API Reference

Namespace: `SolidCore.Coroutines`

All types are auto-referenced from the bundled assemblies — just add `using SolidCore.Coroutines;`. Inline docs for every member are also available in your IDE via the shipped `.xml` files.

Looking for other assemblies?

- **Burst Reference** — Unmanaged collections, hashing, and SIMD utilities.
- **Collections Reference** — Managed `FastList`, `IndexMap`, and the zero-alloc Query engine.
- Front doors — `CoroutineRunner`, `CoroutineJobSystem`
- Authoring — `SequenceBuilder`, `Yield`, `InstructionSequence`
- Handles & requests — `CoroutineHandle`, `CoroutineSpawnRequest`
- In-coroutine helpers — `CoroutineRuntime`
- Diagnostics — `CoroutineDiagnostics`
- Timing — `CoroutineTime`
- Advanced / low-level

Front doors

`CoroutineRunner` (single-threaded)

The main entry point. Owns a scheduler and allocator so you never touch native buffers directly.

Creation

- `static CoroutineRunner Create(int expectedPeak)` — create pre-allocated for a known peak count.
- `new CoroutineRunner(int capacity)` — scheduler prewarmed for `capacity` coroutines.

Authoring

- `SequenceBuilder Sequence(int capacity = …)` — start a fluent builder bound to this runner's allocator (finish with the no-arg `Build()`).

Spawning

- `CoroutineHandle Start(InstructionSequence sequence)` — start one instance; returns its handle.
- `void Start(InstructionSequence sequence, int count)` — start `count` instances of a shared template (batch, allocation-free).
- `void Stop(CoroutineHandle handle)` — stop a running coroutine by handle.

Ticking

- `void Tick(float deltaTime)` — advance one frame. Call this in your game loop.
- `int RunToCompletion(float deltaTime, int maxTicks)` — headless drain in **simulated** time (as fast as the CPU allows); returns ticks executed. For tests/batch sim.
- `int StartAndRun(InstructionSequence sequence, int count, float deltaTime)` — start a wave and drain it headless in one call.
- `int RunRealtime(float fixedDeltaTime, float maxSeconds)` — **wall-clock** pump: paces fixed steps to real time so `WaitForSeconds(2)` blocks ~2 s. `maxSeconds = 0` means no timeout. Blocks the calling thread.
- `int StartAndRunRealtime(InstructionSequence sequence, int count, float fixedDeltaTime, float maxSeconds)` — wall-clock mirror of `StartAndRun`.

State / diagnostics

- `int ActiveCount` — live coroutine count. O(1).
- `long TotalStarted`, `long TotalCompleted` — lifetime counters.
- `CoroutineAllocator Allocator` — the owned template/buffer allocator.
- `CoroutineScheduler Scheduler` — underlying scheduler (advanced; exposes `LastTickTimeMs`, `PeakTickTimeMs`, etc.).
- `string GetPerformanceSummary()` — formatted performance report.
- `static CoroutineRunner Current` — the runner ticking on this thread (or null), for local spawns.

Lifecycle

- `void Dispose()` — frees the scheduler and allocator. Required.

`CoroutineJobSystem` (multi-threaded)

Pool of lock-free worker shards over a shared allocator, driven fork/join. Call scheduling/tick methods from one owning thread; workers are idle between ticks.

Creation

- `static CoroutineJobSystem Create(int expectedPeak, int workerCount = 0)` — `workerCount = 0` ⇒ `ProcessorCount - 1`.
- `new CoroutineJobSystem(int workerCount, int shardCapacity)`

Authoring

- `SequenceBuilder Sequence(int capacity = …)` — builder bound to this system's allocator.
- `CoroutineAllocator Allocator`

Scheduling / spawning

- `void Schedule(in CoroutineSpawnRequest request)` — distribute `request.Count` instances evenly across shards.
- `void Schedule(InstructionSequence template, int count)` — convenience overload.
- `void ScheduleOn(int shard, InstructionSequence template, int count)` — spawn all onto one specific shard.

- `void Spawn(InstructionSequence template, int count)` — alias for `Schedule`.
- `void SpawnOne(InstructionSequence template)` — spawn a single instance.
- `void FlushSpawns()` — create all queued coroutines in parallel without ticking (front-load a wave's creation cost).

Ticking

- `void Tick(float deltaTime)` / `void TickParallel(float deltaTime)` — advance every shard in parallel, then join (pending spawns created first).
- `int RunToCompletion(float deltaTime, int maxTicks)`
- `int SpawnAndRun(InstructionSequence template, int count, float deltaTime)`

State / diagnostics (read between frames)

- `int WorkerCount`
- `int ActiveCount` — total live coroutines across all shards.
- `CoroutineDiagnostics GetDiagnostics()` — merged cross-shard stats.

Lifecycle

- `void Dispose()` — stops all workers and frees the shared allocator.

Authoring

SequenceBuilder

Fluent, authoring-time builder. Obtain via `runner.Sequence()` / `jobs.Sequence()`. Bakes into an allocator-owned template on `Build()`.

- `SequenceBuilder Do(Action action)` — run a managed action, then advance.
- `SequenceBuilder WaitForSeconds(float seconds)` — wait in simulated seconds.
- `SequenceBuilder WaitForFrames(int frames)` — wait N scheduler ticks.
- `SequenceBuilder WaitUntil(Func<bool> predicate)` — block until the predicate is `true`.
- `SequenceBuilder Add(in YieldInstructionData instruction)` — append a raw instruction (prefer the helpers above).
- `int Count` — instructions added so far.
- `InstructionSequence Build()` — bake into the bound allocator (throws if unbound).
- `InstructionSequence Build(CoroutineAllocator allocator)` — bake into a specific allocator.
- `void Dispose()` — free scratch memory if never built.

`Do` *and* `WaitUntil` *take ordinary managed delegates (lambdas, method groups, closures). This is the IL2CPP/Mono-compatible form.*

`Yield`

Lower-level factory for `YieldInstructionData` values (the `SequenceBuilder` helpers wrap these):

- `Yield.WaitForSeconds(float duration)`
- `Yield.WaitForFrames(int frames)`
- `Yield.WaitUntil(Func<bool> predicate)`
- `Yield.Do(Action action)`

`InstructionSequence`

Immutable native instruction buffer, shared as a flyweight template across many coroutine instances. Owned and freed by the allocator. Treat it as an opaque, reusable handle — build once, spawn many.

Handles & requests

`CoroutineHandle`

Value-type identifier for one running instance (slot index + generation version). Stale handles are invalidated automatically when a slot is recycled.

- `int Index`, `int Version`
- `bool IsValid`

`CoroutineSpawnRequest`

A unit of batch work: spawn `Count` instances of one shared `Template`.

- `new CoroutineSpawnRequest(InstructionSequence template, int count)` — `count` clamped to ≥ 1.
- `InstructionSequence Template`, `int Count`

In-coroutine helpers

`CoroutineRuntime`

Static helpers callable from **inside** a running coroutine (a `Do` action) to spawn into the runner/shard currently ticking on this thread — lock-free, never crosses a thread boundary.

- `bool HasCurrentShard`
- `int CurrentShardId` — `-1` outside a tick or for a single-threaded runner.
- `bool SpawnLocal(InstructionSequence template)` — spawn one; no-op (false) if outside a tick.
- `bool SpawnLocal(InstructionSequence template, int count)` — spawn many.

Diagnostics

`CoroutineDiagnostics`

Merged cross-shard snapshot from `CoroutineJobSystem.GetDiagnostics()` (gather between frames).

- `int ShardCount`, `int ActiveCount`

- `long TotalStarted`, `long TotalCompleted`, `long TotalInstructions`, `long TotalTicks`
- `double AggregateCpuTimeMs` — total work across all worker threads.
- `double CriticalPathMs` — slowest shard's tick time (the parallel frame's cost).
- `double PeakTickMs` — worst single tick on any shard.
- `double ThroughputPerSecond` — instructions/sec of wall-clock time.
- `double ParallelSpeedup` — aggregate CPU work ÷ critical-path time (≈ shard count when ideal).
- `string Summary()` — formatted multi-line report.

Timing

`CoroutineTime`

Deterministic fixed-timestep clock (accumulator pattern); pure value type, zero-alloc.

- `new CoroutineTime(float fixedDeltaTime, int maxStepsPerFrame)`
- `float FixedDeltaTime`, `float Accumulator`, `float TotalTime`, `int FrameCount`, `int MaxStepsPerFrame`
- `int Accumulate(float realDeltaTime)` — feed real elapsed time; returns how many fixed steps to run.

- `void Step()` — advance the clock one fixed step (call once per executed step).
- `float Alpha` — interpolation factor [0,1) for smoothing rendering between fixed steps.

Advanced / low-level

You typically won't touch these directly — `CoroutineRunner` / `CoroutineJobSystem` own them — but they're public for advanced scenarios:

Type	Role
`CoroutineScheduler`	Single-shard, zero-GC scheduler over contiguous native state. Not thread-safe. Exposes `LastTickTimeMs`, `PeakTickTimeMs`, `TotalTickTimeMs`, `ActiveCount`, `Update(dt)`, `GetPerformanceSummary()`.
`CoroutineAllocator`	Owns/recycles the native memory backing sequences. `CreateTemplate`, `Prewarm`, `Rent`/`Return`, `Dispose`.
`CoroutineWorker`	One background thread owning a scheduler shard (used by the job system).
`CoroutineCallbacks`	Process-wide registry mapping callback ids → managed delegates (the indirection that keeps instructions blittable).

`CoroutineState`, `CoroutineStatus`	Per-instance state and lifecycle enum (`Inactive`, `Running`, `Waiting`, `Completed`).
`YieldInstructionData`, `YieldInstructionType`	Blittable tagged-union describing one instruction (`None`, `WaitForSeconds`, `WaitForFrames`, `WaitUntil`, `Do`).