EmbedSim Framework

Architecture & Design Document

Block-Diagram Simulation for 32-bit Embedded Platforms

Version 2.0 · EmbedSim Framework

1. Purpose and Design Goals

EmbedSim is a Python-based block-diagram simulation framework built for one specific mission: let a control engineer write logic once in Python, verify it in simulation, then deploy the same algorithm as compiled C on a 32-bit microcontroller (Aurix TriCore, Cortex-M4) — with no manual rewrites, no numerical inconsistencies, and no surprise.

Why not Simulink? Simulink costs tens of thousands of dollars per seat and its code generator is a separate product. EmbedSim is open-source and targets engineers and research groups who need the same workflow at zero licence cost.

Design Constraints That Shaped the Architecture

- **float32 by default:** Embedded MCUs operate natively in 32-bit single precision. Using float64 wastes bandwidth on the CAN bus and slows DSP pipelines. The entire framework defaults to float32; float64 is available per-block when needed (e.g. FMU co-simulation).
- **Single-backend switch flag:** Every block supports use_c_backend=True/False. Switching one flag at runtime redirects compute() to either a pure-Python path or a Cython-compiled C path — no architectural changes to the diagram.
- **Feedback loops are first-class:** Real control systems always have feedback. A framework that cannot handle cycles is useless for its target domain. The engine's two-pass DFS and LoopBreaker interface make feedback as easy to express as any feedforward chain.
- **FMU interoperability:** Plant models from Modelica, Dymola, or OpenModelica are delivered as FMU files. FMUBlock wraps the FMI 2.0 Co-Simulation protocol as a first-class VectorBlock so any Modelica model snaps into the diagram unchanged.
- **Scriptable control logic:** A ScriptBlock lets engineers prototype any algorithm in Python inline, then auto-generate equivalent C from the same AST — the single source of truth for both simulation and deployment.

2. Package Structure

The package is installed as embedsim/ and imports everything from a flat namespace via __init__.py. This lets user code write from embedsim import EmbedSim, VectorGain without knowing which module a class lives in.

Module	Key Classes	Responsibility
core_blocks.py	`VectorBlock, VectorSignal`	Abstract base class. Dual-backend dispatch (Python/C). >> wiring operator. float32 default precision.
source_blocks.py	`VectorConstant, VectorStep, ThreePhaseGenerator, VectorRamp`	Signal generators. No inputs; produce signals from mathematical functions or fixed values.
processing_blocks.py	`VectorGain, VectorSum, VectorDelay, VectorSaturation`	Stateless combinational blocks. One or more inputs, one output per timestep.

dynamic_blocks.py	`VectorIntegrator, StateSpaceBlock, TransferFunctionBlock, VectorEnd`	Blocks with continuous internal state. Expose get_derivative() for ODE solvers.
simulation_engine.py	`EmbedSim, LoopBreaker, VectorDelay(LB), VectorScope, ODESolver`	Orchestrates the full simulation. Graph traversal, ODE stepping, scope recording, topology printing.
script_blocks.py	`ScriptBlock`	User-written Python scripts with AST-based validation and C code generation.
code_generator.py	`CodeGenStart, CodeGenEnd, SimBlockBase, MCUTarget`	Boundary markers and generators. Produces .h header, .pyx Cython wrapper, _simblock.py stub.
fmu_blocks.py	`FMUBlock`	Adapter for FMI 2.0 Co-Simulation FMUs (Modelica plants, etc.).
__init__.py		Flat re-export of all public symbols. Silently skips fmu_blocks if fmpy is not installed.

3. Core Abstractions

3.1 VectorSignal — the data unit

Everything that flows between blocks is a VectorSignal: a named NumPy array of dtype float32 (or float64 per block). It has no methods beyond construction and __repr__. Its simplicity is intentional — signal objects are created, passed, read, and discarded at every timestep. Keeping them as thin wrappers avoids allocation overhead in the hot simulation loop.

```
sig = VectorSignal([1.0, 2.0, 3.0], name='current')   # float32 by default
sig = VectorSignal([1.0, 2.0],       dtype=np.float64) # float64 override
```

3.2 VectorBlock — the computation unit

VectorBlock is the abstract base class every block inherits from. It enforces three contracts:

- **Wiring contract:** The >> operator appends self to other.inputs. This builds an explicit input list on every block, forming the block graph. No graph object is ever created separately — the graph is encoded implicitly in the inputs attributes.
- **Backend contract:** compute() dispatches to compute_py() or compute_c() based on the use_c_backend flag. Subclasses override compute_py() for their Python logic and optionally compute_c() for their compiled path. compute() itself is never overridden.
- **Integration contract:** Dynamic blocks expose get_derivative(t, inputs) and integrate_state(dt, solver). The engine calls these during the ODE step; static blocks inherit no-op defaults.

```
# Minimal concrete block
class ScalarGain(VectorBlock):
    def __init__(self, name, gain, **kw):
        super().__init__(name, **kw)
        self.gain = gain

    def compute_py(self, t, dt, input_values=None):
        y = input_values[0].value * self.gain
        self.output = VectorSignal(y, self.name, dtype=self.dtype)
        return self.output
```

4. Python / C Dual-Backend Execution

This is EmbedSim's central design feature. The same block runs in either Python (for development) or compiled C (for production) without changing the block diagram. There are two separate mechanisms for this: the block-level backend switch and the region-level code generation pipeline.

4.1 Block-level backend switch

Every VectorBlock carries a use_c_backend flag set at construction. The compute() dispatcher checks this flag on every call:

```
def compute(self, t, dt, input_values=None):
    if self.use_c_backend:
        return self.compute_c(t, dt, input_values)   # Cython path
    return self.compute_py(t, dt, input_values)       # Python path
```

The C path calls into a compiled Cython wrapper that holds C structs on the stack and releases the GIL:

```
# Inside the generated Cython wrapper:
cpdef void compute(self):
    """Call C function — GIL released."""
    with nogil:
        my_block_compute(&self._in, &self._out)  # pure C, no Python objects
```

You can flip the backend at runtime:

```
block = MyPidBlock('pid', use_c_backend=False)  # start in Python
# ... verify behaviour ...
block.switch_backend(True)                        # flip to C
```

4.2 Region-level code generation pipeline (CodeGenStart / CodeGenEnd)

For a subsystem containing many blocks, marking the entry and exit with CodeGenStart and CodeGenEnd boundary blocks allows generating a complete C interface in one step.

Code generation workflow

1. Insert CodeGenStart and CodeGenEnd as transparent pass-through blocks around the region.
2. Run the simulation for at least one step so all signal sizes are known.
3. Call cg_end.generate_pyx_stub(cg_start, 'my_block', './codegen/').
4. Three files are written:

my_block.h	— C header with InputSignals / OutputSignals structs and void my_block_compute(const InputSignals*, OutputSignals*)
my_block_wrapper.pyx	— Cython glue that packs/unpacks flat NumPy arrays into structs

my_block_simblock.py — Ready-to-use SimBlockBase subclass with compute_py() stub and complete compute_c() implementation

5. C developer implements my_block_compute() in my_block.c.
6. Build: python setup_my_block.py build_ext --inplace
7. Switch: block = MyBlockSimBlock('b', use_c_backend=True)

4.3 ScriptBlock — inline Python with automatic C generation

ScriptBlock offers a third path: the user writes a short Python script inline as a string. The script is parsed at construction time into an AST, validated for forbidden operations (no import, no open, no eval), and then either executed directly or translated to a C function.

Mode	How it works
'python' (default)	exec() runs the script string in a prepared context dict each timestep. Variables that survive execution are saved into script_locals and injected back on the next call, giving the script persistent state (integrators, accumulators, etc.)
'c'	AST visitor translates the script to a C function body at construction time. The C code is written to a temp file, compiled via subprocess to a shared library, loaded with ctypes, and called via a fixed ABI: void compute(double* in, int n_in, double* out, int n_out, double* params, int n_params, double t, double dt).

5. Block Graph, Dependency Resolution, and DFS

5.1 How the graph is represented

EmbedSim has no explicit graph object. The block graph is stored entirely in the inputs lists: every block holds direct Python references to its upstream blocks. The >> operator simply appends:

```
def __rshift__(self, other):
    other.inputs.append(self)
    return other

# Wiring a chain:
source >> gain >> integrator >> sink
# is equivalent to:
gain.inputs       = [source]
integrator.inputs = [gain]
sink.inputs       = [integrator]
```

EmbedSim walks from sinks backward — the topology always describes signal flow from left (sources) to right (sinks), so traversal from sinks gives the execution order in reverse, which is then reversed back. This sink-anchored approach means unreachable blocks (ones not connected to any sink) are automatically excluded from the simulation.

5.2 Why DFS and not BFS?

Post-order DFS naturally produces a topologically sorted list. When the DFS finishes visiting all of block B's dependencies, it appends B — meaning B always appears after every block it depends on. BFS would require a separate counting step to achieve the same ordering guarantee. DFS also detects cycles trivially through a visiting set: if the traversal re-encounters a block currently on the call stack, a cycle exists.

5.3 Two-pass DFS — handling feedback loops

A naive single-pass DFS fails on feedback loops because it recurses into the same block it started from, causing infinite recursion. EmbedSim uses a two-pass strategy:

Pass 1 — find_loop_breakers()

A full recursive traversal of the entire graph from every sink.

Purpose: collect every block that implements the LoopBreaker interface.

This pass accepts cycles because it only records instances, never backtracks on them.

Result: a Python set of loop-breaker block references.

Pass 2 — dfs() with loop-breaker pruning

Standard post-order DFS. A visiting set detects true algebraic loops.

When a dependency edge points to a block in the loop_breakers set:

— the loop-breaker block is added to the execution order immediately,

— its own inputs are NOT followed (the cycle is cut here).

When a block in visiting is encountered again: ValueError — true algebraic loop,

no LoopBreaker is present to resolve it.

Result: execution order list, topologically sorted, with feedback cycles resolved.

```
# Simplified view of the two-pass logic
def traverse_blocks_from_sinks_with_loops(sinks):
    loop_breakers = set()
    blocks_order  = []

    # Pass 1: find every LoopBreaker anywhere in the graph
    def find_loop_breakers(block):
        if isinstance(block, LoopBreaker):
            loop_breakers.add(block)
        for inp in block.inputs:
            find_loop_breakers(inp)

    # Pass 2: topological sort, cutting at loop-breaker edges
    visiting = set()
    def dfs(block):
        if block in visiting:
            raise ValueError(f'Algebraic loop at {block.name}')
        visiting.add(block)
        for inp in block.inputs:
            if inp in loop_breakers:
                if inp not in blocks_set: blocks_order.append(inp)   # cut edge
            else:
                dfs(inp)   # normal recursion
        visiting.remove(block)
        blocks_order.append(block)
    for sink in sinks: dfs(sink)
    return blocks_order
```

5.4 LoopBreaker — breaking the cycle causally

A LoopBreaker block resolves a feedback cycle by outputting its value from the previous timestep rather than its current input. This is mathematically correct for any discrete-time feedback system: the delayed signal is causal and introduces exactly one sample of latency, which is identical to a unit-delay element (z^{-1} in the z-domain).

Before the main compute pass, the engine pre-seeds every LoopBreaker's output attribute with its stored previous value:

```
# At the start of every timestep:
for block in self.blocks:
    if isinstance(block, LoopBreaker) and block.output is None:
        block.output = block.get_loop_breaking_output()   # previous value

# Now the main pass runs — downstream blocks read the pre-seeded value
# when they encounter the loop-breaker as their input.
```

VectorDelay in simulation_engine.py is the standard LoopBreaker. It stores the current input as last_output each step and returns that on the next step. Any subclass (e.g. StickyDelay in the parallel parking example) that also inherits LoopBreaker is automatically detected by the engine during graph traversal.

6. ODE Integration

Dynamic blocks (VectorIntegrator, StateSpaceBlock, TransferFunctionBlock) hold a continuous internal state vector x. At each timestep the engine advances x using the selected ODE solver. Two solvers are implemented.

Solver	Description
ODESolver.EULER (forward Euler)	$x(t+dt) = x(t) + dt \cdot f(x,u,t)$. One derivative evaluation per step. First-order accuracy: $O(dt)$ error per step. Use for fast prototyping or when dt is very small relative to system dynamics. Best for non-stiff systems.
ODESolver.RK4 (Runge-Kutta 4)	$x(t+dt) = x(t) + (dt/6)(k1 + 2k2 + 2k3 + k4)$. Four derivative evaluations per step, requiring three extra full compute passes (_compute_all_blocks at t+dt/2 and t+dt). Fourth-order accuracy: $O(dt^4)$ error. Use for accurate simulation of stiff or fast-changing systems. Roughly 4× the cost of Euler.

Important design point: for RK4, the engine re-evaluates the entire block graph at intermediate time points to get consistent input values for each stage. This is more expensive but necessary — using stale inputs from t for stages at t+dt/2 would corrupt the accuracy of the integration.

7. Simulation Loop

The EmbedSim.run() method orchestrates everything. The loop is fixed-step:

```
for step in range(int(T / dt)):
    1. _compute_all_blocks(t)
       a. Pre-seed all LoopBreaker outputs from previous timestep
       b. Iterate blocks in topological order
       c. For each block: gather input_values from upstream .output attributes
       d. Call block.compute(t, dt, input_values)  → routes to Python or C

    2. scope.record(t)
       Sample every monitored block's .output into the data dict

    3. ODE integration
       Euler: one get_derivative() + integrate_state() per dynamic block
       RK4:   four stages, each requiring a full compute pass

    4. t += dt

# Final sample at t = T (captures the last state)
_compute_all_blocks(T)
scope.record(T)
```

Blocks are reset (output = None, last_output = None) before the loop starts, clearing any state from a previous run. The block's reset() is responsible for clearing internal state vectors in dynamic blocks.

8. Signal Recording — VectorScope

VectorScope is EmbedSim's oscilloscope. You register blocks before the simulation run; the scope samples their .output at every timestep.

```
sim.scope.add(motor_block, indices=[0], label='speed')   # record component 0 only
sim.scope.add(abc_block)                                  # record all components
sim.run()
speed = sim.scope.get_signal('speed', index=0)            # np.ndarray shape (N,)
abc   = sim.scope.get_full_signal('abc_block')            # shape (N, 3)
```

Data is stored in two parallel structures:

- **scope.data**: dict keyed by 'label[i]' — fast scalar access for plotting.
- **scope.full_signals**: dict keyed by label — full vector snapshots as list of arrays.
- **scope.t**: flat list of simulation timestamps.

9. FMU Co-Simulation Integration

FMUBlock is an adapter from VectorSignal semantics to the FMI 2.0 Co-Simulation protocol. An FMU (.fmu file) is a ZIP archive containing compiled model binaries and a modelDescription.xml listing all variables and their integer value references. FMUBlock manages the complete FMU lifecycle:

Phase	What happens

Construction	Extracts the FMU ZIP, parses modelDescription.xml with fmpy, caches value references (integer IDs) for all named input and output variables.
Initialization	Instantiates FMU2Slave, calls setupExperiment(), enterInitializationMode(), sets parameter values via setReal(), calls exitInitializationMode().
Per-step	EmbedSim calls compute(t, dt, input_values). FMUBlock calls fmu.setReal(refs, vals) to push input signals, then fmu.doStep(t, dt) to advance the model's internal solver, then fmu.getReal(refs) to pull output values. Returns a VectorSignal.
Termination	FMU.terminate() and FMU.freeInstance() are called on del or explicit reset.

Why value references matter: FMI does not look up variables by string name at runtime (that would be too slow). Every variable is assigned an integer valueReference at compile time. FMUBlock caches these references once during initialization, so the hot per-step path is purely numeric integer-indexed calls.

10. C Code Generation Pipeline (CodeGenStart / CodeGenEnd)

The code generation system exists to answer one question: given a region of the simulation diagram, produce everything a C developer needs to write and integrate an equivalent embedded implementation.

10.1 How boundary markers work

CodeGenStart and CodeGenEnd are transparent pass-through blocks. They produce no computation of their own — they just let signals flow through while recording the signal names and sizes at their boundary. After running the simulation at least one step, block.output is populated and signal dimensions are known.

```
sensor   >> cg_start >> pid_block >> cg_end >> actuator
#            ↑ region entry          ↑ region exit
#
cg_end.generate_pyx_stub(cg_start, 'pid_controller', './codegen/')
```

10.2 Generated files

File	Purpose
pid_controller.h	C header. Defines InputSignals and OutputSignals typedef structs using exact field names taken from block.name attributes. Declares void pid_controller_compute (const InputSignals*, OutputSignals*). The C developer implements this signature in pid_controller.c.
pid_controller_wrapper.pyx	Cython wrapper. cimports the C header, allocates InputSignals and OutputSignals structs on the stack (no heap). set_inputs() packs a flat double array into the struct. compute() calls the C function with nogil. get_outputs() unpacks the result into a NumPy array.
pid_controller_simblock.py	SimBlockBase subclass. Has a complete compute_c() calling the Cython wrapper and a stub compute_py() showing the expected input/output layout for a Python reference implementation.

setup_pid_controller.py	Setuptools build script. Detects compiler (MSVC /O2 vs GCC -O3 -ffast-math). Compiles .pyx and .c together into a single extension module so no separate .so/.dll linking step is needed.

10.3 Why Cython and not cffi or ctypes?

Cython was chosen because it uniquely combines three properties that matter for this use case:

- **Struct-on-stack semantics:** cdef struct members live in C stack memory, not the Python heap. There is literally zero allocation on the hot compute path.
- **GIL release:** with nogil: allows the C function to run with full thread scheduling freedom. This matters when EmbedSim is later parallelised across subsystems.
- **Type declarations:** cpdef void compute() and double[::1] (C-contiguous view) eliminate all Python boxing overhead for numerical types.
- **Generated code is debuggable:** The annotated .html file produced by cythonize(annotate=True) shows exactly which lines become C and which remain Python, making performance tuning transparent.

11. Precision Architecture

Embedded control runs in float32. Desktop simulation tools default to float64. EmbedSim resolves this mismatch with a three-level precision system:

Level	How to use it
Global default (float32)	Set once in core_blocks.py: DEFAULT_DTYPE = np.float32. All blocks, all signals, all state vectors default to this. Nothing else needs to change.
Per-block override	Pass dtype=np.float64 to any block constructor. That block and its outputs use float64, while the rest of the diagram stays float32. Useful for FMU interfaces and ScriptBlock C execution (C ABI uses double).
Global override	import embedsim.core_blocks as cb; cb.DEFAULT_DTYPE = np.float64 before any blocks are created. All subsequent construction uses float64.

Why float32 matters on Aurix TriCore

The TriCore instruction set has a 32-bit FPU. float64 operations require software emulation, roughly 4×10× slower. CAN/FlexRay frame payloads are typically 8 bytes — one float64 fills an entire slot; two float32s fit. The framework matching the MCU's native type eliminates type conversion at the simulation boundary.

12. Topology Visualisation

EmbedSim.print_topology() prints a concise execution-order table.
EmbedSim.print_topology_sources2sink() renders the full block diagram as ASCII art in the terminal. The ASCII renderer:

- Assigns each block a lane (row) based on topological depth.
- Draws signal flow as horizontal arrows with box labels.
- Renders fan-out branches with box-drawing characters (┬, └, ┘, ├).
- Renders feedback arcs below the main diagram with ◀ re-entry arrows labelled by the loop-breaker block name and type.
- ⚡ marks dynamic blocks; 🔄 marks loop-breaker blocks in the execution table.

13. End-to-End Developer Workflow

1. **Prototype in Python.** Build the full diagram using VectorBlock subclasses with use_c_backend=False. Use ScriptBlock for any algorithm you have not yet formalised.
2. **Verify numerically.** Run EmbedSim with ODESolver.RK4, record signals with VectorScope, plot results. Fix the algorithm in Python — iteration is fast with no compilation step.
3. **Identify the codegen region.** Insert CodeGenStart and CodeGenEnd around the subsystem that will run on the MCU (typically the control law, excluding the plant model).
4. **Generate the C interface.** Run the simulation one step, call generate_pyx_stub(). Hand the .h file and the signal documentation to the embedded software team.
5. **C developer implements the algorithm.** They write the .c file against the generated header. The struct field names match the block names from the Python diagram — no translation layer.
6. **Compile the Cython wrapper.** python setup_pid.py build_ext --inplace. Produces a .pyd/.so extension loadable by Python.
7. **Switch and verify.** Set use_c_backend=True on the generated SimBlock, re-run the simulation. Outputs should match the Python run to floating-point tolerance. Any divergence indicates a bug in the C implementation.
8. **Deploy.** The C source compiled in step 6 is the identical code that runs on the MCU. Replace the Cython wrapper glue with a direct hardware call in the embedded firmware.

14. Key Design Decisions and Trade-offs

Decision	Rationale and trade-off
Graph stored in .inputs lists, no separate graph object	Zero overhead — no adjacency matrix, no node map. Trade-off: harder to serialise the diagram, no built-in graph query API. Acceptable because the primary use case is construction + run, not graph inspection.
Sink-anchored traversal	Automatically excludes unconnected blocks. Natural match for block-diagram thinking where data flows to outputs. Trade-off: requires at least one sink; pure sources with no sink are never executed.
Two-pass DFS rather than iterative Kahn's algorithm	DFS maps naturally to the recursive inputs structure and gives free cycle detection. Kahn's algorithm would be more memory-efficient for huge graphs but requires building an explicit adjacency structure first — overkill for control diagrams that rarely exceed 100 blocks.
LoopBreaker as a mixin interface	Any block can become a loop breaker by inheriting LoopBreaker alongside VectorBlock. This composability allows StickyDelay (the parallel parking example) to add custom reset and dimension-guard logic while still being recognised by the

	engine as a cycle-cutter. Trade-off: requires knowing to inherit both base classes.
ScriptBlock AST translation rather than a DSL	Python is already the language engineers are using. Translating the same Python expressions through the AST means no new syntax to learn. Trade-off: only a subset of Python is translatable to C (no object methods, no list comprehensions); complex algorithms need the full code-gen pipeline instead.
Cython rather than ctypes for the wrapper	Cython gives struct-on-stack allocation and GIL release, which ctypes cannot provide. Trade-off: requires a build step. The generated setup.py automates this completely for both Windows and Linux.
float32 global default	Matches the target MCU (Aurix TriCore native FPU width). Numerical validation runs in the same precision as production. Trade-off: less headroom for iterative accumulation in long simulations; override to float64 when needed.

15. Glossary

Term	Definition
VectorBlock	Abstract base class. Every simulation element inherits from this.
VectorSignal	Thin NumPy array wrapper. The data unit flowing between blocks.
LoopBreaker	Mixin interface marking a block that provides its previous-timestep output to cut a feedback cycle.
DFS	Depth-First Search. Used to build the topological execution order from sink to source.
Topological order	Execution sequence in which every block appears after all its non-loop-breaking dependencies.
compute_py()	Python implementation of a block's transfer function. Override in subclasses.
compute_c()	C/Cython implementation. Calls the compiled wrapper with GIL released.
CodeGenStart/End	Transparent boundary markers. After a simulation run, generate .h, .pyx, _simblock.py.
FMU	Functional Mock-up Unit. A compiled simulation model following the FMI standard.
FMI	Functional Mock-up Interface. Standard for portable simulation model exchange.
Value reference	Integer ID used by FMI to identify variables; faster than string lookup at runtime.
Cython	Python superset that compiles to C. Used to wrap hand-written C blocks with zero GIL overhead.
DEFAULT_DTYPE	Global precision constant in core_blocks.py. Change to np.float64 to switch the whole framework.

VectorScope	Signal recorder. Samples monitored block outputs at every timestep for post-run analysis.
SimulationStats	Dataclass holding total_steps, compute_time, loop_breakers_count after a run.
MCUTarget	Namespace class listing MCU identifier strings (AURIX_TRICORE, CORTEX_M4) used by code generators.