network-required intersphinx/linkcheck resources are available.

Useful starting points:

- *Installation* — installation and native source-build prerequisites;
- *Command-line interfaces* — current command-line entry points and compatibility commands;
- *Atomic-data setup* — `atdb.fits` / `coheat.dat` discovery and setup;
- *Python guide* — Python execution, atomic-data, output, and XSPEC-table workflows;
- *C++ and native CLI reference* — complete native executable reference;
- *Qualification policy* — current compact qualification/freeze policy.

PYTHON MODULE INDEX

X

- `xstar_tools`, [39](#)
- `xstar_tools.backends`, [39](#)
- `xstar_tools.config`, [39](#)
- `xstar_tools.data`, [39](#)
- `xstar_tools.execution`, [39](#)
- `xstar_tools.result`, [39](#)

Symbols

__init__() (*xstar_tools.ExecutionMode* method), 29
 __init__() (*xstar_tools.XStarConfig* method), 32
 __init__() (*xstar_tools.XStarData* method), 34
 __init__() (*xstar_tools.XStarProducts* method), 35
 __init__() (*xstar_tools.XStarResult* method), 36
 __init__() (*xstar_tools.api.XSTARAtomic* method), 39
 __init__() (*xstar_tools.tables.XSpecTableProducts* method), 44

A

available() (*in module xstar_tools.backends*), 38

B

BackendMode (*in module xstar_tools*), 29
 build_xspec_tables() (*in module xstar_tools.tables*), 45
 build_xstinitable() (*in module xstar_tools.tables*), 46

D

describe() (*in module xstar_tools.backends*), 38

E

ExecutionMode (*class in xstar_tools*), 29

L

list_fits_hdus() (*in module xstar_tools.xstar_outputs*), 42

M

module
 xstar_tools, 39
 xstar_tools.backends, 39
 xstar_tools.config, 39
 xstar_tools.data, 39
 xstar_tools.execution, 39
 xstar_tools.result, 39

P

parse_ion() (*in module xstar_tools.api*), 41

R

read_fits_table() (*in module xstar_tools.xstar_outputs*), 42
 read_fits_table_hdus() (*in module xstar_tools.xstar_outputs*), 42
 read_xout_abundances() (*in module xstar_tools.xstar_outputs*), 42
 read_xout_lines() (*in module xstar_tools.xstar_outputs*), 43
 read_xout_parameters() (*in module xstar_tools.xstar_outputs*), 43
 read_xout_spectra() (*in module xstar_tools.xstar_outputs*), 43
 run_xstar() (*in module xstar_tools*), 38
 run_xstar2xspec() (*in module xstar_tools.tables*), 46

W

write_csv() (*in module xstar_tools.xstar_outputs*), 44
 write_xout_spectra_csv() (*in module xstar_tools.xstar_outputs*), 44

X

XSpecTableProducts (*class in xstar_tools.tables*), 44
 xstar_tools
 module, 39
 xstar_tools.backends
 module, 39
 xstar_tools.config
 module, 39
 xstar_tools.data
 module, 39
 xstar_tools.execution
 module, 39
 xstar_tools.result
 module, 39
 XSTARAtomic (*class in xstar_tools.api*), 39
 XStarConfig (*class in xstar_tools*), 32
 XStarData (*class in xstar_tools*), 34
 XStarProducts (*class in xstar_tools*), 35
 XStarResult (*class in xstar_tools*), 36