Burst Reference

Namespace: `SolidCore.Burst`

The `SolidCore.Burst` assembly provides low-level, zero-allocation utilities and collections optimized for unmanaged memory. These are used internally by the coroutine runner but can be used standalone for high-performance scenarios.

- Unmanaged Collections — `BurstArray<T>`, `BurstList<T>`, `BurstMap<TKey, TValue>`
- Hashing Utilities — `BurstHash`
- SIMD Math — `Simd`

Unmanaged Collections

All unmanaged collections must be manually disposed to avoid memory leaks, as they are not tracked by the C# Garbage Collector.

`BurstArray<T>`

Fixed-length array in unmanaged memory.

Properties

- `int Length` — Element count.

Methods

- `BurstArray(int length)` — Constructor. Allocates `length` zero-initialized elements.
- `ref T this[int index]` — Reference to the element at `index`. No bounds check.
- `Span<T> AsSpan()` — Mutable view over all elements. Invalidated by `Dispose()`.
- `ReadOnlySpan<T> AsReadOnlySpan()` — Read-only view over all elements. Invalidated by `Dispose()`.
- `void Dispose()` — Frees the backing memory and resets to empty.

BurstList<T>

Growable list in unmanaged memory. Doubling capacity when full.

Properties / Fields

- `int Count` — Element count.
- `int Capacity` — Current allocated slot count.
- `bool IsEmpty` — True when `Count` is zero.
- `T* Data` — Raw pointer to the backing buffer. No bounds check; invalidated by any growth.

Methods

- `BurstList(int capacity)` — Constructor. Allocates `capacity` zero-initialized slots (min 4).
- `ref T this[int index]` — Reference to the element at `index`. Throws if out of range.
- `void Add(in T value)` / `void Push(in T value)` — Appends `value`.
- `T Pop()` — Removes and returns the last element. Throws if empty.
- `bool TryPop(out T item)` — Pops the last element into `item`; returns false if empty.
- `void RemoveLast()` — Drops the last element. Throws if empty.
- `void RemoveAt(int index)` — Removes the element at `index` in O(1) by swapping in the last (unordered). Throws if out of range.
- `bool Remove(in T item)` — Removes the first matching `item` (unordered swap); false if absent. O(n) scan.
- `bool Contains(in T item)` — True if `item` is present. O(n) scan.
- `void Clear()` — Resets to empty; keeps allocated capacity.
- `void EnsureCapacity(int capacity)` — Grows to at least `capacity` slots if larger than current.
- `ReadOnlySpan<T> AsSpan()` — Read-only view over the live elements. Invalidated by any growth.
- `Enumerator GetEnumerator()` — Allocation-free forward enumerator.
- `void Dispose()` — Frees the backing memory and resets to empty.

`BurstMap<TKey, TValue>`

Open-addressed hash map in unmanaged memory using Robin-Hood probing.

Properties

- `int Count` — Live entry count.
- `int Capacity` — Current slot count (always a power of two).

Methods

- `BurstMap(int capacity, float maxLoadFactor)` — Constructor. Allocates buffers rounded up to a power of two (min 4); resizes past `maxLoadFactor`.
- `bool TryAdd(in TKey key, in TValue value)` — Inserts the pair; overwrites and returns false if `key` already exists, else true.
- `bool TryGet(in TKey key, out TValue value)` — Retrieves the value for `key` into `value`; returns false if absent.
- `ref TValue GetRef(in TKey key)` — Reference to the value for `key`. Throws `KeyNotFoundException` if absent.
- `bool Remove(in TKey key)` — Removes `key` with backward-shift compaction; returns false if absent.
- `void Dispose()` — Frees all backing buffers and resets to empty.

Hashing Utilities

`BurstHash` **(static)**

Allocation-free hashing helpers for unmanaged keys.

- `static ulong FNV1A64<T>(in T key)` — 64-bit FNV-1a hash over the raw bytes of `key`. Processes 8 bytes per step.
- `static int ToIndex(ulong hash, int capacity)` — Folds `hash` to a slot index. Requires power-of-two `capacity` (masks, no modulo).
- `static bool Equals<T>(in T a, in T b)` — Equality of two unmanaged keys.

SIMD Math

`Simd` (static)

Portable SIMD over contiguous `float` buffers via `Vector<T>` — accelerated on RyuJIT, software fallback on Mono/IL2CPP.

- `bool IsHardwareAccelerated` — True when `Vector<T>` is JIT-accelerated on this platform.
- `int FloatWidth` — Lane count of `Vector<T>` for `float` on this platform.
- `static void SubtractInPlace(float* data, int count, float scalar)` — Subtracts `scalar` from the first `count` floats at `data`, in place.
- `static bool SubtractInPlaceAnyLessOrEqualZero(float* data, int count, float scalar)` — As above, also returning whether any result is ≤ 0 — a fast-reject "did any timer expire" signal.

- `static void MultiplyAddInPlace(float* acc, float* factor, int count, float scalar)` – `acc[i] += factor[i] * scalar`, in place – vectorized fused multiply-add.