MCS Engine: SDK API Reference

Contents

Osinix (osinix.com)

This is the function-level reference for the MCS SDK. For installation, linking, and a first working program, see the SDK Getting Started guide. For the deliberation concepts and the session config format, see the User Guide.

Each entry below gives the C function and its Python equivalent side by side, with arguments, return values, and the errors it can produce.

Conventions

Return values (C). Every C function returns `MCS_OK` (`0`) on success and a negative `MCS_ERR_*` code on failure. Functions that return a pointer return `NULL` on failure. Functions that return a count or a byte length return a non-negative number on success.

Error reporting (C). After any failed call, `mcs_errno()` holds the numeric code and `mcs_error()` holds a human-readable message. A successful call does not clear them, so check each call's own return value rather than polling the globals.

Errors (Python). The Python SDK raises exceptions instead of returning codes: `MCSError` (base), and `MCSLicenseError`, `MCSTimeoutError`, `MCSConfigError`.

The session handle. `mcs_session_start` returns an opaque handle. You own it from that moment and must call `mcs_session_free` to release it, whatever the outcome. In Python the `Session` object owns the handle and frees it on context exit.

One session per process. Only one session runs at a time, enforced by a lockfile. A second concurrent start returns `MCS_ERR_SESSION_ALREADY_RUNNING`.

Thread safety. The error-state functions (`mcs_errno`, `mcs_error`) use process-global state and are not thread-safe. Treat one session and its handle as owned by one thread.

Library lifecycle

Initialize the library

```
int mcs_library_init(const char *engine_path, const char *license_path);
```

```
mcs.init(license_path=None, engine_path=None)   # raises on failure
```

Loads and verifies the license, caches the licensed tier, and prepares internal state. Call it once at program start. (If you never call it, the first `mcs_session_start` initializes lazily.)

- `engine_path`: where to find the engine binary. `NULL` searches `PATH` for the tier-matching name (`mcs_basic` / `mcs_pro` / `mcs_pro_max`); a directory is searched for that binary; a file path is exec'd directly.
- `license_path`: the `license.key` to use. `NULL` uses the standard search order (the `license_path` argument, then `$MCS_LICENSE_KEY`, then the default config location).

Repeat calls are a smart merge: the license from the first successful call is cached for the process and not reloaded; a later call may add an `engine_path`, or pass `NULL` to keep what it has, but a later call with a different `license_path` is rejected with `MCS_ERR_INVALID_PARAMETER`.

Returns `MCS_OK`, or `MCS_ERR_LICENSE_INVALID`, `MCS_ERR_LICENSE_EXPIRED`, `MCS_ERR_INVALID_PARAMETER`, `MCS_ERR_INTERNAL`. (Python raises `MCSLicenseError` for the license cases.)

Get the SDK version

```
const char *mcs_library_version(void);
```

```
mcs.version()    # -> str
```

Returns the version string ("MAJOR.MINOR.PATCH"). Safe to call before init. The string is owned by the library in C; never free it.

License and locations

License information

```
int mcs_license_info(mcs_license_info_t *out);
```

```
mcs.license_info()  # -> LicenseInfo(licensee, tier, expires, days_remaining, has_sdk)
```

Fills `out` from the license cached at init (no file I/O); unlike `locations()`, no running session is required. In Python, `has_sdk` is returned as a `bool`.

```
typedef struct {
    char licensee[256];   /* person or company on the license */
    char tier[32];        /* "Basic" / "Pro" / "Pro Max" / "Enterprise" */
    char expires[16];     /* "YYYY-MM-DD" */
    int  days_remaining;  /* negative if already expired */
    int  has_sdk;         /* 1 if this license grants SDK access */
} mcs_license_info_t;
```

Returns `MCS_OK`, `MCS_ERR_NOT_INITIALIZED`, or `MCS_ERR_INVALID_PARAMETER` (`out` is `NULL`).

Resolved file locations

```
int mcs_locations(mcs_locations_t *out);
```

```
mcs.locations()    # -> Locations(install_dir, config_path, sessions_dir, temp_dir)
```

Fills the engine's resolved paths: `install_dir`, `config_path`, `sessions_dir`, `temp_dir` (each a NUL-terminated string in C). These are reported by the engine, so `config_path` is empty until a config has been loaded. Returns `MCS_OK`, or `MCS_ERR_NOT_INITIALIZED` / `MCS_ERR_LOCATIONS_UNAVAILABLE` / `MCS_ERR_NO_ACTIVE_SESSION` when the paths are not yet available.

Session lifecycle

Start a session

```
mcs_session_handle_t *mcs_session_start(const char *mcs_conf_path,
                                        const char *session_conf_path);
```

```
mcs.start(mcs_conf, session_conf)    # -> Session
```

Forks and execs the tier-matching engine as a subprocess with your `mcs.conf` and session config, then returns immediately with a handle. The license and tier limits are checked before the subprocess is spawned, so an over-tier config fails here rather than mid-run.

- `mcs_conf_path`: path to `mcs.conf`. Must exist and be readable.
- `session_conf_path`: path to the session config (panel, agenda, mode).

Returns a non-`NULL` handle on success, or `NULL` with `mcs_errno` set to one of: `MCS_ERR_NOT_INITIALIZED`, `MCS_ERR_LICENSE_INVALID`, `MCS_ERR_LICENSE_EXPIRED`, `MCS_ERR_FEATURE_NOT_LICENSED`, `MCS_ERR_ENGINE_BINARY_NOT_FOUND`, `MCS_ERR_CONFIG_NOT_FOUND`, `MCS_ERR_CONFIG_INVALID`, `MCS_ERR_TIER_LIMIT_EXCEEDED`, `MCS_ERR_SESSION_ALREADY_RUNNING`, `MCS_ERR_SUBPROCESS_SPAWN_FAILED`, `MCS_ERR_INVALID_PARAMETER`, `MCS_ERR_INTERNAL`. (Python returns a `Session` or raises the matching exception.)

Wait for natural completion

```
int mcs_session_wait(mcs_session_handle_t *handle, int timeout_sec);
```

```
session.wait(timeout=None)
```

Blocks until the session ends on its own (moderator ended it, the limit was reached, or an engine error). It does not signal the engine. Use `0` (C) or `None` (Python) to wait indefinitely; a positive timeout returns `MCS_ERR_TIMEOUT` (C) or raises `MCSTimeoutError` (Python) if the session is still running when it elapses, leaving the session alive to wait on again or stop.

Returns `MCS_OK`, `MCS_ERR_TIMEOUT`, `MCS_ERR_HANDLE_INVALID`, `MCS_ERR_INVALID_PARAMETER` (negative timeout), or `MCS_ERR_INTERNAL`.

Stop a session

```
int mcs_session_stop(mcs_session_handle_t *handle, int mode);
```

```
session.stop(force=False)
```

Ends a running session.

- `MCS_STOP_GRACEFUL` (C) / `force=False` (Python): sends an interrupt; the engine finishes the current turn, writes its final transcript, summary, and status, then exits. Blocks until it exits or a 30-second grace cap, returning `MCS_ERR_TIMEOUT` if the engine has not exited (escalate to force).
- `MCS_STOP_FORCE` (C) / `force=True` (Python): kills the engine immediately; final output may be missing. Use for emergencies or after a graceful timeout.

Idempotent: stopping a finished session is a no-op that returns `MCS_OK`. Other returns: `MCS_ERR_HANDLE_INVALID`, `MCS_ERR_INVALID_PARAMETER` (unknown mode), `MCS_ERR_IO`.

Free the handle

```
void mcs_session_free(mcs_session_handle_t *handle);
```

```
session.free()      # usually automatic via the context manager
```

Releases the handle and its resources. If the session is still running it is stopped and reaped first, so the call may block briefly while the engine flushes. After this the handle is invalid. `mcs_session_free(NULL)` is a safe no-op. In Python, leaving a `with mcs.start(...) as session:` block frees it automatically.

Session monitoring

All monitoring calls read the engine's per-turn `status.json` in the session directory. Very early in a session that file may not exist yet, in which case the C calls return `MCS_ERR_STATUS_FILE_MISSING`; retry briefly or call `mcs_session_wait(handle, 1)` to give the engine time to write it.

Is the session done?

```
int mcs_session_is_done(mcs_session_handle_t *handle);
```

```
session.is_done()    # -> bool
```

Returns `1` if the session has reached a terminal phase (completed, stopped, or error), `0` if still running; or `MCS_ERR_HANDLE_INVALID`, `MCS_ERR_STATUS_FILE_MISSING`, `MCS_ERR_IO`.

Turn and round numbers

```
int mcs_session_get_turn(mcs_session_handle_t *handle, int which);
int mcs_session_get_round(mcs_session_handle_t *handle, int which);
/* convenience inlines: mcs_session_get_current_turn(h),
   mcs_session_get_last_completed_turn(h), and the round equivalents */
```

```
session.current_turn()           # -> int
session.last_completed_turn()    # -> int
session.current_round()          # -> int
session.last_completed_round()   # -> int
```

`which` is `MCS_CURRENT` (the turn/round being processed now) or `MCS_LAST_COMPLETED` (the most recent finished one, whose output file exists on disk). Returns the number (`>= 0`), or `MCS_ERR_HANDLE_INVALID`, `MCS_ERR_INVALID_PARAMETER`, `MCS_ERR_STATUS_FILE_MISSING`, `MCS_ERR_IO`.

Phase and status strings

```
int mcs_session_get_phase(mcs_session_handle_t *handle, char *out, size_t out_sz);
int mcs_session_get_status(mcs_session_handle_t *handle, char *out, size_t out_sz);
```

```
session.phase()     # -> str
session.status()    # -> str
```

`phase` is one of `deliberation`, `completed`, `aborted` (a 32-byte buffer is safe). `status` is one of `running`, `completed`, `stopped`, `error` (a 16-byte buffer is safe). The C calls copy the string into your buffer and return the byte count written (excluding NUL), or `MCS_ERR_HANDLE_INVALID`, `MCS_ERR_INVALID_PARAMETER` (NULL/zero/too-small buffer), `MCS_ERR_STATUS_FILE_MISSING`, `MCS_ERR_IO`.

Files

The session keeps files in three categories. The C API exposes basenames; build an absolute path by joining the session directory, the category subfolder, and the basename. Python returns ready-to-use paths.

- `MCS_FILES_INPUT` (`input/`): files you supplied before the session began.
- `MCS_FILES_OUTPUT` (`output/`): engine outputs (transcript, summary, metadata, per-turn files).
- `MCS_FILES_MEDIA` (`media/`): files models produced and exchanged during the session.

Session directory

```
int mcs_session_get_session_dir(mcs_session_handle_t *handle, char *out, size_t out_sz);
```

```
session.session_dir()    # -> str
```

Copies the absolute session directory path. Returns bytes written, or `MCS_ERR_HANDLE_INVALID`, `MCS_ERR_INVALID_PARAMETER`, `MCS_ERR_STATUS_FILE_MISSING`, `MCS_ERR_IO`.

Listing files

```
int mcs_session_get_file_count(mcs_session_handle_t *handle, int category);
int mcs_session_get_file(mcs_session_handle_t *handle, int category,
                         int index, char *out, size_t out_sz);
/* convenience inlines exist per category, e.g.
   mcs_session_get_output_file_count / mcs_session_get_output_file */
```

```
session.input_files()     # -> list[str]
session.output_files()    # -> list[str]
session.media_files()     # -> list[str]
```

`get_file_count` returns the number of files in a category; `get_file` copies the basename of the `index`-th file. Returns `MCS_ERR_INVALID_PARAMETER` for an unrecognized category or out-of-range index, plus the usual `MCS_ERR_HANDLE_INVALID`, `MCS_ERR_STATUS_FILE_MISSING`, `MCS_ERR_IO`. In C, join `session_dir + "/" + category-folder + "/" + basename` for the full path.

Building a session config from JSON

```
int mcs_session_config_save_conf(const char *json, const char *out_path);
int mcs_session_config_save_json(const char *json, const char *out_path);
```

```
mcs.save_session_config_as_conf(json_str, out_path)
mcs.save_session_config_as_json(json_str, out_path)
```

Serialize a JSON session description (the same shape the web UI uses) to a `.conf` or `.json` file you can then pass to `start`. The `.conf` form is preferred for human editing. Returns `MCS_OK`, or `MCS_ERR_INVALID_PARAMETER` (NULL arguments), `MCS_ERR_CONFIG_INVALID` (unparseable JSON or missing required fields), or `MCS_ERR_IO` (could not write). You can also author the config file by hand in the format documented in the User Guide and skip these entirely.

Error state (C)

```
int mcs_errno(void);
const char *mcs_error(void);
```

`mcs_errno` returns the most recent error code (`MCS_OK` if none since init). `mcs_error` returns a human-readable description of the last error, owned by the library. Both are overwritten only by later failed calls, not by successful ones. The Python SDK surfaces this through exception types and messages instead.

Constants

All constants are plain integers. In C they carry the `MCS_` prefix and are defined in `libmcs.h`; in Python they are attributes on the `mcs` module with the prefix dropped (for example `MCS_CURRENT` is `mcs.CURRENT`). Each group below is the set of allowed values for one function argument.

Turn / round selectors

```
MCS_CURRENT          0     /* the in-progress turn or round */
MCS_LAST_COMPLETED   1     /* the most recently finished turn or round */
```

Passed as the `which` argument to `mcs_session_get_turn()` and `mcs_session_get_round()` to say which number you want: the one being processed right now (`MCS_CURRENT`), or the most recent one whose output file is already on disk (`MCS_LAST_COMPLETED`). During an active session the current value is one ahead of the last completed; after the session ends they are equal. In Python, use the named methods instead: `session.current_turn()`, `session.last_completed_turn()`, and the round equivalents.

For example:

```
int cur  = mcs_session_get_turn(s, MCS_CURRENT);
int done = mcs_session_get_turn(s, MCS_LAST_COMPLETED);
```

```
cur  = session.current_turn()
done = session.last_completed_turn()
```

File categories

```
MCS_FILES_INPUT      0     /* <session_dir>/input/  */
MCS_FILES_OUTPUT     1     /* <session_dir>/output/ */
MCS_FILES_MEDIA      2     /* <session_dir>/media/  */
```

Passed as the `category` argument to `mcs_session_get_file_count()` and `mcs_session_get_file()` to choose which set of files to list: the inputs you supplied before the session, the engine's structured outputs (transcript, summary, metadata, per-turn files), or the media files the models produced and exchanged. Each category maps to the matching subdirectory of the session directory. In Python, use `session.input_files()`, `session.output_files()`, and `session.media_files()`.

For example, to list the output files:

```
char name[256];
int n = mcs_session_get_file_count(s, MCS_FILES_OUTPUT);
for (int i = 0; i < n; i++)
    if (mcs_session_get_file(s, MCS_FILES_OUTPUT, i, name, sizeof name) >= 0)
        printf("%s\n", name);
```

```
for name in session.output_files():
    print(name)
```

Stop modes

```
MCS_STOP_GRACEFUL    0     /* interrupt; let the engine flush, 30s grace */
MCS_STOP_FORCE       1     /* kill immediately */
```

Passed as the `mode` argument to `mcs_session_stop()`. `MCS_STOP_GRACEFUL` lets the engine finish the current turn and write its final transcript, summary, and status before exiting (within a grace window; see `mcs_session_stop` above). `MCS_STOP_FORCE` kills the engine immediately, so the final output files may be missing. In Python, use `session.stop(force=False)` (graceful, the default) or `session.stop(force=True)`.

For example, stop gracefully and escalate to force if it does not exit in time:

```
if (mcs_session_stop(s, MCS_STOP_GRACEFUL) == MCS_ERR_TIMEOUT)
    mcs_session_stop(s, MCS_STOP_FORCE);
```

```
try:
    session.stop()              # graceful (force=False)
except mcs.MCSTimeoutError:
    session.stop(force=True)
```

Error codes

`MCS_OK` is `0`; all errors are negative. The Value column is the exact integer a C function returns (and what the matching `mcs.ERR_*` constant equals in Python), so a raw return value like `-10` can be mapped back to its meaning.

Code	Value	Meaning
`MCS_OK`	`0`	Success
`MCS_ERR_NOT_INITIALIZED`	`-1`	`mcs_library_init` has not succeeded
`MCS_ERR_LICENSE_INVALID`	`-2`	License missing, malformed, or bad signature
`MCS_ERR_LICENSE_EXPIRED`	`-3`	License valid but past expiry
`MCS_ERR_FEATURE_NOT_LICENSED`	`-18`	Valid license, but no SDK access
`MCS_ERR_ENGINE_BINARY_NOT_FOUND`	`-4`	No matching engine binary found
`MCS_ERR_CONFIG_NOT_FOUND`	`-5`	A config file path does not exist
`MCS_ERR_CONFIG_INVALID`	`-6`	A config file is malformed
`MCS_ERR_TIER_LIMIT_EXCEEDED`	`-16`	Config exceeds the tier's limits
`MCS_ERR_FILE_FORMAT_NOT_ALLOWED`	`-19`	A file type the tier cannot use
`MCS_ERR_SESSION_ALREADY_RUNNING`	`-7`	Another session is active
`MCS_ERR_SESSION_ALREADY_DONE`	`-17`	Operation needs a running session
`MCS_ERR_NO_ACTIVE_SESSION`	`-20`	No session to read from
`MCS_ERR_SUBPROCESS_SPAWN_FAILED`	`-8`	The engine process failed to start
`MCS_ERR_HANDLE_INVALID`	`-9`	NULL or invalid handle
`MCS_ERR_TIMEOUT`	`-10`	Wait or graceful-stop time elapsed
`MCS_ERR_STATUS_FILE_MISSING`	`-11`	`status.json` not written yet
`MCS_ERR_SESSION_ABORTED`	`-12`	Session ended abnormally
`MCS_ERR_LOCATIONS_UNAVAILABLE`	`-21`	Engine paths not yet resolved
`MCS_ERR_INVALID_PARAMETER`	`-14`	A bad argument was passed
`MCS_ERR_IO`	`-13`	An I/O operation failed
`MCS_ERR_INTERNAL`	`-15`	Unexpected internal error

In Python the same codes are available as `mcs.ERR_*` (again with the `MCS_` prefix dropped, so `MCS_ERR_TIMEOUT` is `mcs.ERR_TIMEOUT`) and appear as the integer `.code` on a raised `MCSError`.

Support

- **Developer and integration help, and bug reports:** support@osinix.com
- **Licensing, SDK access, and add-ons:** licensing@osinix.com
- **Documentation:** https://osinix.com/docs
