

Source: shell docs/programmers_guide_public.md @ 5ef71f1 — curated by scripts/curate_ccs_shell.py; edits land in the source first.

CCS Shell Programmer's Guide — Draft 0.1

CCS target compilers as **in-process library calls** — no fork, no exec, no temporary process — with one C implementation surface and four language bindings: **C/C++**, **Python**, **JavaScript**, **Rust**.

This guide accompanies `ccs_shell.zip` from `b3u.dev`. Every example is runnable from the `examples/` directory of that delivery, against the shipped library — nothing here requires tooling you don't have.

1. The one-implementation architecture

```
your program (C / C++ / Python / JS / Rust)
  |
  | operation-shaped calls: compile / compile_string /
  | decompile / raw_binary / help + settings
  ▼
include/ccs_shell.h — the stable C ABI
  |
  | (extern "C", 19 functions; Python, JS and Rust
  | each bind THIS – they add no compiler logic)
  ▼
lib/libccsshell.so
  |
  | the CCS compile engine; one shell object per
  | target, as many as you like in one process
  ▼
artifacts on disk (+ optionally the SCB image in memory, §6)
```

Key properties:

- **A library call is the same compile.** The artifacts a library compile writes are byte-identical to the CLI frontends delivered by `b3u.dev` on the same input.
- **Many targets, one process.** Construct shells for different targets side by side; each is independent.

- **No exception ever crosses the C ABI.** Every entry point catches and converts (a C++ exception unwinding into CPython or Node is undefined behaviour in practice).
- **Compile-side only.** The surface is SCB *production* and frontend operation (§10).

2. Targets

A **target** is one grammar's compiler: `json` compiles `.json` documents, `metta` compiles `.metta` programs. The shipped set in this delivery is `json`, `metta`.

Every surface can enumerate the linked targets, and constructing a shell for an unlinked target fails **loudly, with the linked set in the message** — never a silent fallback.

3. Operations, artifacts, and return codes

operation	CLI twin	writes
<code>compile(file)</code>	<code>ccsjson f</code>	<code>f.lis f.urt f.fgr f.bfgr f.log</code>
<code>compile_string(text, name)</code>	<i>(API only)</i>	the same set, named from <code>name</code>
<code>decompile(fgr)</code>	<code>-u f.fgr</code>	<code>f.fgr.ubr</code>
<code>raw_binary(fgr)</code>	<code>-b f.fgr</code>	<code>f.bfgr</code> (native byte order)
<code>help()</code>	<code>-h</code>	usage text on stdout

Artifact naming, the input-extension rule and `.log` placement follow the `file/name` argument exactly as the CLI frontends' do. `.fgr` is the portable decimal-text SCB form; `.bfgr` is the raw native-byte-order form (fast to load, not portable across endianness).

The return-code contract (identical in all four languages):

- `0` — success.
- `1` — **an outcome, not an error**: the input had diagnostics (see `<input>.lis / .log`), or the arguments were refused. $\triangle$ *Check it.* `sh.compile(f)` on its own line, result discarded, is the classic false green.
- `< 0` at the C ABI (`CCS_SHELL_EEXCEPT -1`, `CCS_SHELL_EARG -2`) — a *programming* error (thread-affinity violation, API misuse). The bindings do not return these: Python raises `CcsShellError`, JS throws `Error`, Rust returns `Err(CcsShellError)`.

compile_string: \n means end of line, and the text is treated *exactly* as the same bytes in a file (CRLF text keeps its \rs on Linux; normalize first if you have CRLF input). name is required and non-empty: it names every artifact and drives the extension rule.

Settings (apply to every subsequent operation on the shell): `set_positions(on)` enables source-position tracking in the `.bfgr` (the CLI's `-D 1`); `set_first_follow(on)` prints the grammar analysis report into the `.lis` (the CLI's `-L 1`).

4. The thread contract

A shell belongs to the thread that created it.

- Legal: one thread driving many shells; N threads each driving its own.
- Forbidden: calling operations from a thread that neither created the shell nor explicitly claimed it.
- The hand-off is explicit: the *receiving* thread calls `rebind_to_current_thread()` and ownership **moves** (the previous owner becomes a violator).

How a violation surfaces, per language:

language	violation appears as
C/C++	<code>CCS_SHELL_EEXCEPT + ccs_last_error()</code>
Python	<code>CcsShellError</code> raised (then <code>rebind_to_current_thread()</code> and retry works)
JS	Error thrown (sync-only API in v1)
Rust	△ does not compile: <code>Shell is !Send/!Sync</code> — the policy lives in the type system

Checks are on by default; `CPPCC_SHELL_THREAD_CHECK=0` disables them process-wide, `set_thread_checks(false)` per instance. Concurrent shells must use **distinct input paths** — outputs are named after inputs, so two compiles of the same path race on the filesystem.

5. Building against the delivery

The zip is self-contained for Linux x86-64:

```
lib/libccsshell.so    the engine (targets: json, metta)
include/ccs_shell.h   the C ABI header
```

python/	js/	rust/	the binding sources
examples/			one runnable demo per language + sample data
README.md			build + run, step by step

- **C / C++:** `gcc your.c -I include -L lib -lccshell -Wl,-rpath,'$ORIGIN'/../lib` (see `examples/c_demo.c`).
- **Python:** stdlib-only ctypes binding — no build at all. Point `CCS_SHELL_LIB` at `lib/libccshell.so` and import `ccs_shell`.
- **JavaScript:** a hand-compiled N-API addon — `bash js/build.sh` (needs `g++` and Node headers; no npm, no network).
- **Rust:** `cargo build` in `rust/` (links the shipped `.so`; no crates.io dependencies, builds offline).

The README walks each in order; every step is copy-pasteable.

6. The SCB byte-image bridge

A compile builds a Syntax-Controlled Binary in memory and writes it as `.fgr/.bfgr`. To hand the **just-compiled SCB** to the CCS SCB reader packages (delivered separately in your `b3u.dev` language bundles) *without the disk round trip*:

<code>set_keep_scb(True)</code>	retain each successful compile's image
<code>compile(...)</code>	(default OFF – opt in)
<code>scb_bytes()</code>	the image – BYTE-IDENTICAL to the <code>.bfgr</code> the same compile wrote; valid until the next operation on this shell

Every CCS SCB reader has a **from-bytes** entry that consumes it:

ecosystem	reader package	from-bytes entry
C++	<code>ccsscb</code>	<code>readBinaryRawFromBuffer</code>
Python	<code>ccs_scb</code>	<code>read_binary_raw_bytes(data)</code>
JS	<code>ccs_js</code>	<code>readBinaryRawBytes(buf)</code>
Rust	<code>ccs_scb_rs</code>	<code>read_binary_raw_bytes(&[u8], None)</code>

So *program text in* → *walked SCB out*, with no file consulted by the reader (§8.4 shows it end to end).

7. The APIs

7.1 C — the ABI directly

`include/ccs_shell.h` is C89-clean and directly usable from plain C (from `examples/c_demo.c`):

```
#include "ccs_shell.h"

ccs_shell* sh = ccs_shell_create("json", NULL);
if (!sh) { fprintf(stderr, "%s\n", ccs_last_error()); return 2; }
int rc = ccs_shell_compile(sh, "input.json");
if (rc < 0) fprintf(stderr, "%s\n", ccs_last_error());
ccs_shell_destroy(sh);

/* enumeration */
for (int i = 0; i < ccs_target_count(); ++i)
    printf("%s\n", ccs_target_name(i));
```

The 19 entry points: `abi_version`, `target_count/name`, `create/destroy`, `compile`, `compile_string`, `decompile`, `raw_binary`, `help`, `set_positions`, `set_first_follow`, `set_thread_checks`, `rebind_to_current_thread`, `set_keep_scb`, `scb_size`, `scb_copy`, `language`, `last_error`. Stability rule: functions are appended, signatures never change; assert `ccs_shell_abi_version() == CCS_SHELL_ABI` at load.

7.2 Python — `import ccs_shell`

`stdlib-only` (ctypes over `libccsshell.so`; set `CCS_SHELL_LIB`). From `examples/python_demo.py`:

```
from ccs_shell import Shell, targets, CcsShellError

targets()                                # ('json', 'metta')

with Shell("json") as sh:
```

```

    rc = sh.compile("input.json")      # 0 ok, 1 = diagnostics (RETURNED)
    if rc != 0:
        ...                            # see input.json.lis / .log

# from a string – '\n' ends a line; name drives the artifacts
with Shell("json") as sh:
    rc = sh.compile_string(text, "input.json")

# settings
with Shell("json") as sh:
    sh.set_positions(True)
    sh.set_first_follow(True)
    sh.compile("input.json")

```

Errors: unknown target → `ValueError` naming targets(); ABI mismatch → `ImportError` at load; use-after-close() → `RuntimeError`; affinity violation / misuse → `CcsShellError`. The hand-off:

```

sh = Shell("json")                # owned by this thread
def worker():
    try:
        sh.compile("input.json")    # raises CcsShellError
    except CcsShellError:
        sh.rebind_to_current_thread() # the explicit claim
        sh.compile("input.json")     # now 0
threading.Thread(target=worker).start()

```

7.3 JavaScript

A hand-compiled N-API addon (no node-gyp, no npm dependencies; build with `bash js/build.sh`). Sync-only in v1 — a compile blocks the event loop. From `examples/js_demo.mjs`:

```

const { Shell, targets } = require("ccs-shell-js");

targets();                                // ['json', 'metta']

const sh = new Shell("json");
const rc = sh.compile("input.json"); // 0 ok, 1 = diagnostics (RETURNED)
sh.compileString(text, "input.json"); // the string arm
sh.decompile("input.json.fgr");

```

```
sh.rawBinary("input.json.fgr");
sh.close();                                // idempotent; finalizer covers
                                           // an unclosed shell
```

Errors: unknown target / rc < 0 → throw new Error(...) carrying ccs_last_error(); use-after-close throws.

7.4 Rust — ccs-shell (+ ccs-shell-sys)

Links the shipped .so; zero crates.io dependencies, builds offline. From rust/examples/demo.rs:

```
use ccs_shell::{Shell, targets};

targets();                                // ["json","metta"]

let sh = Shell::new("json", None)?;
match sh.compile("input.json")? {         // Result<i32, CcsShellError>
    0 => { /* ok */ }
    _ => { /* diagnostics - input.json.lis */ }
}
sh.compile_string(&text, "input.json")?;   // the string arm
sh.decompile("input.json.fgr")?;
sh.raw_binary("input.json.fgr")?;
// drop(sh) frees the native handle
```

Ok(1) is an outcome; Err(_) only where the ABI reports rc < 0. △ Shell is !Send/!Sync: moving it to another thread **does not compile** — the affinity policy enforced by the type system. rebind_to_current_thread exists for unsafe movers.

8. Worked patterns

All shapes below are in examples/, per language, against the shipped sample inputs.

8.1 Compile → decompile as function calls (Python shown)

```
with Shell("json") as sh:
    assert sh.compile("sample.json") == 0          # -> .urt .fgr .bfgr
```

```
assert sh.decompile("sample.json.fgr") == 0    # -> .fgr.ubr
```

8.2 raw_binary and idempotency

```
saved = read("sample.json.bfgr")
assert sh.raw_binary("sample.json.fgr") == 0    # repopulates .bfgr
assert read("sample.json.bfgr") == saved        # byte-identical
# and compiling the same source twice is byte-identical (.fgr + .bfgr)
```

8.3 The same compile from a string

```
text = open("sample.json").read()
with Shell("json") as sh:
    assert sh.compile_string(text, "sample.json") == 0
# artifact set byte-identical to the file compile's
```

8.4 Compile → in-memory SCB → walk (the bridge, Rust shown)

```
let sh = Shell::new("json", None)?;
sh.set_keep_scb(true)?;
sh.compile_string(&doc_text, "sample.json"?;
let img = sh.scb_bytes()?.expect("image held");

let mut scb = ccs_scb_rs::SyntaxControlledBinary::new();
scb.read_binary_raw_bytes(&img, None)?;    // no disk consulted
for i in 0..scb.head().cntnkw {            // walk the kwn records
    let kwn = scb.ptrkwn(i).unwrap();
    // ... kwn.k, children, ...
}
```

Python twin:

```
sh = Shell("json"); sh.set_keep_scb(True)
sh.compile_string(text, "sample.json")
img = sh.scb_bytes()
scb = SyntaxControlledBinary().read_binary_raw_bytes(img)
walk = [scb.ptrkwn(i).k for i in range(scb.head().cntnkw)]
```

(The `ccs_scb_rs` / `ccs_scb` reader packages come with your `b3u.dev` language bundles.)

9. The examples

file	shows
examples/c_demo.c	the C ABI end to end: create, compile, enumerate, error text
examples/python_demo.py	compile file + string, return-code check, the SCB bridge
examples/js_demo.mjs	the addon: compile, decompile, raw_binary
rust/examples/demo.rs	the typed surface, !Send affinity, the bridge

Each prints what it did and exits 0; the README runs them all.

10. The product boundary

This delivery exposes **SCB consumption and frontend operation** — compile, decompile, raw-binary, and the byte-image hand-off, for the shipped targets. Compiling a *grammar* into a new target compiler is a b3u.dev service operation: upload your .sgr on the site and the delivery you download is that grammar's frontend.

11. License and support

Development and evaluation use is covered by your b3u.dev subscription (Terms of Service §5.3). PRODUCTION deployment of components produced with this library requires a separate written agreement — contact support@b3u.dev BEFORE deploying (Terms §5.4).

Platform: Linux x86-64. Questions, platform issues, feedback: support@b3u.dev.