

Concept for the **svah** project

v86

Peter Felecan
assisted by Claude (Anthropic), model Sonnet 5
September 5, 2026

Contents

1	Introduction	2
2	Motivation	2
3	Project Name	2
4	Scope	3
4.1	What is ported	3
4.2	What is <i>not</i> ported	3
5	Architecture	3
6	Target Platforms	4
7	Implementation Language and Standard	4
8	Build System	4
9	Deliverables	4
10	Further Reading	5

List of Figures

List of Tables

1 Introduction

`svah` (समुद्रः वह्— Sanskrit: *the sea carries*) is a C library and companion program for controlling ODrive BLDC motor controllers via the CANSimple protocol.

The project ports the host-side Python library distributed with the ODrive firmware (`tools/odrive`) to pure C, enabling its use on bare-metal microcontrollers, embedded Linux systems, and PC host platforms from a single, portable codebase.

2 Motivation

The official ODrive host library is written in Python and depends on `python-can` and an IPython shell (`odrivetool`). This is suitable for desktop configuration and development but excludes:

Bare-metal MCU integration Python is not available on MCUs; embedding motor control logic in a Python script running on an attached PC introduces latency and a single point of failure.

Real-time determinism Python’s garbage collector and interpreter overhead are incompatible with hard real-time constraints required for BLDC torque and velocity control loops.

Toolchain uniformity Embedded projects typically build with `arm-none-eabi-gcc` or similar. A C library integrates naturally into these toolchains without FFI bridges.

These three goals are met by the project’s artifacts directly: the portable `libsvah` core (`OdDriveApi` and `CanOpenSimple`, see Architecture below) links into any C toolchain with no Python runtime and no garbage collector on the control path, satisfying bare-metal integration and real-time determinism; the platform-specific HAL layer isolates the one part that genuinely differs per target, satisfying toolchain uniformity everywhere else. `svah` demonstrates the API end-to-end against real hardware; `svahsim` makes the same verification possible without hardware attached. What each artifact is and how to use it is detailed in LibraryApi.org, SimScriptLanguage.org, and TestPlan.org — see *Further Reading* at the end of this document.

3 Project Name

The Sanskrit phrase समुद्रः वह् evokes the image of a medium that carries — the library carries control commands across the CAN bus to the motor controller, as the sea carries ships. The name is also pleasantly short for use as a command name and library prefix.

4 Scope

4.1 What is ported

- Encoding and decoding of all CANSimple protocol frames
- Axis state management (`AXIS_STATE_*`) and enumeration
- High-level control commands: position, velocity, torque setpoints
- Error readback and error clearing
- Watchdog feed support

4.2 What is *not* ported

odrivetool interactive shell (IPython) development and calibration tool only; remains in Python.

Live matplotlib plotter visualisation tool only.

DFU / firmware update Python is sufficient and appropriate for this infrequent operation.

5 Architecture

The library is structured in three horizontal layers:

Application / svah program		
OdriveApi – high-level ODrive commands		
CanopenSimple – CANSimple frame codec (portable, platform-independent)		
HalCanLinux	HalCanEsp32	HalCanPeak
HalTimeLin.	HalTimeEsp32	HalTimePeak
(SocketCAN)	(TWAI)	(USB-CAN)
IMPLEMENTED	PLANNED	PLANNED

The horizontal split between the portable layers and the HAL implementations is enforced by two abstract headers:

HalCan.h abstract CAN frame transmit/receive interface, with hybrid polling and callback receive model.

HalTime.h abstract monotonic time source for timeout and watchdog management.

6 Target Platforms

Platform	Toolchain	CAN interface	Status
Linux (RPi, PC)	system gcc	SocketCAN	implemented
ESP32	xtensa-esp32-elf-gcc	TWAI (native)	planned
SAMD51 / SAM3X	arm-none-eabi-gcc	CAN native / MCP2515	planned
PC Windows/Mac	MinGW / clang	PEAK, gs_usb	planned

AVR (classic Arduino Uno/Mega) is explicitly out of scope: the absence of a native CAN controller and severe RAM constraints (2–32 KB) make it an unsuitable target for this library.

7 Implementation Language and Standard

Pure C, compiled at the **C11** standard (`-std=c11`) with the GNU toolchain. C11 provides `_Static_assert` for compile-time HAL contract verification, `<stdatomic.h>` for interrupt-safe state flags on MCU targets, and anonymous structs/unions useful for CAN frame payload layout.

8 Build System

The project uses the GNU autotools suite (`autoconf` / `automake` / GNU Make). This provides:

- Platform detection via `configure`
- Toolchain selection for cross-compilation (`--host` flag)
- Standard `make dist` delivery archive generation
- Consistency with the FSF open-source ecosystem

9 Deliverables

libsvah.a / **libsvah.so** static and shared library (built by default via `libtool`; the primary deliverable other software links against)

svah command-line test and demonstration program; exercises the full API against one ODrive axis

svahsim standalone ODrive protocol simulator, for testing without physical hardware — see SimScriptLanguage.org

svah.1, **svahsim.1** manual pages

10 Further Reading

This document covers the project's goals and architecture. For the rest:

LibraryApi.org the complete `libsvah` public API — every function, type, and usage example, organised by the same `OdriveApi` / `CanopenSimple` / `HalCan` / `HalTime` layering introduced above.

SimScriptLanguage.org the `.svs` scripting language for `svahsim`, for testing against a simulated ODrive without hardware.

TestPlan.org how the library and programs are verified, from unit tests through physical integration tests on real CAN hardware.

Index.org (the project's HTML documentation index) links to all of the above, plus the project's internal process documents.